Programação Orientada para Objectos

Manuel Menezes de Sequeira

18 de Fevereiro de 2004

Who are the learned? They who practice what they know.

Muhammad, *The Sayings of Muhammad*,
Allama Sir Abdullah e Al-Mamun Al-Suhrawardy, editores, 10 (1949)

Conteúdo

Prefácio **xi**

0.1 Exercícios sobre classes . . . xiii

1 Introdução à Programação **1**

1.1 Computadores . . . 1

1.2 Programação . . . 2

1.3 Algoritmos: resolvendo problemas . . . 3

1.3.1 Regras do jogo . . . 4

1.3.2 Desenvolvimento e demonstração de correcção . . . 8

1.4 Programas . . . 16

1.5 Resumo: resolução de problemas . . . 18

2 Conceitos básicos de programação **19**

2.1 Introdução . . . 19

2.1.1 Consola e canais . . . 23

2.1.2 Definição de variáveis . . . 25

2.1.3 Controlo de fluxo . . . 26

2.2 Variáveis . . . 28

2.2.1 Memória e inicialização . . . 28

2.2.2 Identificadores . . . 30

2.2.3 Inicialização . . . 31

2.3 Tipos básicos . . . 31

2.3.1 Tipos aritméticos . . . 33

2.3.2 Booleanos ou lógicos . . . 39

2.3.3 Caracteres . . . 39

2.4 Valores literais 42
2.5 Constantes 43
2.6 Instâncias 46
2.7 Expressões e operadores 46
2.7.1 Operadores aritméticos 47
2.7.2 Operadores relacionais e de igualdade 49
2.7.3 Operadores lógicos 50
2.7.4 Operadores *bit*-a-*bit* 51
2.7.5 Operadores de atribuição 53
2.7.6 Operadores de incrementação e decrementação 54
2.7.7 Precedência e associatividade 55
2.7.8 Efeitos laterais e mau comportamento 55
2.7.9 Ordem de cálculo 57

3 Modularização: rotinas 59
3.1 Introdução à modularização 59
3.2 Funções e procedimentos: rotinas 62
3.2.1 Abordagens descendente e ascendente 63
3.2.2 Definição de rotinas 70
3.2.3 Sintaxe das definições de funções 73
3.2.4 Contrato e documentação de uma rotina 74
3.2.5 Integração da função no programa 76
3.2.6 Sintaxe e semântica da invocação ou chamada 77
3.2.7 Parâmetros 78
3.2.8 Argumentos 79
3.2.9 Retorno e devolução 79
3.2.10 Significado de `void` 79
3.2.11 Passagem de argumentos por valor e por referência 81
3.2.12 Variáveis locais e globais 89
3.2.13 Blocos de instruções ou instruções compostas 89
3.2.14 Âmbito ou visibilidade de variáveis 90
3.2.15 Duração ou permanência de variáveis 93
3.2.16 Nomes de rotinas 94

3.2.17 Declaração vs. definição 95
3.2.18 Parâmetros constantes 99
3.2.19 Instruções de asserção 102
3.2.20 Melhorando módulos já produzidos 112
3.3 Rotinas recursivas 115
3.4 Mecanismo de invocação de rotinas 117
3.5 Sobrecarga de nomes 124
3.6 Parâmetros com argumentos por omissão 126

4 Controlo do fluxo dos programas 129
4.1 Instruções de selecção 129
4.1.1 As instruções `if` e `if else` 130
4.1.2 Instruções de selecção encadeadas 133
4.1.3 Problemas comuns 136
4.2 Asserções 137
4.2.1 Dedução de asserções 137
4.2.2 Predicados mais fortes e mais fracos 139
4.2.3 Dedução da pré-condição mais fraca de uma atribuição 139
4.2.4 Asserções em instruções de selecção 141
4.3 Desenvolvimento de instruções de selecção 146
4.3.1 Escolha das instruções alternativas 147
4.3.2 Determinação das pré-condições mais fracas 148
4.3.3 Determinação das guardas 149
4.3.4 Verificação das guardas 149
4.3.5 Escolha das condições 150
4.3.6 Alterando a solução 151
4.3.7 Metodologia 152
4.3.8 Discussão 153
4.3.9 Outro exemplo de desenvolvimento 156
4.4 Variantes das instruções de selecção 159
4.4.1 O operador `? :` 159
4.4.2 A instrução `switch` 160
4.5 Instruções de iteração 164

4.5.1 A instrução de iteração `while` . . . 165
4.5.2 Variantes do ciclo `while`: `for` e `do while` . . . 166
4.5.3 Exemplo simples . . . 175
4.5.4 `return`, `break`, `continue`, e `goto` em ciclos . . . 175
4.5.5 Problemas comuns . . . 182
4.6 Asserções com quantificadores . . . 183
4.6.1 Somas . . . 184
4.6.2 Produtos . . . 185
4.6.3 Conjunções e o quantificador universal . . . 186
4.6.4 Disjunções e o quantificador existencial . . . 187
4.6.5 Contagens . . . 188
4.6.6 O resto da divisão . . . 189
4.7 Desenvolvimento de ciclos . . . 189
4.7.1 Noção de invariante . . . 193
4.7.2 Correcção de ciclos . . . 199
4.7.3 Melhorando a função `potência()` . . . 202
4.7.4 Metodologia de Dijkstra . . . 204
4.7.5 Um exemplo . . . 217
4.7.6 Outro exemplo . . . 229

5 Matrizes, vectores e outros agregados **235**
5.1 Matrizes clássicas do C++ . . . 236
5.1.1 Definição de matrizes . . . 237
5.1.2 Indexação de matrizes . . . 238
5.1.3 Inicialização de matrizes . . . 240
5.1.4 Matrizes multidimensionais . . . 241
5.1.5 Matrizes constantes . . . 242
5.1.6 Matrizes como parâmetros de rotinas . . . 242
5.1.7 Restrições na utilização de matrizes . . . 247
5.2 Vectores . . . 248
5.2.1 Definição de vectores . . . 249
5.2.2 Indexação de vectores . . . 249
5.2.3 Inicialização de vectores . . . 250

5.2.4 Operações 251
5.2.5 Acesso aos itens de vectores 251
5.2.6 Alteração da dimensão de um vector 252
5.2.7 Inserção e remoção de itens 253
5.2.8 Vectores multidimensionais? 254
5.2.9 Vectores constantes 255
5.2.10 Vectores como parâmetros de rotinas 255
5.2.11 Passagem de argumentos por referência constante 257
5.2.12 Outras operações com vectores 261
5.3 Algoritmos com matrizes e vectores 262
5.3.1 Soma dos elementos de uma matriz 262
5.3.2 Soma dos itens de um vector 266
5.3.3 Índice do maior elemento de uma matriz 266
5.3.4 Índice do maior item de um vector 272
5.3.5 Elementos de uma matriz num intervalo 272
5.3.6 Itens de um vector num intervalo 277
5.3.7 Segundo elemento de uma matriz com um dado valor 278
5.3.8 Segundo item de um vector com um dado valor 286
5.4 Cadeias de caracteres 288
5.4.1 Cadeias de caracteres clássicas 289
5.4.2 A classe `string` 291

6 Tipos enumerados **297**
6.1 Sobrecarga de operadores 299

7 Tipos abstractos de dados e classes C++ **301**
7.1 De novo a soma de fracções 302
7.2 Tipos Abstractos de Dados e classes C++ 307
7.2.1 Definição de TAD 308
7.2.2 Acesso aos membros 311
7.2.3 Alguma nomenclatura 311
7.2.4 Operações suportadas pelas classes C++ 312
7.3 Representação de racionais por fracções 312

7.3.1 Operações aritméticas elementares . . . 313
7.3.2 Canonicidade do resultado . . . 313
7.3.3 Aplicação à soma de fracções . . . 314
7.3.4 Encapsulamento e categorias de acesso . . . 318
7.3.5 Rotinas membro: operações e métodos . . . 319
7.4 Classes C++ como módulos . . . 325
7.4.1 Construtores . . . 326
7.4.2 Construtores por cópia . . . 332
7.4.3 Condição invariante de classe . . . 332
7.4.4 Porquê o formato canónico das fracções? . . . 334
7.4.5 Explicitação da condição invariante de classe . . . 335
7.5 Sobrecarga de operadores . . . 343
7.6 Testes de unidade . . . 347
7.7 Devolução por referência . . . 351
7.7.1 Mais sobre referências . . . 351
7.7.2 Operadores ++ e -- prefixo . . . 358
7.7.3 Operadores ++ e -- sufixo . . . 361
7.8 Mais operadores para o TAD `Racional` . . . 364
7.8.1 Operadores de atribuição especiais . . . 364
7.8.2 Operadores aritméticos . . . 368
7.9 Construtores: conversões implícitas e valores literais . . . 370
7.9.1 Valores literais . . . 370
7.9.2 Conversões implícitas . . . 371
7.9.3 Sobrecarga de operadores: operações ou rotinas? . . . 372
7.10 Operadores igualdade, diferença e relacionais . . . 372
7.10.1 Inspectores e interrogações . . . 373
7.10.2 Operadores de igualdade e diferença . . . 375
7.10.3 Operadores relacionais . . . 376
7.11 Constância: verificando erros durante a compilação . . . 377
7.11.1 Passagem de argumentos . . . 378
7.11.2 Constantes implícitas: operações constantes . . . 383
7.11.3 Devolução por valor constante . . . 387

7.11.4 Devolução por referência constante . . . 390
7.12 Reduzindo o número de invocações com `inline` . . . 391
7.13 Optimização dos cálculos com racionais . . . 395
7.13.1 Adição e subtracção . . . 396
7.13.2 Multiplicação . . . 399
7.13.3 Divisão . . . 400
7.13.4 Simétrico e identidade . . . 401
7.13.5 Operações de igualdade e relacionais . . . 401
7.13.6 Operadores especiais . . . 402
7.14 Operadores de inserção e extracção . . . 403
7.14.1 Sobrecarga do operador << . . . 404
7.14.2 Sobrecarga do operador >> . . . 407
7.14.3 Lidando com erros . . . 408
7.14.4 Coerência entre os operadores << e >> . . . 411
7.14.5 Leitura e escrita de ficheiros . . . 413
7.15 Amizades e promiscuidades . . . 417
7.15.1 Rotinas amigas . . . 417
7.15.2 Classes amigas . . . 419
7.15.3 Promiscuidades . . . 419
7.16 Código completo do TAD `Racional` . . . 420
7.17 Outros assuntos acerca de classes C++ . . . 434
7.17.1 Constantes membro . . . 434
7.17.2 Membros de classe . . . 435
7.17.3 Destrutores . . . 437
7.17.4 De novo os membros de classe . . . 439
7.17.5 Construtores por omissão . . . 440
7.17.6 Matrizes de classe . . . 443
7.17.7 Conversões para outros tipos . . . 443
7.17.8 Uma aplicação mais útil das conversões . . . 446

8 Programação baseada em objectos **449**
8.1 Desenho de classes . . . 451

9 Modularização de alto nível 453
9.1 Modularização física . . . 455
9.1.1 Constituição de um módulo . . . 456
9.2 Fases da construção do ficheiro executável . . . 456
9.2.1 Pré-processamento . . . 457
9.2.2 Compilação . . . 464
9.2.3 Fusão . . . 466
9.2.4 Arquivos . . . 469
9.3 Ligação dos nomes . . . 473
9.3.1 Vantagens de restringir a ligação dos nomes . . . 474
9.3.2 Espaços nominativos sem nome . . . 477
9.3.3 Ligação de rotinas, variáveis e constantes . . . 479
9.3.4 Ligação de classes C++ e tipos enumerados . . . 486
9.4 Conteúdo dos ficheiros de interface e implementação . . . 490
9.4.1 Relação entre interface e implementação . . . 491
9.4.2 Ferramentas de utilidade interna ao módulo . . . 491
9.4.3 Rotinas não-membro . . . 492
9.4.4 Variáveis globais . . . 492
9.4.5 Constantes globais . . . 492
9.4.6 Tipos enumerados não-membro . . . 493
9.4.7 Classes C++ não-membro . . . 494
9.4.8 Operações (rotinas membro) . . . 494
9.4.9 Atributos de instância . . . 495
9.4.10 Variáveis membro de classe . . . 495
9.4.11 Constantes membro de classe . . . 496
9.4.12 Classes C++ membro (embutidas) . . . 497
9.4.13 Enumerados membro . . . 497
9.4.14 Evitando erros devido a inclusões múltiplas . . . 498
9.4.15 Ficheiro auxiliar de implementação . . . 500
9.5 Construção automática do ficheiro executável . . . 500
9.6 Modularização em pacotes . . . 507
9.6.1 Colisão de nomes . . . 507

9.6.2 Espaços nominativos . . . 509
9.6.3 Directivas de utilização . . . 511
9.6.4 Declarações de utilização . . . 512
9.6.5 Espaços nominativos e modularização física . . . 513
9.6.6 Ficheiros de interface . . . 515
9.6.7 Ficheiros de implementação e ficheiros auxiliares de implementação . . . 515
9.6.8 Pacotes e espaços nominativos . . . 516
9.7 Exemplo final . . . 517

10 Listas e iteradores 523
10.1 Listas . . . 523
10.1.1 Operações com listas . . . 524
10.2 Iteradores . . . 527
10.2.1 Operações com iteradores . . . 528
10.2.2 Operações se podem realizar com iteradores e listas . . . 529
10.2.3 Itens fictícios . . . 530
10.2.4 Operações que invalidam os iteradores . . . 532
10.2.5 Conclusão . . . 533
10.3 Interface . . . 535
10.3.1 Interface de `ListaDeInt` . . . 535
10.3.2 Interface de `ListaDeInt::Iterador` . . . 540
10.3.3 Usando a interface das novas classes . . . 543
10.3.4 Teste dos módulos . . . 545
10.4 Implementação simplista . . . 547
10.4.1 Implementação de `ListaDeInt` . . . 547
10.4.2 Implementação de `ListaDeInt::Iterador` . . . 550
10.4.3 Implementação dos métodos públicos de `ListaDeInt` . . . 552
10.4.4 Implementação dos métodos públicos de `ListaDeInt::Iterador` . . . 556
10.5 Uma implementação mais eficiente . . . 558
10.5.1 Cadeias ligadas . . . 559

11 Ponteiros e variáveis dinâmicas 563

12 Herança e polimorfismo 565

13 Programação genérica **567**

14 Excepções e tratamento de erros **569**

A Notação e símbolos **587**

A.1 Notação e símbolos 587

A.2 Abreviaturas e acrónimos 590

B Um pouco de lógica **591**

C Curiosidades e fenómenos estranhos **593**

C.1 Inicialização 593

C.1.1 Inicialização de membros 594

C.2 Rotinas locais 594

C.3 Membros acessíveis "só para leitura" 595

C.4 Variáveis virtuais 597

C.5 Persistência simplificada 603

D Nomes e seu formato: recomendações **615**

E Palavras-chave do C++ **617**

F Precedência e associatividade no C++ **621**

G Tabelas de codificação ISO-8859-1 (Latin-1) e ISO-8859-15 (Latin-9) **625**

H Listas e iteradores: listagens **633**

H.1 Versão simplista 633

H.1.1 Ficheiro de interface: `lista_int.H` 633

H.1.2 Ficheiro de implementação auxiliar: `lista_int_impl.H` 638

H.1.3 Ficheiro de implementação: `lista_int.C` 643

I Listas e iteradores seguros **651**

Prefácio

Colocar aqui ideias sobre evolução das folhas:

- Notas marginais.
- Notas de pé de página.
- Minipages para Saber mais.
- Minipages para Curiosidades.
- Minipages para Exercícios de auto-avaliação.

!!Vou em 3.7.9! 2003/12/4

!!Atenção! Digo que é comum não usar const por valor nas declarações. Mas a verdade é que, sem o usar, na condição objectivo não há forma de garantir que não mudaram de valor!

!!Retirar referências a Arquitectura de Computadores e Sistemas Operativos e CA e LP.

!!Rever os capítulos posteriores ao das classes! CII devia ser CIC, etc.

!!Rever todo o texto de modo a usar instância no sentido de variável ou constante!

!!Avisar para os exemplos de sintaxe: não os copiar. Pôr algures agradecimentos aos colegas e alunos.

!!Terminar índice!

!!Nas asserções (controlo de fluxo), distinguir claramente entre demonstração directa e inversa (ver resumo das aulas práticas em HTML).

!!Substituir "idêntico" por "igual" excepto onde se pretender dizer "a mesma entidade".

!!Deixar claro que PBO e POO o ênfase não são os algoritmos: é a modelação. O que existe e que propriedades tem. E isso também é resolver problemas. Neste texto far-se-á uma (discutível) abordagem histórica da programação: primeiro o ênfase serão os algoritmos (programação procedimental estruturada), depois os tipos de dados (programação baseada em objectos) e finalmente as entidades existentes no problema e suas relações e comportamento (programação orientada para objectos).

!!Eliminar todos os "!!"

!!Verificar ortografia.

!!Nas classes usar CII logo desde o início.

!!Nas classes introduzir noção de inspector cedo.

!!Herança: usar "tipo estático" vs. "tipo dinâmico". "É um", mais rigor: "Funciona como um", mais rigor: "Usável como um".

!!Herança privada: só se for necessário fazer sobreposição de operação polimórfica ou se for necessário acesso protegido.

!!"Tem um" pode significar "implementada à custa de".

!!Herança (Exceptional C++): métodos não exigem mais, nem prometem menos.

!!Em ADT deve-se definir sempre um swap externo.

!!Usar PilhaDeInt e PilhaDe<T>.

!!Ser sistemático na devolução de constantes em rotinas ou métodos que devolvam instâncias de classes sem ser por referência (ou em modelos, quanto o tipo devolvido for parâmetro do modelo, pois pode vir a ser uma classe).

!!Políticas de gestão de objectos dinâmicos:

quem constrói destrói (bom para composição, mau para agregação). Usar auto_ptr sem transferência de posse, por exemplo.

quem tem destrói (bom para agregação, mas não permite referências múltiplas). Usar auto_ptr com transferência de posse.

último fecha a porta (agregação com posse partilhada). Ponteiros espertos com contagem de referências.

!!Verificar se material sobre canais existe nas folhas o suficiente. Capítulo ou espalhado pelos capítulos?

!!Fazer breve demonstração da preservação do invariante nos métodos desenvolvidos para a classe Racional? É útil?

!!No capítulo de sobre modularização física, fazer figuras UML? Seria útil, particularmente para o exemplo final.

Capítulo 2:

!! Explicar algures int() e o que são variáveis temporárias (sem nome!). UML: objecto sem nome!

Capítulo 3:

!!Atenção! Usar nomes de funções como absolutoDe(). Corrigir tudo.

!!Usar a expressão "parâmetros com argumentos por omissão". Colocar na modularização!

!!Na modularização referir que normalmente se pretendem módulos com PC fraca e CO forte.

!! Aqui deveriam vir melhores recomendações! Ver erros mais frequentes do Problema 1.

!! Atenção às figuras! Os valores devolvidos devem ser objectos UML sem nome!!!! Variáveis temporárias!

Capítulo 7:

!! Discutir comentários de documentação: ver nos erros mais frequentes.

Capítulo 9:

!! Usar como exemplo a classe dos racionais dividida em ficheiros!

!! Dizer que modularização física acelera o desenvolvimento.

!! using namespace X nunca em .H ou _impl.H. Nunca em .C se o pacote pertencer ao espaço X!

!! Em X.H de namespace Y usar #ifndef Y_X_H, e por X.H em Y/X.H.

0.1 Exercícios sobre classes

3. O cálculo da raiz quadrada de um valor v pode ser feito, de uma forma algo ineficiente, procurando a solução da equação f(x) = v, onde f(x) := x2 (em que se usou a notação := com o significado "é definida por"), usando o método de Newton. Este método advoga que se deve construir uma sequência r0, r1, ... de raízes definida, de uma forma recorrente, por rn+1 = rn - (rn2 - v) / (2rn). Esta sequência converge para a raiz quadrada (positiva) de v quando n tende para infinito, desde que o valor inicial r0 seja positivo. Escreva uma função Racional raizQuadrada(Racional v) que calcule um valor racional que aproxime a raiz quadrada do racional v. Considere a aproximação razoável quando a diferença entre termos sucessivos da sequência for inferior a 1/100 (ou seja, quando |rn+1 - rn| = |(rn2 - v) / (2rn)| < 1/100).

Se procurar a raiz quadrada de 2 usando o método sugerido, chegará à surpreendente fracção 577/408, que é uma excelente aproximação (577/408 = 1,41421568..., cujo quadrado é 2,000006007...).

4. [difícil] Os inteiros são limitados, pelo que, em máquinas em que os int são representados em complemento para 2 e têm 32 bits, o valor racional mais pequeno (em módulo) representável pela classe Racional tal como desenvolvida é 2-32 (aproximadamente 10-10) e o maior (em módulo) 232 (aproximadamente 1010). Se se pretendesse tornar a classe dos racionais verdadeiramente útil, seria necessário estender a gama dos int consideravelmente. Não sendo isso possível sem mudar de linguagem, compilador, sistema operativo e/ou computador, a solução pode passar por construir uma nova classe Inteiro, à custa da qual a classe Racional pode ser construída, e que represente inteiros de precisão (virtualmente) arbitrária. Construa essa classe, usando sequências de int para representar os inteiros de precisão arbitrária. Pode ter de usar memória dinâmica, a ensinar posteriormente.

Capítulo 1

Introdução à Programação

It has often been said that a person does not really understand something until after teaching it to someone else. Actually, a person does not really understand something until after teaching it to a computer (...)

Donald E. Knuth, *Selected Papers in Computer Science*, 10 (1996)

1.1 Computadores

O que é um computador? As respostas que surgem com mais frequência são que um computador é um conjunto de circuitos integrados, uma ferramenta ou uma máquina programável. Todas estas respostas são verdadeiras, até certo ponto. No entanto, um rádio também é um conjunto de circuitos integrados, um martelo uma ferramenta, e uma máquina de lavar roupa uma máquina programável... Algo mais terá de se usar para distinguir um computador de um rádio, de um martelo, ou de uma máquina de lavar roupa. A distinção principal está em que um computador pode ser programado para resolver virtualmente qualquer problema ou realizar praticamente qualquer tarefa, ao contrário da máquina de lavar roupa em que existe um conjunto muito pequeno de possíveis programas à escolha e que estão pré-definidos. I.e., um computador tipicamente não tem nenhuma aplicação específica: é uma máquina genérica. Claro que um computador usado para meras tarefas de secretariado (onde se utilizam os ditos "conhecimentos de informática do ponto de vista do utilizador") não é muito diferente de uma máquina de lavar roupa ou de uma máquina de escrever electrónica.

Um computador é portanto uma máquina programável de aplicação genérica. Serve para resolver problemas de muitos tipos diferentes, desde o problema de secretaria mais simples, passando pelo apoio à gestão de empresas, até ao controlo de autómatos e à construção de sistemas inteligentes. O objectivo desta disciplina é ensinar os princípios da resolução de problemas através de um computador.

Os computadores têm uma arquitectura que, nos seus traços gerais, não parece muito complicada. No essencial, os computadores consistem num processador (o "cérebro") que, para além de dispor de um conjunto de registos (memória de curta duração), tem acesso a uma memória (memória de média duração) e a um conjunto de dispositivos de entrada e saída de dados, entre os quais tipicamente discos rígidos (memória de longa duração), um teclado e um ecrã. Esta é uma descrição simplista de um computador. Na disciplina de Arquitectura de Computadores os vários componentes de um computador serão estudados com mais profundidade. Na disciplina de Sistemas Operativos, por outro lado, estudar-se-ão os programas que normalmente equipam os computadores de modo a simplificar a sua utilização.

1.2 Programação

Como se programa um computador? Os computadores, como se verá na disciplina de Arquitectura de Computadores, entendem uma linguagem própria, usualmente conhecida por linguagem máquina. Esta linguagem caracteriza-se por não ter qualquer tipo de ambiguidades: a interpretação das instruções é única. Por outro lado, esta linguagem também se caracteriza por consistir num conjunto de comandos ou instruções que são executados "cegamente" pelo computador: é uma linguagem imperativa. Os vários tipos de linguagens (imperativas, declarativas, etc.), as várias formas de classificar as linguagens e as vantagens e desvantagens de cada tipo de linguagem serão estudados na disciplina de Linguagens de Programação.

A linguagem máquina caracteriza-se também por ser penosa de utilizar para os humanos: todas as instruções correspondem a códigos numéricos, difíceis de memorizar, e as instruções são muito elementares. Uma solução parcial passa por atribuir a cada instrução uma dada mnemónica, ou seja, substituir os números que representam cada operação por nomes mais fáceis de decorar. Às linguagens assim definidas chama-se linguagens *assembly* e aos tradutores de *assembly* para linguagem máquina *assemblers* ou assembladores.

Uma solução preferível passa por equipar os computadores com compiladores, isto é com programas que são capazes de traduzir para linguagem máquina programas escritos em linguagens mais fáceis de usar pelo programador e mais poderosas que o *assembly*. Infelizmente, não existem ainda compiladores para as linguagens naturais, que é o nome que se dá às linguagens humanas (e.g., português, inglês, etc.). Mas existem as chamadas linguagens de programação de alto nível (alto nível entre as linguagens de programação, bem entendido), que se aproximam um pouco mais das linguagens naturais na sua facilidade de utilização pelos humanos, sem no entanto introduzirem as imprecisões, ambiguidades e dependências de contextos externos que são características das linguagens naturais. Nesta disciplina usar-se-á o C++ como linguagem de programação[1].

Tal como o conhecimento profundo do português não chega para fazer um bom escritor, o conhecimento profundo de uma linguagem de programação não chega para fazer um bom programador, longe disso. O conhecimento da linguagem é necessário, mas não é de todo suficiente. Programar não é o simples acto de escrever ideias de outrem: é ter essas ideias, é ser criativo e engenhoso. Resolver problemas exige conhecimentos da linguagem, conhecimento das técnicas conhecidas de ataque aos problemas, inteligência para fazer analogias com outros

[1] Pode-se pensar num computador equipado com um compilador de uma dada linguagem de programação como um novo computador, mais poderoso. É de toda a conveniência usar este tipo de abstracção, que de resto será muito útil para perceber sistemas operativos, onde se vão acrescentando camada após camada de *software* para acrescentar "inteligência" aos computadores.

problemas, mesmo em áreas totalmente desconexas, criatividade, intuição, engenho, persistência, etc. Programar não é um acto mecânico. Assim, aprender a programar consegue-se através do estudo e, fundamentalmente, do treino.

1.3 Algoritmos: resolvendo problemas

Dado um problema que é necessário resolver, como desenvolver uma solução? Como expressá-la? À última pergunta responde-se muito facilmente: usando uma linguagem. A primeira é mais difícil. Se a solução desenvolvida corresponder a um conjunto de instruções bem definidas e sem qualquer ambiguidade, podemos dizer que temos um *algoritmo* que resolve um problema, i.e., que a partir de um conjunto de entradas produz determinadas saídas.

A noção de algoritmo não é simples de perceber, pois é uma abstracção. Algoritmos são métodos de resolver problemas. Mas à concretização de um algoritmo numa dada linguagem já não se chama algoritmo: chama-se *programa*. Sob esse ponto de vista, todas as versões escritas de um algoritmo são programas, mesmo que expressos numa linguagem natural (desde que não façam uso da sua característica ambiguidade). Abusando um pouco das definições, no entanto, chamaremos algoritmo a um método de resolução de um dado problema expresso em linguagem natural, e programa à concretização de um algoritmo numa dada linguagem de programação.

A definição de algoritmo é um pouco mais completa do que a apresentada. De acordo com Knuth [10] os algoritmos têm cinco características importantes:

Finitude Um algoritmo tem de terminar sempre ao fim de um número finito de passos. De nada nos serve um algoritmo se existirem casos em que não termina.

Definitude[2] Cada passo do algoritmo tem de ser definido com precisão; as acções a executar têm de ser especificadas rigorosamente e sem ambiguidade. No fundo isto significa que um algoritmo tem de ser tão bem especificado que até um computador possa seguir as suas instruções.

Entrada Um algoritmo pode ter zero ou mais entradas, i.e., entidades que lhe são dadas inicialmente, antes do algoritmo começar. Essas entidades pertencem a um conjunto bem definido (e.g., o conjunto dos números inteiros). Existem algoritmos interessantes que não têm qualquer entrada, embora sejam raros.

Saída Um algoritmo tem uma ou mais saídas, i.e., entidades que têm uma relação bem definida com as entradas (o problema resolvido pelo algoritmo é o de calcular as saídas correspondentes às entradas).

Eficácia Todas as operações executadas no algoritmo têm de ser suficientemente básicas para, em princípio, poderem ser feitas com exactidão e em tempo finito por uma pessoa usando um papel e um lápis.

Para além destas características dos algoritmos, pode-se também falar da sua eficiência, isto é, do tempo que demoram a ser executados para dadas entradas. Em geral, portanto, pretende-se que os algoritmos sejam não só finitos mas também suficientemente rápidos [10]. Ou seja, devem resolver o problema em tempo útil (e.g., enquanto ainda somos vivos para estarmos interessados na sua resolução) mesmo para a mais desfavorável combinação possível das entradas.

O estudo dos algoritmos, a algoritmia (*algorithmics*), é um campo muito importante da ciência da computação, que envolve por exemplo o estudo da sua correcção (verificação se de facto resolvem o problema), da sua finitude (será de facto um algoritmo, ou não passa de um método computacional [10] inútil na prática?) e da sua eficiência (análise de algoritmos). Estes temas serão inevitavelmente abordados nesta disciplina, embora informalmente, e serão fundamentados teoricamente na disciplina de Computação e Algoritmia. A disciplina de Introdução à Programação serve portanto de ponte entre as disciplinas de Arquitectura de Computadores e Computação e Algoritmia, fornecendo os conhecimentos necessários para o trabalho subsequente em outras disciplinas da área da informática.

1.3.1 Regras do jogo

No jogo de resolução de problemas que é o desenvolvimento de algoritmos, há duas regras simples:

1. as variáveis são os únicos objectos manipulados pelos algoritmos e
2. os algoritmos só podem memorizar valores em variáveis.

As *variáveis* são os objectos sobre os quais as instruções dos algoritmos actuam. Uma variável corresponde a um local onde se podem guardar valores. Uma variável tem três características:

1. Um nome, que é fixo durante a vida da variável, e pelo qual a variável é conhecida. É importante distinguir as variáveis sobre as quais os algoritmos actuam das variáveis matemáticas usuais. Os nomes das variáveis de algoritmo ou programa serão grafados com um tipo de largura fixa, enquanto as variáveis matemáticas serão grafadas em itálico. Por exemplo, `k` e `número_de_alunos` são variáveis de algoritmo ou programa enquanto k é uma variável matemática.
2. Um tipo, que também é fixo durante a vida da variável, e que determina o conjunto de valores que nela podem ser guardados e, sobretudo, as operações que se podem realizar com os valores guardados nas variáveis desse tipo. Para já, considerar-se-á que todas as variáveis têm tipo inteiro, i.e., que guardam valores inteiros suportando as operações usuais com números inteiros. Nesse caso dir-se-á que o tipo é das variáveis é inteiro ou $\mathbb{Z}$.
3. Um e um só valor em cada instante de tempo. No entanto, o valor pode variar no tempo, à medida que o algoritmo decorre.

Costuma-se fazer uma analogia entre algoritmo e receita de cozinha. Esta analogia é atractiva, mas falha em alguns pontos. Tanto uma receita como um algoritmo consistem num conjunto de instruções. Porém, no caso das receitas, as instruções podem ser muito vagas. Por exemplo, "ponha ao lume e vá mexendo até alourar". Num algoritmo as instruções tem de ser precisas, sem margem para interpretações diversas. O pior ponto da analogia diz respeito às variáveis. Pode-se dizer que as variáveis de um algoritmo correspondem aos recipientes usados para realizar uma receita. O problema é que um recipiente pode (a) estar vazio e (b) conter qualquer tipo de ingrediente, enquanto *uma variável contém sempre valores do mesmo tipo* e, além disso, *uma variável contém sempre um qualquer valor*: as variáveis não podem estar vazias! É absolutamente fundamental entranhar esta característica pouco intuitiva das variáveis.

É muito conveniente ter uma notação para representar graficamente uma variável. A Figura 1.1 mostra uma variável inteira de nome `idade` com valor 24. É comum, na linguagem corrente,

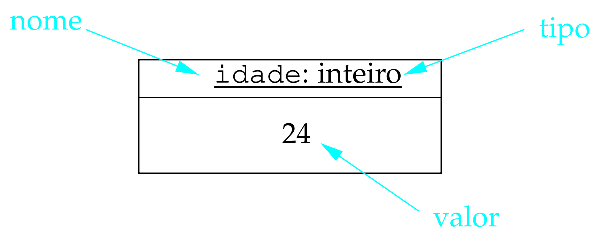


Figura 1.1: Notação para uma variável. A azul explicações sobre a notação gráfica usada.

não distinguir entre a variável, o seu nome e o valor que guarda. Esta prática é abusiva, mas simplifica a linguagem. Por exemplo, acerca da variável na Figura 1.1, é costume dizer-se que "a `idade` é 24". Em rigor dever-se-ia dizer que "a variável de nome `idade` guarda actualmente o valor 24".

Suponha-se um problema simples. São dadas duas variáveis `n` e `soma`, ambas de tipo inteiro. Pretende-se que a execução do algoritmo coloque na variável `soma` a soma dos primeiros `n` inteiros não-negativos. Admite-se portanto que a variável `soma` tem inicialmente um valor arbitrário enquanto a variável `n` contém inicialmente o número de termos da soma a realizar.

Durante a execução de um algoritmo, as variáveis existentes vão mudando de valor. Aos valores das variáveis num determinado instante da execução do algoritmo chama-se *estado*. Há dois estados particularmente importantes durante a execução de um algoritmo: o estado inicial e o estado final.

O estado inicial é importante porque os algoritmos só estão preparados para resolver os problemas se determinadas condições mínimas se verificarem no início da sua execução. Para o problema dado não faz qualquer sentido que a variável `n` possua inicialmente o valor -3, por exemplo. Que poderia significar a "soma dos primeiros -3 inteiros não-negativos"? Um algoritmo é uma receita para resolver um problema, mas apenas se as variáveis verificaram inicialmente determinadas condições mínimas, expressas na chamada *pré-condição* ou PC. Neste caso a pré-condição diz simplesmente que a variável `n` não pode ser negativa, i.e., $PC \equiv 0 \leq \mathtt{n}$.

O estado final é ainda mais importante que o estado inicial. O estado das variáveis no final de um algoritmo deve ser o necessário para que o problema que o algoritmo é suposto resolver esteja de facto resolvido. Para especificar quais os estados aceitáveis para as variáveis no final do algoritmo usa-se a chamada *condição-objectivo* ou CO. Neste caso a condição objectivo é $CO \equiv \mathtt{soma} = \sum_{j=0}^{\mathtt{n}-1} j$, ou seja, a variável `soma` deve conter a soma dos inteiros entre 0 e $\mathtt{n}-1$

inclusive[3].

O par de condições pré-condição e condição objectivo representa de forma compacta o problema que um algoritmo é suposto resolver. Neste caso, porém, este par de condições está incompleto: uma vez que o algoritmo pode alterar o valor das variáveis, pode perfeitamente alterar o valor da variável `n`, pelo que um algoritmo perverso poderia simplesmente colocar o valor zero quer em `n` quer em `soma` e declarar resolver o problema! Em rigor, portanto, dever-se-ia indicar claramente que `n` não pode mudar de valor ao longo do algoritmo, i.e., que `n` é uma *constante*, ou, alternativamente, indicar claramente na condição objectivo que o valor de `n` usado é o valor de `n` no *início* do algoritmo. Em vez disso admitir-se-á simplesmente que o valor de `n` não é alterado pelo algoritmo.

O problema proposto pode ser resolvido de uma forma simples. Considere-se uma variável adicional `i` que conterá os inteiros a somar (um de cada vez...). Comece-se por colocar o valor zero quer na variável `soma` quer na variável `i`. Enquanto o valor da variável `i` não atingir o valor da variável `n`, deve-se somar o valor da variável `i` ao valor da variável `soma` e guardar o resultado dessa soma na própria variável `soma` (que vai servindo para acumular o resultado), e em seguida deve-se aumentar o valor de `i` de uma unidade. Quando o valor de `i` atingir `n`, o algoritmo termina. Um pouco mais formalmente[4]:

$\{PC \equiv 0 \leq \mathtt{n}.\}$
$\mathtt{i} \leftarrow 0$
$\mathtt{soma} \leftarrow 0$
enquanto $\mathtt{i} \neq \mathtt{n}$ faça-se:
    $\mathtt{soma} \leftarrow \mathtt{soma} + \mathtt{i}$
    $\mathtt{i} \leftarrow \mathtt{i} + 1$
$\{CO \equiv \mathtt{soma} = \sum_{j=0}^{\mathtt{n}-1} j.\}$

O símbolo $\leftarrow$ deve ser lido "fica com o valor de" e chama-se a *atribuição*. Tudo o que se coloca entre chavetas são comentários, não fazendo parte do algoritmo propriamente dito. Neste caso usaram-se comentários para indicar as condições que se devem verificar no início e no final do algoritmo.

Um algoritmo é lido e executado pela ordem normal de leitura em português: de cima para baixo e da esquerda para a direita, excepto quando surgem construções como um **enquanto**, que implicam voltar atrás para repetir um conjunto de instruções.

É muito importante perceber a evolução dos valores das variáveis ao longo da execução do algoritmo. Para isso é necessário arbitrar os valores iniciais da variáveis. Suponha-se que `n` tem inicialmente o valor 4. O estado inicial, i.e., imediatamente antes de começar a executar o algoritmo, é o indicado na Figura 1.2. Dois aspectos são de notar. Primeiro que este estado verifica a pré-condição indicada. Segundo que os valores iniciais das variáveis `i` e `soma` são irrelevantes, o que é indicado através do símbolo '?'.

Neste caso é evidente que o estado final, i.e., imediatamente após a execução do algoritmo, é o indicado na Figura 1.3. Mas em geral pode não ser tão evidente, pelo que é necessário

[3] Mais tarde usar-se-á uma notação diferente para o somatório: $CO \equiv \mathtt{soma} = (\mathbf{S}\, j : 0 \leq j < \mathtt{n} : j)$.

[4] Para o leitor mais atento deverá ser claro que uma forma mais simples de resolver o problema é simplesmente colocar na variável `soma` o resultado de $\frac{\mathtt{n}(\mathtt{n}-1)}{2}$...

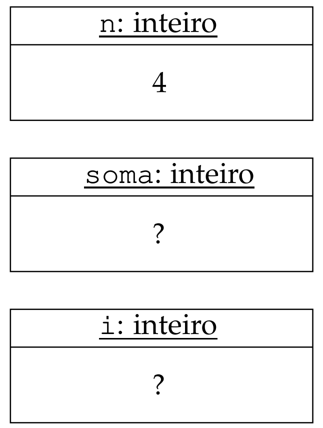


Figura 1.2: Estado inicial do algoritmo.

Figura 1.3: Estado final do algoritmo.

fazer o traçado da execução do algoritmo, i.e., a verificação do estado ao longo da execução do algoritmo.

O traçado da execução de um algoritmo implica, pois, registar o valor de todas as variáveis antes e depois de todas as instruções executadas. Para que isso se torne claro, é conveniente numerar as transições entre as instruções:

$\{PC \equiv 0 \leq \mathtt{n}.\}$

1

$\mathtt{i} \leftarrow 0$

2

$\mathtt{soma} \leftarrow 0$

3

enquanto $\mathtt{i} \neq \mathtt{n}$ **faça-se:**

4

    $\mathtt{soma} \leftarrow \mathtt{soma} + \mathtt{i}$

5

    $\mathtt{i} \leftarrow \mathtt{i} + 1$

6

fim do enquanto.

7

$\{CO \equiv \mathtt{soma} = \sum_{j=0}^{\mathtt{n}-1} j.\}$

Introduziu-se explicitamente o final do enquanto de modo a separar claramente os intervalos 6 e 7. O intervalo 6 está após o aumento de um da variável `i` e antes de se verificar se o seu valor atingiu já o valor de `n`. O intervalo 7, pelo contrário, está depois do enquanto, quando `i` atingiu já o valor de `n`, no final do algoritmo.

Estes intervalos ficam mais claros se se recorrer a um diagrama de actividade[5] para representar o algoritmo, como se pode ver na Figura 1.4. É fácil agora seguir o fluxo de execução e analisar os valores das variáveis em cada transição entre instruções. O resultado dessa análise pode ser visto na Tabela 1.1.

A realização do traçado de um algoritmo, como se verá, não demonstra a sua correcção. Para o fazer há que usar técnicas um pouco mais formais, abordadas brevemente na próxima secção.

1.3.2 Desenvolvimento e demonstração de correcção

Seja o problema de, dados quaisquer dois inteiros positivos, calcular os seu máximo divisor comum[6].

O algoritmo que resolve este problema tem de começar por instruções dizendo para os dois inteiros serem pedidos a alguém (i.e., lidos) e guardados em duas variáveis, a que se darão

[5]Como estes diagramas mostram claramente o fluxo de execução do algoritmo, também são conhecidos por diagramas de fluxo ou fluxogramas.

[6]Este exemplo faz uso de alguma simbologia a que pode não estar habituado. Recomenda-se a leitura do Apêndice A.

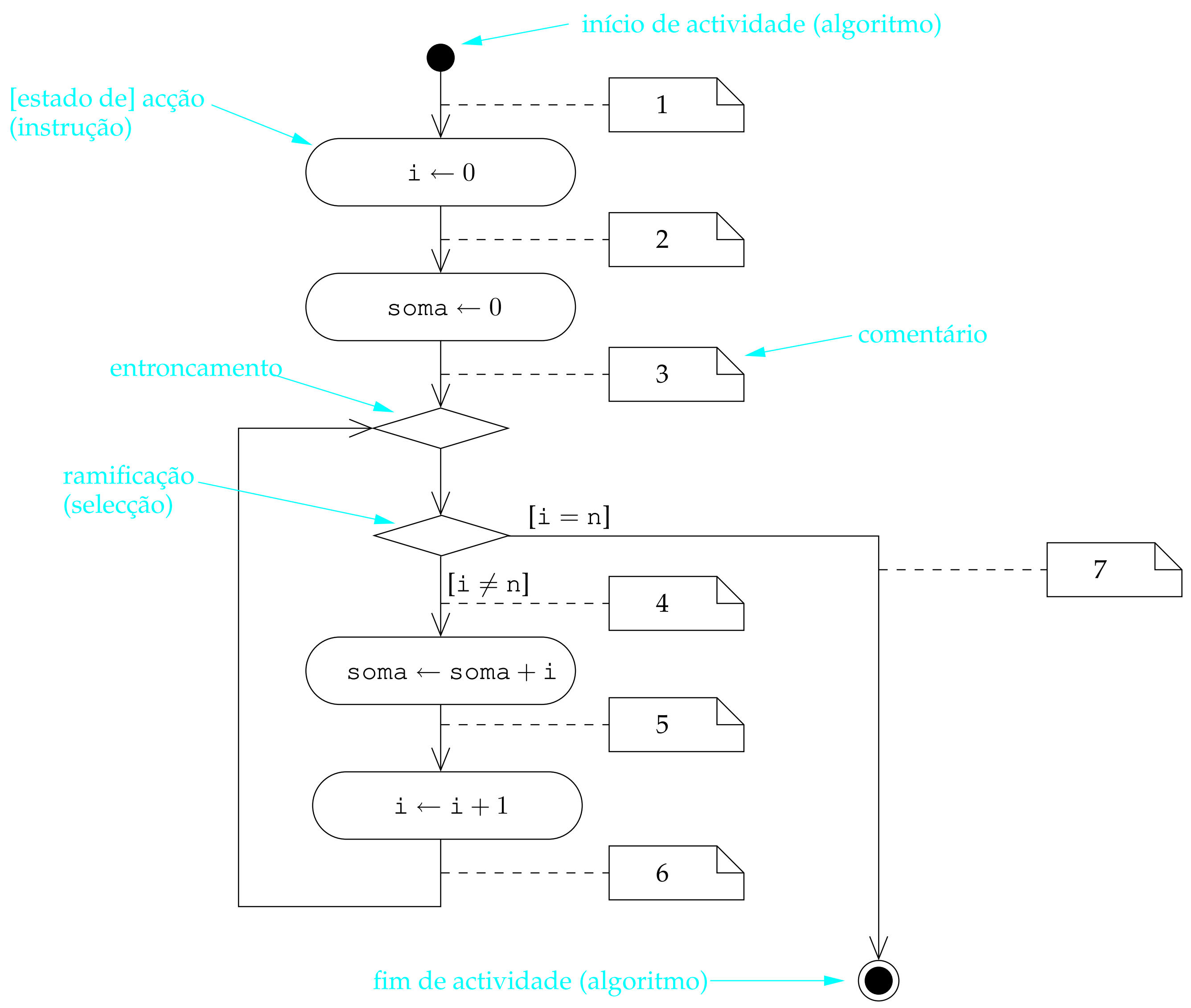


Figura 1.4: Diagrama de actividade do algoritmo. As setas indicam a sequência de execução das acções (instruções). Os losangos têm significados especiais: servem para unir fluxos de execução alternativos (entroncamentos) ou para os começar (ramificações). Em cada um dos ramos saídos de uma ramificação indicam-se as condições que têm de se verificar para que seja escolhido esse ramo. A azul encontram-se explicações sobre a notação gráfica usada. Os comentários contêm a numeração das transições usadas no algoritmo, para que se possa fazer uma mais fácil correspondência entre este e o diagrama.

Tabela 1.1: Traçado do algoritmo.

Transição	`n`	`i`	`soma`	Comentários
1	4	?	?	Verifica-se $PC \equiv 0 \leq \mathtt{n}$.
2	4	0	?	
3	4	0	0	
4	4	0	0	
5	4	0	0	
6	4	1	0	
4	4	1	0	
5	4	1	1	
6	4	2	1	
4	4	2	1	
5	4	2	3	
6	4	3	3	
4	4	3	3	
5	4	3	6	
6	4	4	6	
7	4	4	6	Verifica-se $CO \equiv \mathtt{soma} = \sum_{j=0}^{\mathtt{n}-1} j = \sum_{j=0}^{3} j = 0+1+2+3 = 6$.

os nomes `m` e `n`, e terminar por uma instrução dizendo para o máximo divisor comum ser comunicado a alguém (i.e., escrito). Para que o resultado possa ser comunicado a alguém, é necessário que esteja guardado em algum lado. Para isso é necessária uma variável adicional, a que se dará o nome de `k`. Os passos intermédios do algoritmo, entre a leitura dos dois inteiros e a escrita do resultado, indicam a forma de cálculo do máximo divisor comum, e são os passos que nos interessam neste momento. Assim, considerar-se-á inicialmente o problema mais simples de, dados dois valores inteiros positivos arbitrários colocados nas duas variáveis `m` e `n`, colocar na variável `k` o seu máximo divisor comum (mdc). Um exemplo é representado na Figura 1.5.

A abordagem deste problema passa em primeiro lugar pela identificação da chamada pré-condição (PC), i.e., pelas condições que se sabe que se verificam (ou que pelo menos se admite que se verificam) *a priori*. Neste caso a pré-condição é

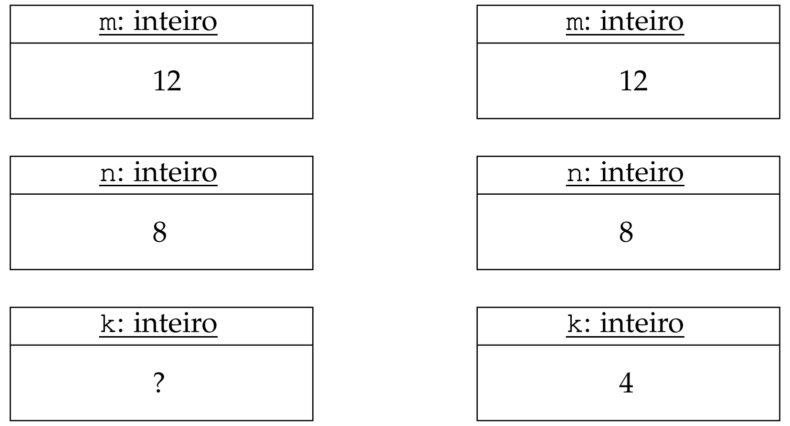
$$PC \equiv \mathtt{m} \text{ e } \mathtt{n} \text{ são inteiros positivos.}$$

Depois, deve-se formalizar a condição objectivo (CO). Neste caso a condição objectivo é

$$CO \equiv \mathtt{k} = \mathrm{mdc}(\mathtt{m}, \mathtt{n}),$$

onde se assume que `m` e `n` não mudam de valor ao longo do algoritmo. O objectivo do problema é, pois, encontrar um valor para `k` que verifique a condição objectivo CO.

Para que o problema faça sentido, é necessário que, quaisquer que sejam `m` e `n` tais que a PC é verdadeira, exista pelo menos um inteiro positivo `k` tal que a CO é verdadeira. No



(a) Antes de executar o algoritmo (b) Depois de executar o algoritmo

Figura 1.5: Exemplo de estado das variáveis antes 1.5(a) e depois 1.5(b) de executado o algoritmo.

problema em causa não há quaisquer dúvidas: todos os pares de inteiros positivos têm um correspondente máximo divisor comum.

Para que a solução do problema não seja ambígua, é necessário garantir mais. Não basta que exista solução: é necessário que seja única, ou seja, que quaisquer que sejam `m` e `n` tais que a PC é verdadeira, existe um e um só inteiro positivo `k` tal que a CO é verdadeira. Quando isto se verifica, pode-se dizer que o problema está bem colocado (a demonstração de que um problema está bem colocado muitas vezes se faz desenvolvendo um algoritmo: são as chamadas demonstrações construtivas). É fácil verificar que, neste caso, o problema está bem colocado: existe apenas um mdc de cada par de inteiros positivos.

Depois de se verificar que de facto o problema está bem colocado dadas a pré-condição e a condição objectivo, é de todo o interesse identificar propriedades interessantes do mdc. Duas propriedades simples são:

a) O mdc é sempre superior ou igual a 1, i.e., quaisquer que sejam m e n inteiros positivos, $1 \leq \mathrm{mdc}(m, n)$.

b) O mdc não excede nunca o menor dos dois valores, i.e., quaisquer que sejam m e n inteiros positivos, $\mathrm{mdc}(m, n) \leq \min(m, n)$, em que $\min(m, n)$ é o menor dos dois valores m e n.

A veracidade destas duas propriedades é evidente, pelo que não se demonstra aqui.

Seja $m \div k$ a notação para o resto da divisão inteira de m por k. Ou seja, dados dois inteiros $0 \leq m$ e $0 < k$, e sendo $m = qk + r$ com $0 \leq r < k$ (onde q e r são respectivamente o quociente e o resto da divisão de m por k), então $r = m \div k$. Nesse caso, dizer que $0 < k$ é divisor de $0 \leq m$ é o mesmo que dizer que $m \div k = 0$, i.e., que a divisão inteira de m por k tem resto zero.

Duas outras propriedades que se verificará serem importantes são:

c) Quaisquer que sejam k, m e n inteiros positivos,

$$\mathrm{mdc}(m,n) \leq k \wedge (m \div k = 0 \wedge n \div k = 0) \Rightarrow k = \mathrm{mdc}(m,n).$$

Ou seja, se k é superior ou igual ao máximo divisor comum de m e n e se k é divisor comum de m e n, então k é o máximo divisor comum de m e n.

A demonstração pode ser feita por absurdo. Suponha-se que k não é o máximo divisor comum de m e n. Como k é divisor comum, então conclui-se que $k < \mathrm{mdc}(m,n)$, isto é, há divisores comuns maiores que k. Mas então $\mathrm{mdc}(m,n) \leq k < \mathrm{mdc}(m,n)$, que é uma impossibilidade. Logo, $k = \mathrm{mdc}(m,n)$.

d) Quaisquer que sejam k, m e n inteiros positivos,

$$\mathrm{mdc}(m,n) \leq k \wedge (m \div k \neq 0 \vee n \div k \neq 0) \Rightarrow \mathrm{mdc}(m,n) < k.$$

Ou seja, se k é superior ou igual ao máximo divisor comum de m e n e se k não é divisor comum de m e n, então k é maior que o máximo divisor comum de m e n. Neste caso a demonstração é trivial.

Em que é que estas propriedades nos ajudam? As duas primeiras propriedades restringem a gama de possíveis divisores, i.e., o intervalo dos inteiros onde o mdc deve ser procurado, o que é obviamente útil. As duas últimas propriedades serão úteis mais tarde.

Não há nenhum método mágico de resolver problemas. Mais tarde ver-se-á que existem metodologias que simplificam a abordagem ao desenvolvimento de algoritmos, nomeadamente aqueles que envolvem iterações (repetições). Mesmo essas metodologias não são substituto para o engenho e a inteligência. Para já, usar-se-á apenas a intuição.

Onde se deve procurar o mdc? A intuição diz que a procura deve ser feita a partir de $\min(\mathtt{m},\mathtt{n})$, pois de outra forma, começando por 1, é fácil descobrir divisores comuns, mas não é imediato se eles são ou não o máximo divisor comum: é necessário testar todos os outros potenciais divisores até $\min(\mathtt{m},\mathtt{n})$. Deve-se portanto ir procurando divisores comuns partindo de $\min(\mathtt{m},\mathtt{n})$ para baixo até se encontrar algum, que será forçosamente o mdc.

Como transformar estas ideias num algoritmo e, simultaneamente, demonstrar a sua correcção? A variável `k`, de acordo com a CO, tem de terminar o algoritmo com o valor do máximo divisor comum. Mas pode ser usada entretanto, durante o algoritmo, para conter inteiros que são candidatos a máximo divisor comum. Assim, se se quiser começar pelo maior valor possível, deve-se atribuir a `k` (ou seja, registar na "caixa" `k`) o menor dos valores `m` e `n`:

```
se m < n então:
    k ← m
senão:
    k ← n
```

Depois destas instruções, é evidente que se pode afirmar que $\mathrm{mdc}(\mathtt{m},\mathtt{n}) \leq \mathtt{k}$, dada a propriedade b).

Em seguida, deve-se verificar se `k` divide `m` e `n` e, no caso contrário, passar ao próximo candidato, que é o valor inteiro imediatamente abaixo:

enquanto $\mathtt{m} \div \mathtt{k} \neq 0 \vee \mathtt{n} \div \mathtt{k} \neq 0$ **faça-se:**
  $\mathtt{k} \leftarrow \mathtt{k} - 1$

A condição que controla o ciclo (chama-se ciclo porque é uma instrução que implica a repetição cíclica de outra, neste caso $\mathtt{k} \leftarrow \mathtt{k} - 1$) chama-se guarda ($G$) do ciclo. Neste caso a guarda é

$$G \equiv \mathtt{m} \div \mathtt{k} \neq 0 \vee \mathtt{n} \div \mathtt{k} \neq 0,$$

que é verdadeira se k não for divisor comum a m e n.

A instrução $\mathtt{k} \leftarrow \mathtt{k} - 1$ chama-se o progresso (*prog*) do ciclo pois, como se verá, é ela que garante que o ciclo progride em direcção ao seu fim.

Quando este ciclo terminar, $\mathtt{k}$ é o mdc. Como demonstrá-lo? Repare-se que, dadas as propriedades c) e d), quer a inicialização de $\mathtt{k}$, quer a guarda e o progresso, garantem que há uma condição que se mantém sempre verdadeira, desde o início ao fim do ciclo, e que por isso recebe o nome de condição invariante (CI) do ciclo:

$$CI \equiv \mathrm{mdc}(\mathtt{m}, \mathtt{n}) \leq \mathtt{k}.$$

Esta condição diz simplesmente que k é sempre superior ou igual ao máximo divisor comum que se procura.

Dada a inicialização do ciclo

se $\mathtt{m} < \mathtt{n}$ **então:**
  $\mathtt{k} \leftarrow \mathtt{m}$
senão:
  $\mathtt{k} \leftarrow \mathtt{n}$

e a pré-condição, tem-se que $\mathtt{k} = \min(\mathtt{m}, \mathtt{n})$, o que, conjugado com a propriedade b), garante que $\mathrm{mdc}(\mathtt{m}, \mathtt{n}) \leq \mathtt{k}$. Ou seja, a inicialização leva à veracidade da condição invariante.

Durante o ciclo, que acontece se a G for verdadeira? Diminui-se $\mathtt{k}$ de uma unidade, isto é progride-se[7]. Mas, admitindo que a condição invariante é verdadeira e sendo a guarda também verdadeira, conclui-se pela propriedade d) que

$$\mathrm{mdc}(\mathtt{m}, \mathtt{n}) < \mathtt{k}$$

logo após a verificação da guarda e imediatamente antes do progresso.

Como o valor guardado em $\mathtt{k}$ é inteiro, isso é o mesmo que dizer que $\mathrm{mdc}(\mathtt{m}, \mathtt{n}) \leq \mathtt{k} - 1$. Donde, depois do progresso $\mathtt{k} \leftarrow \mathtt{k} - 1$, a veracidade da condição invariante é recuperada, i.e., $\mathrm{mdc}(\mathtt{m}, \mathtt{n}) \leq \mathtt{k}$. Ou seja, demonstrou-se que se a condição invariante for verdadeira antes do progresso, também o será depois.

A demonstração da veracidade da condição invariante do ciclo após a inicialização do ciclo e da preservação da sua veracidade durante a execução do ciclo correspondem, no fundo, à demonstração por indução de que a condição invariante se verifica durante todo o ciclo, *inclusive quando este termina*. I.e., demonstrou-se que a condição invariante é, de facto, invariante.

[7] É uma progressão porque se fica mais próximo do final do ciclo, ou seja, fica-se mais próximo do real valor do $\mathrm{mdc}(\mathtt{m}, \mathtt{n})$.

Que acontece se, durante o ciclo, a guarda for falsa? Nesse caso o ciclo termina, concluindo-se que há duas condições verdadeiras: a condição invariante CI e também a negação da guarda, i.e., $\neg G$. A conjunção destas condições, pela propriedade c), implica que a condição objectivo é verdadeira, i.e., que `k` termina de facto com o máximo divisor comum de `m` e `n`.

A demonstração da correcção (parcial) do algoritmo passa, pois, por demonstrar que a condição invariante é verdadeira do início ao fim do ciclo (i.e., que é de facto invariante) e que, quando o ciclo termina (i.e., quando a guarda é falsa), se tem forçosamente que a condição objectivo é verdadeira, i.e.,

$$CI \wedge \neg G \Rightarrow CO.$$

Assim, o algoritmo completo, "decorado" com comentários onde se indica claramente quais as condições verdadeiras em cada transição entre intruções, é:

$\{PC \equiv\ 0 < \mathtt{m} \wedge 0 < \mathtt{n}.\}$
se $\mathtt{m} < \mathtt{n}$ **então:**
    $\mathtt{k} \leftarrow \mathtt{m}$
senão:
    $\mathtt{k} \leftarrow \mathtt{n}$
$\{$*Aqui sabe-se que* $\mathrm{mdc}(m, n) \leq \min(m, n) = k$*, ou seja, que a* CI *é verdadeira..*$\}$
enquanto $\mathtt{m} \div \mathtt{k} \neq 0 \vee \mathtt{n} \div \mathtt{k} \neq 0$ **faça-se:**
    $\{$*Aqui a* G *é verdadeira. Logo, pela propriedade d),*
    a CI *mantém-se verdadeira depois do seguinte progresso:*$\}$
    $\mathtt{k} \leftarrow \mathtt{k} - 1$
$\{$*Aqui a* G *é falsa. Logo, pela propriedade c),*
$k = \mathrm{mdc}(m, n)$*, ou seja a* CO *é verdadeira..*$\}$

Uma questão importante, que ficou por demonstrar, é se é garantido que o ciclo termina sempre. Se não se garantir a terminação, não se pode chamar a esta sequência de instruções um algoritmo, como se viu. Esta demonstração é essencial para garantir a correcção *total* do algoritmo.

A demonstração é simples. Em primeiro lugar, a inicialização garante que o valor inicial de `k` é superior ou igual a 1 (pois `m` e `n` são inteiros positivos). O progresso, por outro lado, faz o valor de `k` diminuir a cada iteração do ciclo. Como 1 é divisor comum de qualquer par de inteiros positivos, o ciclo, na pior das hipóteses, termina com $\mathtt{k} = 1$, i.e., na pior das hipóteses o ciclo termina ao fim de $\min(\mathtt{m}, \mathtt{n}) - 1$ iterações. Assim, o algoritmo é de facto um algoritmo, pois verifica a condição de finitude: está totalmente correcto.

Resumindo, o conjunto de instruções que se apresentou é um algoritmo porque:

1. é finito, terminando sempre ao fim de no máximo $\min(\mathtt{m}, \mathtt{n}) - 1$ iterações do ciclo;

2. está bem definido, pois cada passo do algoritmo está definido com precisão e sem ambiguidades;

3. tem duas entradas, que são os valores colocados inicialmente em `m` e `n`, pertencentes ao conjunto dos inteiros positivos;

4. tem uma saída que é o valor que se encontra em k no final do algoritmo e que verifica $k = \mathrm{mdc}(m, n)$; e

5. é eficaz, pois todas as operações do algoritmo podem ser feitas com exactidão e em tempo finito por uma pessoa usando um papel e um lápis (e alguma paciência).

Quanto à sua eficiência, pode-se desde já afirmar que é suficientemente rápido para se poder usar em muitos casos, embora existam algoritmos consideravelmente mais eficientes, que serão abordados posteriormente.

Mais uma vez se poderia fazer o traçado deste algoritmo admitindo, por exemplo, o estado inicial indicado na Figura 1.5(a). Esse traçado fica como exercício para o leitor, que pode recorrer ao diagrama de actividade correspondente ao algoritmo mostrado na Figura 1.6.

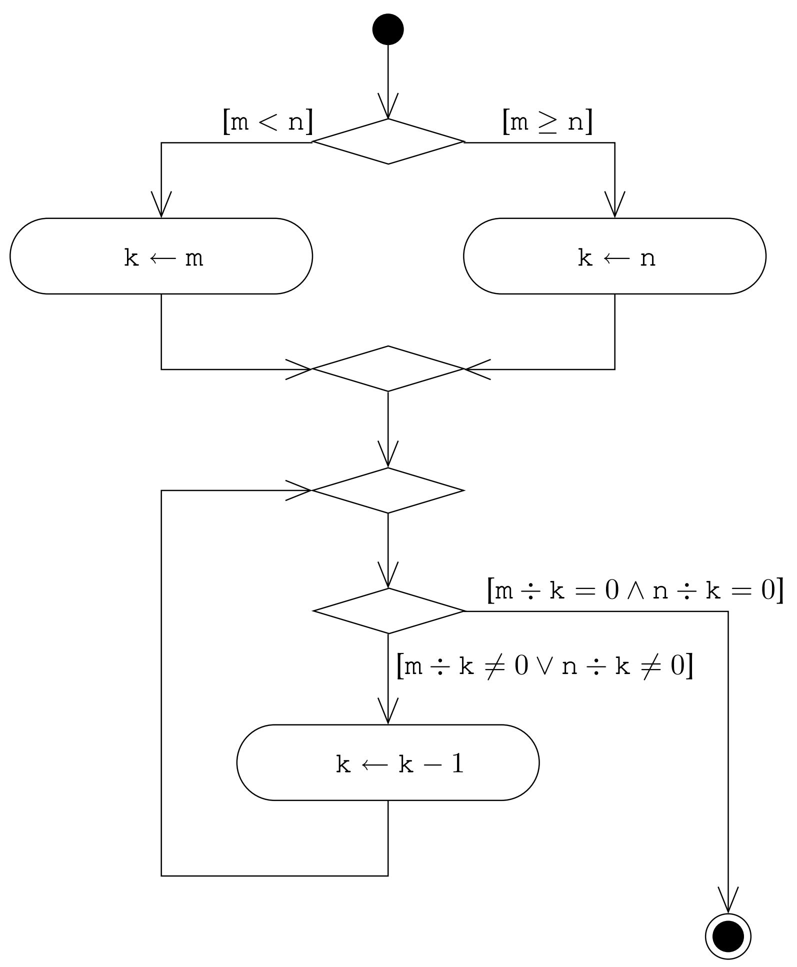


Figura 1.6: Diagrama de actividade correspondente ao algoritmo para cálculo do mdc.

Observações

O desenvolvimento de algoritmos em simultâneo com a demonstração da sua correcção é um exercício extremamente útil, como se verá no Capítulo 4. Mas pode-se desde já adiantar a uma das razões: não é possível demonstrar que um algoritmo funciona através de testes, excepto se se testarem rigorosamente todas as possíveis combinações de entradas válidas (i.e., entradas que verificam pré-condição do algoritmo). A razão é simples: o facto de um algoritmo funcionar correctamente para n diferentes combinações da entrada, por maior que n seja, não garante que não dê um resultado errado para uma outra combinação ainda não testada. Claro está que o oposto é verdadeiro: basta que haja uma combinação de entradas válidas para as quais o algoritmo produz resultados errados para se poder afirmar que ele está errado!

No caso do algoritmo do mdc é evidente que jamais se poderá mostrar a sua correcção através de testes: há infinitas possíveis combinações das entradas. Ainda que se limitasse a demonstração a entradas inferiores ou iguais a $1\,000\,000$, ter-se-iam $1\,000\,000\,000\,000$ testes para realizar. Admitindo que se conseguiam realizar $1\,000\,000$ de testes por segundo, a demonstração demoraria $1\,000\,000$ de segundos, cerca de 12 dias. Bastaria aumentar os valores das entradas até $10\,000\,000$ para se passar para um total de 1157 dias, cerca de três anos. Pouco prático, portanto...

O ciclo usado no algoritmo não foi obtido pela aplicação directa da metodologia de Dijkstra que será ensinada na Secção 4.7, mas poderia ter sido. Pede-se ao leitor que regresse mais tarde a esta secção para verificar que o ciclo pode ser obtido usando a factorização da condição objectivo

$$CO \equiv \overbrace{\mathtt{m} \div \mathtt{k} = 0 \wedge \mathtt{n} \div \mathtt{k} = 0}^{\neg G} \wedge \overbrace{0 < \mathtt{k} \wedge (\mathbf{Q}\, i \,:\, 0 \leq i < \mathtt{k} \,:\, \mathtt{m} \div i \neq 0 \vee \mathtt{n} \div i \neq 0)}^{CI},$$

onde **Q** significa “qualquer que seja”.

Finalmente, para o leitor mais interessado, fica a informação de que algoritmos mais eficientes para o cálculo do mdc podem ser obtidos pela aplicação das seguintes propriedades do mdc:

- Quaisquer que sejam a e b inteiros positivos $\mathrm{mdc}(a, b) = \mathrm{mdc}(b \div a, a)$.
- Se $a = 0 \wedge 0 < b$, então $\mathrm{mdc}(a, b) = b$.

Estas propriedades permitem, em particular, desenvolver o chamado algoritmo de Euclides para obtenção do máximo divisor comum.

1.4 Programas

Como se viu, um programa é a concretização, numa dada linguagem de programação, de um determinado algoritmo (que resolve um problema concreto). Nesta disciplina ir-se-á utilizar uma linguagem de alto nível chamada C++. Esta linguagem tem o seu próprio léxico, a sua sintaxe, a sua gramática e a sua semântica (embora simples). Todos estes aspectos da linguagem serão tratados ao longo deste texto. Para já, no entanto, apresenta-se sem mais explicações a tradução do algoritmo do mdc para C++, embora agora já com instruções explícitas para leitura das entradas a partir de um teclado e de escrita da saída no ecrã. O programa resultante é suficientemente simples para que possa ser entendido quase na sua totalidade. No entanto, é normal que o leitor fique com dúvidas: serão esclarecidas no próximo capítulo.

```
// Todo o texto após // são comentários, não fazendo parte do programa.

// Os programas começam tipicamente por um conjunto de inclusões.
// Neste caso incluem-se ferramentas de entradas e saídas (Input/Output) a partir
// de canais (stream):
#include <iostream>

// Para evitar ter de escrever std::cout, std::cin, etc.
using namespace std;

/// Este programa calcula o máximo divisor comum de dois números.
int main()
{
    // Instruções de inserção de informação no canal de saída cout, ligado ao ecrã.
    // Mostra ao utilizador informação sobre o programa e pede-lhe para inserir os
    // dois inteiros dos quais quer obter o máximo divisor comum:
    cout << "Máximo divisor comum de dois números."
         << endl;
    cout << "Introduza dois inteiros positivos: ";

    // Definição de duas variáveis inteiras:
    int m, n;

    // Instrução de extracção de informação do canal de entrada, ligado ao teclado.
    // Serve para obter do utilizador os valores dos quais ele pretende saber o mdc:
    cin >> m >> n; // Assume-se que m e n são positivos!

    // Como um divisor é sempre menor ou igual a um número, escolhe-se
    // o mínimo dos dois!
    int k;
    if(m < n)
        k = m;
    else
        k = n;
    // Neste momento sabe-se que mdc(m, n) ≤ k.

    while(m % k != 0 or n % k != 0) {
        // Se o ciclo não parou, então k não divide m e n, logo,
        // k ≠ mdc(m, n) ∧ mdc(m, n) ≤ k. Ou seja, mdc(m, n) < k.
        --k; // Decrementa k. É o progresso do ciclo.
        // Neste momento, mdc(m, n) ≤ k outra vez! É o invariante do ciclo!
```

```
    }

    // Como mdc(m, n) ≤ k (invariante do ciclo) e k divide m e n
    // (o ciclo terminou, não foi?), conclui-se que k = mdc(m, n)!

    // Insere no canal de saída uma mensagem para o utilizador dizendo-lhe qual o
    // resultado:
    cout << "O máximo divisor comum de " << m
         << " e " << n << " é " << k << '.' << endl;
}
```

1.5 Resumo: resolução de problemas

Resumindo, a resolução de problemas usando linguagens de programação de alto nível tem vários passos:

1. Especificação do problema, feita por humanos.
2. Desenvolvimento de um algoritmo que resolve o problema, feito por humanos. É neste passo que se faz mais uso da inteligência e criatividade.
3. Concretização do algoritmo na linguagem de programação: desenvolvimento do programa, feito por humanos. O resultado deste passo é um programa, consistindo numa sequência de instruções, à qual também se chama *código*. Este passo é mecânico e relativamente pouco interessante.
4. Tradução do programa para linguagem máquina, feita pelo computador, ou melhor, por uma programa chamado compilador.
5. Execução do programa para resolver um problema particular (e.g., cálculo de $\mathrm{mdc}(131, 47)$), feita pelo computador.

Podem ocorrer erros em todos estes passos. No primeiro, se o problema estiver mal especificado. No segundo ocorrem os chamados *erros lógicos*: o algoritmo desenvolvido na realidade não resolve o problema tal como especificado. É aqui que os erros são mais graves, pelo que é extremamente importante assegurar a correcção dos algoritmos desenvolvidos. No terceiro passo, os erros mais benévolos são os que conduzem a *erros sintácticos* ou gramaticais, pois o compilador, que é o programa que traduz o programa da linguagem de programação em que está escrito para a linguagem máquina, assinala-os. Neste passo os piores erros são *gralhas* que alteram a semântica do programa sem o invalidar sintacticamente. Por exemplo, escrever `l` (letra 'l') em vez de `1` pode ter consequências muito graves se `l` for o nome de uma variável. Estes erros são normalmente muito difíceis de detectar[8]. Finalmente, a ocorrência de erros nos dois últimos passos é tão improvável que mais vale assumir que não podem ocorrer de todo.

[8]Estes erros assemelham-se ao "não" introduzido (acidentalmente?) pelo revisor de provas em "História do cerco de Lisboa", de José Saramago.

Capítulo 2

Conceitos básicos de programação

¿Qué sería cada uno de nosotros sin su memória? Es una memória que en buena parte está hecha del ruido pero que es esencial. (...) Ése es el problema que nunca podremos resolver: el problema de la identidad cambiante. (...) Porque si hablamos de cambio de algo, no decimos que algo sea reemplazado por otra cosa.

Jorge Luis Borges, *Borges Oral*, 98-99 (1998)

Um programa, como se viu no capítulo anterior, consiste na concretização prática de um algoritmo numa dada linguagem de programação. Um programa numa linguagem de programação imperativa consiste numa sequência de instruções[1], sem qualquer tipo de ambiguidade, que são executadas ordenadamente. As instruções:

1. alteram o valor de variáveis, i.e., alteram o estado do programa e consequentemente da memória usada pelo programa;
2. modificam o fluxo de controlo, i.e., alteram a execução sequencial normal dos programas permitindo executar instruções repetida, condicional ou alternativamente; ou
3. definem (ou declaram) entidades (variáveis, constantes, funções, procedimentos, etc.).

Neste capítulo começa por se analisar informalmente o programa apresentado no capítulo anterior e depois discutem-se em maior pormenor os conceitos básicos de programação, nomeadamente variáveis, constantes, tipos, expressões e operadores.

2.1 Introdução

Um programa em C++ tem normalmente uma estrutura semelhante à seguinte (todas as linhas precedidas de `//` e todo o texto entre `/*` e `*/` são comentários, sendo portanto ignorados pelo compilador e servindo simplesmente para documentar os programas, i.e., explicar ou clarificar as intenções do programador):

[1] E não só, como se verá.

```
/* As duas linhas seguintes são necessárias para permitir a
   apresentação de variáveis no ecrã e a inserção de valores
   através do teclado (usando os canais [ou streams] cin e cout): */
#include <iostream>

using namespace std;

/* A linha seguinte indica o ponto onde o programa vai começar a ser executado.
   Indica também que o programa não usa argumentos e que o valor por ele
   devolvido ao sistema operativo é um inteiro (estes assuntos serão vistos em
   pormenor mais tarde): */
int main()
{ // esta chaveta assinala o início do programa.

    // Aqui aparecem as instruções que compõem o programa.

} // esta chaveta assinala o fim do programa.
```

A primeira linha,

```
#include <iostream>
```

serve para obter as declarações do canal de leitura de dados do teclado (`cin`) e do canal de escrita de dados no ecrã (`cout`).

A linha seguinte,

```
using namespace std;
```

é uma directiva de utilização do espaço nominativo `std`, e serve para se poder escrever simplesmente `cout` em vez de `std::cout`. Nem sempre o seu uso é recomendável [12, pág.171], sendo usado aqui apenas para simplificar a apresentação dos programas sem os tornar inválidos (por razões de espaço, neste texto estas duas linhas iniciais serão muitas vezes omitidas, devendo o leitor colocá-las se decidir experimentar os exemplos apresentados). Os espaços nominativos serão abordados no Capítulo 9.

Quanto a `main()`, é uma função que tem a particularidade de ser a primeira a ser invocada no programa. As funções serão vistas em pormenor no Capítulo 3.

Esta estrutura pode ser vista claramente no programa de cálculo do mdc introduzido no capítulo anterior:

```
// Todo o texto após // são comentários, não fazendo parte do programa.

// Os programas começam tipicamente por um conjunto de inclusões.
// Neste caso incluem-se ferramentas de entradas e saídas (Input/Output) a partir
```

```
// de canais (stream):
#include <iostream>

// Para evitar ter de escrever std::cout, std::cin, etc.
using namespace std;

/// Este programa calcula o máximo divisor comum de dois números.
int main()
{
    // Instruções de inserção de informação no canal de saída cout, ligado ao ecrã.
    // Mostra ao utilizador informação sobre o programa e pede-lhe para inserir os
    // dois inteiros dos quais quer obter o máximo divisor comum:
    cout << "Máximo divisor comum de dois números."
         << endl;
    cout << "Introduza dois inteiros positivos: ";

    // Definição de duas variáveis inteiras:
    int m, n;

    // Instrução de extracção de informação do canal de entrada, ligado ao teclado.
    // Serve para obter do utilizador os valores dos quais ele pretende saber o mdc:
    cin >> m >> n; // Assume-se que m e n são positivos!

    // Como um divisor é sempre menor ou igual a um número, escolhe-se
    // o mínimo dos dois!
    int k;
    if(m < n)
        k = m;
    else
        k = n;
    // Neste momento sabe-se que mdc(m, n) ≤ k.

    while(m % k != 0 or n % k != 0) {
        // Se o ciclo não parou, então k não divide m e n, logo,
        // k ≠ mdc(m, n) ∧ mdc(m, n) ≤ k. Ou seja, mdc(m, n) < k.
        --k; // Decrementa k. É o progresso do ciclo.
        // Neste momento, mdc(m, n) ≤ k outra vez! É o invariante do ciclo!
    }

    // Como mdc(m, n) ≤ k (invariante do ciclo) e k divide m e n
    // (o ciclo terminou, não foi?), conclui-se que k = mdc(m, n)!

    // Insere no canal de saída uma mensagem para o utilizador dizendo-lhe qual o
    // resultado:
    cout << "O máximo divisor comum de " << m
         << " e " << n << " é " << k << '.' << endl;
```

```
}
```

Este programa foi apresentado como concretização em C++ do algoritmo de cálculo do mdc também desenvolvido no capítulo anterior. Em rigor isto não é verdade. O programa está dividido em três partes, das quais só a segunda parte é a concretização directa do algoritmo desenvolvido:

```
// Como um divisor é sempre menor ou igual a um número, escolhe-se
// o mínimo dos dois!
int k;
if(m < n)
    k = m;
else
    k = n;
// Neste momento sabe-se que mdc(m, n) ≤ k.

while(m % k != 0 or n % k != 0) {
    // Se o ciclo não parou, então k não divide m e n, logo,
    // k ≠ mdc(m, n) ∧ mdc(m, n) ≤ k. Ou seja, mdc(m, n) < k.
    --k; // Decrementa k. É o progresso do ciclo.
    // Neste momento, mdc(m, n) ≤ k outra vez! É o invariante do ciclo!
}

// Como mdc(m, n) ≤ k (invariante do ciclo) e k divide m e n
// (o ciclo terminou, não foi?), conclui-se que k = mdc(m, n)!
```

As duas partes restantes resolvem dois problemas que foram subtilmente ignorados no capítulo anterior:

1. De onde vêem as entradas do programa?
2. Para onde vão as saídas do programa?

Claramente as entradas do programa têm de vir de uma entidade exterior ao programa. Que entidade exactamente depende da aplicação que se pretende dar ao programa. Por exemplo, as entradas poderiam ser originadas:

- por um humano utilizador do programa,
- por um outro programa qualquer ou
- a partir de um ficheiro no disco rígido do computador.

Também é óbvio que as saídas (neste caso a saída) do programa têm de ser enviadas para alguma entidade externa ao programa, pois de outra forma não mereceriam o seu nome.

No caso do programa apresentado assume-se que existe um utilizador humano, que digita as entradas num teclado e vê as saídas num ecrã.

2.1.1 Consola e canais

É típico os programas actuais usarem interfaces mais ou menos sofisticadas com o seu utilizador. Ao longo deste texto, no entanto, adoptar-se-á um modelo muito simplificado (e primitivo) de interacção com o utilizador: admite-se que os programas desenvolvidos são executados numa consola de comandos, sendo as entradas do programa lidas de um teclado no qual um utilizador humano as digita (ou lidas de um ficheiro de texto guardado no disco rígido do computador) e sendo as saídas do programa escritas na consola de comandos onde o utilizador humano as pode ler (ou escritas num ficheiro de texto guardado no disco rígido do computador).

A consola de comandos tem um modelo muito simples. É uma grelha rectangular de células, com uma determinada altura e largura, em que cada célula pode conter um qualquer dos caracteres (símbolos gráficos) disponíveis na tabela de codificação usada na máquina em causa (ver explicação mais abaixo). Quando um programa é executado[2], a grelha de células fica à sua disposição. As saídas do programa fazem-se escrevendo caracteres nessa grelha, o que se consegue inserindo-os no chamado *canal*[3] *de saída* `cout`. Pelo contrário as entradas fazem-se lendo caracteres do teclado, o que se consegue extraindo-os do chamado *canal de entrada* `cin`. No modelo de consola usado, os caracteres correspondentes às teclas premidas pelo utilizador do programa não são simplesmente postos à disposição do programa em execução: são também mostrados na grelha de células da consola de comandos, que assim mostrará uma mistura de informação inserida pelo utilizador do programa e de informação gerada pelo próprio programa.

O texto surge na consola de comandos de cima para baixo e da esquerda para a direita. Quando uma linha de células está preenchida, o texto continua a ser escrito na linha seguinte. Se a consola estiver cheia, a primeira linha é descartada e o conteúdo de cada uma das linhas restantes é deslocado para a linha imediatamente acima, deixando uma nova linha disponível na base da grelha.

Num programa em C++ o canal de saída para o ecrã[4] é designado por `cout`. O canal de entrada de dados pelo teclado é designado por `cin`. Para efectuar uma operação de escrita no ecrã usa-se o operador de inserção <<. Para efectuar uma operação de leitura de dados do teclado usa-se o operador de extracção >>.

No programa do mdc o resultado é apresentado pela seguinte instrução:

```
cout << "O máximo divisor comum de " << m
     << " e " << n << " é " << k << '.' << endl;
```

Admitindo, por exemplo, que as variáveis do programa têm os valores indicados na Figura 1.5(b), o resultado será aparecer escrito na consola:

```
O máximo divisor comum de 12 e 8 é 4.
```

[2]O que se consegue em Linux escrevendo na consola o seu nome (precedido de `./`) depois do "pronto" (*prompt*).
[3]Optou-se por traduzir *stream* por canal em vez de fluxo, com se faz noutros textos, por parecer uma abstracção mais apropriada.
[4]Para simplificar chamar-se-á muitas vezes "ecrã" à grelha de células que constitui a consola.

Por outro lado, as entradas são lidas do teclado pelas seguintes instruções:

```
cout << "Introduza dois inteiros positivos: ";
int m, n;
cin >> m >> n; // Assume-se que m e n são positivos!
```

A primeira instrução limita-se a escrever no ecrã uma mensagem pedindo ao utilizador para introduzir dois números inteiros positivos, a segunda serve para definir as variáveis para onde os valores dos dois números serão lidos e a terceira procede à leitura, ou extração, propriamente dita.

Após uma operação de extracção de valores do canal de entrada, a variável para onde a extracção foi efectuada toma o valor que foi inserido no teclado pelo utilizador do programa, desde que esse valor possa ser tomado por esse tipo de variável[5].

Ao ser executada uma operação de leitura como acima, o computador interrompe a execução do programa até que seja introduzido algum valor no teclado. Todos os espaços em branco são ignorados. Consideram-se espaços em branco os espaços propriamente ditos (' '), os tabuladores (que são os caracteres que se insere quando se carrega na tecla `tab` ou ⇥) e os fins-de-linha (que são caracteres especiais que assinalam o fim de uma linha de texto e que se insere quando se carrega na tecla `return` ou ↵). Para tornar evidente a presença de um espaço quando se mostra um sequência de caracteres usar-se-á o símbolo ␣. Compare-se

```
Este texto tem um tabulador aqui        e um espaço no fim
```

com

```
Este texto tem um tabulador aqui⇥e um espaço no fim␣
```

O *manipulador* `endl` serve para mudar a impressão para a próxima linha da grelha de células sem que a linha corrente esteja preenchida. Por exemplo, as instruções

```
cout << "****";
cout << "***";
cout << "**";
cout << "*";
```

resultam em

```
**********
```

Mas se se usar o manipulador `endl`

[5] Mais tarde se verá como verificar se o valor introduzido estava correcto, ou seja, como verificar se a operação de extracção teve sucesso.

```
cout << "****" << endl;
cout << "***" << endl;
cout << "**" << endl;
cout << "*" << endl;
```

o resultado é

```
****
***
**
*
```

Pode-se usar uma única instrução de inserção para produzir o mesmo resultado, embora seja recomendável dividir essa instrução em várias linhas, por forma a que seja mais evidente o aspecto gráfico com que o ecrã ficará depois de executada a instrução:

```
cout << "****" << endl
     << "***" << endl
     << "**" << endl
     << "*" << endl;
```

O mesmo efeito do manipulador `endl` pode ser obtido incluindo no texto a escrever no ecrã a sequência de escape `'\n'` (ver explicação mais abaixo). Ou seja;

```
cout << "****\n"
     << "***\n"
     << "**\n"
     << "*\n";
```

A utilização de canais de entrada e saída, associados não apenas ao teclado e ao ecrã mas também a ficheiros arbitrários no disco, será vista mais tarde.

2.1.2 Definição de variáveis

Na maior parte das linguagens de programação de alto nível as variáveis tem de ser *definidas*[6] antes de utilizadas. A definição das variáveis serve para indicar claramente quais são as suas duas características estáticas, nome e tipo, e qual o seu valor inicial, se for possível indicar um que tenha algum significado no contexto em causa.

No programa em análise podem ser encontradas três definições de variáveis em duas instruções de definição:

```
int m, n;
int k;
```

[6]Ou pelo menos *declaradas*.

Estas instruções definem as variáveis `m` e `n`, que conterão os valores inteiros dos quais se pretende saber o `mdc`, e a variável `k` que conterá o desejado valor do `mdc`.

Ver-se-á mais tarde que é de toda a conveniência inicializar as variáveis, i.e., indicar o seu valor inicial durante a sua definição. Por exemplo, para inicializar `k` com o valor 121 poder-se-ia ter usado:

```
int k = 121;
```

No entanto, no exemplo dado, nenhuma das variáveis pode ser inicializada com um valor que tenha algum significado. As variáveis `m` e `n` têm de ser lidas por operações de extracção, pelo que a sua inicialização acabaria por não ter qualquer efeito prático, uma vez que os valores iniciais seriam imediatamente substituídos pelos valores extraídos do canal de entrada. À variável `k`, por outro lado, só se pode atribuir o valor desejado depois de saber qual é a menor das outras duas variáveis, pelo que a sua inicialização também seria inútil[7].

A definição de variáveis e os seus possíveis tipos serão vistos em pormenor nas Secções 2.2 e 2.3.

2.1.3 Controlo de fluxo

A parte mais interessante do programa apresentado é a segunda, correspondente ao algoritmo desenvolvido no capítulo anterior. Aí encontram-se as instruções do programa que permitem alterar o fluxo normal de execução, que ocorre normalmente de cima para baixo e da esquerda para a direita ao longo das instruções dos programas. São as chamadas *instruções de selecção* (neste caso um `if else`) e as *instruções iterativas* (neste caso um `while`).

O objectivo de uma instrução de selecção é permitir a selecção de duas instruções alternativas de acordo com o valor lógico de uma determinada condição. No programa existe uma instrução de selecção:

```
if(m < n)
```

[7] Na realidade seria possível proceder à inicialização recorrendo à função (ver Capítulo 3) `min()`, desde que se acrescentasse a inclusão do ficheiro de interface apropriado (`algorithm`):

```
#include <iostream>
#include <algorithm>

using namespace std;

int main()
{
    ...

   int k = min(m, n);

    ...
}
```

```
    k = m;
else
    k = n;
```

Esta instrução de selecção coloca em `k` o menor dos valores das variáveis `m` e `n`. Para isso executa alternativamente duas instruções de *atribuição*, que colocam na variável `k` o valor de `m` se `m` $<$ `n`, ou o valor de `n` no caso contrário, i.e., se `n` $\leq$ `m`. As instruções de selecção `if` têm sempre um formato semelhante ao indicado, com as palavras-chave `if` e `else` a preceder as instruções alternativas, e com a condição entre parênteses logo após a palavra-chave `if`. Uma instrução de selecção pode controlar a execução de sequências de instruções, bastando para isso colocá-las entre chavetas:

```
if(x < y) {
    int aux = x;
    x = y;
    y = aux;
}
```

Tal como no exemplo anterior, é possível omitir a segunda instrução alternativa, após a palavra-chave `else`, embora nesse caso a instrução `if` se passe a chamar *instrução condicional* e já não instrução de selecção.

É importante notar que a atribuição se representa, em C++, pelo símbolo =. Assim, a instrução

```
k = m;
```

não deve ser lida como "`k` é igual a `m`", mas sim, "`k` fica com o valor de `m`".

O objectivo de uma instrução iterativa é repetir uma instrução controlada, i.e., construir um *ciclo*. No caso de uma instrução `while`, o objectivo é, enquanto uma condição for verdadeira, repetir a instrução controlada. No programa existe uma instrução iterativa `while`:

```
while(m % k != 0 or n % k != 0) {
    --k;
}
```

O objectivo desta instrução é, enquanto `k` não for divisor comum de `m` e `n`, ir diminuindo o valor de `k`. Tal como no caso das instruções de selecção, também nas instruções iterativas pode haver apenas uma instrução controlada ou uma sequência delas, desde que envoltas por chavetas. No exemplo acima as chavetas são redundantes, pelo que se pode escrever simplesmente

```
while(m % k != 0 or n % k != 0)
    --k;
```

Nesta instrução iterativa há duas expressões importantes. A primeira é a expressão `--k` na instrução controlada pelo `while`, consistindo simplesmente na aplicação do operador de decrementação prefixo, que neste caso se limita a diminuir de um o valor guardado em `k`. A segunda é a expressão usada como condição do ciclo, ou seja, como guarda. Na expressão usada como guarda encontram-se três operadores:

1. O operador `or` que calcula a disjunção dos valores lógicos de duas expressões condicionais. Ou seja, é um operador lógico correspondente ao "ou" da linguagem natural e ao símbolo $\vee$ em notação matemática (embora com particularidades que serão vistas mais tarde).

2. O operador `!=`, que resulta no valor $\mathcal{V}$ (verdadeiro) se os seus operandos forem diferentes e no valor $\mathcal{F}$ (falso) se eles forem iguais. A notação usada para a diferença deve-se à inexistência do símbolo $\neq$ na generalidade dos teclados (e mesmo das tabelas de codificação de caracteres).

3. O operador `%`, que resulta no resto da divisão inteira do primeiro operando pelo segundo. O símbolo usado (percentagem) deve-se à inexistência do símbolo $\div$ na generalidade do teclados.

As instruções de selecção e de iteração e o seu desenvolvimento serão estudados no Capítulo 4. Os operadores e as expressões com eles construídas serão pormenorizado na Secção 2.7.

2.2 Variáveis

2.2.1 Memória e inicialização

Os computadores têm memória, sendo através da sua manipulação que os programas eventualmente acabam por chegar aos resultados pretendidos (as saídas). As linguagens de alto nível, como o C++, "escondem" a memória por trás do conceito de variável, que são uma forma estruturada de lhe aceder. As variáveis são, na realidade, pedaços de memória a que se atribui um nome, que têm um determinado conteúdo ou valor, e cujo conteúdo é interpretado de acordo com o tipo da variável. Todas as variáveis têm um dado tipo. Uma variável pode ser, por exemplo, do tipo `int` em C++. Se assim for, essa variável terá sempre um valor inteiro dentro de uma gama de valores admissível.

Os tipos das variáveis não passam, na realidade, de uma abstracção. Todas as variáveis, independentemente do seu tipo, são representadas na memória do computador por padrões de *bits*[8], os famosos "zeros e uns", colocados na zona de memória atribuída a essa variável. O tipo de uma variável indica simplesmente como o padrão de *bits* associado à variável deve ser interpretado.

[8]Dígitos binários, do inglês *binary digit*.

Cada variável tem duas características estáticas, o nome e o tipo, e uma característica dinâmica, o seu valor[9], tal como nos algoritmos. Antes de usar uma variável é necessário indicar ao compilador qual o seu nome e tipo, de modo a que a variável possa ser construída, i.e., ficar associada a uma zona de memória, e de modo a que a forma de interpretar os padrões de *bits* nessa zona de memória fique estabelecida. Uma instrução onde se contrói uma variável com um dado nome, de um determinado tipo, e com um determinado valor inicial, denomina-se *definição*. A instrução

```
int a = 10;
```

é a definição de uma variável chamada `a` que pode guardar valores do tipo `int` (inteiros) e cujo valor inicial é o inteiro 10. A sintaxe das definições pode ser algo complicada, mas em geral tem a forma acima, isto é, o nome do tipo, seguido do nome da variável, e seguido de uma inicialização.

Uma forma intuitiva de ver uma variável é imaginá-la como uma folha de papel com um nome associado e onde se decidiu escrever apenas números inteiros, por exemplo. No entanto, essa folha de papel pode conter apenas um valor em cada instante, pelo que a escrita de um novo valor implica o apagamento do anterior, e tem de conter sempre um valor, mesmo inicialmente (é como se a folha viesse já preenchida de fábrica). A notação usada para representar graficamente uma variável é a introduzida no capítulo anterior. A Figura 2.1 mostra a representação gráfica da variável `a` definida acima.

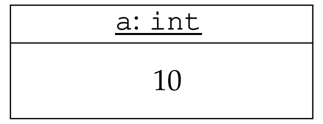


Figura 2.1: Notação gráfica para uma variável definida por `int a = 10;`. O nome e o tipo da variável estão num compartimento à parte, no topo, e têm de ser sublinhados.

Quando uma variável é definida, o computador reserva para ela uma zona da memória com o número de *bits* necessário para guardar um valor do tipo indicado (essas reservas são feitas em múltiplos de uma unidade básica de memória, tipicamente com oito *bits*, ou seja, um octeto ou *byte*[10]). Se a variável não for explicitamente inicializada, essa posição de memória conterá inicialmente um padrão de *bits* arbitrário, desconhecido, a que se dá usualmente o nome de "lixo". Por exemplo, se se tivesse usado a definição

```
int a;
```

a variável `a` conteria um padrão de *bits* arbitrário e portanto o seu valor seria arbitrário. Diz-se-ia que `a` continha lixo. Para evitar esta arbitrariedade, que pode ter consequências nefastas no

[9]Mais tarde se aprenderá que existe um outro género de variáveis que não têm nome. Estas variáveis, que se dizem anónimas, podem ser variáveis temporárias, que existem apenas durante o cálculo de expressões, ou as variáveis dinâmicas introduzidas no Capítulo 11. De igual forma se verá que existem *constantes*, que partilham todas as características das variáveis, embora o seu valor também seja estático.

[10]Em rigor a dimensão dos *bytes* pode variar de máquina para máquina, não tendo de ter forçosamente oito *bits*.

comportamento do programa se não se tiver cuidado, ao definir uma variável deve-se, sempre que possível e razoável, inicializá-la com um dado valor inicial, tal como indicado na primeira definição. A inicialização de uma variável pode ser feita de duas formas alternativas[11]:

```
int a = 10; // como originalmente.
int a(10);  // forma alternativa.
```

A sintaxe das definições de variáveis também permite que se defina mais do que uma variável numa só instrução. Por exemplo,

```
int a = 0, b = 1, c = 2;
```

define três variáveis, todas do tipo `int`.

2.2.2 Identificadores

Os nomes de variáveis, e em geral de todas as entidades em C++, i.e., os *identificadores*, podem ser constituídos por letras[12] (sendo as minúsculas distinguidas das maiúsculas), dígitos decimais, e também pelo caractere '_' (sublinhado ou *underscore*), que se usa normalmente para aumentar a legibilidade do nome. Um identificador não pode conter espaços nem pode começar por um dígito.

Durante a tradução do programa, o compilador, antes de fazer uma análise sintáctica do programa, durante a qual verifica a gramática do programa, faz uma análise lexical, durante a qual identifica os símbolos (ou *tokens*) que constituem o texto. Esta análise lexical é "gulosa": os símbolos detectados (palavras, sinais de pontuação, etc.) são tão grandes quanto possível, pelo que `lêValor` é sempre interpretado como um único identificador, e não como o identificador `lê` seguido do identificador `Valor`.

Os identificadores devem ser tão auto-explicativos e claros quanto possível. Se uma variável guarda, por exemplo, o número de alunos numa turma, deve chamar-se

```
número_de_alunos_na_turma
```

ou então

```
número_de_alunos
```

[11] Na realidade as duas formas não são rigorosamente equivalentes (ver Secção C.1).

[12] A norma do C++ especifica que de facto se pode usar qualquer letra. Infelizmente a maior parte dos compiladores existentes recusa-se a aceitar letras acentuadas ou letras especiais, tais como "ç" ou "ñ". Este texto foi escrito ignorando esta restrição prática por duas razões:

1. Mais tarde ou mais cedo os compiladores serão corrigidos e o autor não precisará de mais do que eliminar esta nota :-).
2. O autor claramente prefere escrever com acentos. Por exemplo, `Cágado`...

se o complemento "na turma" for evidente dado o contexto no programa. É claro que o nome usado para uma variável não tem qualquer importância para o compilador, mas uma escolha apropriada dos nomes pode aumentar grandemente a legibilidade dos programas (pelos humanos...).

2.2.3 Inicialização

As zonas de memória correspondentes às variáveis contêm sempre um padrão de *bits*: não há *bits* "vazios". Assim, as variáveis têm sempre um valor qualquer. Depois de uma definição, se não for feita uma inicialização explícita conforme sugerido atrás, a variável definida contém um valor arbitrário[13]. Uma fonte frequente de erros é o esquecimento de inicializar as variáveis definidas. Por isso, é recomendável a inicialização de todas as variáveis tão cedo quanto possível, desde que essa inicialização tenha algum significado para o resto do programa (e.g., no exemplo da Secção 1.4 essa inicialização não era possível[14]). Da mesma forma, as variáveis devem-se definir tão tarde quanto possível, i.e., tão próximo quanto for possível da sua primeira utilização, por forma a facilitar a sua inicialização com um valor que faça sentido e a evitar a utilização dessa variável por engano em instruções intermédias.

Existem algumas circunstâncias nas quais as variáveis de tipos básicos do C++ são inicializadas implicitamente com zero (ver Secção 3.2.15). Deve-se evitar fazer uso desta característica do C++ e inicializar explicitamente todas as variáveis.

2.3 Tipos básicos

O tipo das variáveis está relacionado directamente com o conjunto de possíveis valores tomados pelas variáveis desse tipo. Para poder representar e manipular as variáveis na memória do computador, o compilador associa a cada tipo não só o número de *bits* necessário para a representação de um valor desse tipo na memória do computador, mas também a forma como os padrões de *bits* guardados em variáveis desse tipo devem ser interpretados. A linguagem C++ tem definidos *a priori* alguns tipos: os chamados tipos básicos ou primitivos do C++.

O paradigma de programação que usa na parte inicial destas folhas é o paradigma de programação *procedimental*. Só depois se estudarão dois paradigmas de programação alternativos que passam por acrescentar à linguagem novos tipos mais apropriados para os problemas em causa: a *programação baseada em objectos* (Capítulo 7 e, sobretudo, Capítulo 8) e a *programação orientada para objectos* (Capítulo 12). Até lá os únicos tipos disponíveis são os tipos básicos.

Nas tabelas seguintes são apresentados os tipos básicos existentes no C++ e a gama de valores que podem representar em Linux sobre Intel[15]. É muito importante notar que os computadores

[13] Desde que seja de um tipo básico do C++ e em alguns outros casos. A afirmação não é verdadeira para classes em geral (ver Capítulo 7).

[14] Ou melhor, a inicialização é possível desde que se use o operador (meio exótico) `?:` (ver Secção 4.4.1):

```
int k = m < n ? m : n;
```

[15] Ambas as tabelas se referem ao compilador de C++ `g++` da GNU Compiler Collection (GCC 3.2) num sistema Linux, distribuição Caixa Mágica, núcleo 2.4.14, correndo sobre a arquitectura Intel, localizado para a Europa Ocidental.

são máquinas finitas: tudo é limitado, desde a memória ao tamanho da representação dos tipos em memória. Assim, a gama de diferentes valores possíveis de guardar em variáveis de qualquer tipo é limitada. A gama de valores representável para cada tipo de dados pode variar com o processador e o sistema operativo. Por exemplo, no sistema operativo Linux em processadores Alpha, os `long int` (segunda tabela) têm 64 bits. Em geral só se pode afirmar que a gama dos `long` é sempre suficiente para abarcar qualquer `int` e a gama dos `int` é sempre suficiente para abarcar qualquer `short`, o mesmo acontecendo com os `long double` relativamente aos `double` e com os `double` relativamente aos `float`.

Tabela 2.1: Tipos básicos elementares.

Tipo	Descrição	Gama	*Bits*
`bool`	valor booleano ou lógico	$\mathcal{V}$ e $\mathcal{F}$	8
`int`	número inteiro	-2^{31} a $2^{31}-1$ (-2147483648 a 2147483647)	32
`float`	número racional (representação IEEE 754 [6])	1,17549435 $\times 10^{-38}$ a 3,40282347 $\times 10^{38}$ (e negativos)	32
`char`	caractere (código Latin-1)	A maior parte dos caracteres gráficos em uso. Exemplos: 'a', 'A', '1', '!', '*', etc.	8

Tabela 2.2: Outros tipos básicos.

Tipo	Descrição	Gama	*Bits*
`short` [`int`]	número inteiro	-2^{15} a 2^{15-1} (-32768 a 32767)	16
`unsigned short` [`int`]	número inteiro positivo	0 a $2^{16}-1$ (0 a 65535)	16
`unsigned` [`int`]	número inteiro positivo	0 a $2^{32}-1$ (0 a 4294967295)	32
`long` [`int`]	número inteiro	a mesma que `int`	32
`unsigned long` [`int`]	número inteiro positivo	a mesma que `unsigned int`	32
`double`	número racional (representação IEEE 754 [6])	2,2250738585072014 $\times 10^{-308}$ a 1,7976931348623157 $\times 10^{308}$ (e negativos)	64
`long double`	número racional (representação IEEE 754 [6])	3,36210314311209350626 $\times 10^{-4932}$ a 1,18973149535723176502 $\times 10^{4932}$ (e negativos)	96

Alguns dos tipos derivados de `int` podem ser escritos de uma forma abreviada: os parênteses rectos na Tabela 2.2 indicam a parte opcional na especificação do tipo. O qualificador `signed` também pode ser usado, mas `signed int` e `int` são sinónimos. O único caso em que este qualificador faz diferença é na construção `signed char`, mas apenas em máquinas onde os `char` não têm sinal.

A representação interna dos vários tipos pode ser ou não relevante para os programas. A maior parte das vezes não é relevante, excepto quanto ao facto de que se deve sempre estar ciente das limitações de qualquer tipo. Por vezes, no entanto, a representação é muito relevante, nomeadamente quando se programa ao nível do sistema operativo e se tem de aceder a *bits* individuais. Assim, apresentam-se em seguida algumas noções sobre a representação usual dos tipos básicos do C++. Esta matéria será pormenorizada na disciplina de Arquitectura de Computadores.

2.3.1 Tipos aritméticos

Os tipos aritméticos são todos os tipos que permitem representar números (`int`, `float` e seus derivados). Variáveis (e valores literais, ver Secção 2.4) destes tipos podem ser usadas para realizar operações aritméticas e relacionais, que serão vistas nas próximas secções. Os tipos derivados de `int` chamam-se tipos inteiros. Os tipos derivados de `float` chamam-se tipos de vírgula flutuante. Os `char`, em rigor, também são tipos aritméticos e inteiros, mas serão tratados à parte.

Representação de inteiros

Admita-se, para simplificar, que num computador hipotético as variáveis do tipo `unsigned int` têm 4 *bits* (normalmente têm 32 bits). Pode-se representar esquematicamente uma dada variável do tipo `unsigned int` como

b_3	b_2	b_1	b_0

em que os b_i com $i = 0, \cdots, 3$ são *bits*, tomando portanto os valores 0 ou 1. É fácil verificar que existem apenas $2^4 = 16$ padrões diferentes de *bits* possíveis de colocar numa destas variáveis. Como associar valores inteiros a cada um desses padrões? A resposta mais óbvia é a que é usada na prática: considera-se que o valor representado é $(b_3b_2b_1b_0)_2$, i.e., que os *bits* são dígitos de um número expresso na base binária (ou seja, na base 2). Por exemplo:

1	0	0	1

é o padrão de *bits* correspondente ao valor $(1001)_2$, ou seja $1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 9$, em decimal[16].

Suponha-se agora que a variável contém

[16] Os números inteiros podem-se representar de muitas formas. A representação mais evidente, corresponde a desenhar um traço por cada unidade: ||||||||||||| representa o inteiro treze ("treze" por sua vez é outra representação, por extenso). A representação romana do mesmo inteiro é XIII. A representação árabe é *posicional* e é a mais prática. Nela usam-se sempre os mesmos 10 dígitos, que vão aumentando de peso da direita para a esquerda (o peso é multiplicado sucessivamente por 10), e onde o dígito zero é fundamental. A representação árabe do mesmo inteiro é 13, que significa $1 \times 10 + 3$. A representação árabe usa 10 dígitos e por isso diz-se que usa a base 10, ou

1	1	1	1

e que se soma 1 ao seu conteúdo: o resultado é $(1111+1)_2 = (10000)_2$. Mas este valor não é representável num `unsigned int` de quatro *bits*! Um dos *bits* tem de ser descartado. Normalmente escolhe-se guardar apenas os 4 *bits* menos significativos do resultado, pelo que, no computador hipotético em que os inteiros têm 4 *bits*, $(1111+1)_2 = (0000)_2$. Ou seja, nesta aritmética binária com um número limitado de dígitos, tudo funciona como se os valores possíveis estivessem organizados em torno de um relógio, neste caso em torno de um relógio com 16 horas, ver Figura 2.2, onde após as $(1111)_2 = 15$ horas fossem de novo $(0000)_2 = 0$ horas[17].

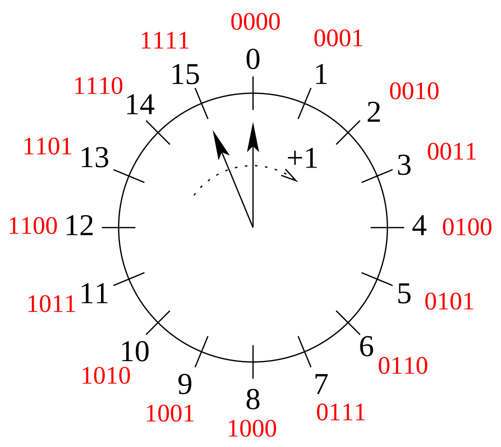


Figura 2.2: Representação de inteiros sem sinal com quatro *bits*.

A extensão destas ideias para, por exemplo, 32 *bits* é trivial: nesse caso o relógio seria enorme, pois teria 2^{32} horas, de 0 a $2^{32}-1$, que é, de facto, a gama dos `unsigned int` no Linux com a configuração apresentada.

É extremamente importante recordar as limitações dos tipos. Por exemplo, como os valores das variáveis do tipo `int` não podem crescer indefinidamente, ao se atingir o topo do relógio

---

que a representação é decimal. Pode-se usar a mesma representação posicional com qualquer base, desde que se forneçam os respectivos dígitos. Em geral a notação posicional tem a forma $(d_{n-1}d_{n-2}\cdots d_1d_0)_B$ onde B é a base da representação, n é o número de dígitos usado neste número em particular, e d_i com $i = 0,\cdots,n-1$ são os sucessivos dígitos, sendo que cada um deles pertence a um conjunto com B possíveis dígitos possuindo valores de 0 a $B-1$. O número representado por $(d_{n-1}d_{n-2}\cdots d_1d_0)_B$ pode ser calculado como

$$\sum_{i=0}^{n-1} d_iB^i = d_{n-1}B^{n-1} + d_{n-2}B^{n-2} + \cdots + d_1B^1 + d_0B^0.$$

Para $B = 2$ (numeração binária) o conjunto de possíveis dígitos, por ordem crescente de valor, é $\{0,1\}$, para $B = 8$ (numeração octal) é $\{0,1,2,3,4,5,6,7\}$, para $B = 10$ é $\{0,1,2,3,4,5,6,7,8,9\}$, e para $B = 16$ (numeração hexadecimal) é $\{0,1,2,3,4,5,6,7,8,9,\mathrm{A},\mathrm{B},\mathrm{C},\mathrm{D},\mathrm{E},\mathrm{F}\}$, onde à falta de símbolos se recorreu às letras de A a F. Quando se omite a indicação explícita da base, assume-se que esta é 10.

[17]Tipicamente no topo do relógio estaria 16, e não zero, mas optar-se-á pela numeração a começar em zero, que é quase sempre mais vantajosa (ver Nota 4 na página 239).

volta-se a zero!

Falta verificar como representar inteiros com sinal, i.e., incluindo não só valores positivos mas também valores negativos. Suponha-se que os `int` têm, de novo num computador hipotético, apenas 4 *bits*. Admita-se uma representação semelhante à dos `unsigned int`, mas diga-se que, no lugar das 15 horas (padrão de *bits* 1111), mesmo antes de chegar de novo ao padrão 0000, o valor representado é x, ver Figura 2.3. Pelo que se disse anteriormente, somar 1 a x corresponde a rodar o ponteiro do padrão 1111 para o padrão 0000. Admitindo que o padrão 0000 representa de facto o valor 0, tem-se $x + 1 = 0$. Conclui, por isso, que o padrão 1111 é um bom candidato para representar o valor $x = -1$! Estendendo o argumento anterior, pode-se dizer que as 14 horas correspondem à representação de -2, e assim sucessivamente, como se indica na Figura 2.4.

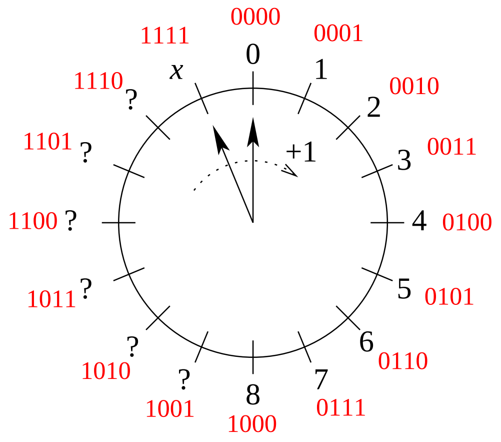


Figura 2.3: Como representar os inteiros com sinal?

O salto nos valores no relógio deixa de ocorrer do padrão 1111 para o padrão 0000 (15 para 0 horas, se interpretados como inteiros sem sinal), para passar a ocorrer na passagem do padrão 0111 para o padrão 1000 (das 7 para as -8 horas, se interpretados como inteiros com sinal). A escolha deste local para a transição não foi arbitrária. Em primeiro lugar permite representar um número semelhante de valores positivos e negativos: 7 positivos e 8 negativos. Deslocando a transição de uma hora no sentido dos ponteiros do relógio, poder-se-ia alternativamente representar 8 positivos e 7 negativos. A razão para a escolha apresentada na figura acima prende-se com o facto de, dessa forma, a distinção entre não-negativos (positivos ou zero) e negativos se poder fazer olhando apenas para o *bit* mais significativo (o *bit* mais à esquerda), que quando é 1 indica que o valor representado é negativo. A esse *bit* chama-se, por isso, *bit* de sinal. Esta representação chama-se representação em *complemento para 2*. Em Arquitectura de Computadores ficarão mais claras as vantagens desta representação na simplificação do *hardware* do computador encarregue de realizar operações com valores inteiros.

Como saber, olhando para um padrão de *bits*, qual o valor representado, no caso de se estarem a usar inteiros com sinal? Primeiro olha-se para o *bit* de sinal. Se for 0, interpreta-se o padrão de *bits* como um número binário: é esse o valor representado. Se o *bit* de sinal for 1, então o valor

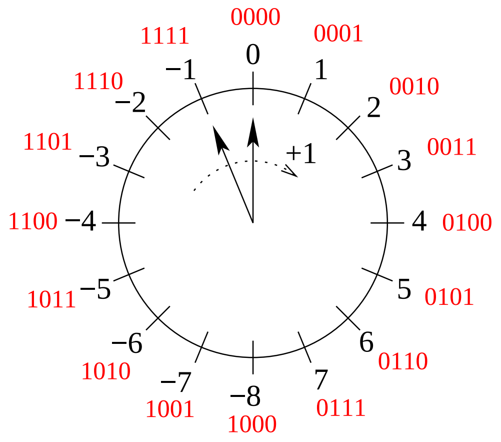


Figura 2.4: Representação de inteiros com sinal com quatro *bits*.

representado é igual ao valor binário correspondente ao padrão de *bits* subtraído de 16. Por exemplo, observando o padrão 1011, conclui-se que representa o valor negativo $(1011)_2 - 16 = 11 - 16 = -5$, como se pode confirmar na Figura 2.4.

A extensão destas ideias para o caso dos `int` com 32 bits é muito simples. Nesse caso os valores representados variam de -2^{31} a $2^{31} - 1$ e a interpretação dos valores representados faz-se como indicado no parágrafo acima, só que subtraindo 2^{32}, em vez de 16, no caso dos valores negativos. Por exemplo, os padrões

00000000000000000000010000000001

e

10000000000000000000000000000111

representam os valores 1025 e -2147483641, como se pode verificar facilmente com uma máquina de calcular.

Representação de valores de vírgula flutuante

Os tipos de vírgula flutuante destinam-se a representar valores decimais, i.e., uma gama limitada dos números racionais. Porventura a forma mais simples de representar valores racionais passa por usar exactamente a mesma representação que para os inteiros com sinal (complemento para dois), mas admitindo que todos os valores devem ser divididos por uma potência fixa de 2. Por exemplo, admitindo que se usam 8 *bits* na representação e que a divisão é feita por $16 = 2^4 = (10000)_2$, tem-se que o padrão 01001100 representa o valor $\frac{(01001100)_2}{(10000)_2} = (0100,1100)_2 = 4,75$.

Esta representação corresponde a admitir que os *bits* representam um valor binário em que a vírgula se encontra quatro posições para a esquerda

b_3	b_2	b_1	b_0	,	b_{-1}	b_{-2}	b_{-3}	b_{-4}

do que acontecia na representação dos inteiros, onde a vírgula está logo após o *bit* menos significativo (o *bit* mais à direita)

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0	,

O valor representado por um determinado padrão de *bits* $b_3b_2b_1b_0b_{-1}b_{-2}b_{-3}b_{-4}$ é dado por

$$\sum_{i=-4}^{3} b_i 2^i.$$

O menor valor positivo representável nesta forma é $\frac{1}{16} = 0,0625$. Por outro lado, só se conseguem representar valores de -8 a -0,0625, o valor 0, e valores de 0,0625 a 7,9375. O número de dígitos decimais de precisão está entre dois e três (um antes da vírgula e entre um e dois depois da vírgula). Caso se utilize 32 *bits* e se desloque a vírgula 16 posições para a esquerda, o menor valor positivo representável é $\frac{1}{2^{16}} = 0,00001526$, e são representáveis valores de -32768 a -0,00001526, o valor 0, e valores de 0,00001526 a 32767,99998, aproximadamente, correspondendo a entre nove e 10 dígitos decimais de precisão, cinco antes da vírgula e entre quatro e cinco depois depois da vírgula.

A este tipo de representação chama-se *vírgula fixa*, por se colocar a vírgula numa posição fixa e pré-determinada.

A representação em vírgula fixa impõe limitações consideráveis na gama de valores representáveis: no caso dos 32 *bits* com vírgula 16 posições para a esquerda, representam-se apenas valores entre aproximadamente $1,5 \times 10^{-5}$ e $3,8 \times 10^{5}$, em módulo. Este problema resolve-se usando uma representação em que a vírgula não está fixa, mas varia consoante as necessidades: a vírgula passa a "flutuar". Uma vez que especificar a posição da vírgula é o mesmo que especificar uma potência de dois pela qual o valor deve ser multiplicado, a representação de vírgula flutuante corresponde a especificar um número da forma $m \times 2^e$, em que a m se chama mantissa e a e se chama expoente. Na prática esta representação não usa a representação em complemento para dois em nenhum dos seus termos, pelo que inclui um termo de sinal

$$s \times m \times 2^e,$$

em que m é sempre não-negativo e s é o termo de sinal, valendo -1 ou 1.

A representação na memória do computador de valores de vírgula flutuante passa, pois, por dividir o padrão de *bits* disponível em três zonas com representações diferentes: o sinal, a mantissa e o expoente. Como parte dos *bits* têm de ser usados para o expoente, que especifica a localização da vírgula "flutuante", o que se ganha na gama de valores representáveis perde-se na precisão desses mesmo valores.

Na maior parte dos processadores utilizados hoje em dia usa-se uma das representações especificadas na norma IEEE 754 [6] que, no caso de valores representados em 32 *bits* (chamados de precisão simples e correspondentes ao tipo *float* do C++),

1. atribui um *bit* (s_0) ao sinal (em que 0 significa positivo e 1 negativo),

2. atribui 23 *bits* (m_{-1} a m_{-23}) à mantissa, que são interpretados como um valor de vírgula fixa, sem sinal, com vírgula imediatamente antes do *bit* mais significativo, e

3. atribui oito *bits* (e_7 a e_0) ao expoente, que são interpretados como um inteiro entre 0 e 255 a que se subtrai 127, pelo que o expoente varia entre -126 e 127 (os valores 0 e 255 antes da subtracção são especiais, pelo que o expoente não pode tomar os valores -127 nem 128);

ou seja,

$$\boxed{s_0}\ 1,\boxed{m_{-1}}\boxed{\cdots}\boxed{m_{-23}}\ \boxed{e_7}\boxed{\cdots}\boxed{e_0}$$

Os valores representados desta forma são calculados como se indica na Tabela 2.3.

Tabela 2.3: Valor representado em formato de vírgula flutuante de precisão simples de acordo com IEEE 754 [6]. O sinal é dado por s_0.

Mantissa	Expoente	Valor representado
$m_{-1}\cdots m_{-23} = 0\cdots 0$	$e_7\cdots e_0 = 0\cdots 0$	± 0
$m_{-1}\cdots m_{-23} \neq 0\cdots 0$	$e_7\cdots e_0 = 0\cdots 0$	$\pm\,(0,m_{-1}\cdots m_{-23})_2 \times 2^{-126}$ (valores não-normalizados[18])
$m_{-1}\cdots m_{-23}$	$e_7\cdots e_0 \neq 0\cdots 0\ \wedge$ $e_7\cdots e_0 \neq 1\cdots 1$	$\pm\,(1,m_{-1}\cdots m_{-23})_2 \times 2^{(e_7\cdots e_0)_2-127}$ (valores normalizados[18])
$m_{-1}\cdots m_{-23} = 0\cdots 0$	$e_7\cdots e_0 = 1\cdots 1$	$\pm\infty$
$m_{-1}\cdots m_{-23} \neq 0\cdots 0$	$e_7\cdots e_0 = 1\cdots 1$	(valores especiais)

Quando os *bits* do expoente não são todos 0 nem todos 1 (i.e., $(e_7\cdots e_0)_2$ não é 0 nem 255), a mantissa tem um *bit* adicional à esquerda, implícito, com valor sempre 1. Nesse caso diz-se que os valores estão normalizados[18].

É fácil verificar que o menor valor positivo normalizado representável é

$$+\,(1{,}0\cdots 0)_2 \times 2^{(00000001)_2-127} = 2^{-126} = 1{,}17549435 \times 10^{-38}$$

e o maior é

$$+\,(1{,}1\cdots 1)_2 \times 2^{(11111110)_2-127} = \frac{(2^{24}-1)}{2^{23}} \times 2^{127} = 3{,}40282347 \times 10^{38}$$

[18]Os valores normalizados do formato IEEE 754 [6] têm sempre, portanto, o *bit* mais significativo, que é implícito, a 1. Os valores não-normalizados têm, naturalmente, menor precisão (dígitos significativos) que os normalizados, pois têm o *bit* mais significativo, que é implícito, com valor 0.

Comparem-se estes valores com os indicados para o tipo `float` na Tabela 2.1. Com esta representação conseguem-se cerca de seis dígitos decimais de precisão, menos quatro que a representação de vírgula fixa apresentada em primeiro lugar, mas uma gama bastante maior de valores representáveis. Os tipos `double` têm uma representação semelhante à apresentada, embora com maior precisão e gama, pois a sua representação é feita usando respectivamente 64 e 96 *bits* no total.

No que diz respeito à programação, a utilização de valores de vírgula flutuante deve ser evitada sempre que possível: se for possível usar inteiros nos cálculos é preferível usá-los a recorrer aos tipos de vírgula flutuante, que, apesar da sua aparência inocente, reservam muitas surpresas. Em particular, é importante recordar que os valores são representados com precisão finita, o que implica arredondamentos e acumulações de erros. Outra fonte comum de erros prende-se com a conversão de valores na base decimal para os formatos de vírgula flutuante em base binária: valores inocentes como 0,2 não são representáveis na base 2 usando um número finito de dígitos, pois são dízimos infinitos periódicos! Ao estudo da forma de lidar com estes tipos de erros sem surpresas desagradáveis dá-se o nome de análise numérica [3].

2.3.2 Booleanos ou lógicos

Existe um tipo `bool` cujas variáveis guardam valores lógicos. A forma de representação usual tem oito *bits* e reserva o padrão 00000000 para representar o valor falso ($\mathcal{F}$), representando todos os outros padrões o valor verdadeiro ($\mathcal{V}$). Os valores verdadeiro e falso são representados em C++ pelos identificadores reservados (ou *palavras-chave*) `true` e `false` (que são também *valores literais*, ver Secção 2.4). Os valores booleanos podem ser convertidos em inteiros. Nesse caso o valor falso é convertido em 0 e o valor verdadeiro é convertido em 1. Por exemplo,

```
int i = int(true);
```

inicializa a variável `i` com o valor inteiro 1.

2.3.3 Caracteres

Caracteres são símbolos representando letras, dígitos decimais, sinais de pontuação, etc. Cada caractere tem variadas representações gráficas, dependendo do seu tipo tipográfico. Por exemplo, a primeira letra do alfabeto em maiúscula tem as seguintes representações gráficas (entre muitas outras):

A A $\mathcal{A}$ A

É possível definir variáveis que guardam caracteres. O tipo `char` é usado em C++ para esse efeito. Em cada variável do tipo `char` é armazenado o código de um caractere. Esse código consiste num padrão de *bits*, correspondendo cada padrão de *bits* a um determinado caractere. Cada padrão de *bits* pode também ser interpretado como um número inteiro em binário, das formas que se viram atrás. Assim cada caractere é representado por um determinado valor inteiro, o seu código, correspondente a um determinado padrão de *bits*.

Em Linux sobre uma arquitectura Intel, as variáveis do tipo `char` são representadas em memória usando 8 *bits*, pelo que existem $2^8 = 256$ diferentes caracteres representáveis. Existem várias tabelas de codificação de caracteres diferentes, que a cada padrão de *bits* fazem corresponder um caractere diferente. De longe a mais usada neste canto da Europa é a tabela dos códigos Latin-9 (ou melhor, ISO-8859-15, ver Apêndice G), que é uma extensão da tabela ASCII (*American Standard Code for Information Interchange*) incluindo os caracteres acentuados em uso na Europa Ocidental[19]. Existem muitas destas extensões, que podem ser usadas de acordo com os caracteres que se pretendem representar, i.e., de acordo com o alfabeto da língua mais utilizada localmente. Essas extensões são possíveis porque o código ASCII fazia uso apenas dos 7 *bits* menos significativos de um caractere (i.e., apenas 128 das possíveis combinações de zeros e uns)[20].

Não é necessário, conhecer os códigos dos caracteres de cor: para indicar o código do caractere correspondente à letra 'b' usa-se o valor literal `'b'`. Claro está que o que fica guardado numa variável não é `'b'`, é um padrão de *bits* correspondente ao inteiro 98, pelo menos se se usar a tabela de codificação Latin-9. Esta tradução de uma representação habitual, `'b'`, para um código específico, 98, é feita pelo compilador e permite escrever programas sem quaisquer preocupações relativamente à tabela de codificação em uso.

Decorre naturalmente do que se disse que, em C++, é possível tratar um `char` como um pequeno número inteiro. Por exemplo, se se executar o conjunto de instruções seguinte:

```
char caractere = 'i';
caractere = caractere + 1;
cout << "O caractere seguinte é: " << caractere << endl;
```

aparece no ecrã

```
O caractere seguinte é: j
```

O que sucedeu foi que se adicionou 1 ao código do caractere 'i', de modo que a variável `caractere` passou a conter o código do caractere 'j'. Este pedaço de código só produz o efeito apresentado se, na tabela de codificação que está a ser usada, as letras sucessivas do alfabeto possuírem códigos sucessivos. Isso pode nem sempre acontecer. Por exemplo, na tabela EBCDIC *(Extended Binary Coded Decimal Interchange Code*), já pouco usada, o caractere 'i' tem

[19] A tabela de codificação Latin-1, ou ISO-8859-1, esteve em vigor na Europa Ocidental até à criação do euro. O suporte para o novo símbolo obrigou à mudança para a nova tabela ISO-8859-15.

[20] Existe um outro tipo de codificação, o Unicode, que suporta todos os caracteres de todas as expressões escritas vivas ou mortas em simultâneo. Mas exige uma codificação diferente, com maior número de *bits* (o C++ possui um outro tipo, `wchar_t`, para representar caracteres com estas características). No unicode existem duas formas de codificação. O UTF-16 propõe representações com 16 *bits*, extensíveis a 32 *bits*. O UTF-8 propõe representações onde os caracteres são representados por sequências de um, dois, três ou quatro *bytes*, sendo os caracteres da tabela ASCII representados com apenas oito *bits* e usando exactamente os mesmos padrões de *bits* do ASCII. É espectável que a prazo o Unicode seja o único tipo de codificação em uso.

código 137 e o caractere 'j' tem código 145! Num sistema que usasse esse código, as instruções acima certamente não escreveriam a letra 'j' no ecrã.

Os `char` são interpretados como inteiros em C++. Estranhamente, se esses inteiros têm ou não sinal não é especificado pela linguagem. Quer em Linux sobre arquitecturas Intel quer no Windows NT, os `char` são interpretados como inteiros com sinal (ou seja, char≡`signed char`), com uma gama de valores que varia entre -128 e 127, devendo-se usar o tipo `unsigned char` se se pretender forçar uma interpretação dos caracteres como inteiros sem sinal, i.e., com uma gama de valores variando entre 0 e 255.

O programa seguinte imprime no ecrã todos os caracteres da tabela ASCII (que só especifica os caracteres correspondentes aos códigos de 0 a 127, isto é, todos os valores positivos dos `char` em Linux e a primeira metade da tabela Latin-9)[21]:

```
#include <iostream>

using namespace std;

int main()
{
    for(int i = 0; i != 128; ++i)
        cout << "'" << char(i) << "' (" << i << ")" << endl;
}
```

Quando é realizada uma operação de extracção de um canal para uma variável do tipo `char`, é lido apenas um caractere, mesmo que no teclado seja inserido um código (ou mais do que um caractere), e são descartados os espaços em branco. I.e., o resultado das instruções

```
cout << "Insira um caractere: ";
char caractere;
cin >> caractere;
cout << "O caractere inserido foi: '" << caractere << "'" << endl;
```

seria

```
Insira um caractere:␣␣␣␣␣48
O caractere inserido foi: '4'
```

caso o utilizador inserisse 48 (após uns quantos espaços). O dígito oito foi ignorado visto que se leu apenas um caractere, e não o seu código. Para ler qualquer caractere, incluindo os espaços em branco, deve-se usar a operação `get()` com o canal `cin`. Por exemplo, o resultado das instruções

[21] Alguns dos caracteres escritos são especiais, representando mudanças de linha, etc. Por isso, o resultado de uma impressão no ecrã de todos os caracteres da tabela de codificação ASCII pode ter alguns efeitos estranhos.

```
cout << "Insira um caractere: ";
char caractere;
cin.get(caractere);
cout << "O caractere inserido foi: '" << caractere << "'" << endl;
```

seria

```
Insira um caractere:␣␣␣␣␣48
O caractere inserido foi: ' '
```

admitindo que o utilizador digitava a mesma sequência de caracteres que anteriormente.

2.4 Valores literais

Os valores literais permitem indicar explicitamente num programa valores dos tipos básicos do C++. Alguns exemplos de valores literais (repare-se bem na utilização dos sufixos `U`, `L` e `F`) são:

```
false    // do tipo bool, representa o valor lógico F.
'a'      // do tipo char, representa o código do caractere 'a'.
100      // do tipo int, valor 100 (em decimal).
100U     // do tipo unsigned.
100L     // do tipo long.
100.0    // do tipo double.
100.0F   // do tipo float (e não double).
100.0L   // do tipo long double.
1.1e230  // do tipo double, valor 1,1 × 10^230, usa a chamada notação científica.
```

Os valores inteiros podem ainda ser especificados em octal (base 8) ou hexadecimal (base 16). Inteiros precedidos de `0x` são considerados como representados na base hexadecimal e portanto podem incluir, para além dos 10 dígitos usuais, as letras entre A a F (com valores de 10 a 15), em maiúsculas ou em minúsculas. Inteiros precedidos de 0 simplesmente são considerados como representados na base octal e portanto podem incluir apenas dígitos entre 0 e 7. Por exemplo[22]:

```
0x1U   // o mesmo que 1U.
0x10FU // o mesmo que 271U, ou seja (00000000000000000000000100001111)₂.
077U   // o mesmo que 63U, ou seja (00000000000000000000000000111111)₂.
```

Em alguns casos a utilização de valores literais nas instruções de um programa são um mau sinal. Isso ocorre quando esses valores literais representam valores importantes do ponto de

[22]Os exemplos assumem que os `int` têm 32 *bits*.

vista do problema a resolver, que têm uma semântica própria, que ocorrem em várias instruções do programa, e que, por isso, seriam melhor representados recorrendo a constantes (ver Secção 2.5).

Para além dos valores literais para os tipos básicos do C++ apresentados acima, há um tipo adicional de valores literais: as cadeias de caracteres. Correspondem a sequências de caracteres colocados entre `""`. Para já não se dirá qual o tipo destes valores literais, bastando saber-se que se podem usar para escrever texto no ecrã (ver Secção 2.1.1 para mais informação sobre canais e Secção 5.4 para informação acerca de cadeias de caracteres).

Há alguns caracteres que são especiais. O seu objectivo não é serem representados por um determinado símbolo, como se passa com as letras e dígitos, por exemplo, mas sim controlar a forma como o texto está organizado. Normalmente considera-se um texto como uma sequência de caracteres, dos quais fazem parte caracteres especiais que indicam o final das linhas, os tabuladores, ou mesmo se deve soar uma campainha quando o texto for mostrado. Os caracteres de controlo não podem ser especificados directamente como valores literais, uma vez que não têm representação gráfica. Para os representar usam-se as chamadas *sequências de escape*, que correspondem a sequências de caracteres com significado especial e que são iniciadas por um caractere chamado de escape. Quando esse caractere ocorre num valor literal do tipo `char` (ou numa cadeia de caracteres), indica que se deve *escapar* da interpretação normal destes valores literais e passar a uma interpretação especial. Esse caractere é a barra para trás ('\') e serve para construir as sequências de escape indicadas na Tabela 2.4.

2.5 Constantes

Nos programas em C++ também se pode definir constantes, i.e., "variáveis" que não mudam de valor durante toda a sua existência. Nas constantes o valor é uma característica estática e não dinâmica, como no caso das variáveis. A definição de constantes faz-se colocando a palavra-chave[23] `const` após o tipo pretendido para a constante[24]. As constantes, justamente por o serem, têm obrigatoriamente de ser inicializadas no acto da definição. No caso das constantes de tipos básicos, isso significa que o seu valor estático tem de ser indicado na própria definição. Por exemplo:

```
int const primeiro_primo = 2;
char const primeira_letra_do_alfabeto_latino = 'a';
```

A constante `primeiro_primo` é representada pela notação apresentada na Figura 2.5.

As constantes devem ser usadas como alternativa aos valores literais quando estes tiverem uma semântica (um significado) particular. O nome dado à constante deve reflectir exactamente esse significado. Por exemplo, em vez de

[23]Palavras-chave são palavras cujo significado está pré-determinado pela própria linguagem e portanto que não podem ser usadas para outros fins. Ver Apêndice E.

[24]A palavra-chave `const` também pode ser colocada antes do tipo. Porém, embora essa prática seja corrente, não é recomendável dadas as confusões a que induz quando aplicada a ponteiros. Ver !!.

Tabela 2.4: Sequências de escape. A negrito as sequências mais importantes.

Sequência	**Nome**	**Significado**
`\n`	fim-de-linha	Assinala o fim de uma linha.
`\t`	tabulador	Salta para a próxima posição de tabulação.
`\v`	tabulador vertical	
`\b`	espaço para trás	
`\r`	retorno do carreto	
`\f`	nova página	
`\a`	campainha	
`\\`	caractere '\'	O caractere '\' serve de escape, num valor literal tem de se escrever `'\\'`.
`\?`	caractere '?'	Evita a sequência `??` que tem um significado especial.
`\'`	caractere '''	Valor literal correspondente ao caractere plica é `'\''`.
`\"`	caractere '"'	Dentro duma cadeia de caracteres o caractere '"' escreve-se `"\""`.
`\`*`ooo`*	caractere com código *`ooo`* em octal.	Pouco recomendável: o código dos caracteres muda com a tabela de codificação.
`\x`*`hhh`*	caractere com código *`hhh`* em hexadecimal.	Pouco recomendável: o código dos caracteres muda com a tabela de codificação.

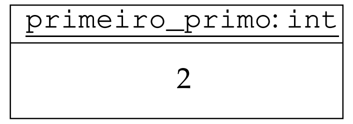


Figura 2.5: Notação usada para as constantes.

```
double raio = 3.14;
cout << "O perímetro é " << 2.0 * 3.14 * raio << endl;
cout << "A área é " << 3.14 * raio * raio << endl;
```

é preferível[25]

```
double const pi = 3.14;
double raio = 3.14;
cout << "O perímetro é " << 2 * pi * raio << endl;
cout << "A área é " << pi * raio * raio << endl;
```

Há várias razões para ser preferível a utilização de constantes no código acima:

1. A constante tem um nome apropriado (com um significado claro), o que torna o código C++ mais legível.

2. Se se pretender aumentar a precisão do valor usado para π, basta alterar a inicialização da constante:

   ```
   double const pi = 3.1415927;
   double raio = 3.14;
   cout << "O perímetro é " << 2 * pi * raio << endl;
   cout << "A área é " << pi * raio * raio << endl;
   ```

3. Se, no código original, se pretendesse alterar o valor inicial do raio para 10,5, por exemplo, seria natural a tentação de recorrer à facilidade de substituição de texto do editor. Uma pequena distracção seria suficiente para substituir por 10,5 também a aproximação de π, cujo valor original era, por coincidência, igual ao do raio. O resultado seria desastroso:

   ```
   double raio = 10.5;
   // Erro! Substituição desastrosa:
   cout << "O perímetro é " << 2 * 10.5 * raio << endl;
   // Erro! Substituição desastrosa:
   cout << "A área é " << 10.5 * raio * raio << endl;
   ```

Aos valores literais usados directamente no código, sem recurso a constantes que lhes atribuam uma semântica clara, chama-se muitas vezes *numeros mágicos*. A programação é simultaneamente uma arte e uma ciência, pelo que o recurso à magia deve-se pôr de parte: *nunca se devem usar números mágicos num programa*!

[25] A subtileza neste caso é que o valor literal `3.14` tem dois significados completamente diferentes! É o valor do raio mas também é uma aproximação de π!

2.6 Instâncias

A uma variável ou constante de um dado tipo é usual chamar-se, no jargão da programação orientada para os objectos, uma *instância* desse tipo. Esta palavra, já existindo em português, foi importada do inglês recentemente com o novo significado de "exemplo concreto" ou "exemplar" (cf. a expressão "*for instance*", com o significado de "por exemplo"). Assim, uma variável ou constante de um tipo é uma instância ou exemplar dessa classe, tal como uma mulher ou um homem são instâncias dos humanos. Instância é, por isso, um nome que se pode dar quer a variáveis quer a constantes.

2.7 Expressões e operadores

O tipo de instrução mais simples, logo após a instrução de definição de variáveis, consiste numa expressão terminada por ponto-e-vírgula. Assim, as próximas são instruções válidas, se bem que inúteis:

```
; // expressão nula (instrução nula).
2;
1 + 2 * 3;
```

Para que os programas tenham algum interesse, é necessário que "algo mude" à medida que são executados, i.e., que o estado da memória (ou de dispositivos associados ao computador, como o ecrã, uma impressora, etc.) seja alterado. Isso consegue-se, por exemplo, alterando os valores das variáveis através do operador de atribuição. As próximas instruções são potencialmente mais úteis, pois agem sobre o valor de uma variável:

```
int i = 0;

// Atribuições:
i = 2;
i = 1 + 2 * 3;
```

Uma expressão é composta por valores (valores literais, valores de instâncias, etc) e operações. Muitas vezes numa expressão existe um operador de atribuição = ou suas variantes[26] , ver Secção 2.7.5. A expressão

```
a = b + 3;
```

significa "atribua-se à variável `a` o resultado da soma do valor da instância `b` com o valor literal inteiro (do tipo `int`) 3".

A expressão na última instrução de

[26]Excepto quando a expressão for argumento de alguma função ou quando controlar alguma instrução de selecção ou iteração, como se verá mais tarde.

```
int i, j;
bool são_iguais;
...
são_iguais = i == j;
```

significa "atribua-se à variável `são_iguais` o valor lógico (booleano, do tipo `bool`) da afirmação 'os valores das instâncias `i` e `j` são iguais"'. Depois desta operação o valor de `são_iguais` é verdadeiro se `i` for igual a `j` e falso no caso contrário.

Os operadores usuais em C++ são de um de três tipos: unários, se tiverem apenas um operando, binários, se tiverem dois operandos, e ternários, se tiverem três operandos. Existem ainda outros tipos de operadores com uma *aridade* (número de operandos) maior, como é o caso da invocação de rotinas, como se verá mais tarde.

2.7.1 Operadores aritméticos

Os operadores aritméticos são

+ adição (binário), e.g., `2.3 + 6.7`;

+ identidade (unário), e.g., `+5.5`;

- subtracção (binário), e.g., `10 - 4`;

- simétrico (unário), e.g., `-3`;

***** multiplicação (binário), e.g., `4 * 5`;

/ divisão (binário), e.g., `10.5 / 3.0`; e

% resto da divisão inteira (binário), e.g., `7 % 3`.

As operações aritméticas preservam os tipos dos operandos, i.e., a soma de dois `float` resulta num valor do tipo `float`, etc.

O significado do operador divisão depende do tipo dos operandos usados. Por exemplo, o resultado de `10 / 20` é 0 (zero), e não 0,5. I.e., se os operandos da divisão forem de algum dos tipos aritméticos inteiros, então a divisão usada é a divisão inteira, que "não usa casas decimais". Por outro lado, o operador `%` só pode ser usado com operandos de um tipo inteiro, pois a divisão inteira só faz sentido nesse caso. Os operadores de divisão e resto da divisão inteira estão especificados de tal forma que, quaisquer que sejam dois valores inteiros `a` e `b`, se verifique a condição `a == (a / b) * b + a % b`.

É possível, embora não recomendável, que os operandos de um operador sejam de tipos aritméticos diferentes. Nesse caso é feita a conversão automática do operando com tipo menos "abrangente" para o tipo do operando mais "abrangente". Por exemplo, o código

```
double const pi = 3.1415927;
double x = 1 + pi;
```

leva o compilador a converter automaticamente o valor literal `1` do tipo `int` para o tipo `double`. Esta é uma das chamadas conversões aritméticas usuais, que se processam como se segue:

- Se algum operando for do tipo `long double`, o outro operando é convertido para `long double`;
- caso contrário, se algum operando for do tipo `double`, o outro operando é convertido para `double`;
- caso contrário, se algum operando for do tipo `float`, o outro operando é convertido para `float`;
- caso contrário, todos os operandos do tipo `char`, `signed char`, `unsigned char`, `short int` ou `unsigned short int` são convertidos para `int`, se um `int` puder representar todos os possíveis valores do tipo de origem, ou para um `unsigned int` no caso contrário (a estas conversões chama-se "promoções inteiras");
- depois, se algum operando for do tipo `unsigned long int`, o outro operando é convertido para `unsigned long int`;
- caso contrário, se um dos operandos for do tipo `long int` e o outro `unsigned int`, então se um `long int` puder representar todos os possíveis valores de um `unsigned int`, o `unsigned int` é convertido para `long int`, caso contrário ambos os operandos são convertidos para `unsigned long int`;
- caso contrário, se algum dos operandos for do tipo `long int`, o outro operando é convertido para `long int`; e
- caso contrário, se algum dos operandos for do tipo `unsigned int`, o outro operando é convertido para `unsigned int`.

Estas regras, apesar de se ter apresentado apenas uma versão resumida, são complexas e pouco intuitivas. Além disso, algumas destas conversões podem resultar em perda de precisão (e.g., converter um `long int` num `float` pode resultar em erros de arredondamento, se o `long int` for suficientemente grande). É preferível, portanto, evitar as conversões tanto quanto possível. Assim, o código acima deveria ser reescrito como

```
double const pi = 3.1415927;
double x = 1.0 + pi;
```

de modo a que o valor literal fosse do mesmo tipo que a constante `pi`. Se não se tratar de um valor literal mas sim de uma instância, então é preferível converter explicitamente um ou ambos os operandos para compatibilizar os seus tipos

```
double const pi = 3.1415927;
int i = 1;
double x = double(i) + pi; // as conversões têm o formato
                           // tipo(expressão).
```

pois fica muito claro que o programador está consciente da conversão e, presume-se, das suas consequências.

Em qualquer dos casos é sempre boa ideia repensar o código para perceber se as conversões são mesmo necessárias, pois há algumas conversões que podem introduzir erros de aproximação indesejáveis e, em alguns casos, mesmo desastrosos.

2.7.2 Operadores relacionais e de igualdade

Os operadores relacionais (todos binários) são

`>` maior,

`<` menor,

`>=` maior ou igual, e

`<=` menor ou igual,

com os significados óbvios.

Para comparar a igualdade ou diferença de dois operandos usam-se os operadores de igualdade

`==` igual a, e

`!=` diferente de.

Quer os operadores relacionais, quer os de igualdade, têm como resultado não um valor aritmético mas sim um valor lógico, do tipo `bool`, sendo usadas comummente para controlar instruções de selecção e iterativas, que serão estudadas em pormenor mais tarde. Por exemplo:

```
if(m < n)
    k = m;
else
    k = n;
```

ou

```
while(n % k != 0 or m % k != 0)
    --k;
```

Os tipos dos operandos destes operadores são compatibilizados usando as conversões aritméticas usuais apresentadas atrás.

É muito importante não confundir o operador de igualdade com o operador de atribuição. Em C++ a atribuição é representada por = e a verificação de igualdade por ==.

2.7.3 Operadores lógicos

Os operadores lógicos (ou booleanos) aplicam-se a operandos lógicos e são[27]

`and` conjunção ou “e” (binário), também se pode escrever `&&`;

`or` disjunção ou “ou” (binário), também se pode escrever `||`; e

`not` negação ou “não” (unário), também se pode escrever `!`.

Por exemplo:

```
a > 5              // verdadeira se a for maior que 5.
not (a > 5)        // verdadeira se a não for maior que 5.
a < 5 and b <= 7   // verdadeira se a for menor que 5 e b for menor ou igual a 7.
a < 5 or b <= 7    // verdadeira se a for menor que 5 ou b for menor ou igual a 7.
```

Estes operadores podem operar sobre operandos booleanos, mas também sobre operandos aritméticos. Neste último caso, os operandos aritméticos são convertidos para valores lógicos, correspondendo o valor zero a $\mathcal{F}$ e qualquer outro valor diferente de zero a $\mathcal{V}$. É sempre má ideia recorrer a estas interpretações de valores aritméticos como se de valores booleanos se tratasse.

Esta característica da linguagem C++, como tantas outras, é apresentada aqui não por ser uma boa característica, a explorar, mas exactamente por ser uma má característica, que ocasionalmente dá azo a erros de difícil detecção.[28]

---

[27] Fez-se todo o esforço neste texto para usar a versão mais intuitiva destes operadores, uma introdução relativamente recente na linguagem C++. No entanto, o hábito de anos prega rasteiras, pelo que o leitor poderá encontrar ocasionalmente um `&&` ou outro...

Nem todos os compiladores aceitam as versões mais recentes dos operadores lógicos. Quando isso não acontecer, e para o programador não ser forçado a usar os velhos e desagradáveis símbolos, recomenda-se que coloque no início dos seus programas as directivas de pré-processamento seguintes (Secção 9.2.1):

```
#define and &&
#define or ||
#define not !
#define bitand &
#define bitor |
#define xor ^
#define compl ~
```

[28] Ou seja, não se deve nunca usar o valor de uma expressão aritmética como se de um valor booleano se tratasse. É infelizmente comum encontrar-se código como

```
int i = ...;
...
if(i)
    ...
```

onde se poderia usar o código

```
int i = ...;
```

A ordem de cálculo dos operandos de um operador não é, em geral, especificada. Os dois operadores binários `and` e `or` são uma das excepções. Os seus operandos (que podem ser sub-expressões) são calculados da esquerda para a direita, sendo o cálculo atalhado logo que o resultado seja conhecido. Se o primeiro operando de um `and` é $\mathcal{F}$, o resultado é $\mathcal{F}$, se o primeiro operando de um `or` é $\mathcal{V}$, o resultado é $\mathcal{V}$, e em ambos os casos o segundo operando não chega a ser calculado. Esta característica será de grande utilidade no controlo de fluxo dos programas (Capítulo 4).

Para os mais esquecidos apresentam-se as tabelas de verdade das operações lógicas na Figura 2.6.

$$\boxed{\begin{array}{rcl} \mathcal{F}\wedge\mathcal{F} & = & \mathcal{F} \\ \mathcal{F}\wedge\mathcal{V} & = & \mathcal{F} \\ \mathcal{V}\wedge\mathcal{F} & = & \mathcal{F} \\ \mathcal{V}\wedge\mathcal{V} & = & \mathcal{V} \end{array}} \quad \boxed{\begin{array}{rcl} \mathcal{F}\vee\mathcal{F} & = & \mathcal{F} \\ \mathcal{F}\vee\mathcal{V} & = & \mathcal{V} \\ \mathcal{V}\vee\mathcal{F} & = & \mathcal{V} \\ \mathcal{V}\vee\mathcal{V} & = & \mathcal{V} \end{array}} \quad \boxed{\begin{array}{rcl} \mathcal{F}\oplus\mathcal{F} & = & \mathcal{F} \\ \mathcal{F}\oplus\mathcal{V} & = & \mathcal{V} \\ \mathcal{V}\oplus\mathcal{F} & = & \mathcal{V} \\ \mathcal{V}\oplus\mathcal{V} & = & \mathcal{F} \end{array}}$$

$$\boxed{\begin{array}{rcl} \neg\mathcal{F} & = & \mathcal{V} \\ \neg\mathcal{V} & = & \mathcal{F} \end{array}}$$

$\mathcal{F}$	falso
$\mathcal{V}$	verdadeiro
$\wedge$	conjunção
$\vee$	disjunção
$\oplus$	ou exclusivo
$\neg$	negação

Figura 2.6: Tabelas de verdade das operações lógicas elementares.

2.7.4 Operadores *bit*-a-*bit*

Há alguns operadores do C++ que permitem fazer manipulações de valores de muito baixo nível, ao nível do *bit*. São chamadas operações *bit*-a-*bit* e apenas admitem operandos de tipos básicos inteiros. Muito embora estejam definidos para tipos inteiros com sinal, alguns têm resultados não especificados quando os operandos são negativos. Assim, assume-se aqui que os operandos são de tipos inteiros sem sinal, ou pelo menos que são garantidamente positivos.

Estes operadores pressupõem naturalmente que se conhecem as representações dos tipos básicos na memória. Isso normalmente é contraditório com a programação de alto nível. Por isso, estes operadores devem ser usados apenas onde for estritamente necessário.

Os operadores são

`bitand` conjunção *bit*-a-*bit* (binário), também se pode escrever `&`;

```
...
if(i != 0)
    ...
```

que é muito mais claro e directo.

`bitor` disjunção *bit*-a-*bit* (binário), também se pode escrever $|$;

`xor` disjunção exclusiva *bit*-a-*bit* (binário), também se pode escrever `^`;

`compl` negação *bit*-a-*bit (unário)*, também se pode escrever `~`;

`<<` deslocamento para a esquerda (binário); e

`>>` deslocamento para a direita (binário).

Estes operadores actuam sobre os *bits* individualmente. Admitindo, para simplificar, que o tipo `unsigned int` é representado com apenas 16 *bits*,

```
123U bitand 0xFU == 11U e
123U bitand 0xFU == 0xBU,
```

pois sendo $123 = (0000000001111011)_2$ e $(F)_{16} = (0000000000001111)_2$, a conjunção calculada para pares de *bits* correspondentes na representação de `123U` e `0xFU` resulta em

$$\begin{array}{cccccccccccccccc}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 1 & 1 \\
\wedge & \wedge & \wedge & \wedge & \wedge & \wedge & \wedge & \wedge & \wedge & \wedge & \wedge & \wedge & \wedge & \wedge & \wedge & \wedge \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1
\end{array}$$

`ou` seja, $(0000000000001011)_2 = 11 = (\mathrm{B})_{16}$.

As operações de deslocamento deslocam o padrão de *bits* correspondente ao valor do primeiro operando de tantas posições quanto o valor do segundo operando, inserindo zeros (0) à direita, quando o deslocamento é para a esquerda, e à esquerda, quando o deslocamento é para a direita, e eliminando os *bits* do extremo oposto. Por exemplo, admitindo de novo representações de 16 *bits*:

```
1U << 4U == 16U
52U >> 4U == 3U
```

Pois $1 = (0000000000000001)_2$ deslocado para a esquerda de quatro dígitos binários resulta em $(0000000000010000)_2 = 16$ e $52 = (0000000000110100)_2$ deslocado para a direita de quatro posições resulta em $(0000000000000011)_2 = 3$. O deslocamento para a esquerda de n *bits* corresponde à multiplicação do inteiro por 2^n e o deslocamento de n *bits* para a direita corresponde à divisão inteira por 2^n.

Os operadores << e >> têm significados muito diferentes quando o seu primeiro operando é um canal, como se viu na Secção 2.1.1.

2.7.5 Operadores de atribuição

A operação de atribuição, indicada pelo símbolo =, faz com que a variável que está à esquerda do operador (o primeiro operando) passe a tomar o valor do segundo operando. Uma consequência importante é que, ao contrário do que acontece quando os operadores que se viu até aqui são calculados, o estado do programa é afectado pelo cálculo de uma operação de atribuição: há uma variável que muda de valor. Por isso se diz que a atribuição é uma operação com *efeitos laterais*. Por exemplo[29]:

```
a = 3 + 5; // a toma o valor 8.
```

A atribuição é uma operação, e não uma instrução, ao contrário do que se passa noutras linguagens (como o Pascal). Isto significa que a operação tem um resultado, que é o valor que ficou guardado no seu operando esquerdo. Este facto, conjugado com a associatividade à direita deste tipo de operadores (ver Secção 2.7.7), permite escrever

```
a = b = c = 1;
```

para, numa única instrução, atribuir 1 às três variáveis `a`, `b` e `c`.

Ao contrário do que acontece, por exemplo, com os operadores aritméticos e relacionais, quando os operandos de uma atribuição são de tipos diferentes, é sempre o segundo operando que é convertido para se adaptar ao tipo do primeiro operando. Assim, o resultado de

```
int i;
float f;
f = i = 1.9f;
```

é que

1. a variável `i` fica com o valor 1, pois a conversão de `float` para `int` elimina a parte fraccionária (i.e., *não arredonda, trunca*), e
2. a variável `f` fica com o valor 1,0, pois é o resultado da conversão do valor 1 para `float`, sendo 1 o resultado da atribuição a `i`[30].

Por outro lado, do lado esquerdo de uma atribuição, i.e., como primeiro operando, tem de estar uma variável ou, mais formalmente, um chamado *lvalue* (de *left value*), ou seja, algo a que se possa mudar o valor. Deve ser claro que não faz qualquer sentido colocar uma constante ou um valor literal do lado esquerdo de uma atribuição:

[29]Mais uma vez são de notar os significados distintos dos operadores = (atribuição) e == (comparação, igualdade). É frequente o programador confundir estes dois operadores, levando a erros de programação muito difíceis de detectar.

[30]Recorda-se que o resultado de uma atribuição é o valor que fica na variável a que se atribui o valor.

```
double const pi = 3.1415927;
pi = 4.5;  // Absurdo! Não faz sentido alterar o que é constante!
4.6 = 1.5; // Pior ainda! Que significado poderia ter semelhante instrução?
```

Existem vários outros operadores de atribuição em C++ que são formas abreviadas de escrever expressões comuns. Assim, `i += 4` tem (quase) o mesmo significado que `i = i + 4`. Todos eles têm o mesmo formato: *op*= em que *op* é uma das operações binárias já vistas (mas nem todas têm o correspondente operador de atribuição, ver Apêndice F). Por exemplo:

```
i = i + n; // ou i += n;
i = i - n; // ou i -= n;
i = i * n; // ou i *= n;
i = i % n; // ou i %= n;
i = i / n; // ou i /= n;
```

2.7.6 Operadores de incrementação e decrementação

As expressões da forma `i += 1` e `i -= 1`, por serem tão frequentes, merecem também uma forma especial de abreviação em C++: os operadores de incrementação e decrementação ++ e `--`. Estes dois operadores têm, por sua vez, duas versões: a versão prefixo e a versão sufixo. Quando o objectivo é simplesmente incrementar ou decrementar uma variável, as duas versões podem ser consideradas equivalentes, embora a versão prefixo deva em geral ser preferida[31]:

```
i += 1; // ou ++i; (preferível)
        // ou i++;
```

Porém, se o resultado da operação for usado numa expressão envolvente, as versões prefixo e sufixo têm resultados muito diferentes: o valor da expressão `i++` é o valor de `i` *antes de incrementado*, ou seja, `i` é incrementado depois de o seu valor ser extraído como resultado da operação, enquanto o valor da expressão `++i` é o valor de `i` *depois de incrementado*, ou seja, `i` é incrementado antes de o seu valor ser extraído como resultado da operação. Assim,

```
int i = 0;
int j = i++;
cout << i << ' ' << j << endl;
```

escreve no ecrã os valores 1 e 0, enquanto

```
int i = 0;
int j = ++i;
cout << i << ' ' << j << endl;
```

escreve no ecrã os valores 1 e 1.

As mesmas observações aplicam-se às duas versões do operador de decrementação.

[31] As razões para esta preferência ficarão claras quando na Secção 7.7.1.

2.7.7 Precedência e associatividade

Qual o resultado da expressão `4 * 3 + 2`? 14 ou 20? Qual o resultado da expressão `8 / 4 / 2`? 1 ou 4? Para que estas expressões não sejam ambíguas, o C++ estabelece um conjunto de regras de precedência e associatividade para os vários operadores possíveis. A Tabela 2.5 lista os operadores já vistos por ordem decrescente de precedência (ver uma tabela completa no Apêndice F). Quanto à associatividade, apenas os operadores unários (com um único operando) e os operadores de atribuição se associam à direita: todos os outros associam-se à esquerda, como é habitual. Para alterar a ordem de cálculo dos operadores numa expressão podem-se usar parênteses. Os parênteses tanto servem para evitar a precedência normal dos operadores, e.g., `x = (y + z) * w`, como para evitar a associatividade normal de operações com a mesma precedência, e.g., `x * (y / z)`.

A propósito do último exemplo, os resultados de `4 * 5 / 6` e `4 * (5 / 6)` são diferentes! O primeiro é 3 e o segundo é 0! O mesmo se passa com valores de vírgula flutuante, devido aos erros de arredondamento. Por exemplo, o seguinte troço de programa

```
// Para os resultados serem mostrados com 20 dígitos (usar #include <iomanip>):
cout << setprecision(20);
cout << 0.3f * 0.7f / 0.001f << ' ' << 0.3f * (0.7f / 0.001f)
     << endl;
```

escreve no ecrã (em máquinas usando o formato IEEE 754 para os `float`)

```
210 209.9999847412109375
```

onde se pode ver claramente que os arredondamentos afectam de forma diferente duas expressões que, do ponto de vista matemático, deveriam ter o mesmo valor.

2.7.8 Efeitos laterais e mau comportamento

Chamam-se expressões sem efeitos laterais as expressões cujo cálculo não afecta o valor de nenhuma variável. O C++, ao classificar as várias formas de fazer atribuições como meros operadores, dificulta a distinção entre instruções com efeitos laterais e instruções sem efeitos laterais. Para simplificar, classificar-se-ão como instruções de atribuição as instruções que consistam numa operação de atribuição, numa instrução de atribuição compacta (*op*=) ou numa simples incrementação ou decrementação. Se a expressão do lado direito da atribuição não tiver efeitos laterais, a instrução de atribuição não tem efeitos laterais e vice-versa. Pressupõe-se, naturalmente, que uma instrução de atribuição tem como efeito principal (não lateral) atribuir o valor da expressão do lado direito à entidade (normalmente uma variável) do lado esquerdo.

Uma instrução diz-se mal comportada se o seu resultado não estiver definido. As instruções sem efeitos laterais são sempre bem comportadas. As instruções com efeitos laterais são bem comportadas se puderem ser decompostas numa sequência de instruções sem efeitos laterais.

Assim:

Tabela 2.5: Precedência e associatividade de alguns dos operadores do C++. Operadores colocados na mesma célula da tabela têm a mesma precedência. As células são apresentadas por ordem decrescente de precedência. Apenas os operadores unários (com um único operando) e os operadores de atribuição se associam à direita: todos os outros associam-se à esquerda.

Descrição	**Sintaxe** (itálico: partes variáveis da sintaxe)
construção de valor	*tipo* (*lista_expressões*)
incrementação sufixa	*lvalue* ++
decrementação sufixa	*lvalue* --
incrementação prefixa	++ *lvalue*
decrementação prefixa	-- *lvalue*
negação *bit*-a-*bit* ou complemento para um	compl *expressão* (ou ~)
negação	not *expressão*
simétrico	- *expressão*
identidade	+ *expressão*
endereço de	& *lvalue*
conteúdo de	* *expressão*
multiplicação	*expressão* * *expressão*
divisão	*expressão* / *expressão*
resto da divisão inteira	*expressão* % *expressão*
adição	*expressão* + *expressão*
subtracção	*expressão* - *expressão*
deslocamento para a esquerda	*expressão* << *expressão*
deslocamento para a direita	*expressão* >> *expressão*
menor	*expressão* < *expressão*
menor ou igual	*expressão* <= *expressão*
maior	*expressão* > *expressão*
maior ou igual	*expressão* >= *expressão*
igual	*expressão* == *expressão*
diferente	*expressão* != *expressão* (ou not_eq)
conjunção *bit*-a-*bit*	*expressão* bitand *expressão* (ou &)
disjunção exclusiva *bit*-a-*bit*	*expressão* xor *expressão* (ou ^)
disjunção *bit*-a-*bit*	*expressão* bitor *expressão* (ou |)
conjunção	*expressão* and *expressão* (ou &&)
disjunção	*expressão* or *expressão* (ou ||)
atribuição simples	*lvalue* = *expressão*
multiplicação e atribuição	*lvalue* *= *expressão*
divisão e atribuição	*lvalue* /= *expressão*
resto e atribuição	*lvalue* %= *expressão*
adição e atribuição	*lvalue* += *expressão*
subtracção e atribuição	*lvalue* -= *expressão*
deslocamento para a esquerda e atribuição	*lvalue* <<= *expressão*
deslocamento para a direita e atribuição	*lvalue* >>= *expressão*
conjunção *bit*-a-*bit* e atribuição	*lvalue* &= *expressão* (ou and_eq)
disjunção *bit*-a-*bit* e atribuição	*lvalue* |= *expressão* (ou or_eq)
disjunção exclusiva *bit*-a-*bit* e atribuição	*lvalue* ^= *expressão* (ou xor_eq)

`x = y = z = 1;` Instrução de atribuição com efeitos laterais, pois a expressão `y = z = 1` tem efeitos laterais (altera `y` e `z`), mas bem comportada, pois pode ser transformada numa sequência de instruções sem efeitos laterais:

```
z = 1;
y = z;
x = y;
```

`x = y + (y = z = 1);` Instrução com efeitos laterais e mal comportada. Esta instrução não tem remissão. Está simplesmente errada. Ver mais abaixo discussão de caso semelhante.

`++x;` Instrução sem efeitos laterais, equivalente a `x = x + 1`.

`if(x == 0) ...` Sem efeitos laterais, pois a expressão não altera qualquer variável.

`if(x++ == 0) ...` Com efeitos laterais, pois `x` muda de valor, mas bem comportada. Se `x` for uma variável inteira, pode ser transformada numa sequência de instruções sem efeitos laterais:

```
x = x + 1;
if(x == 1)
    ...
```

`while(cin >> x) ...` Com efeitos laterais mas bem comportada. Pode ser decomposta em:

```
cin >> x;
while(cin.good()) {
    ...
    cin >> x;
}
```

2.7.9 Ordem de cálculo

A ordem de cálculo dos operandos é indefinida para a maior parte dos operadores. As excepções são as seguintes:

`and` O operando esquerdo é calculado primeiro. Se for $\mathcal{F}$, o resultado é $\mathcal{F}$ e o segundo operando não chega a ser calculado.

`or` O operando esquerdo é calculado primeiro. Se for V, o resultado é $\mathcal{V}$ e o segundo operando não chega a ser calculado.

`,` O primeiro operando é sempre calculado primeiro.

`? :` O primeiro operando é sempre calculado primeiro. O seu valor determina qual dos dois restantes operandos será também calculado, ficando sempre um deles por calcular.

Para os restantes operadores a ordem de cálculo dos operandos de um operador é indefinida. Assim, na expressão:

```
y = sin(x) + cos(x) + sqrt(x)
```

não se garante que `sin(x)` (seno de `x`) seja calculada em primeiro lugar e `sqrt(x)` (raiz quadrada de `x`) em último[32]. Se uma expressão não envolver operações com efeitos laterais, esta indefinição não afecta o programador. Quando a expressão tem efeitos laterais que afectam variáveis usadas noutros locais da mesma expressão, esta pode deixar de ter resultados bem definidos devido à indefinição quanto é ordem de cálculo. Nesse caso a expressão é mal comportada.

Por exemplo, depois das instruções

```
int i = 0;
int j = i + i++;
```

o valor de `j` pode ser 0 ou 1, consoante o operando `i` seja calculado antes ou depois do operando `i++`. No fundo o problema é que é impossível saber *a priori* se a segunda instrução pode ser decomposta em

```
int j = i;
i++;
j = j + i;
```

ou em

```
int j = i;
j = j + i;
i++;
```

Este tipo de comportamento deve-se a que, como se viu, no C++ as atribuições e respectivas abreviações são operadores que têm um resultado e que portanto podem ser usados dentro de expressões envolventes. Se fossem instruções especiais, como em Pascal, era mais difícil escrever instruções mal comportadas[33]. Este tipo de operações, no entanto, fazem parte do estilo usual de programação em C++, pelo que devem ser bem percebidas as suas consequências: *uma expressão com efeitos laterais não pode ser interpretada como uma simples expressão matemática*, pois há variáveis que mudam de valor durante o cálculo! Expressões com efeitos laterais são pois de evitar, salvo nas "expressões idiomáticas" da linguagem C++. Como se verá, esta é também uma boa razão para se fazer uma distinção clara entre funções (sem efeitos laterais) e procedimentos (com efeitos laterais mas cujas invocações não podem constar numa expressão) quando se introduzir a modularização no próximo capítulo.

---

[32]Repare-se que é a ordem de cálculo dos operandos que é indefinida. A ordem de cálculo das operações neste caso é bem definida. Como a adição tem associatividade à esquerda, a expressão é equivalente a `y = (sin(x) + cos(x)) + sqrt(x)`, ou seja, a primeira adição é forçosamente calculada antes da segunda.

[33]Mas não impossível, pois uma função com argumentos passados por referência (`&` em C++ e `var` em Pascal) pode alterar argumentos envolvidos na mesma expressão que invoca a função.

Capítulo 3

Modularização: rotinas

Methods are more important than facts.

Donald E. Knuth, *Selected Papers in Computer Science*, 176 (1996)

A modularização é um conceito extremamente importante em programação. Neste capítulo abordar-se-á o nível atómico de modularização: as funções e os procedimentos. Este tipo de modularização é fundamental em programação procedimental. Quando se começar a abordar a programação baseada em objectos, no Capítulo 7, falar-se-á de um outro nível de modularização: as classes. Finalmente a modularização regressará a um nível ainda mais alto no Capítulo 9.

3.1 Introdução à modularização

Exemplos de modularização, i.e., exemplos de sistemas constituídos por módulos, são bem conhecidos. A maior parte dos bons sistemas de alta fidelidade são compostos por módulos: o amplificador, o equalizador, o leitor de DVD, o sintonizador, as colunas, etc. Para o produtor de um sistema deste tipo a modularização tem várias vantagens:

1. reduz a complexidade do sistema, pois cada módulo é mais simples que o sistema na globalidade e pode ser desenvolvido por um equipa especializada;
2. permite alterar um módulo independentemente dos outros, por exemplo porque se desenvolveu um novo circuito para o amplificador, melhorando assim o comportamento do sistema na totalidade;
3. facilita a assistência técnica, pois é fácil verificar qual o módulo responsável pela avaria e consertá-lo isoladamente; e
4. permite fabricar os módulos em quantidades diferentes de modo à produção se adequar melhor à procura (e.g., hoje em dia os leitores de DVD vendem-se mais que os CD, que estão a ficar obsoletos).

Também para o consumidor final do sistema a modularização traz vantagens:

1. permite a substituição de um único módulo do sistema, quer por avaria quer por se pretender uma maior fidelidade usando, por exemplo, um melhor amplificador;
2. em caso de avaria apenas o módulo avariado fica indisponível, podendo-se continuar a usar todos os outros (excepto, claro, se o módulo tiver um papel fundamental no sistema);
3. permite a evolução do sistema por acrescento de novos módulos com novas funções (e.g., é possível acrescentar um leitor de DVD a um sistema antigo ligando-o ao amplificador);
4. evita a redundância, pois os módulos são reutilizados com facilidade (e.g., o amplificador amplifica os sinais do sintonizador, leitor de DVD, etc.).

Estas vantagens não são exclusivas dos sistemas de alta fidelidade: são gerais. Qualquer sistema pode beneficiar de pelo menos algumas destas vantagens se for modularizado. A arte da modularização está em identificar claramente que módulos devem existir no sistema. Uma boa modularização atribui uma única função bem definida a cada módulo, minimiza as ligações entre os módulos e maximiza a coesão interna de cada módulo. No caso de um bom sistema de alta fidelidade, tal corresponde a minimizar a complexidade dos cabos entre os módulos e a garantir que os módulos contêm apenas os circuitos que contribuem para a função do módulo. A coesão tem portanto a ver com as ligações internas a um módulo, que idealmente devem ser maximizadas. Normalmente, um módulo só pode ser coeso se tiver uma única função, bem definida.

Há algumas restrições adicionais a impor a uma boa modularização. Não basta que um módulo tenha uma função bem definida: tem de ter também uma interface bem definida. Por interface entende-se aquela parte de um módulo que está acessível do exterior e que permite a sua utilização. É claro, por exemplo, que um dono de uma alta fidelidade não pode substituir o seu amplificador por um novo modelo se este tiver ligações e cabos que não sejam compatíveis com o modelo mais antigo.

A interface de um módulo é a parte que está acessível ao consumidor. Tudo o resto faz parte da sua implementação, ou mecanismo, e é típico que esteja encerrado numa caixa fora da vista, ou pelo menos fora do alcance do consumidor. Num sistema bem desenhado, cada módulo mostra a sua interface e esconde a complexidade da sua implementação: cada módulo está encapsulado numa "caixa", a que se costuma chamar uma "caixa preta". Por exemplo, num relógio vê-se o mostrador, os ponteiros e o manípulo para acertar as horas, mas o mecanismo está escondido numa caixa. Num automóvel toda a mecânica está escondida sob o *capot*.

Para o consumidor, o interior (a implementação) de um módulo é irrelevante: o audiófilo só se importa com a constituição interna de um módulo na medida em que ela determina o seu comportamento externo. O consumidor de um módulo só precisa de conhecer a sua função e a sua interface. A sua visão de um módulo permite-lhe, abstraindo-se do seu funcionamento interno, preocupar-se apenas com aquilo que lhe interessa: ouvir som de alta fidelidade.

A modularização, o encapsulamento e a abstracção são conceitos fundamentais em engenharia da programação para o desenvolvimento de programas de grande escala. Mesmo para pequenos programas estes conceitos são úteis, quando mais não seja pelo treino que proporciona a sua utilização e que permite ao programador mais tarde lidar melhor com projectos de maior escala. Estes conceitos serão estudados com mais profundidade em disciplinas posteriores, como Concepção e Desenvolvimento de Sistemas de Informação e Engenharia da Programação. Neste capítulo far-se-á uma primeira abordagem aos conceitos de modularização e de abstracção em programação. Os mesmos conceitos serão revisitados ao longo dos capítulos subsequentes.

As vantagens da modularização para a programação são pelo menos as seguintes [2]:

1. facilita a detecção de erros, pois é em princípio simples identificar o módulo responsável pelo erro, reduzindo-se assim o tempo gasto na identificação de erros;

2. permite testar os módulos individualmente, em vez de se testar apenas o programa completo, o que reduz a complexidade do teste e permite começar a testar antes de se ter completado o programa;

3. permite fazer a manutenção do programa (correcção de erros, melhoramentos, etc.) módulo a módulo e não no programa globalmente, o que reduz a probabilidade de essa manutenção ter consequências imprevistas noutras partes do programa;

4. permite o desenvolvimento independente dos módulos, o que simplifica o trabalho em equipa, pois cada elemento ou cada sub-equipa tem a seu cargo apenas alguns módulos do programa; e

5. permite a reutilização do código[1] desenvolvido, que é porventura a mais evidente vantagem da modularização em programas de pequena escala.

Um programador assume, ao longo do desenvolvimento de um programa, dois papeis distintos: por um lado é produtor, pois é sua responsabilidade desenvolver módulos; por outro é consumidor, pois fará com certeza uso de outros módulos, desenvolvidos por outrem ou por ele próprio no passado. Esta é uma noção muito importante. É de toda a conveniência que um programador possa ser um mero consumidor dos módulos já desenvolvidos, sem se preocupar com o seu funcionamento interno: basta-lhe, como consumidor, sabe qual a função módulo e qual a sua interface.

À utilização de um sistema em que se olha para ele apenas do ponto de vista do seu funcionamento externo chama-se *abstracção*. A capacidade de abstracção é das qualidades mais importantes que um programador pode ter (ou desenvolver), pois permite-lhe reduzir substancialmente a complexidade da informação que tem de ter presente na memória, conduzindo por isso a substanciais ganhos de produtividade e a uma menor taxa de erros. A capacidade de abstracção é tão fundamental na programação como no dia-a-dia. Ninguém conduz o automóvel com a preocupação de saber se a vela do primeiro cilindro produzirá a faísca no momento certo para a próxima explosão! Um automóvel para o condutor normal é um objecto que lhe permite deslocar-se e que possui um interface simples: a ignição para ligar o automóvel, o volante para ajustar a direcção, o acelerador para ganhar velocidade, etc. O encapsulamento dos módulos, ao esconder do consumidor o seu mecanismo, facilita-lhe esta visão externa dos módulos e portanto facilita a sua capacidade de abstracção.

[1] Dá-se o nome de código a qualquer pedaço de programa numa dada linguagem de programação.

3.2 Funções e procedimentos: rotinas

A modularização é, na realidade, um processo hierárquico: muito provavelmente cada módulo de um sistema de alta fidelidade é composto por sub-módulos razoavelmente independentes, embora invisíveis para o consumidor. O mesmo se passa na programação. Para já, no entanto, abordar-se-ão apenas as unidades atómicas de modularização em programação: *funções* e *procedimentos*[2].

Função Conjunto de instruções, com interface bem definida, que efectua um dado cálculo.

Procedimento Conjunto de instruções, com interface bem definida, que faz qualquer coisa.

Por uma questão de simplicidade daqui em diante chamar-se-á *rotina* quer a funções quer a procedimentos. Ou seja, a unidade atómica de modularização são as rotinas, que se podem ser ou funções e ou procedimentos.

As rotinas permitem isolar pedaços de código com objectivos bem definidos e torná-los reutilizáveis onde quer que seja necessário. O "fabrico" de uma rotina corresponde em C++ àquilo que se designa por *definição*. Uma vez definida "fabricada", uma rotina pode ser utilizada sem que se precise de conhecer o seu funcionamento interno, da mesma forma que o audiófilo não está muito interessado nos circuitos dentro do amplificador, mas simplesmente nas suas características vistas do exterior. Rotinas são pois como caixas pretas: uma vez definidas (e correctas), devem ser usadas sem preocupações quanto ao seu funcionamento interno.

Qualquer linguagem de programação, e o C++ em particular, fornece um conjunto de tipos básicos e de operações que se podem realizar com variáveis, constantes e valores desses tipos. Uma maneira de ver as rotinas é como extensões a essas operações disponíveis na linguagem "não artilhada". Por exemplo, o C++ não fornece qualquer operação para calcular o `mdc` (máximo divisor comum), mas no Capítulo 1 viu-se uma forma de o calcular. O pedaço de programa que calcula o `mdc` pode ser colocado numa caixa preta, com uma interface apropriada, de modo a que possa ser reutilizado sempre que necessário. Isso corresponde a definir uma função chamada `mdc` que pode mais tarde ser utilizada onde for necessário calcular o máximo divisor comum de dois inteiros:

```
cout << "Introduza dois inteiros: ";
int m, n;
cin >> m >> n;
cout << "mdc(" << m << ", " << n << ") = " << mdc(m, n) << endl;
```

Assim, ao se produzirem rotinas, está-se como que a construir uma versão mais potente da linguagem de programação utilizada. É interessante que muitas tarefas em programação podem ser interpretadas exactamente desta forma. Em particular, ver-se-á mais tarde que é possível aplicar a mesma ideia aos tipos de dados disponíveis: o programador pode não apenas "artilhar" a linguagem com novas operações sobre tipos básicos, como também com novos tipos!

[2] A linguagem C++ não distingue entre funções e procedimentos: ambos são conhecidos simplesmente por funções nas referências técnicas sobre a linguagem.

A este último tipo de programação chama-se programação centrada nos dados, e é a base da programação baseada em objectos e, consequentemente, da programação orientada para objectos.

3.2.1 Abordagens descendente e ascendente

Nos capítulos anteriores introduziram-se vários conceitos, como os de algoritmos, dados e programas. Explicaram-se também algumas das ferramentas das linguagens de programação, tais como variáveis, constantes, tipos, valores literais, expressões, operações, etc. Mas, como usar todos estes conceitos para resolver um problema em particular?

Existem muitas possíveis abordagens à resolução de problemas em programação, quase todas com um paralelo perfeito com as abordagens que se usam no dia-a-dia. Porventura uma das abordagens mais clássicas em programação é a abordagem descendente (ou *top-down*).

Abordar um problema "de cima para baixo" corresponde a olhar para ele na globalidade e identificar o mais pequeno número de sub-problemas independentes possível. Depois, sendo esses sub-problemas independentes, podem-se resolver independentemente usando a mesma abordagem: cada sub-problema é dividido num conjunto de sub-sub-problemas mais simples. Esta abordagem tem a vantagem de limitar a quantidade de informação a processar pelo programador em cada passo e de, por divisão sucessiva, ir reduzindo a complexidade dos problema até à trivialidade. Quando os problemas identificados se tornam triviais pode-se escrever a sua solução na forma do passo de um algoritmo ou instrução de um programa. Cada problema ou sub-problema identificado corresponde normalmente a uma rotina no programa final.

Esta abordagem não está isenta de problemas. Um deles é o de não facilitar o reaproveitamento de código. É que dois sub-problemas podem ser iguais sem que o programador dê por isso, o que resulta em duas rotinas iguais ou, pelo menos, muito parecidas. Assim, é muitas vezes conveniente alternar a abordagem descendente com a abordagem ascendente.

Na abordagem ascendente, começa por se tentar perceber que ferramentas fazem falta para resolver o problema mas não estão ainda disponíveis. Depois desenvolvem-se essas ferramentas, que correspondem tipicamente a rotinas, e repete-se o processo, indo sempre acrescentando camadas de ferramentas sucessivamente mais sofisticadas à linguagem. A desvantagem deste método é que dificilmente se pode saber que ferramentas fazem falta sem um mínimo de abordagem descendente. Daí a vantagem de alternar as abordagens.

Suponha-se que se pretende escrever um programa que some duas fracções positivas introduzidas do teclado e mostre o resultado na forma de uma fracção reduzida (ou em termos mínimos). Recorda-se que uma fracção $\frac{n}{d}$ está em termos mínimos se não existir qualquer divisor comum ao numerador e ao denominador com excepção de 1, ou seja, se $\mathrm{mdc}(m, n) = 1$. Para simplificar admite-se que as fracções introduzidas são representadas, cada uma, por um par de valores inteiros positivos: numerador e denominador. Pode-se começar por escrever o "esqueleto" do programa:

```
#include <iostream>
```

```
using namespace std;

/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
{
    ...
}
```

Olhando o problema na globalidade, verifica-se que pode ser dividido em três sub-problemas: ler as fracções de entrada, obter a fracção soma em termos mínimos e escrever o resultado. Traduzindo para C++:

```
#include <iostream>

using namespace std;

/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
{
    // Leitura das fracções do teclado:
    ...

    // Cálculo da fracção soma em termos mínimos:
    ...

    // Escrita do resultado:
    ...
}
```

Pode-se agora abordar cada sub-problema independentemente. Começando pela leitura das fracções, identificam-se dois sub-sub-problemas: pedir ao utilizador para introduzir as fracções e ler as fracções. Estes problemas são tão simples de resolver que se passa directamente ao código:

```
#include <iostream>

using namespace std;

/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
{
    // Leitura das fracções do teclado:
```

```
    cout << "Introduza duas fracções: ";
    int n1, d1, n2, d2;
    cin >> n1 >> d1 >> n2 >> d2;

    // Cálculo da fracção soma em termos mínimos:
    ...

    // Escrita do resultado:
    ...
}
```

Usou-se um única instrução para definir quatro variáveis do tipo `int` que guardarão os numeradores e denominadores das duas fracções lidas: o C++ permite definir várias variáveis na mesma instrução.

De seguida pode-se passar ao sub-problema final da escrita do resultado. Suponha-se que, sendo as fracções de entrada $\frac{6}{9}$ e $\frac{7}{3}$, se pretendia que surgisse no ecrã:

```
A soma de 6/9 com 7/3 é 3/1.
```

Então o problema pode ser resolvido como se segue:

```
#include <iostream>

using namespace std;

/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
{
    // Leitura das fracções do teclado:
    cout << "Introduza duas fracções: ";
    int n1, d1, n2, d2;
    cin >> n1 >> d1 >> n2 >> d2;

    // Cálculo da fracção soma em termos mínimos:
    ...

    // Escrita do resultado:
    cout << "A soma de ";
    escreveFracção(n1, d1);
    cout << " com ";
    escreveFracção(n2, d2);
    cout << " é ";
    escreveFracção(?, ?);
    cout << '.' << endl;
}
```

Neste caso adiou-se um problema: admitiu-se que está disponível algures um procedimento chamado `escreveFracção()` que escreve uma fracção no ecrã. É evidente que mais tarde será preciso definir esse procedimento, que, usando um pouco de abordagem ascendente, se percebeu vir a ser utilizado em três locais diferentes. Podia-se ter levado a abordagem ascendente mais longe: se se vai lidar com fracções, não seria útil um procedimento para ler uma fracção do teclado? E uma outra para reduzir uma fracção a termos mínimos? O resultado obtido como uma tal abordagem, dada a pequenez do problema, seria semelhante ao que se obterá prosseguindo a abordagem descendente.

Sobrou outro problema: como escrever a fracção resultado sem saber onde se encontram o seu numerador e o seu denominador? Claramente é necessário, para a resolução do sub-problema do cálculo da soma, definir duas variáveis adicionais onde esses valores serão guardados:

```
#include <iostream>

using namespace std;

/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
{
    // Leitura das fracções do teclado:
    cout << "Introduza duas fracções: ";
    int n1, d1, n2, d2;
    cin >> n1 >> d1 >> n2 >> d2;

    // Cálculo da fracção soma em termos mínimos:
    int n;
    int d;
    ...

    // Escrita do resultado:
    cout << "A soma de ";
    escreveFracção(n1, d1);
    cout << " com ";
    escreveFracção(n2, d2);
    cout << " é ";
    escreveFracção(n, d);
    cout << '.' << endl;
}
```

É necessário agora resolver o sub-problema do cálculo da fracção soma em termos mínimos. Dadas duas fracções, a sua soma é simples se desde que tenham o mesmo denominador. A forma mais simples de reduzir duas fracções diferentes ao mesmo denominador consiste em multiplicar ambos os termos da primeira fracção pelo denominador da segunda e vice versa. Ou seja,

$$\frac{a}{b} + \frac{c}{d} = \frac{ad}{bd} + \frac{bc}{bd} = \frac{ad + bc}{bd},$$

pelo que o programa fica:

```
#include <iostream>

using namespace std;

/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
{
    // Leitura das fracções do teclado:
    cout << "Introduza duas fracções: ";
    int n1, d1, n2, d2;
    cin >> n1 >> d1 >> n2 >> d2;

    // Cálculo da fracção soma em termos mínimos:
    int n = n1 * d2 + n2 * d1;
    int d = d1 * d2;
    ...

    // Escrita do resultado:
    cout << "A soma de ";
    escreveFracção(n1, d1);
    cout << " com ";
    escreveFracção(n2, d2);
    cout << " é ";
    escreveFracção(n, d);
    cout << '.' << endl;
}
```

Usando o mesmo exemplo que anteriormente, se as fracções de entrada forem $\frac{6}{9}$ e $\frac{7}{3}$, o programa tal como está escreve

```
A soma de 6/9 com 7/3 é 81/27.
```

Ou seja, a fracção resultado não está reduzida. Para a reduzir é necessário dividir o numerador e o denominador da fracção pelo seu mdc:

```
#include <iostream>

using namespace std;

/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
```

```
{
    // Leitura das fracções do teclado:
    cout << "Introduza duas fracções: ";
    int n1, d1, n2, d2;
    cin >> n1 >> d1 >> n2 >> d2;

    // Cálculo da fracção soma em termos mínimos:
    int n = n1 * d2 + n2 * d1;
    int d = d1 * d2;
    int k = mdc(n, d);
    n /= k; // o mesmo que n = n / k;
    d /= k; // o mesmo que d = d / k;

    // Escrita do resultado:
    cout << "A soma de ";
    escreveFracção(n1, d1);
    cout << " com ";
    escreveFracção(n2, d2);
    cout << " é ";
    escreveFracção(n, d);
    cout << '.' << endl;
}
```

Neste caso adiou-se mais um problema: admitiu-se que está disponível algures uma função chamada `mdc()` que calcula o mdc de dois inteiros. Isto significa que mais tarde será preciso definir esse procedimento. Recorda-se, no entanto, que o algoritmo para o cálculo do mdc foi visto no Capítulo 1 e revisto no Capítulo 2.

A solução encontrada ainda precisa de ser refinada. Suponha-se que o programa é compilado e executado num ambiente onde valores do tipo `int` são representados com apenas 6 *bits*. Nesse caso, de acordo com a discussão do capítulo anterior, essas variáveis podem conter valores entre -32 e 31. Que acontece quando, sendo as fracções de entrada $\frac{6}{9}$ e $\frac{7}{3}$, se inicializa a variável `n`? O valor da expressão `n1 * d2 + n2 * d1` é 81, que excede em muito a gama dos `int` com 6 *bits*! O resultado é desastroso. Não é possível evitar totalmente este problema, mas é possível minimizá-lo se se reduzir a termos mínimos as fracções de entrada logo após a sua leitura. Se isso acontecesse, como $\frac{6}{9}$ em termos mínimos é $\frac{2}{3}$, a expressão `n1 * d2 + n2 * d1` teria o valor 27, dentro da gama de valores hipotética dos `int`.

Nos ambientes típicos os valores do tipo `int` são representados por 32 *bits*, pelo que o problema acima só se põe para numeradores e denominadores muito maiores. Mas não é pelo facto de ser um problema mais raro que deixa de ser problema, pelo que convém alterar o programa para:

```
#include <iostream>

using namespace std;
```

```
/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
{
    // Leitura das fracções do teclado:
    cout << "Introduza duas fracções: ";
    int n1, d1, n2, d2;
    cin >> n1 >> d1 >> n2 >> d2;
    int k = mdc(n1, d1);
    n1 /= k;
    d1 /= k;
    int k = mdc(n2, d2);
    n2 /= k;
    d2 /= k;

    // Cálculo da fracção soma em termos mínimos:
    int n = n1 * d2 + n2 * d1;
    int d = d1 * d2;
    int k = mdc(n, d);
    n /= k;
    d /= k;

    // Escrita do resultado:
    cout << "A soma de ";
    escreveFracção(n1, d1);
    cout << " com ";
    escreveFracção(n2, d2);
    cout << " é ";
    escreveFracção(n, d);
    cout << '.' << endl;
}
```

O programa acima precisa ainda de ser corrigido. Como se verá mais à frente, não se podem definir múltiplas variáveis com o mesmo nome no mesmo contexto. Assim, a variável `k` deve ser definida uma única vez e reutilizada quando necessário:

```
#include <iostream>

using namespace std;

/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
{
    // Leitura das fracções do teclado:
    cout << "Introduza duas fracções: ";
```

```
    int n1, d1, n2, d2;
    cin >> n1 >> d1 >> n2 >> d2;
    int k = mdc(n1, d1);
    n1 /= k;
    d1 /= k;
    k = mdc(n2, d2);
    n2 /= k;
    d2 /= k;

    // Cálculo da fracção soma em termos mínimos:
    int n = n1 * d2 + n2 * d1;
    int d = d1 * d2;
    k = mdc(n, d);
    n /= k;
    d /= k;

    // Escrita do resultado:
    cout << "A soma de ";
    escreveFracção(n1, d1);
    cout << " com ";
    escreveFracção(n2, d2);
    cout << " é ";
    escreveFracção(n, d);
    cout << '.' << endl;
}
```

3.2.2 Definição de rotinas

Antes de se poder utilizar uma função ou um procedimento, é necessário defini-lo (antes de usar a aparelhagem há que fabricá-la). Falta, portanto, definir a função `mdc()` e o procedimento `escreveFracção()`.

Um possível algoritmo para o cálculo do mdc de dois inteiros positivos foi visto no primeiro capítulo. A parte relevante do correspondente programa é:

```
int m; int n; // Como inicializar m e n?

int k;
if(m < n)
    k = m;
else
    k = n;

while(m % k != 0 or n % k != 0)
    --k;
```

Este troço de programa calcula o `mdc` dos valores de `m` e `n` e coloca o resultado na variável `k`. É necessário colocar este código numa função, ou seja, num módulo com uma interface e uma implementação escondida numa "caixa". Começa-se por colocar o código numa "caixa", i.e., entre `{}`:

```
{
    int m; int n; // Como inicializar m e n?

    int k;
    if(m < n)
        k = m;
    else
        k = n;

    while(m % k != 0 or n % k != 0)
        --k;
}
```

Tudo o que fica dentro da caixa está inacessível do exterior.

É necessário agora atribuir um nome ao módulo, tal como se atribui o nome "Amplificador" ao módulo de uma aparelhagem que amplifica os sinais áudio vindos de outros módulos. Neste caso o módulo chama-se `mdc`:

```
mdc
{
    int m; int n; // Como inicializar m e n?

    int k;
    if(m < n)
        k = m;
    else
        k = n;

    while(m % k != 0 or n % k != 0)
        --k;
}
```

Qual é a interface deste módulo, ou melhor, desta função? Uma caixa sem interface é inútil. A função deve calcular o `mdc` de dois números. Mas de onde vêm eles? O resultado da função fica guardado na variável `k`, que está dentro da caixa. Como comunicar esse valor para o exterior?

No programa da soma de fracções a função `mdc()` é utilizada, ou melhor, invocada (ou ainda chamada), em três locais diferentes. Em cada um deles escreveu-se o nome da função seguida de uma lista de duas expressões. A estas expressões chama-se *argumentos* passados à função.

Quando calculadas, essas expressões têm os valores que se pretende que sejam usados para inicializar as variáveis m e n definidas na função mdc(). Para o conseguir, as variáveis m e n não devem ser variáveis normais definidas dentro da caixa: devem ser *parâmetros* da função. Os parâmetros da função são definidos entre parênteses logo após o nome da função, e não dentro da caixa ou *corpo* da função, e servem como entradas da função, fazendo parte da sua interface:

```
mdc(int m, int n)
{
    int k;
    if(m < n)
        k = m;
    else
        k = n;

    while(m % k != 0 or n % k != 0)
        --k;
}
```

A função mdc(), que é um módulo, já tem entradas, que correspondem aos dois parâmetros definidos. Quando a função é invocada *os valores dos argumentos são usados para inicializar os parâmetros respectivos*. Claro que isso implica que o número de argumentos numa invocação de uma função tem de ser rigorosamente igual ao número de parâmetros da função, salvo em alguns casos que se verão mais tarde. E os tipos dos argumentos também têm de ser compatíveis com os tipos dos parâmetros.

É muito importante distinguir entre a definição de uma rotina (neste caso um função) e a sua invocação ou chamada. A definição de uma rotina é única, e indica a sua interface (i.e., como a rotina se utiliza e que nome tem) e a sua implementação (i.e., como funciona). Uma invocação de uma rotina é feita onde quer que seja necessário recorrer aos seus serviços para calcular algo (no caso de uma função) ou para fazer alguma coisa (no caso de um procedimento).

Finalmente falte definir como se fazem as saídas da função. Uma função em C++ só pode ter uma saída. Neste caso a saída é o valor guardado em k no final da função, que é o maior divisor comum dos dois parâmetros m e n. Para que o valor de k seja usado como saída da função, usa-se uma instrução de retorno:

```
mdc(int m, int n)
{
    int k;
    if(m < n)
        k = m;
    else
        k = n;

    while(m % k != 0 or n % k != 0)
        --k;
```

```
    return k;
}
```

A definição da função tem de indicar claramente que a função tem uma saída de um dado tipo. Neste caso a saída é um valor do tipo `int`, pelo que a definição da função fica:

```
int mdc(int m, int n)
{
    int k;
    if(m < n)
        k = m;
    else
        k = n;

    while(m % k != 0 or n % k != 0)
        --k;

    return k;
}
```

3.2.3 Sintaxe das definições de funções

A definição de uma rotina é constituída por um *cabeçalho* seguido de um corpo, que consiste no conjunto de instruções entre `{}`. No cabeçalho são indicados o tipo do valor calculado ou *devolvido* por essa rotina, o nome da rotina e a sua lista de parâmetros (cada parâmetro é representado por um par *`tipo nome`*, sendo os pares separados por vírgulas). Isto é:

```
tipo_de_devolução nome(lista_de_parâmetros)
```

O cabeçalho de uma rotina corresponde à sua interface, tal como as tomadas para cabos nas traseiras de um amplificador e os botões de controlo no seu painel frontal constituem a sua interface. Quando a rotina é uma função é porque calcula um valor de um determinado tipo. Esse tipo é indicado em primeiro lugar no cabeçalho. No caso de um procedimento, que não calcula nada, é necessário colocar a palavra-chave `void` no lugar desse tipo, como se verá mais à frente. Logo a seguir indica-se o nome da rotina, e finalmente uma lista de parâmetros, que consiste simplesmente numa lista de definições de variáveis com uma sintaxe semelhante (embora não idêntica) à que se viu no Capítulo 2.

No exemplo anterior definiu-se uma função que tem dois parâmetros (ambos do tipo `int`) e que devolve um valor inteiro. O seu cabeçalho é:

```
int mdc(int m, int n)
```

A sintaxe de especificação dos parâmetros é diferente da sintaxe de definição de variáveis "normais", pois não se podem definir vários parâmetros do mesmo tipo separando os seus nomes por vírgulas: o cabeçalho

```
int mdc(int m, n)
```

é inválido, pois falta-lhe a especificação do tipo do parâmetro n.

O corpo desta função, i.e., a sua implementação, corresponde às instruções entre {}:

```
{
    int k;
    if(m < n)
        k = m;
    else
        k = n;

    while(m % k != 0 or n % k != 0)
        --k;

   return k;
}
```

Idealmente o corpo das rotinas deve ser pequeno, contendo entre uma e dez instruções. Muito raramente haverá boas razões para ultrapassar as 60 linhas. A razão para isso prende-se com a dificuldade dos humanos (sim, os programadores são humanos) em abarcar demasiados assuntos de uma só vez: quanto mais curto for o corpo de uma rotina, mais fácil foi de desenvolver e mais fácil é de corrigir ou melhorar. Por outro lado, quanto maior for uma rotina, mais difícil é reutilizar o seu código.

3.2.4 Contrato e documentação de uma rotina

A definição de uma rotina (ou melhor, a sua declaração, como se verá na Secção 3.2.17) só fica realmente completa quando incluir um comentário indicando exactamente aquilo que calcula (se for uma função) ou aquilo que faz (se for um procedimento). I.e., a definição de uma rotina só fica completa se contiver a sua *especificação*:

```
/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre   PC ≡ 0 < m ∧ 0 < n.
    @post  CO ≡ mdc = mdc(m, n). Assume-se que m e n não mudam de valor. */
int mdc(int m, int n)
{
    int k;
    if(m < n)
        k = m;
    else
        k = n;
```

```
        while(m % k != 0 or n % k != 0)
            --k;

        return k;
    }
```

Todo o texto colocado entre `/*` e `*/` é ignorado pelo compilador: é um comentário de bloco (os comentários começados por `//` são comentários de linha). Este comentário contém:

- uma descrição do que a função calcula ou do que o procedimento faz, em português vernáculo;
- a pré-condição ou PC da rotina, ou seja, a condição que as entradas (i.e., os valores iniciais dos parâmetros) têm de verificar de modo a assegurar o bom funcionamento da rotina; e
- a condição objectivo ou CO, mais importante ainda que a PC, que indica a condição que deve ser válida quando a rotina termina numa instrução de retorno. No caso de uma função, o seu nome pode ser usado na condição objectivo para indicar o valor devolvido no seu final.

Na definição acima, colocou-se o comentário junto ao cabeçalho da rotina por ser fundamental para se perceber *o que a rotina faz* (ou calcula). O cabeçalho de uma rotina, por si só, não diz o que ela faz, simplesmente *como se utiliza*. Por outro lado, o corpo de uma rotina diz *como funciona*. Uma rotina é uma caixa preta: no seu interior fica o mecanismo (o corpo da rotina), no exterior a interface (o cabeçalho da rotina) e pode-se ainda saber para que serve e como se utiliza lendo o seu manual de utilização (os comentários contendo a descrição em português e a pré-condição e a condição objectivo).

Estes comentários são parte da *documentação* do programa. Os comentários de bloco começados por `/**` e os de linha começados por `///` são considerados comentários de documentação por alguns sistemas automáticos que extraem a documentação de um programa a partir deste tipo especial de comentário[3]. Da mesma forma, as construções @pre e @post servem para identificar ao sistema automático de documentação a pré-condição e a condição objectivo, que também é conhecida por pós-condição.

As condições PC e CO funcionam como um *contrato* que o programador produtor da rotina estabelece com o seu programador consumidor:

> Se o programador consumidor desta rotina garantir que as variáveis do programa respeitam a pré-condição PC imediatamente antes de a invocar, o programador produtor desta rotina garante que a condição objectivo CO será verdadeira imediatamente depois de esta terminar.

[3]Em particular existe um sistema de documentação disponível em Linux chamado `doxygen` que é de grande utilidade.

Esta visão "legalista" da programação está por trás de uma metodologia relativamente recente de desenvolvimento de programas a que se chama "desenho por contracto". Em algumas linguagens de programação, como o Eiffel, as pré-condições e as condições objectivo fazem parte da própria linguagem, o que permite fazer a sua verificação automática. Na linguagem C++ um efeito semelhante, embora mais limitado, pode ser obtido usando as chamadas instruções de asserção, discutidas mais abaixo.

3.2.5 Integração da função no programa

Para que a função `mdc()` possa ser utilizada no programa desenvolvido, é necessário que a sua definição se encontre antes da primeira utilização:

```
#include <iostream>

using namespace std;

/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre   PC ≡ 0 < m ∧ 0 < n.
    @post  CO ≡ mdc = mdc(m, n). Assume-se que m e n não mudam de valor. */
int mdc(int m, int n)
{
    int k;
    if(m < n)
        k = m;
    else
        k = n;

    while(m % k != 0 or n % k != 0)
        --k;

    return k;
}

/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
{
    // Leitura das fracções do teclado:
    cout << "Introduza duas fracções: ";
    int n1, d1, n2, d2;
    cin >> n1 >> d1 >> n2 >> d2;
    int k = mdc(n1, d1);
    n1 /= k;
    d1 /= k;
```

```
        k = mdc(n2, d2);
        n2 /= k;
        d2 /= k;

        // Cálculo da fracção soma em termos mínimos:
        int n = n1 * d2 + n2 * d1;
        int d = d1 * d2;
        k = mdc(n, d);
        n /= k;
        d /= k;

        // Escrita do resultado:
        cout << "A soma de ";
        escreveFracção(n1, d1);
        cout << " com ";
        escreveFracção(n2, d2);
        cout << " é ";
        escreveFracção(n, d);
        cout << '.' << endl;
    }
```

3.2.6 Sintaxe e semântica da invocação ou chamada

Depois de definidas, as rotinas podem ser utilizadas noutros locais de um programa. A utilização típica corresponde a invocar ou chamar a rotina para que seja executada com um determinado conjunto de entradas. A invocação da função `mdc()` definida acima pode ser feita como se segue:

```
int x = 5;                // 1
int divisor;              // 2
divisor = mdc(x + 3, 6); // 3
cout << divisor << endl; // 4
```

A sintaxe da invocação de rotinas consiste simplesmente em colocar o seu nome seguido de uma lista de expressões (separadas por vírgulas) em número igual aos dos seus parâmetros. A estas expressões chama-se *argumentos*.

Os valores recebidos pelos parâmetros de uma rotina e o valor devolvido por uma função podem ser de qualquer tipo básico do C++ ou de tipos de dados definidos pelo programador (começar-se-á a falar destes tipos no Capítulo 5). O tipo de um argumento tem de ser compatível com o tipo do parâmetro respectivo[4]. Como é evidente, uma função pode devolver um único valor do tipo indicado no seu cabeçalho.

Uma invocação de uma função pode ser usada em expressões mais complexas, tal como qualquer operador disponível na linguagem, uma vez que as funções devolvem um valor calcu-

[4]Não esquecer que todas as expressões em C++ são de um determinado tipo.

lado. No exemplo acima, a chamada à função `mdc()` é usada como segundo operando de uma operação de atribuição (instrução 3).

Que acontece quando o código apresentado é executado?

Instrução 1: É construída uma variável `x` inteira com valor inicial 5.

Instrução 2: É construída uma variável `divisor`, também inteira, mas sem que seja inicializada, pelo que contém "lixo" (já se viu que não é boa ideia não inicializar, isto é apenas um exemplo!).

Instrução 3: Esta instrução implica várias ocorrências sequenciais, pelo que se separa em duas partes:

```
          mdc(x + 3, 6)  // 3A
divisor =              ; // 3B
```

Instrução 3A: É invocada a função `mdc()`:

1. São construídos os parâmetros `m` e `n`, que funcionam como quaisquer outras variáveis, excepto quanto à inicialização.
2. Cada um dos parâmetros `m` e `n` é inicializado com o valor do argumento respectivo na lista de argumentos colocados entre parênteses na chamada da função. Neste caso o parâmetro `m` é inicializado com o valor 8 e o parâmetro `n` é inicializado com o valor 6.
3. A execução do programa passa para a primeira instrução do corpo da função.
4. O corpo da função é executado. A primeira instrução executada constrói uma nova variável `k` com "lixo". A função termina quando se atinge a chaveta final do seu corpo ou quando ocorre uma instrução de retorno, que consiste na palavra-chave `return` seguida de uma expressão (apenas no caso das funções, como se verá). O valor dessa expressão é o valor calculado e devolvido pela função. Neste caso o seu valor é 2 (valor de `k` depois da procura do `mdc`).
5. Ao ser atingida a instrução de retorno a função termina.
6. São destruídas as variáveis `k`, `m` e `n`.
7. A execução do programa passa para a instrução seguinte à de invocação da função (neste caso 3B).

Instrução 3B: É atribuído à variável `divisor` o valor calculado pela função (neste caso é 2). Diz-se que a função *devolveu* o valor calculado.

Instrução 4: O valor de `divisor` é escrito no ecrã.

3.2.7 Parâmetros

Parâmetros são as variáveis listadas entre parênteses no cabeçalho da definição de uma rotina. São variáveis locais (ver Secção 3.2.12), embora com uma particularidade: são automaticamente inicializadas com o valor dos argumentos respectivos em cada invocação da rotina.

3.2.8 Argumentos

Argumentos são as expressões listadas entre parênteses numa invocação ou chamada de uma rotina. O seu valor é utilizado para inicializar os parâmetros da rotina invocada.

3.2.9 Retorno e devolução

Em inglês a palavra *return* tem dois significados distintos: retornar (ou regressar) e devolver. O português é neste caso mais rico, pelo que se usarão palavras distintas: dir-se-á que uma rotina *retorna* quando termina a sua execução e o fluxo de execução regressa ao ponto de invocação, e dir-se-á que uma função, ao retornar, *devolve* um valor que pode ser usado na expressão em que a função foi invocada. No exemplo do `mdc` acima o valor inteiro devolvido é usado numa expressão envolvendo o operador de atribuição.

Uma função termina quando o fluxo de execução atinge uma instrução de retorno. As instruções de retorno consistem na palavra-chave `return` seguida de uma expressão e de um `;`. A expressão tem de ser de um tipo compatível com o tipo de devolução da função. O resultado da expressão, depois de convertido no tipo de devolução, é o valor devolvido ou calculado pela função.

No caso da função `mdc()` o retorno e a devolução fazem-se com a instrução

```
return k;
```

O valor devolvido neste caso é o valor contido na variável `k`, que é o `mdc` dos valores iniciais de `m` e `n`.

3.2.10 Significado de `void`

Um procedimento tem a mesma sintaxe de uma função, mas não devolve qualquer valor. Esse facto é indicado usando a palavra-chave `void` como tipo do valor de devolução. Um procedimento termina quando se atinge a chaveta final do seu corpo ou quando se atinge uma instrução de retorno simples, sem qualquer expressão, i.e., `return;`.

Os procedimentos têm tipicamente efeitos laterais, e.g., afectam valores de variáveis que lhes são exteriores (e.g., usando passagem de argumentos por referência, descrita na próxima secção). Assim sendo, para evitar maus comportamentos, não se devem usar procedimentos em expressões (ver Secção 2.7.8). A utilização do tipo de devolução `void` impede a chamada de procedimentos em expressões, pelo que o seu uso é recomendado[5].

Resumindo: é de toda a conveniência que os procedimentos tenham tipo de devolução `void` e que as funções se limitem a devolver um valor calculado e não tenham qualquer efeito lateral. O respeito por esta regra simples pode poupar muitas dores de cabeça ao programador.

No programa da soma de fracções ficou em falta a definição do procedimento `escreveFracção()`. A sua definição é muito simples:

[5] Isto é uma simplificação. Na realidade podem haver expressões envolvendo operandos do tipo `void`. Mas a sua utilidade muito é restrita.

```
/** Escreve no ecrã uma fracção, no formato usual, que lhe é passada na forma de dois
    argumentos inteiros positivos.
    @pre  PC ≡ V (ou seja, nenhuma pré-condição).
    @post CO ≡ o ecrã contém n/d em que n e d são os valores de n e d
               representados em base decimal. */
void escreveFracção(int n, int d)
{
    cout << n << '/' << d;
}
```

Não é necessária qualquer instrução de retorno, pois o procedimento retorna quando a execução atinge a chaveta final.

O programa completo é então:

```
#include <iostream>

using namespace std;

/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre  PC ≡ 0 < m ∧ 0 < n.
    @post CO ≡ mdc = mdc(m, n). Assume-se que m e n não mudam de valor. */
int mdc(int m, int n)
{
    int k;
    if(m < n)
        k = m;
    else
        k = n;

    while(m % k != 0 or n % k != 0)
        --k;

    return k;
}

/** Escreve no ecrã uma fracção, no formato usual, que lhe é passada na forma de dois
    argumentos inteiros positivos.
    @pre  PC ≡ V (ou seja, nenhuma pré-condição).
    @post CO ≡ o ecrã contém n/d em que n e d são os valores de n e d
               representados em base decimal. */
void escreveFracção(int n, int d)
{
    cout << n << '/' << d;
}
```

```
/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
{
    // Leitura das fracções do teclado:
    cout << "Introduza duas fracções: ";
    int n1, d1, n2, d2;
    cin >> n1 >> d1 >> n2 >> d2;
    int k = mdc(n1, d1);
    n1 /= k;
    d1 /= k;
    k = mdc(n2, d2);
    n2 /= k;
    d2 /= k;

    // Cálculo da fracção soma em termos mínimos:
    int n = n1 * d2 + n2 * d1;
    int d = d1 * d2;
    k = mdc(n, d);
    n /= k;
    d /= k;

    // Escrita do resultado:
    cout << "A soma de ";
    escreveFracção(n1, d1);
    cout << " com ";
    escreveFracção(n2, d2);
    cout << " é ";
    escreveFracção(n, d);
    cout << '.' << endl;
}
```

3.2.11 Passagem de argumentos por valor e por referência

Observe o seguinte exemplo de procedimento. O seu programador pretendia que o procedimento trocasse os valores de duas variáveis passadas como argumentos:

```
// Atenção! Este procedimento não funciona!
void troca(int x, int y)
{
    int const auxiliar = x;
    x = y;
    y = auxiliar;
    /* Não há instrução de retorno explícita, pois trata-se de um
```

```
        procedimento que não devolve qualquer valor.
        Alternativamente poder-se-ia usar return;. */
}
```

O que acontece ao se invocar este procedimento como indicado na segunda linha do seguinte código?

```
int a = 1, b = 2;
troca(a, b);
/* A invocação não ocorre dentro de qualquer expressão, dado que o procedi-
mento não devolve qualquer valor. */
cout << a << ' ' << b << endl;
```

1. São construídas as variáveis `x` e `y`.

2. Sendo parâmetros do procedimento, a variável `x` é inicializada com o valor 1 (valor de `a`) e a variável `y` é inicializada com o valor 2 (valor de `b`). Assim, os parâmetros são *cópias* dos argumentos.

3. Durante a execução do procedimento os valores guardados em `x` e `y` são trocados, ver Figura 3.1.

4. Antes de o procedimento terminar, as variáveis `x` e `y` têm valores 2 e 1 respectivamente.

5. Quando termina a execução do procedimento, as variáveis `x` e `y` são destruídas (ver explicação mais à frente)

Ou seja, não há qualquer efeito sobre os valores das variáveis `a` e `b`! Os parâmetros mudaram de valor *dentro do procedimento* mas as variáveis `a` e `b` não mudaram de valor: `a` continua a conter 1 e `b` a conter 2. Este tipo de comportamento ocorre quando numa função ou procedimento se usa a chamada *passagem de argumentos por valor*. Normalmente, este é um comportamento desejável. Só em alguns casos, como neste exemplo, esta é uma característica indesejável.

Para resolver este tipo de problemas, onde é de interesse que o valor das variáveis que são usadas como argumentos seja alterado dentro de um procedimento, existe o conceito de passagem de argumentos por referência. A passagem de um argumento por referência é indicada no cabeçalho do procedimento colocando o símbolo `&` depois do tipo do parâmetro pretendido, como se pode ver abaixo:

```
void troca(int& x, int& y)
{
    int const auxiliar = x;
    x = y;
    y = auxiliar;
}
```

Ao invocar como anteriormente, ver Figura 3.2:

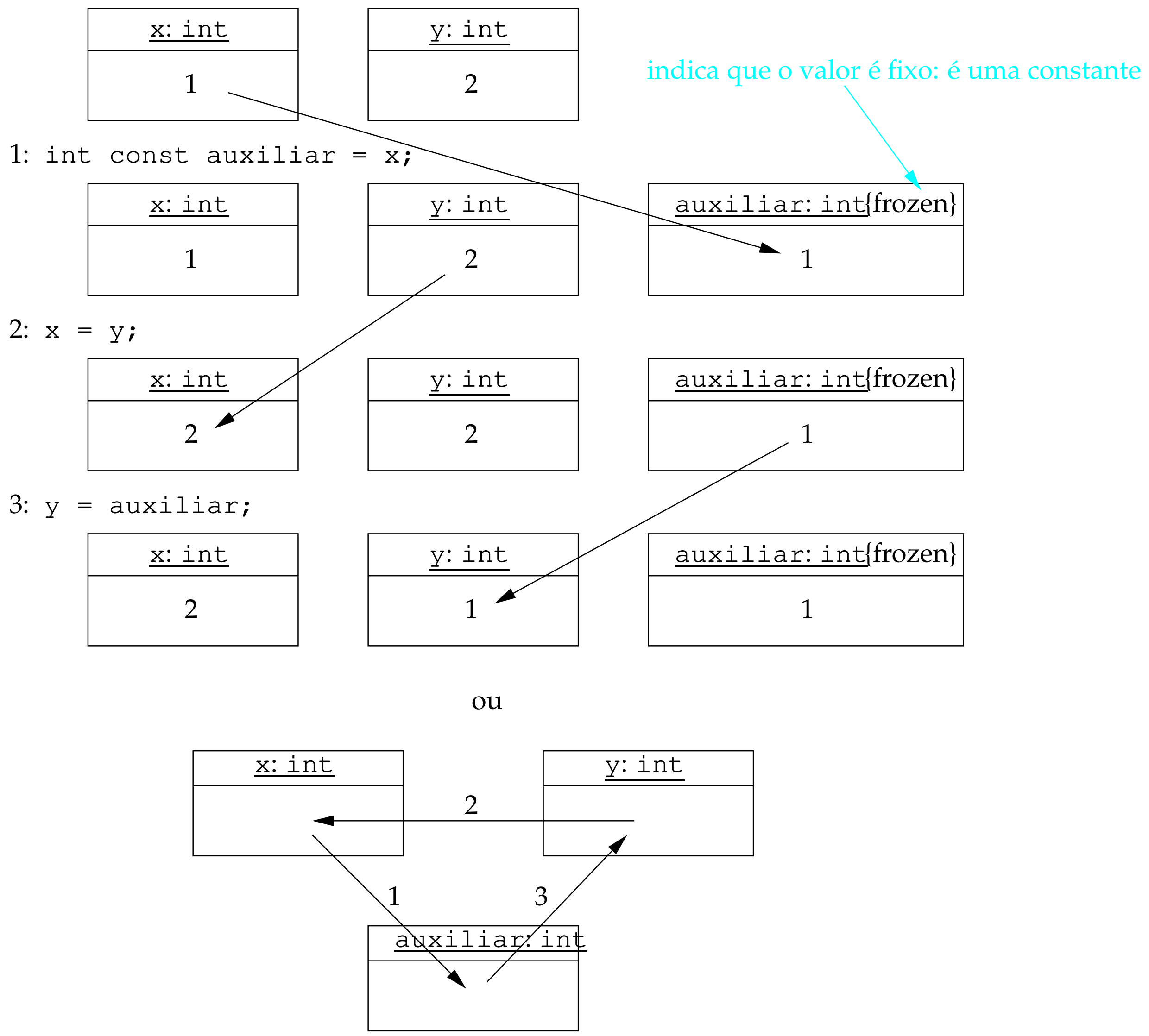


Figura 3.1: Algoritmo usual de troca de valores entre duas variáveis `x` e `y` através de uma constante auxiliar.

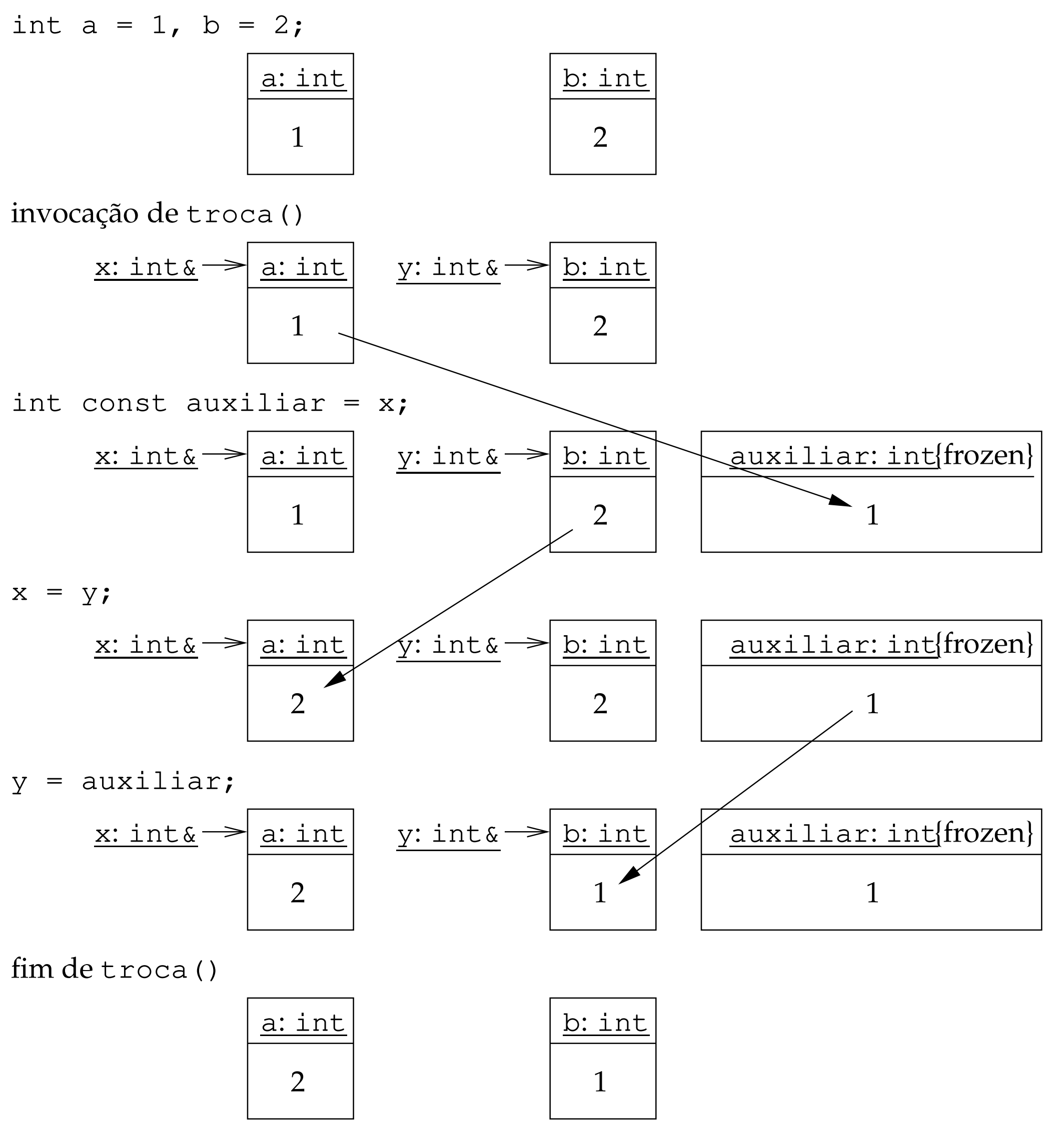


Figura 3.2: Diagramas com evolução do estado do programa que invoca o procedimento `troca()` entre cada instrução.

1. Os parâmetros x e y tornam-se sinónimos (referências) das variáveis a e b. Aqui não é feita a cópia dos valores de a e b para x e y. O que acontece é que os parâmetros x e y passam a referir-se às mesmas posições de memória onde estão guardadas as variáveis a e b. Ao processo de equiparação de um parâmetro ao argumento respectivo passado por referência chama-se também inicialização.

2. No corpo do procedimento o valor que está guardado em x é trocado com o valor guardado em y. Dado que x se refere à mesma posição de memória que a e y à mesma posição de memória que b, uma vez que são sinónimos, ao fazer esta operação está-se efectivamente a trocar os valores das variáveis a e b.

3. Quando termina a execução da função são destruídos os sinónimos x e y das variáveis a e b (que permanecem intactas), ficando os valores destas trocados.

Como só podem existir sinónimos/referências de entidades que, tal como as variáveis, têm posições de memória associadas, a chamada

```
troca(20, a + b);
```

não faz qualquer sentido e conduz a dois erros de compilação.

É de notar que a utilização de passagens por referência deve ser evitada a todo o custo em funções, pois levariam à ocorrência de efeitos laterais nas expressões onde essas funções fossem chamadas. Isto evita situações como

```
int incrementa(int& valor)
{
    return valor = valor + 1;
}

int main()
{
    int i = 0;
    cout << i + incrementa(i) << endl;
}
```

em que o resultado final tanto pode ser aparecer 1 como aparecer 2 no ecrã, dependendo da ordem de cálculo dos operandos da adição. Ou seja, funções com parâmetros que são referências são meio caminho andado para instruções mal comportadas, que se discutiram na Secção 2.7.8.

Assim, as passagens por referência só se devem usar em procedimentos e mesmo aí com parcimónia. Mais tarde ver-se-á que existe o conceito de passagem por referência constante que permite aliviar um pouco esta recomendação (ver Secção 5.2.11).

Uma observação atenta do programa para cálculo das fracções desenvolvido mostra que este contém instruções repetidas, que mereciam ser encapsuladas num procedimento: são as instruções de redução das fracções aos termos mínimos, identificadas abaixo em negrito:

```
#include <iostream>

using namespace std;

/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre  PC ≡ 0 < m ∧ 0 < n.
    @post CO ≡ mdc = mdc(m, n). Assume-se que m e n não mudam de valor. */
int mdc(int m, int n)
{
    int k;
    if(m < n)
        k = m;
    else
        k = n;

    while(m % k != 0 or n % k != 0)
        --k;

    return k;
}

/** Escreve no ecrã uma fracção, no formato usual, que lhe é passada na forma de dois
    argumentos inteiros positivos.
    @pre  PC ≡ V (ou seja, nenhuma pré-condição).
    @post CO ≡ o ecrã contém n/d em que n e d são os valores de n e d
               representados em base decimal. */
void escreveFracção(int n, int d)
{
    cout << n << '/' << d;
}

/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
{
    // Leitura das fracções do teclado:
    cout << "Introduza duas fracções: ";
    int n1, d1, n2, d2;
    cin >> n1 >> d1 >> n2 >> d2;
    int k = mdc(n1, d1);
    n1 /= k;
    d1 /= k;
    k = mdc(n2, d2);
    n2 /= k;
    d2 /= k;
```

```
    // Cálculo da fracção soma em termos mínimos:
    int n = n1 * d2 + n2 * d1;
    int d = d1 * d2;
    k = mdc(n, d);
    n /= k;
    d /= k;

    // Escrita do resultado:
    cout << "A soma de ";
    escreveFracção(n1, d1);
    cout << " com ";
    escreveFracção(n2, d2);
    cout << " é ";
    escreveFracção(n, d);
    cout << '.' << endl;
}
```

É necessário, portanto, definir um procedimento que reduza uma fracção passada como argumento na forma de dois inteiros: numerador e denominador. Não é possível escrever uma função para este efeito, pois seriam necessárias duas saídas, ou dois valores de devolução, o que as funções em C++ não permitem. Assim sendo, usa-se um procedimento que tem de ser capaz de afectar os valores dos argumentos. Ou seja, usa-se passagem de argumentos por referência. O procedimento é então:

```
/** Reduz a fracção passada com argumento na forma de dois inteiros positivos.
    @pre  PC ≡ n = n ∧ d = d ∧ 0 < n ∧ 0 < d
    @post CO ≡ n/d = n/d ∧ mdc(n, d) = 1 */
void reduzFracção(int& n, int& d)
{
    int const k = mdc(n, d);
    n /= k;
    d /= k;
}
```

Note-se que se usaram as variáveis matemáticas n e d para representar os valores iniciais das variáveis do programa `n` e `d`.

O programa completo é então:

```
#include <iostream>

using namespace std;

/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
```

```
    @pre    PC ≡ 0 < m ∧ 0 < n.
    @post   CO ≡ mdc = mdc(m, n). Assume-se que m e n não mudam de valor. */
int mdc(int m, int n)
{
    int k;
    if(m < n)
        k = m;
    else
        k = n;

    while(m % k != 0 or n % k != 0)
        --k;

    return k;
}

/** Reduz a fracção passada com argumento na forma de dois inteiros positivos.
    @pre    PC ≡ n = n ∧ d = d ∧ 0 < n ∧ 0 < d
    @post   CO ≡ n/d = n/d ∧ mdc(n, d) = 1 */
void reduzFracção(int& n, int& d)
{
    int const k = mdc(n, d);
    n /= k;
    d /= k;
}

/** Escreve no ecrã uma fracção, no formato usual, que lhe é passada na forma de dois
    argumentos inteiros positivos.
    @pre    PC ≡ V (ou seja, nenhuma pré-condição).
    @post   CO ≡ o ecrã contém n/d em que n e d são os valores de n e d
                 representados em base decimal. */
void escreveFracção(int n, int d)
{
    cout << n << '/' << d;
}

/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
{
    // Leitura das fracções do teclado:
    cout << "Introduza duas fracções: ";
    int n1, d1, n2, d2;
    cin >> n1 >> d1 >> n2 >> d2;
    reduzFracção(n1, d1);
    reduzFracção(n2, d2);
```

```
        // Cálculo da fracção soma em termos mínimos:
        int n = n1 * d2 + n2 * d1;
        int d = d1 * d2;
        reduzFracção(n, d);

        // Escrita do resultado:
        cout << "A soma de ";
        escreveFracção(n1, d1);
        cout << " com ";
        escreveFracção(n2, d2);
        cout << " é ";
        escreveFracção(n, d);
        cout << '.' << endl;
    }
```

3.2.12 Variáveis locais e globais

Uma observação cuidadosa dos exemplos anteriores revela que afinal `main()` é uma função. Mas é uma função especial: é no seu início que começa a execução do programa.

Assim sendo, verifica-se também que até agora só se definiram variáveis dentro de rotinas. Às variáveis que se definem no corpo de rotinas chama-se *variáveis locais*. As variáveis locais podem ser definidas em qualquer ponto de uma rotina onde possa estar uma instrução. Às variáveis que se definem fora de qualquer rotina chama-se *variáveis globais*. Os mesmos nomes se aplicam no caso das constantes: há constantes locais e constantes globais.

Os parâmetros de uma rotina são variáveis locais como quaisquer outras, excepto quanto à sua forma de inicialização: os parâmetros são inicializados implicitamente com o valor dos argumentos respectivos em cada invocação da rotina.

3.2.13 Blocos de instruções ou instruções compostas

Por vezes é conveniente agrupar um conjunto de instruções e tratá-las como uma única instrução. Para isso envolvem-se as instruções entre `{}`. Por exemplo, no código

```
double raio1, raio2;
...
if(raio1 < raio2) {
    double const aux = raio1;
    raio1 = raio2;
    raio2 = aux;
}
```

as três instruções

```
double const aux = raio1;
raio1 = raio2;
raio2 = aux;
```

estão agrupadas num único *bloco de instruções*, ou numa única *instrução composta*, com execução dependente da veracidade de uma condição. Um outro exemplo simples de um bloco de instruções é o corpo de uma rotina.

Os blocos de instruções podem estar embutidos (ou aninhados) dentro de outros blocos de instruções. Por exemplo, no programa

```
int main()
{
    int n;
    cin >> n;
    if(n < 0) {
        cout << "Valor negativo! Usando o módulo!”;
        n = -n;
    }
    cout << n << endl;
}
```

existem dois blocos de instruções: o primeiro corresponde ao corpo da função `main()` e o segundo à sequência de instruções executada condicionalmente de acordo com o valor de `n`. O segundo bloco de instruções encontra-se embutido no primeiro.

Cada variável tem um contexto de definição. As variáveis globais são definidas no contexto do programa[6] e as variáveis locais no contexto de um bloco de instruções. Para todos os efeitos, os parâmetros de uma rotina pertencem ao contexto do bloco de instruções correspondente ao corpo da rotina.

3.2.14 Âmbito ou visibilidade de variáveis

Cada variável tem um âmbito de visibilidade, determinado pelo contexto no qual foi definida. As variáveis globais são visíveis (isto é, utilizáveis em expressões) desde a sua declaração até ao final do ficheiro (ver-se-á no Capítulo 9 que um programa pode consistir de vários ficheiros). As variáveis locais, por outro lado, são visíveis desde o ponto de definição até à chaveta de fecho do bloco de instruções onde foram definidas.

Por exemplo:

```
#include <iostream>
```

[6]Note-se que as variáveis globais também podem ser definidas no contexto do ficheiro, bastando para isso preceder a sua definição do qualificador `static`. Este assunto será clarificado quando se discutir a divisão de um programa em ficheiros no Capítulo 9.

```
using namespace std;

double const pi = 3.1416;

/** Devolve o perímetro de uma circunferência de raio r.
    @pre  PC ≡ 0 ≤ raio.
    @post CO ≡ perímetro = 2 × π × raio. */
double perímetro(double raio)
{
    return 2.0 * pi * raio;
}

int main()
{
    cout << "Introduza dois raios: ";
    double raio1, raio2;
    cin >> raio1 >> raio2;

    // Ordenação dos raios (por ordem crescente):
    if(raio1 < raio2) {
        double const aux = raio1;
        raio1 = raio2;
        raio2 = aux;
    }

    // Escrita do resultado:
    cout << "raio = " << raio1 << ", perímetro = "
                                << perímetro(raio1) << endl
         << "raio = " << raio2 << ", perímetro = "
                                << perímetro(raio2) << endl;
}
```

Neste código:

1. A constante `pi` é visível desde a sua definição até ao final do corpo da função `main()`.
2. O parâmetro `raio` (que é uma variável local a `perímetro`()), é visível em todo o corpo da função `perímetro()`.
3. As variáveis `raio1` e `raio2` são visíveis desde o ponto de definição (antes da operação de extracção) até ao final do corpo da função `main()`.
4. A constante `aux` é visível desde o ponto de definição até ao fim da instrução composta controlada pelo `if`.

Quanto mais estreito for o âmbito de visibilidade de uma variável, menores os danos causados por possíveis utilizações erróneas. Assim, as variáveis locais devem definir-se tanto quanto possível imediatamente antes da primeira utilização.

Em cada contexto só pode ser definida uma variável com o mesmo nome. Por exemplo:

```
{
    int j;
    int k;
    ...
    int j;    // erro! j definida pela segunda vez!
    float k; // erro! k definida pela segunda vez!
}
```

Por outro lado, o mesmo nome pode ser reutilizado em contextos diferentes. Por exemplo, no programa da soma de fracções utiliza-se o nome `n` em contextos diferentes:

```
int mdc(int m, int n)
{
    ...
}

int main()
{
    ...
    int n = n1 * d2 + n2 * d1;
    ...
}
```

Em cada contexto `n` é uma variável diferente.

Quando um contexto se encontra embutido (ou aninhado) dentro de outro, as variáveis visíveis no contexto exterior são visíveis no contexto mais interior, excepto se o contexto interior definir uma variável com o mesmo nome. Neste último caso diz-se que a definição interior *oculta* a definição mais exterior. Por exemplo, no programa

```
double f = 1.0;

int main()
{
    if(...) {
        f = 2.0;
    } else {
        double f = 3.0;
        cout << f << endl;
    }
    cout << f << endl;
}
```

a variável global `f` é visível desde a sua definição até ao final do programa, incluindo o bloco de instruções após o `if` (que está embutido no corpo de `main()`), mas excluindo o bloco de instruções após o `else` (também embutido em `main()`), no qual uma outra variável como mesmo nome é definida e portanto visível.

Mesmo quando existe ocultação de uma variável global é possível utilizá-la. Para isso basta qualificar o nome da variável global com o operador de resolução de âmbito `::` aplicado ao espaço nominativo global (os espaços nominativos serão estudados na Secção 9.6.2). Por exemplo, no programa

```
double f = 1.0;
int main()
{
    if(...) {
        f = 2.0;
    } else {
        double f = ::f;
        f += 10.0;
        cout << f << endl;
    }
    cout << f << endl;
}
```

a variável `f` definida no bloco após o `else` é inicializada com o valor da variável `f` global.

Um dos principais problemas com a utilização de variáveis globais tem a ver com o facto de estabelecerem ligações entre os módulos (rotinas) que não são explícitas na sua interface, i.e., na informação presente no cabeçalho. Dois procedimentos podem usar a mesma variável global, ficando ligados no sentido em que a alteração do valor dessa variável por um procedimento tem efeito sobre o outro procedimento que a usa. As variáveis globais são assim uma fonte de erros, que ademais são difíceis de corrigir. O uso de variáveis globais é, por isso, fortemente desaconselhado, para não dizer proibido... Já o mesmo não se pode dizer de constantes globais, cuja utilização é fortemente aconselhada.

Outro tipo de prática pouco recomendável é o de ocultar nomes de contextos exteriores através de definições locais com o mesmo nome. Esta prática dá origem a erros de muito difícil correcção.

3.2.15 Duração ou permanência de variáveis

Quando é que as variáveis existem, i.e., têm espaço de memória reservado para elas? As variáveis globais existem sempre desde o início ao fim do programa, e por isso dizem-se *estáticas*. São construídas no início do programa e destruídas no seu final.

As variáveis locais (incluindo parâmetros de rotinas) existem em memória apenas enquanto o bloco de instruções em que estão inseridas está a ser executado, sendo assim potencialmente construídas e destruídas muitas vezes ao longo de um programa. Variáveis com estas características dizem-se *automáticas*.

As variáveis locais também podem ser estáticas, desde que se preceda a sua definição do qualificador `static`. Nesse caso são construídas no momento em que a execução passa pela primeira vez pela sua definição e são destruídas (deixam de existir) no final do programa. Este tipo de variáveis usa-se comummente como forma de definir variáveis locais que preservam o seu valor entre invocações da rotina em que estão definidas.

Inicialização

As variáveis de tipos básicos podem não ser inicializadas explicitamente. Quando isso acontece, as variáveis estáticas são inicializadas implicitamente com um valor nulo, enquanto as variáveis automáticas (por uma questão de eficiência) não são inicializadas de todo, passando portanto a conter "lixo". Recorde-se que os parâmetros, sendo variáveis locais automáticas, são sempre inicializados com o valor dos argumentos respectivos.

Sempre que possível deve-se inicializar explicitamente as variáveis com valores apropriados. Mas nunca se deve inicializar "com qualquer coisa" só para o compilador "não chatear".

3.2.16 Nomes de rotinas

Tal como no caso das variáveis, o nome das funções e dos procedimentos deverá reflectir claramente aquilo que é calculado ou aquilo que é feito, respectivamente. Assim, as funções têm tipicamente o nome da entidade calculada (e.g., `seno()`, `co_seno()`, `comprimento()`) enquanto os procedimentos têm normalmente como nome a terceira pessoa do singular do imperativo do verbo indicador da acção que realizam, possivelmente seguido de complementos (e.g., `acrescenta()`, `copiaSemDuplicações()`). Só se devem usar abreviaturas quando forem bem conhecidas, tal como é o caso em `mdc()`.

Uma excepção a estas regras dá-se para funções cujo resultado é um valor lógico ou booleano. Nesse caso o nome da função deve ser um predicado, sendo o sujeito um dos argumentos da função[7], de modo que a frase completa seja uma proposição verdadeira ou falsa. Por exemplo, `estáVazia(fila)`.

Os nomes utilizados para variáveis, funções e procedimentos (e em geral para qualquer outro identificador criado pelo programador), devem ser tais que a leitura do código se faça da forma mais simples possível, quase como se de português se tratasse.

Idealmente os procedimentos têm um único objectivo, sendo por isso descritíveis usando apenas um verbo. Quando a descrição rigorosa de um procedimento obrigar à utilização de dois ou mais verbos, isso indicia que o procedimento tem mais do que um objectivo, sendo por isso um fortíssimo candidato a ser dividido em dois ou mais procedimentos.

A linguagem C++ é imperativa, i.e., os programas consistem em sequências de instruções. Assim, o corpo de uma função diz como se calcula qualquer coisa, mas não diz *o que* se calcula. É de toda a conveniência que, usando as regras indicadas, esse *o que* fique o mais possível explícito no nome da função, passando-se o mesmo quanto aos procedimentos. Isto, claro está,

[7] No caso de métodos de classes, ver Capítulo 7, o sujeito é o objecto em causa, ou seja, o objecto para o qual o método foi invocado.

não excluindo a necessidade de comentar funções e procedimentos com as respectivas PC e CO.

Finalmente, é de toda a conveniência que se use um estilo de programação uniforme. Tal facilita a compreensão do código escrito por outros programadores ou pelo próprio programador depois de passados uns meses sobre a escrita do código. Assim, sugerem-se as seguintes regras adicionais:

- Os nomes de variáveis e constantes devem ser escritos em minúsculas usando-se o sublinhado _ para separar as palavras. Por exemplo:

```
int const máximo_de_alunos_por_turma = 50;
int alunos_na_turma = 10;
```

- Os nomes de funções ou procedimentos devem ser escritos em minúsculas usando-se letras maiúsculas iniciais em todas as palavras excepto a primeira:

```
int númeroDeAlunos();
```

Recomendações mais gerais sobre nomes e formato de nomes em C++ podem ser encontradas no Apêndice D.

3.2.17 Declaração vs. definição

Antes de se poder invocar uma rotina é necessário que esta seja declarada, i.e., que o compilador saiba que ela existe e qual a sua interface. Declarar uma rotina consiste pois em dizer qual o seu nome, qual o tipo do valor devolvido, quantos parâmetros tem e de que tipo são esses parâmetros. Ou seja, declarar consiste em especificar o cabeçalho da rotina. Por exemplo,

```
void imprimeValorLógico(bool b);
```

ou simplesmente, visto que o nome dos parâmetros é irrelevante numa declaração,

```
void imprimeValorLógico(bool);
```

são possíveis declarações da função `imprimeValorLógico()` que se define abaixo:

```
/** Imprime “verdadeiro” ou “falso” consoante o valor lógico do argumento.
    @pre   PC ≡ V.
    @post  CO ≡ surge escrito no ecrã “verdadeiro” ou “falso” conforme
              o valor de b. */
void imprimeValorLógico(bool b)
{
    if(b)
        cout << "verdadeiro";
    else
        cout << "falso";
}
```

Como se viu, na declaração não é necessário indicar os nomes dos parâmetros. Na prática é conveniente fazê-lo para mostrar claramente ao leitor o significado das entradas da função.

A sintaxe das declarações em sentido estrito é simples: o cabeçalho da rotina (como na definição) mas seguido de `;` em vez do corpo. O facto de uma rotina estar declarada não a dispensa de ter de ser definida mais cedo ou mais tarde: todas as rotinas têm de estar definidas em algum lado. Por outro lado, uma definição, um vez que contém o cabeçalho da rotina, também serve de declaração. Assim, após uma definição, as declarações em sentido estrito são desnecessárias.

Note-se que, mesmo que já se tenha procedido à declaração prévia de uma rotina, é necessário voltar a incluir o cabeçalho na definição.

O facto de uma definição ser também uma declaração foi usado no programa da soma de fracções para, colocando as definições das rotinas antes da função `main()`, permitir que elas fossem usadas no corpo da função `main()`. É possível inverter a ordem das definições através de declarações prévias:

```
#include <iostream>

using namespace std;

/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
{
    // Declaração dos procedimentos necessários. Estas declarações são visíveis
    // apenas dentro da função main().
    void reduzFracção(int& n, int& d);
    void escreveFracção(int n, int d);

    // Leitura das fracções do teclado:
    cout << "Introduza duas fracções: ";
    int n1, d1, n2, d2;
    cin >> n1 >> d1 >> n2 >> d2;
    reduzFracção(n1, d1);
    reduzFracção(n2, d2);

    // Cálculo da fracção soma em termos mínimos:
    int n = n1 * d2 + n2 * d1;
    int d = d1 * d2;
    reduzFracção(n, d);

    // Escrita do resultado:
    cout << "A soma de ";
    escreveFracção(n1, d1);
    cout << " com ";
    escreveFracção(n2, d2);
```

```
    cout << " é ";
    escreveFracção(n, d);
    cout << '.' << endl;
}
```

/** Reduz a fracção passada com argumento na forma de dois inteiros positivos.
@pre $PC \equiv \mathtt{n} = n \wedge \mathtt{d} = d \wedge 0 < n \wedge 0 < d$
@post $CO \equiv \frac{\mathtt{n}}{\mathtt{d}} = \frac{n}{d} \wedge \mathrm{mdc}(\mathtt{n}, \mathtt{d}) = 1$ */

```
void reduzFracção(int& n, int& d)
{
    // Declaração da função necessária. Esta declaração é visível apenas
    // dentro desta função.
    int mdc(int m, int n);

    int const k = mdc(n, d);
    n /= k;
    d /= k;
}
```

/** Escreve no ecrã uma fracção, no formato usual, que lhe é passada na forma de dois argumentos inteiros positivos.
@pre $PC \equiv \mathcal{V}$ (ou seja, nenhuma pré-condição).
@post $CO \equiv$ o ecrã contém `n/d` em que `n` e `d` são os valores de `n` e `d` representados em base decimal. */

```
void escreveFracção(int n, int d)
{
    cout << n << '/' << d;
}
```

/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados como argumentos.
@pre $PC \equiv 0 < \mathtt{m} \wedge 0 < \mathtt{n}$.
@post $CO \equiv \mathtt{mdc} = \mathrm{mdc}(\mathtt{m}, \mathtt{n})$. Assume-se que `m` e `n` não mudam de valor. */

```
int mdc(int m, int n)
{
    int k;
    if(m < n)
        k = m;
    else
        k = n;

    while(m % k != 0 or n % k != 0)
        --k;

    return k;
}
```

A vantagem desta disposição é que aparecem primeiro as rotinas mais globais e só mais tarde os pormenores, o que facilita a leitura do código.

Poder-se-ia alternativamente ter declarado as rotinas fora das funções e procedimentos em que são necessários. Nesse caso o programa seria:

```
#include <iostream>

using namespace std;

/** Reduz a fracção passada com argumento na forma de dois inteiros positivos.
    @pre  PC ≡ n = n ∧ d = d ∧ 0 < n ∧ 0 < d
    @post CO ≡ n/d = n/d ∧ mdc(n, d) = 1 */
void reduzFracção(int& n, int& d);

/** Escreve no ecrã uma fracção, no formato usual, que lhe é passada na forma de dois
    argumentos inteiros positivos.
    PC ≡ V (ou seja, nenhuma pré-condição).
    CO ≡ o ecrã contém n/d em que n e d são os valores de n e d
         representados em base decimal. */
void escreveFracção(int n, int d);

/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre  PC ≡ 0 < m ∧ 0 < n.
    @post CO ≡ mdc = mdc(m, n). Assume-se que m e n não mudam de valor. */
int mdc(int m, int n);

/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
{
    // Leitura das fracções do teclado:
    cout << "Introduza duas fracções: ";
    int n1, d1, n2, d2;
    cin >> n1 >> d1 >> n2 >> d2;
    reduzFracção(n1, d1);
    reduzFracção(n2, d2);

    // Cálculo da fracção soma em termos mínimos:
    int n = n1 * d2 + n2 * d1;
    int d = d1 * d2;
    reduzFracção(n, d);

    // Escrita do resultado:
    cout << "A soma de ";
    escreveFracção(n1, d1);
```

```
        cout << " com ";
        escreveFracção(n2, d2);
        cout << " é ";
        escreveFracção(n, d);
        cout << '.' << endl;
    }

    void reduzFracção(int& n, int& d)
    {
        int const k = mdc(n, d);
        n /= k;
        d /= k;
    }

    void escreveFracção(int n, int d)
    {
        cout << n << '/' << d;
    }

    int mdc(int m, int n)
    {
        int k;
        if(m < n)
            k = m;
        else
            k = n;

        while(m % k != 0 or n % k != 0)
            --k;

        return k;
    }
```

Esta disposição é mais usual que a primeira. Repare-se que neste caso a documentação das rotinas aparece junto com a sua declaração. É que essa documentação é fundamental para saber o que a rotina faz, e portanto faz parte da interface da rotina. Como uma declaração é a especificação completa da interface, é natural que seja a declaração a ser documentada, e não a definição.

3.2.18 Parâmetros constantes

É comum que os parâmetros de uma rotina não mudem de valor durante a sua execução. Nesse caso é bom hábito explicitá-lo, tornando os parâmetros constantes. Dessa forma, enganos do programador serão assinalados prontamente: se alguma instrução da rotina tentar alterar o valor de um parâmetro constante, o compilador assinalará o erro. O mesmo argumento pode ser aplicado não só a parâmetros mas a qualquer variável: se não é suposto que o seu valor mude depois de inicializada, então deveria ser uma constante, e não uma variável.

No entanto, do ponto de vista do consumidor de uma rotina, a constância de um parâmetro correspondente a uma passagem de argumentos por valor é perfeitamente irrelevante: para o consumidor da rotina é suficiente saber que ela não alterará o argumento. A alteração ou não do parâmetro é um pormenor de implementação, e por isso importante apenas para o produtor da rotina. Assim, é comum indicar-se a constância de parâmetros associados a passagens de argumentos por valor *apenas na definição das rotinas e não na sua declaração*.

Estas ideias aplicadas ao programa da soma de fracções conduzem a

```
#include <iostream>

using namespace std;

/** Reduz a fracção passada com argumento na forma de dois inteiros positivos.
    @pre  PC ≡ n = n ∧ d = d ∧ 0 < n ∧ 0 < d
    @post CO ≡ n/d = n/d ∧ mdc(n, d) = 1 */
void reduzFracção(int& n, int& d);

/** Escreve no ecrã uma fracção, no formato usual, que lhe é passada na forma de dois
    argumentos inteiros positivos.
    @pre  PC ≡ V (ou seja, nenhuma pré-condição).
    @post CO ≡ o ecrã contém n/d em que n e d são os valores de n e d
               representados em base decimal. */
void escreveFracção(int n, int d);

/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre  PC ≡ 0 < m ∧ 0 < n.
    @post CO ≡ mdc = mdc(m, n). */
int mdc(int m, int n);

/** Devolve o menor de dois inteiros passados como argumentos.
    @pre  PC ≡ V (ou seja, nenhuma pré-condição).
    @post CO ≡ (mínimo = a ∧ a ≤ b) ∨ (mínimo = b ∧ b ≤ a). */
int mínimo(int a, int b);

/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
{
    // Leitura das fracções do teclado:
    cout << "Introduza duas fracções: ";
    int n1, d1, n2, d2;
    cin >> n1 >> d1 >> n2 >> d2;
```

```
    reduzFracção(n1, d1);
    reduzFracção(n2, d2);

    // Cálculo da fracção soma em termos mínimos:
    int n = n1 * d2 + n2 * d1;
    int d = d1 * d2;
    reduzFracção(n, d);

    // Escrita do resultado:
    cout << "A soma de ";
    escreveFracção(n1, d1);
    cout << " com ";
    escreveFracção(n2, d2);
    cout << " é ";
    escreveFracção(n, d);
    cout << '.' << endl;
}

void reduzFracção(int& n, int& d)
{
    int const k = mdc(n, d);
    n /= k;
    d /= k;
}

void escreveFracção(int const n, int const d)
{
    cout << n << '/' << d;
}

int mdc(int const m, int const n)
{
    int k = mínimo(m, n);

    while(m % k != 0 or n % k != 0)
        --k;

    return k;
}

int mínimo(int const a, int const b)
{
    if(a < b)
        return a;
    else
        return b;
```

```
}
```

Repare-se que se aproveitou para melhorar a modularização do programa acrescentando uma função para calcular o mínimo de dois valores.

3.2.19 Instruções de asserção

Que deve suceder quando o contrato de uma rotina é violado? A violação de um contrato decorre sempre, sem excepção, de um *erro de programação*. É muito importante perceber-se que assim é.

Em primeiro lugar, tem de se distinguir claramente os papéis dos vários intervenientes no processo de escrita e execução de um programa:

[programador] produtor é aquele que escreveu uma ferramenta, tipicamente um módulo (e.g., uma rotina).

[programador] consumidor é aquele que usa uma ferramenta, tipicamente um módulo (e.g., uma rotina), com determinado objectivo.

utilizador do programa é aquele que faz uso do programa.

A responsabilidade pela violação de um contrato nunca é do utilizador do programa. A responsabilidade é sempre de um programador. Se a pré-condição de uma rotina for violada, a responsabilidade é do programador consumidor da rotina. Se a condição objectivo de uma rotina for violada, e admitindo que a respectiva pré-condição não o foi, a responsabilidade é do programador fabricante dessa rotina.

Se uma pré-condição de uma rotina for violada, o contrato assinado entre produtor e consumidor não é válido, e portanto o produtor é livre de escolher o que a rotina faz nessas circunstâncias. Pode fazer o que entender, desde devolver lixo (no caso de uma função), a apagar o disco rígido e escrever uma mensagem perversa no ecrã: tudo é válido.

Claro está que isso não é desejável. O ideal seria que, se uma pré-condição ou uma condição objectivo falhassem, esse erro fosse assinalado claramente. No Capítulo 14 ver-se-á que o mecanismo das excepções é o mais adequado para lidar com este tipo de situações. Para já, à falta de melhor, optar-se-á por abortar imediatamente a execução do programa escrevendo-se uma mensagem de erro apropriada no ecrã.

Considere-se de novo a função para cálculo do máximo divisor comum:

```
/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre  PC ≡ 0 < m ∧ 0 < n.
    @post CO ≡ mdc = mdc(m, n). */
int mdc(int const m, int const n);
{
```

```
        int k = mínimo(m, n);

        while(m % k != 0 or n % k != 0)
            --k;

        return k;
    }
```

Que sucede se a pré-condição for violada? Suponha-se que a função é invocada com argumentos 10 e -6. Então a variável `k` será inicializada com o valor -6. A guarda do ciclo é inicialmente verdadeira, pelo que o valor de `k` passa para -7. Para este valor de `k` a guarda será também verdadeira. E sê-lo-á também para -8, etc. Tem-se portanto um ciclo infinito? Não exactamente. Como se viu no Capítulo 2, as variáveis de tipos inteiros guardam uma gama de valores limitados. Quando se atingir o limite inferior dessa gama, o valor de `k` passará para o maior inteiro representável. Daí descerá, lentamente, inteiro por inteiro, até ao valor 2, que levará finalmente à falsidade da guarda, à consequente terminação do ciclo, e à devolução do valor correcto! Isso ocorrerá muito, mesmo muito tempo depois de chamada a função, visto que exigirá exactamente 4294967288 iterações do ciclo numa máquina onde os inteiros tenham 32 *bits*...

É certamente curioso este resultado. Mesmo que a pré-condição seja violada, o algoritmo, em algumas circunstâncias, devolve o resultado correcto. Há duas lições a tirar deste facto:

1. A função, tal como definida, é demasiado restritiva. Uma vez que faz sentido calcular o máximo divisor comum de quaisquer inteiros, mesmo negativos, desde que não sejam ambos nulos, a função deveria tê-lo previsto desde o início e a pré-condição deveria ter sido consideravelmente relaxada. Isso será feito mais abaixo.

2. Se o contrato é violado qualquer coisa pode acontecer, incluindo uma função devolver o resultado correcto... Quando isso acontece há normalmente uma outra característica desejável que não se verifica. Neste caso é a eficiência.

Claro está que nem sempre a violação de um contrato leva à devolução do valor correcto. Aliás, isso raramente acontece. Repare-se no que acontece quando se invoca a função `mdc()` com argumentos 0 e 10, por exemplo. Nesse caso o valor inicial de `k` é 0, o que leva a que a condição da instrução iterativa `while` tente calcular uma divisão por zero. Logo que isso sucede o programa aborta com uma mensagem de erro pouco simpática e semelhante à seguinte:

```
Floating exception (core dumped)
```

Existe uma forma padronizada de explicitar as condições que devem ser verdadeiras nos vários locais de um programa, obrigando o computador a verificar a sua validade durante a execução do programa: as chamadas instruções de asserção (afirmação). Para se poder usar instruções de asserção tem de ser incluir o ficheiro de interface apropriado:

```
#include <cassert>
```

As instruções de asserção têm o formato

```
assert(condição);
```

onde *condição* é uma condição que deve ser verdadeira no local onde a instrução se encontra para que o programa se possa considerar correcto. Se a condição for verdadeira quando a instrução de asserção é executada, nada acontece: a instrução não tem qualquer efeito. Mas se a condição for falsa o programa aborta e é mostrada uma mensagem de erro com o seguinte aspecto:

```
ficheiro_executável: ficheiro_fonte:linha: cabeçalho: Assertion ‘condição’ failed.
```

onde:

ficheiro_executável Nome do ficheiro executável onde ocorreu o erro.

ficheiro_fonte Nome do ficheiro em linguagem C++ onde ocorreu o erro.

linha Número da linha do ficheiro C++ onde ocorreu o erro.

cabeçalho Cabeçalho da função ou procedimento onde ocorreu o erro (só em alguns compiladores).

condição Condição que deveria ser verdadeira mas não o era.

Deve-se usar uma instrução de asserção para verificar a veracidade da pré-condição da função `mdc()`:

```
/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre   PC ≡ 0 < m ∧ 0 < n.
    @post  CO ≡ mdc = mdc(m, n). */
int mdc(int const m, int const n);
{
    assert(0 < m and 0 < n);

    int k = mínimo(m, n);

    while(m % k != 0 or n % k != 0)
        --k;

    return k;
}
```

Suponha-se o seguinte programa usando a função `mdc()`:

```
#include <iostream>
#include <cassert>

using namespace std;

/** Devolve o menor de dois inteiros passados como argumentos.
    @pre  PC ≡ V (ou seja, nenhuma pré-condição).
    @post CO ≡ (mínimo = a ∧ a ≤ b) ∨ (mínimo = b ∧ b ≤ a). */
int mínimo(int const a, int const b)
{
    if(a < b)
        return a;
    else
        return b;
}

/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre  PC ≡ 0 < m ∧ 0 < n.
    @post CO ≡ mdc = mdc(m, n). */
int mdc(int const m, int const n)
{
    assert(0 < m and 0 < n);

    int k = mínimo(m, n);

    while(m % k != 0 or n % k != 0)
        --k;

    return k;
}

int main()
{
    cout << mdc(0, 10) << endl;
}
```

A execução deste programa leva à seguinte mensagem de erro:

```
teste: teste.C:23: int mdc(int, int): Assertion `0 < m and 0 < n' failed.
Abort (core dumped)
```

É claro que este tipo de mensagens é muito mais útil para o programador que o simples abortar do programa ou, pior, a produção pelo programa de resultados errados.

Suponha-se agora o programa original, para o cálculo da soma de fracções, mas já equipado com instruções de asserção verificando as pré-condições dos vários módulos que o compõem (exceptuando os que não impõem qualquer pré-condição):

```
#include <iostream>

using namespace std;

/** Reduz a fracção passada com argumento na forma de dois inteiros positivos.
    @pre  PC ≡ n = n ∧ d = d ∧ 0 < n ∧ 0 < d
    @post CO ≡ n/d = n/d ∧ mdc(n, d) = 1 */
void reduzFracção(int& n, int& d);

/** Escreve no ecrã uma fracção, no formato usual, que lhe é passada na forma de dois
    argumentos inteiros positivos.
    @pre  PC ≡ V (ou seja, nenhuma pré-condição).
    @post CO ≡ o ecrã contém n/d em que n e d são os valores de n e d
               representados em base decimal. */
void escreveFracção(int n, int d);

/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre  PC ≡ 0 < m ∧ 0 < n.
    @post CO ≡ mdc = mdc(m, n). */
int mdc(int m, int n);

/** Devolve o menor de dois inteiros passados como argumentos.
    @pre  PC ≡ V (ou seja, nenhuma pré-condição).
    @post CO ≡ (mínimo = a ∧ a ≤ b) ∨ (mínimo = b ∧ b ≤ a). */
int mínimo(int a, int b);

/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
{
    // Leitura das fracções do teclado:
    cout << "Introduza duas fracções: ";
    int n1, d1, n2, d2;
    cin >> n1 >> d1 >> n2 >> d2;
    reduzFracção(n1, d1);
    reduzFracção(n2, d2);

    // Cálculo da fracção soma em termos mínimos:
    int n = n1 * d2 + n2 * d1;
    int d = d1 * d2;
    reduzFracção(n, d);
```

```
        // Escrita do resultado:
        cout << "A soma de ";
        escreveFracção(n1, d1);
        cout << " com ";
        escreveFracção(n2, d2);
        cout << " é ";
        escreveFracção(n, d);
        cout << '.' << endl;
    }

    void reduzFracção(int& n, int& d)
    {
        assert(0 < n and 0 < d);

        int const k = mdc(n, d);
        n /= k;
        d /= k;
    }

    void escreveFracção(int const n, int const d)
    {
        cout << n << '/' << d;
    }

    int mdc(int const m, int const n)
    {
        assert(0 < m and 0 < n);

        int k = mínimo(m, n);

        while(m % k != 0 or n % k != 0)
            --k;

        return k;
    }

    int mínimo(int const a, int const b)
    {
        if(a < b)
            return a;
        else
            return b;
    }
```

Que há de errado com este programa? Considere-se o que acontece se o seu utilizador introduzir fracções negativas, por exemplo:

```
-6 7 15 7
```

Neste caso o programa aborta com uma mensagem de erro porque a pré-condição do procedimento `reduzFracção()` foi violada. Um programa não deve abortar nunca. Nunca mesmo. De quem é a responsabilidade disso acontecer neste caso? Do utilizador do programa, que desobedeceu introduzindo fracções negativas apesar de instruído para não o fazer, ou do programador produtor da função `main()` e consumidor do procedimento `reduzFracção()`? A resposta correcta é a segunda: a culpa nunca é do utilizador, é do programador. Isto tem de ficar absolutamente claro: o utilizador tem sempre razão.

Como resolver o problema? Há duas soluções. A primeira diz que deve ser o programador consumidor a garantir que os valores passados nos argumentos de `reduzFracção()` têm de ser positivos, conforme se estabeleceu na sua pré-condição. Em geral é este o caminho certo, embora neste caso se possa olhar para o problema com um pouco mais de cuidado e reconhecer que a redução de fracções só deveria ser proibida se o denominador fosse nulo. Isso implica, naturalmente, refazer o procedimento `reduzFracção()`, relaxando a sua pré-condição. O que ressalta daqui é que quanto mais leonina (forte) uma pré-condição, menos trabalho tem o programador produtor do módulo e mais trabalho tem o programador consumidor. Pelo contrário, se a pré-condição for fraca, isso implica mais trabalho para o produtor e menos para o consumidor.

Em qualquer dos casos, continua a haver circunstâncias nas quais as pré-condições do procedimento podem ser violadas. É sempre responsabilidade do programador consumidor garantir que isso não acontece. A solução mais simples seria usar um ciclo para pedir de novo ao utilizador para introduzir as fracções enquanto estas não verificassem as condições pretendidas. Tal solução não será ensaiada aqui, por uma questão de simplicidade. Adoptar-se-á a solução algo simplista de terminar o programa sem efectuar os cálculos no caso de haver problemas com os valores das fracções:

```
int main()
{
    // Leitura das fracções do teclado:
    cout << "Introduza duas fracções: ";
    int n1, d1, n2, d2;
    cin >> n1 >> d1 >> n2 >> d2;

    if(n1 < 0 or d1 < 0 or n2 < 0 or d2 < 0)
        cout << "Termos negativos.  Nada feito." << endl;
    else {
        reduzFracção(n1, d1);
        reduzFracção(n2, d2);

        // Cálculo da fracção soma em termos mínimos:
        int n = n1 * d2 + n2 * d1;
        int d = d1 * d2;
```

```
            reduzFracção(n, d);

            // Escrita do resultado:
            cout << "A soma de ";
            escreveFracção(n1, d1);
            cout << " com ";
            escreveFracção(n2, d2);
            cout << " é ";
            escreveFracção(n, d);
            cout << '.' << endl;
        }
    }
```

Neste caso é evidente que não haverá violação de nenhuma pré-condição. Muitos terão a tentação de perguntar para que servem as asserções neste momento, e se não seria apropriado eliminá-las. Há várias razões para as asserções continuarem a ser indispensáveis:

1. O programador, enquanto produtor, não deve assumir nada acerca dos consumidores (muito embora em muitos casos produtor e consumidor sejam uma e a mesma pessoa). O melhor é mesmo colocar a asserção: o diabo tece-as.

2. O produtor deve escrever uma rotina pensando em possíveis reutilizações futuras. Pode haver erros nas futuras utilizações, pelo que o mais seguro é mesmo manter a asserção.

3. Se alguém fizer alterações no programa pode introduzir erros. A asserção nesse caso permitirá a sua rápida detecção e correcção.

Parece ter faltado algo em toda esta discussão: a condição objectivo. Tal como se deve usar asserções para verificar as pré-condições, também se deve usar asserções para verificar as condições objectivo. A falsidade de uma condição objectivo, sabendo que a respectiva pré-condição é verdadeira, é também devida a um erro de programação, só que desta vez o responsável pelo erro é o programador produtor. Assim, as asserções usadas para verificar as pré-condições servem para o produtor de uma rotina facilitar a detecção de erros do programador consumidor, enquanto as asserções usadas para verificar as condições objectivo servem para o produtor de uma rotina se proteger dos seus próprios erros.

Transformar as condições objectivo em asserções é, em geral, uma tarefa mais difícil que no caso das pré-condições, que tendem a ser mais simples. As maiores dificuldades surgem especialmente se a condição objectivo contiver quantificadores (somatórios, produtos, quaisquer que seja, existe uns, etc.), se estiverem envolvidas inserções ou extracções de canais (entradas e saídas) ou se a condição objectivo fizer menção aos valores originais das variáveis, i.e., ao valor que as variáveis possuiam no início da rotina em causa.

Considere-se cada uma das rotinas do programa.

O caso da função `mínimo()` é fácil de resolver, pois a condição objectivo traduz-se facilmente para C++. É necessário, no entanto, abandonar os retornos imediatos e guardar o valor a devolver numa variável a usar na asserção[8]:

```
/** Devolve o menor de dois inteiros passados como argumentos.
    @pre  PC ≡ V (ou seja, nenhuma pré-condição).
    @post CO ≡ (mínimo = a ∧ a ≤ b) ∨ (mínimo = b ∧ b ≤ a). */
int mínimo(int const a, int const b)
{
    int mínimo;

    if(a < b)
        mínimo = a;
    else
        mínimo = b;

    assert((mínimo == a and a <= b) or (mínimo == b and b <= a));

    return mínimo;
}
```

Quanto ao procedimento `reduzFracção()`, é fácil verificar o segundo termo da condição objectivo.

```
/** Reduz a fracção passada com argumento na forma de dois inteiros positivos.
    @pre  PC ≡ n = n ∧ d = d ∧ 0 < n ∧ 0 < d
    @post CO ≡ n/d = n/d ∧ mdc(n, d) = 1 */
void reduzFracção(int& n, int& d)
{
    assert(0 < n and 0 < d);

    int const k = mdc(n, d);
    n /= k;
    d /= k;

    assert(mdc(n, d) == 1);
}
```

Mas, e o primeiro termo, que se refere aos valores originais de `n` e `d`? Há linguagens, como Eiffel [11], nas quais as asserções correspondentes às condições objectivo podem fazer recurso aos valores das variáveis no início da respectiva rotina, usando-se para isso uma notação especial. Em C++ não é possível fazê-lo, infelizmente. Por isso o primeiro termo da condição objectivo ficará por verificar.

O procedimento `escreveFracção()` tem um problema: o seu objectivo é escrever no ecrã. Não é fácil formalizar uma condição objectivo que envolve alterações do ecrã, como se pode

[8] É comum usar-se o truque de guardar o valor a devolver por uma função numa variável com o mesmo nome da função, pois nesse caso a asserção tem exactamente o mesmo aspecto que a condição-objectivo.

ver pela utilização de português vernáculo na condição objectivo. É ainda menos fácil escrever uma instrução de asserção nessas circunstâncias. Este procedimento fica, pois, sem qualquer instrução de asserção.

Finalmente, falta a função `mdc()`. Neste caso a condição objectivo faz uso de uma função matemática mdc. Não faz qualquer sentido escrever a instrução de asserção como se segue:

```
/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre  PC ≡ 0 < m ∧ 0 < n.
    @post CO ≡ mdc = mdc(m, n). */
int mdc(int const m, int const n)
{
    assert(0 < n and 0 < d);

    int k = mínimo(m, n);

    while(m % k != 0 or n % k != 0)
        --k;

    assert(k == mdc(m, n)); // absurdo!

    return k;
}
```

Porquê? Simplesmente porque isso implica uma invocação recursiva interminável à função (ver Secção 3.3). Isso significa que se deverá exprimir a condição objectivo numa forma menos compacta:

$$CO \equiv \mathtt{m} \div \mathtt{mdc} = 0 \wedge \mathtt{n} \div \mathtt{mdc} = 0 \wedge (\mathbf{Q}\, j : \mathtt{mdc} < j : \mathtt{m} \div j \neq 0 \vee \mathtt{n} \div j \neq 0)\,.$$

Os dois primeiros termos da condição objectivo têm tradução directa para C++. Mas o segundo recorre a um quantificador que significa "qualquer que seja j maior que o valor devolvido pela função, esse j não pode ser divisor comum de `m` e `n`" (os quantificadores serão abordados mais tarde). Não há nenhuma forma simples de escrever uma instrução de asserção recorrendo a quantificadores, excepto invocando uma função que use um ciclo para verificar o valor do quantificador. Mas essa solução, além de complicada, obriga à implementação de uma função adicional, cuja condição objectivo recorre de novo ao quantificador. Ou seja, não é solução... Assim, o melhor que se pode fazer é reter os dois primeiros termos da condição objectivo reescrita:

```
/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre  PC ≡ 0 < m ∧ 0 < n.
    @post CO ≡ mdc = mdc(m, n). */
int mdc(int const m, int const n)
```

```
{
    assert(0 < n and 0 < d);

    int k = mínimo(m, n);

    while(m % k != 0 or n % k != 0)
        --k;

    assert(m % k == 0 and n % k == 0);

    return k;
}
```

Estas dificuldades não devem levar ao abandono pura e simples do esforço de expressar pré-condições e condições objectivo na forma de instruções de asserção. A vantagem das instruções de asserção por si só é enorme, além de que o esforço de as escrever exige uma completa compreensão do problema, o que leva naturalmente a menos erros na implementação da respectiva resolução.

A colocação criteriosa de instruções de asserção é, pois, um mecanismo extremamente útil para a depuração de programas. Mas tem uma desvantagem aparente: as verificações da asserções consomem tempo. Para quê, então, continuar a fazer essas verificações quando o programa já estiver liberto de erros? O mecanismo das instruções de asserção é interessante porque permite evitar esta desvantagem com elegância: basta definir uma macro de nome `NDEBUG` (*no debug*) para que as asserções deixem de ser verificadas e portanto deixem de consumir tempo, não sendo necessário apagá-las do código. As macros serão explicadas no Capítulo 9, sendo suficiente para já saber que a maior parte dos compiladores para Unix (ou Linux) permitem a definição dessa macro de uma forma muito simples: basta passar a opção `-DNDEBUG` ao compilador.

3.2.20 Melhorando módulos já produzidos

Uma das vantagens da modularização, como se viu, é que se pode melhorar a implementação de qualquer módulo sem com isso comprometer o funcionamento do sistema e sem obrigar a qualquer outra alteração. Na versão do programa da soma de fracções que se segue utiliza-se uma função de cálculo do mdc com um algoritmo diferente, mais eficiente. É o algoritmo de Euclides, que decorre naturalmente das seguintes propriedades do mdc (lembra-se das sugestões no final do Capítulo 1?):

1. $\mathrm{mdc}(m,n) = \mathrm{mdc}(n \div m, m)$ se $0 < m \wedge 0 \leq n$.
2. $\mathrm{mdc}(m,n) = n$ se $m = 0 \wedge 0 < n$.

O algoritmo usado deixa de ser uma busca exaustiva do mdc para passar a ser uma redução sucessiva do problema até à trivialidade: a aplicação sucessiva da propriedade1 vai reduzindo os valores até um deles ser zero. A demonstração da sua correcção faz-se exactamente da mesma forma que no caso da busca exaustiva, e fica como exercício para o leitor. Regresse-se a este algoritmo depois de ter lido sobre metodologias de desenvolvimentos de ciclos no Capítulo 4.

Aproveitou-se ainda para relaxar as pré-condições da função, uma vez que o algoritmo utilizado permite calcular o `mdc` de dois inteiros `m` e `n` qualquer que seja `m` desde que `n` seja positivo. Este relaxar das pré-condições permite que o programa some convenientemente fracções negativas, para o que foi apenas necessário alterar o procedimento `reduzFracção()` de modo a garantir que o denominador é sempre positivo.

```
#include <iostream>

using namespace std;

/** Reduz a fracção passada com argumento na forma de dois inteiros positivos.
    @pre  PC ≡ n = n ∧ d = d ∧ d ≠ 0
    @post CO ≡ n/d = n/d ∧ 0 < d ∧ mdc(n, d) = 1 */
void reduzFracção(int& n, int& d);

/** Escreve no ecrã uma fracção, no formato usual, que lhe é passada na forma de dois
    argumentos inteiros positivos.
    @pre  PC ≡ V (ou seja, nenhuma pré-condição).
    @post CO ≡ o ecrã contém n/d em que n e d são os valores de n e d
               representados em base decimal. */
void escreveFracção(int n, int d);

/** Calcula e devolve o máximo divisor comum de dois inteiros passados
    como argumentos (o segundo inteiro tem de ser positivo).
    @pre  PC ≡ m = m ∧ n = n ∧ 0 < n.
    @post CO ≡ mdc = mdc(|m|, n). */
int mdc(int m, int n);

/** Calcula e escreve a soma em termos mínimos de duas fracções
    lidas do teclado (os denominadores não podem ser nulos!): */
int main()
{
    // Leitura das fracções do teclado:
    cout << "Introduza duas fracções: ";
    int n1, d1, n2, d2;
    cin >> n1 >> d1 >> n2 >> d2;
    reduzFracção(n1, d1);
    reduzFracção(n2, d2);

    // Cálculo da fracção soma em termos mínimos:
    int n = n1 * d2 + n2 * d1;
    int d = d1 * d2;
    reduzFracção(n, d);
```

```
    // Escrita do resultado:
    cout << "A soma de ";
    escreveFracção(n1, d1);
    cout << " com ";
    escreveFracção(n2, d2);
    cout << " é ";
    escreveFracção(n, d);
    cout << '.' << endl;
}

void reduzFracção(int& n, int& d)
{
    assert(d != 0);

    if(d < 0) {
        n = -n;
        d = -d;
    }
    int const k = mdc(n, d);
    n /= k;
    d /= k;

    assert(0 < d and mdc(n, d) == 1);
}

void escreveFracção(int const n, int const d)
{
    cout << n << '/' << d;
}

int mdc(int m, int n)
{
    assert(0 < n);

    if(m < 0)
        m = -m;
    while(m != 0) {
        int const auxiliar = n % m;
        n = m;
        m = auxiliar;
    }

    return n;
}
```

3.3 Rotinas recursivas

O C++, como a maior parte das linguagens de programação imperativas, permite a definição daquilo a que se chama rotinas recursivas. Diz-se que uma rotina é recursiva se o seu corpo incluir chamadas à própria rotina[9]. Por exemplo

```
/** Devolve o factorial do inteiro passado como argumento.
    @pre  PC ≡ 0 ≤ n.
    @post CO ≡ factorial = n! (ou ainda factorial = ∏_{i=1}^{n} i). */
int factorial(int const n)
{
    if(n == 0 or n == 1)
        return 1;
    return n * factorial(n - 1);
}
```

é uma função recursiva que calcula o factorial e que foi obtida de uma forma imediata a partir da definição recorrente do factorial

$$n! = \begin{cases} n(n-1)! & \text{se } 0 < n \\ 1 & \text{se } n = 0 \end{cases},$$

usando-se adicionalmente o facto de que 1! é também 1.

Este tipo de rotinas pode ser muito útil na resolução de alguns problemas, mas deve ser usado com cautela. A chamada de uma rotina recursivamente implica que as variáveis locais (parâmetros incluídos) são construídas tantas vezes quantas a rotina é chamada e só são destruídas quando as correspondentes chamadas retornam. Como as chamadas recursivas se aninham umas dentro das outras, se ocorrerem muitas chamadas recursivas não só pode ser necessária muita memória para as várias versões das variáveis locais (uma versão por cada chamada aninhada), como também a execução pode tornar-se bastante lenta, pois a chamada de funções implica alguma perda de tempo nas tarefas de "arrumação da casa" do processador.

A função `factorial()`, em particular, pode ser implementada usando um ciclo, que resulta em código muito mais eficiente e claro:

```
/** Devolve o factorial do inteiro passado como argumento.
    @pre  PC ≡ 0 ≤ n.
    @post CO ≡ factorial = n! (ou ainda factorial = ∏_{i=1}^{n} i). */
int factorial(int const n)
{
    int factorial = 1;
    for(int i = 0; i != n; ++i)
        factorial *= i + 1;

    return factorial;
}
```

[9]É possível ainda que a recursividade seja entre duas ou mais rotinas, que se chamam mutuamente.

Muito importante é também a garantia de que uma rotina recursiva termina sempre. A função factorial recursiva acima tem problemas graves quando é invocada com um argumento negativo. É que vai sendo chamada recursivamente a mesma função com valores do argumento cada vez menores (mais negativos), sem fim à vista. Podem acontecer duas coisas. Como cada chamada da função implica a construção de uma variável local (o parâmetro `n`) num espaço de memória reservado para a chamada pilha (*stack*), como se verá na próxima secção, esse espaço pode-se esgotar, o que leva o programa a abortar. Ou então, se por acaso houver muito, mas mesmo muito espaço disponível na pilha, as chamadas recursivas continuarão até se atingir o limite inferior dos inteiros nos argumentos. Nessa altura a chamada seguinte é feita com o menor dos inteiros menos uma unidade, que como se viu no Capítulo 2 é o maior dos inteiros. A partir daí os argumentos das chamadas recursivas começam a diminuir e, ao fim de muito, mas mesmo muito tempo, atingirão o valor 0, que terminará a sequência de chamadas recursivas, sendo devolvido um valor errado.

Os problemas não surgem apenas com argumentos negativos, na realidade. É que os valores do factorial crescem depressa, pelo que a função não pode ser invocada com argumentos demasiado grandes. No caso de os inteiros terem 32 *bits*, o limite a impor aos argumentos é que têm de ser inferiores a 13! A função deve sofrer uma actualização na pré-condição e, já agora, ser equipada com as instruções de asserção apropriadas:

```
/** Devolve o factorial do inteiro passado como argumento.
    @pre   PC ≡ 0 ≤ n ≤ 12.
    @post  CO ≡ factorial = n! (ou ainda factorial = ∏_{i=1}^{n} i). */
int factorial(int const n)
{
    assert(0 <= n and n <= 12);

    if(n == 0 or n == 1)
        return 1;
    return n * factorial(n - 1);
}
```

o mesmo acontecendo com a versão não-recursiva:

```
/** Devolve o factorial do inteiro passado como argumento.
    @pre   PC ≡ 0 ≤ n ≤ 12.
    @post  CO ≡ factorial = n! (ou ainda factorial = ∏_{i=1}^{n} i). */
int factorial(int const n)
{
    assert(0 <= n and n <= 12);

    int factorial = 1;
    for(int i = 0; i != n; ++i)
        factorial *= i + 1;

    return factorial;
}
```

Para se compreender profundamente o funcionamento das rotinas recursivas tem de se compreender o mecanismo de chamada ou invocação de rotinas, que se explica na próxima secção.

3.4 Mecanismo de invocação de rotinas

Quando nas secções anteriores se descreveu a chamada da função `mdc()`, referiu-se que os seus parâmetros eram construídos no início da chamada e destruídos no seu final, e que a função, ao terminar, retornava para a instrução imediatamente após a instrução de invocação. Como é que o processo de invocação funciona na prática? Apesar de ser matéria para a disciplina de Arquitectura de Computadores, far-se-á aqui uma descrição breve e simplificada do mecanismo de invocação de rotinas que será útil (embora não fundamental) para se compreender o funcionamento das rotinas recursivas.

O mecanismo de invocação de rotinas utiliza uma parte da memória do computador como se de uma pilha se tratasse, i.e., como um local onde se pode ir acumulando informação de tal forma que a última informação a ser colocada na pilha seja a primeira a ser retirada, um pouco como acontece com as pilhas de processos das repartições públicas, em os processos dos incautos podem ir envelhecendo ao longo de anos na base de uma pilha...

A pilha é utilizada para colocar todas as variáveis locais automáticas quando estas são construídas. O topo da pilha varia quando é executada uma instrução de definição de um variável automática. As variáveis definidas são colocadas (construídas) no topo da pilha e apenas são retiradas (destruídas) quando se abandona o bloco onde foram definidas. É também na pilha que se guarda o endereço da instrução para onde o fluxo de execução do programa deve retornar uma vez terminada a execução de uma rotina. No fundo, a pilha serve para o computador "saber a quantas anda".

Exemplo não-recursivo

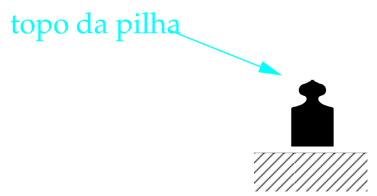
Suponha-se o seguinte programa, incluindo a função `mdc()`,

```
#include <iostream>

using namespace std;

/** Calcula e devolve o máximo divisor comum de dois inteiros passados
    como argumentos (o segundo inteiro tem de ser positivo).
    @pre  PC ≡ m = m ∧ n = n ∧ 0 < n.
    @post CO ≡ mdc = mdc(|m|, n). */
int mdc(int const m, int const n)
{
    assert(0 < n);

    if(m < 0)
        m = -m;
```

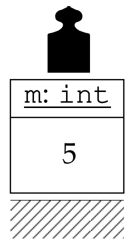
```
    while(m != 0) {
        int const auxiliar = n % m;
        n = m;
        m = auxiliar;
    }

    assert(m % k == 0 and n % k == 0);

    return n;
}

int main()
{
    int m = 5;                       // 1
                   mdc(m + 3, 6)  // 2A
    int divisor =                ; // 2B
    cout << divisor << endl;      // 3
}
```

em que se dividiu a instrução 2 em duas "sub-instruções" 2A e 2B.

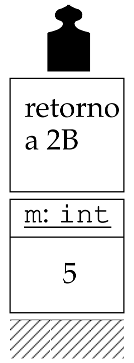
Considera-se, para simplificar, que pilha está vazia quando se começa a executar a função `main()`:



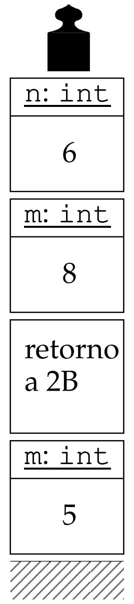
De seguida é executada a instrução 1, que constrói a variável `m`. Como essa variável é automática, é construída na pilha, que fica



Que acontece quando a instrução 2A é executada? A chamada da função `mdc()` na instrução 2A começa por guardar na pilha o endereço da instrução a executar quando a função retornar, i.e., 2B



Em seguida são construídos na pilha os parâmetros da função. Cada parâmetro é inicializado com o valor do argumento respectivo:

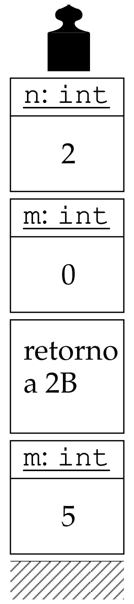


Neste momento existem na pilha duas variáveis de nome `m`: uma pertencente à função `main()` e outra à função `mdc()`. É fácil saber em cada instante quais são as variáveis automática visíveis: são todas as variáveis desde o topo da pilha até ao próximo endereço de retorno. Isto é, neste momento são visíveis apenas as variáveis `m` e `n` de `mdc()`.

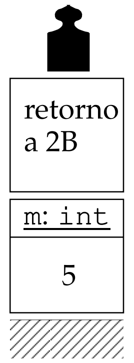
A execução passa então para o corpo da função, onde durante o ciclo a constante local `auxiliar` é construída e destruída no topo da pilha várias vezes[10], até que o ciclo termina com valor desejado na variável `n`, i.e., 2. Assim, imediatamente antes da execução da instrução de retorno, a pilha contém:

---

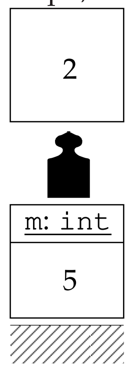
[10]Qualquer compilador minimamente inteligente evita este processo de construção e destruição repetitivas construindo a variável `auxiliar` na pilha logo no início da invocação da função.



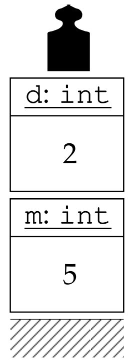
A instrução de retorno começa por calcular o valor a devolver (neste caso é o valor de `n`, i.e., 2), retira da pilha (destrói) todas as variáveis desde o topo até ao próximo endereço de retorno



e em seguida retira da pilha a instrução para onde o fluxo de execução deve ser retomado (i.e., 2B) colocando na pilha (mas para lá do seu topo, em situação periclitante...) o valor a devolver



Em seguida a execução continua em 2B, que constrói a variável `divisor`, inicializando-a com o valor devolvido, colocado após o topo da pilha, que depois se deixa levar por uma "corrente de ar":



O valor de `d` é depois escrito no ecrã na instrução 3.

Finalmente atinge-se o final da função `main()`, o que leva à retirada da pilha (destruição) de todas as variáveis até à base (uma vez que não há nenhum endereço de retorno na pilha):

No final, a pilha fica exactamente como no início: vazia.

Exemplo recursivo

Suponha-se agora o seguinte exemplo, que envolve a chamada à função recursiva `factorial()`:

```
#include <iostream>

using namespace std;

/** Devolve o factorial do inteiro passado como argumento.
    @pre   PC ≡ 0 ≤ n ≤ 12.
    @post  CO ≡ factorial = n!  (ou ainda factorial = ∏_{i=1}^{n} i).  */
int factorial(int const n)
{
    assert(0 <= n and n <= 12);

    if(n == 0 or n == 1)           // 1
        return 1;                  // 2
    return n * factorial(n - 1);   // 3
}
int main()
{
    cout << factorial(3) << endl; // 4
}
```

Mais uma vez é conveniente dividir a instrução 4 em duas “sub-instruções”

```
         factorial(3)          // 4A
cout <<              << endl; // 4B
```

uma vez que a função é invocada antes da escrita do resultado no ecrã. Da mesma forma, a instrução 3 pode ser dividida em duas "sub-instruções"

```
            factorial(n - 1)  // 3A
return n *                  ; // 3B
```

Ou seja,

```
#include <iostream>

using namespace std;

/** Devolve o factorial do inteiro passado como argumento.
    @pre  PC ≡ 0 ≤ n ≤ 12.
    @post CO ≡ factorial = n! (ou ainda factorial = ∏_{i=1}^{n} i). */
int factorial(int const n)
{
    assert(0 <= n and n <= 12);

    if(n == 0 or n == 1)          // 1
        return 1;                 // 2
               factorial(n - 1)  // 3A
    return n *                  ; // 3B
}
int main()
{
             factorial(3)          // 4A
    cout <<              << endl; // 4B
}
```

Que acontece ao ser executada a instrução 4A? Essa instrução contém uma chamada à função `factorial()`. Assim, tal como se viu antes, as variáveis locais da função (neste caso apenas o parâmetro constante `n`) são colocadas na pilha logo após o endereço da instrução a executar quando função retornar. Quando a execução passa para a instrução 1, já a pilha está já como indicado em (b) na Figura 3.3.

Em seguida, como a constante `n` contém 3 e portanto é diferente de 0 e de 1, é executada a instrução após o `if`, que é a instrução 3A (se tiver dúvidas acerca do funcionamento do if, consulte a Secção 4.1.1). Mas a instrução 3A consiste numa nova chamada à função, pelo que os passos acima se repetem, mas sendo agora o parâmetro inicializado com o valor do argumento, i.e., 2, e sendo o endereço de retorno 3B, resultando na pilha indicada em (c) na Figura 3.3.

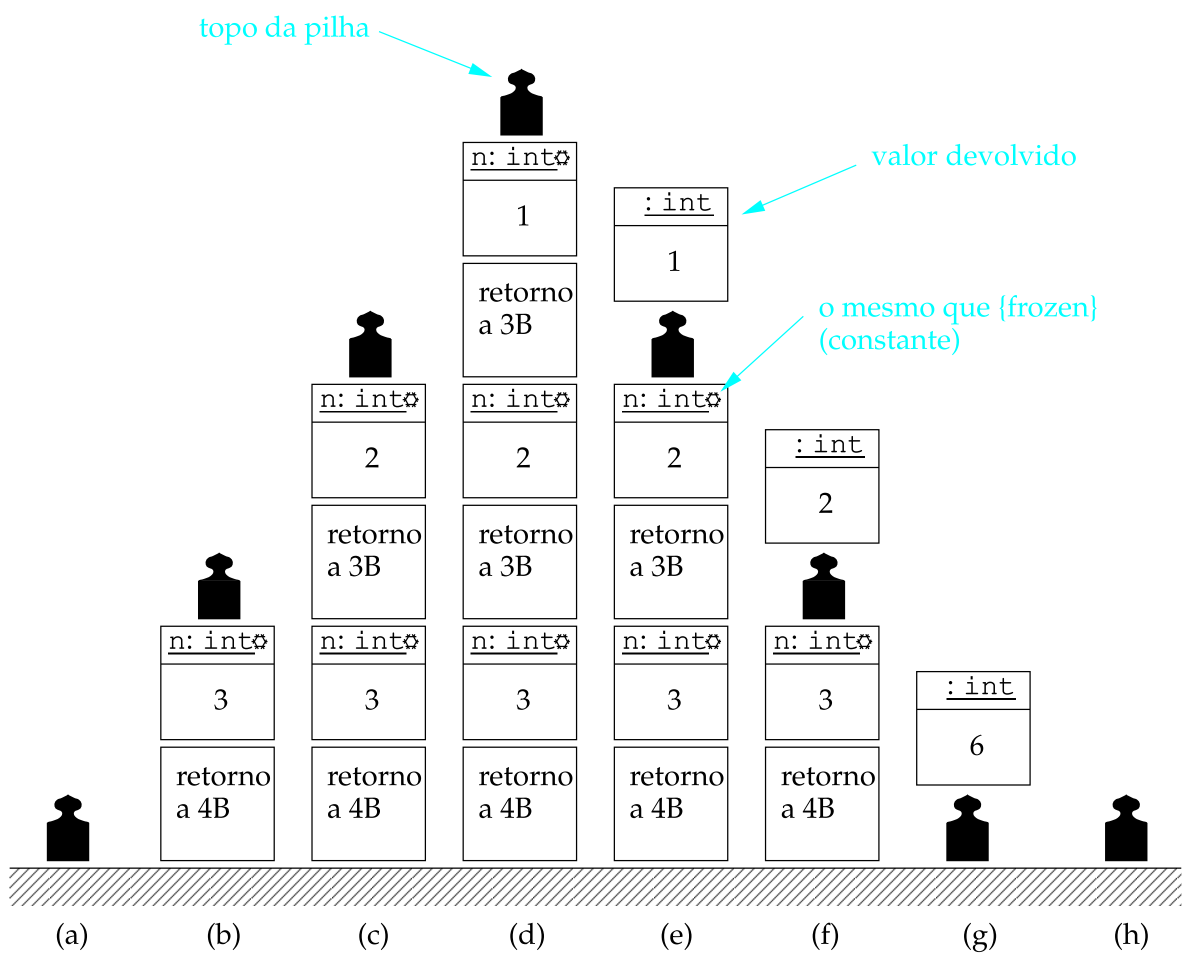


Figura 3.3: Evolução da pilha durante invocações recursivas da função `factorial()`. Admite-se que a pilha inicialmente está vazia, como indicado em (a).

Neste momento existem duas versões da constante `n` na pilha, uma por cada chamada à função que ainda não terminou (há uma chamada "principal" e outra "aninhada"). É esta repetição das variáveis e constantes locais que permite às funções recursivas funcionarem sem problemas.

A execução passa então para o início da função (instrução 1). De novo, como constante `n` da chamada em execução correntemente é 2, e portanto diferente de 0 e de 1, é executada a instrução após o `if`, que é a instrução 3A. Mas a instrução 3A consiste numa nova chamada à função, pelo que os passos acima se repetem, mas sendo agora o parâmetro inicializado com o valor do argumento, i.e., 1, e sendo o endereço de retorno 3B, resultando na pilha indicada em (d) na Figura 3.3.

A execução passa então para o início da função (instrução 1). Agora, como a constante `n` da chamada em execução correntemente é 1, é executada a instrução condicionada pelo `if`, que é a instrução 2. Mas a instrução 2 consiste numa instrução de retorno com devolução do valor 1. Assim, as variáveis e constantes locais são retiradas da pilha, o endereço de retorno (3B) é retirado da pilha, o valor de devolução 1 é colocado após o topo da pilha, e a execução continua na instrução 3B, ficando a pilha indicada em (e) na Figura 3.3.

A instrução 3B consiste numa instrução de retorno com devolução do valor `n * 1`(ou seja 2), em que `n` tem o valor 2 e o 1 é o valor de devolução da chamada anterior, que ficou após o topo da pilha. Assim, as variáveis e constantes locais são retiradas da pilha, o endereço de retorno (3B) é retirado da pilha, o valor de devolução 2 é colocado após o topo da pilha, e a execução continua na instrução 3B, ficando a pilha indicada em (f) na Figura 3.3.

A instrução 3B consiste numa instrução de retorno com devolução do valor `n * 2` (ou seja 6), em que `n` tem o valor 3 e o 2 é o valor de devolução da chamada anterior, que ficou após o topo da pilha. Assim, as variáveis locais e constantes locais são retiradas da pilha, o endereço de retorno (4B) é retirado da pilha, o valor de devolução 6 é colocado após o topo da pilha, e a execução continua na instrução 4B, ficando a pilha indicada em (g) na Figura 3.3.

A instrução 4B corresponde simplesmente a escrever no ecrã o valor devolvido pela chamada à função, ou seja, 6 (que é 3!, o factorial de 3). É de notar que, terminadas todas as chamadas à função, a pilha voltou à sua situação original (que se supôs ser vazia)indicada em (h) na Figura 3.3.

A razão pela qual as chamadas recursivas funcionam como espectável é que, em cada chamada aninhada, são criadas novas versões das variáveis e constantes locais e dos parâmetros (convenientemente inicializados) da rotina. Embora os exemplos acima se tenham baseado em funções, é evidente que o mesmo mecanismo é usado para os procedimentos, embora simplificado pois estes não devolvem qualquer valor.

3.5 Sobrecarga de nomes

Em certos casos é importante ter rotinas que fazem conceptualmente a mesma operação ou o mesmo cálculo, mas que operam com tipos diferentes de dados. Seria pois de todo o interesse que fosse permitida a definição de rotinas com nomes idênticos, distintos apenas no tipo dos seus parâmetros. De facto, a linguagem C++ apenas proíbe a definição no mesmo contexto de funções ou procedimentos com a mesma *assinatura*, i.e., não apenas com o mesmo nome, mas também com a mesma lista dos tipos dos parâmetros[11]. Assim, é de permitida a definição de múltiplas rotinas com o mesmo nome, desde que difiram no número ou tipo de parâmetros. As rotinas com o mesmo nome dizem-se *sobrecarregadas*. A invocação de rotinas sobrecarregadas faz-se como habitualmente, sendo a rotina que é de facto invocada determinada a partir do número e tipo dos argumentos usados na invocação. Por exemplo, suponha-se que estão definidas as funções:

```
int soma(int const a, int const b)
{
    return a + b;
}

int soma(int const a, int const b, int const c)
{
    return a + b + c;
}

float soma(float const a, float const b)
{
    return a + b;
}

double soma(double const a, double const b)
{
    return a + b;
}
```

Ao executar as instruções

```
int i1, i2;
float f1, f2;
double d1, d2;
i2 = soma(i1, 4);      // invoca int soma(int, int).
i2 = soma(i1, 3, i2); // invoca int soma(int, int, int).
f2 = soma(5.6f, f1);  // invoca float soma(float, float).
d2 = soma(d1, 10.0);  // invoca double soma(double, double).
```

são chamadas as funções apropriadas para cada tipo de argumentos usados. Este tipo de comportamento emula para as funções definidas pelo programador o comportamento normal dos operadores do C++. A operação `+`, por exemplo, significa soma de `int` se os operandos forem `int`, significa soma de `float` se os operandos forem `float`, etc. O exemplo mais claro talvez seja o do operador divisão (`/`). As instruções

[11] A noção de assinatura usual é um pouco mais completa que na linguagem C++, pois inclui o tipo de devolução. Na linguagem C++ o tipo de devolução não faz parte da assinatura.

```
cout << 1 / 2 << endl;
cout << 1.0 / 2.0 << endl;
```

têm como resultado no ecrã

```
0 0.5
```

porque no primeiro caso, sendo os operandos inteiros, a divisão usada é a divisão inteira. Assim, cada operador básico corresponde na realidade a vários operadores com o mesmo nome, i.e., sobrecarregados, cada um para determinado tipo dos operandos.

A assinatura de uma rotina corresponde à sequência composta pelo seu nome, pelo número de parâmetros, e pelos tipos dos parâmetros. Por exemplo, as funções `soma()` acima têm as seguintes assinaturas[12]:

- `soma, int, int`
- `soma, int, int, int`
- `soma, float, float`
- `soma, double, double`

O tipo de devolução de uma rotina não faz parte da sua assinatura, não servindo portanto para distinguir entre funções ou procedimentos sobrecarregados.

Num capítulo posterior se verá que é possível sobrecarregar os significados dos operadores básicos (como o operador +) quando aplicados a tipos definidos pelo programador, o que transforma o C++ numa linguagem que se "artilha" de uma forma muito elegante e potente.

3.6 Parâmetros com argumentos por omissão

O C++ permite a definição de rotinas em que alguns parâmetros têm argumentos por omissão. I.e., se não forem colocados os argumentos respectivos numa invocação da rotina, os parâmetros serão inicializados com os valores dos argumentos por omissão. Mas com uma restrição: os parâmetros com argumentos por omissão têm de ser os últimos da rotina.

Por exemplo, a definição

```
int soma(int const a = 0, int const b = 0,
         int const c = 0, int const d = 0)
{
    return a + b + c + d;
}
```

[12] A constância de um parâmetro (desde que não seja um referência) não afecta a assinatura, pois esta reflecte a interface da rotina, e não a sua implementação.

permite a invocação da função `soma()` com qualquer número de argumentos até 4:

```
cout << soma() << endl             // Surge 0.
cout << soma(1) << endl            // Surge 1.
cout << soma(1, 2) << endl         // Surge 3.
cout << soma(1, 2, 3) << endl      // Surge 6.
cout << soma(1, 2, 3, 4) << endl; // Surge 10.
```

Normalmente os argumentos por omissão indicam-se apenas na declaração de um rotina. Assim, se a declaração da função `soma()` fosse feita separadamente da respectiva definição, o código deveria passar a ser:

```
int soma(int const a = 0, int const b = 0,
         int const c = 0, int const d = 0);

...

int soma(int const a, int const b, int const c, int const d)
{
    return a + b + c + d;
}
```

Capítulo 4

Controlo do fluxo dos programas

Se temos...! Diz ela
mas o problema não é só de aprender
é saber a partir daí que fazer

Sérgio Godinho, 2º andar direito, *Pano Cru*.

Quase todas as resoluções de problemas envolvem tomadas de decisões e repetições. Dificilmente se consegue encontrar um algoritmo interessante que não envolva, quando lido em português, as palavras "se" e "enquanto", correspondendo aos conceitos de selecção e de iteração. Neste capítulo estudar-se-ão em pormenor os mecanismos da programação imperativa que suportam esses conceitos e discutir-se-ão metodologias de resolução de problemas usando esses mecanismos.

4.1 Instruções de selecção

A resolução de um problema implica quase sempre a tomada de decisões ou pelo menos a selecção de alternativas.

Suponha-se que se pretendia desenvolver uma função para calcular o valor absoluto de um inteiro. O "esqueleto" da função pode ser o que se segue

```
/** Devolve o valor absoluto do argumento.
    @pre   PC ≡ V (ou seja, sem restrições).
    @post  CO ≡ absoluto = |x|, ou seja,
              0 ≤ absoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    ...
}
```

Este esqueleto inclui, como habitualmente, documentação clarificando o que a função faz, o cabeçalho que indica como a função se utiliza, e um corpo, onde se colocarão as instruções que resolvem o problema, i.e., que explicitam como a função funciona.

A resolução deste problema requer que sejam tomadas acções diferentes consoante o valor de `x` seja positivo ou negativo (ou nulo). São portanto fundamentais as chamadas instruções de selecção ou alternativa.

4.1.1 As instruções `if` e `if else`

As instruções `if` e `if else` são das mais importantes instruções de controlo do fluxo de um programa, i.e., instruções que alteram a sequência normal de execução das instruções de um programa. Estas instruções permitem executar uma instrução caso uma condição seja verdadeira e, no caso da instrução `if else`, uma outra instrução caso a condição seja falsa. Por exemplo, no troço de programa

```
if(x < 0)  // 1
    x = 0; // 2
else       // 3
    x = 1; // 4
...        // 5
```

as linhas 1 a 4 correspondem a uma única *instrução de selecção*. Esta instrução de selecção é composta de uma condição (na linha 1) e duas instruções alternativas (linhas 2 e 4). Se `x` for menor do que zero quando a instrução `if` é executada, então a próxima instrução a executar é a instrução na linha 2, passando o valor de `x` a ser zero. No caso contrário, se `x` for inicialmente maior ou igual a zero, a próxima instrução a executar é a instrução na linha 4, passando a variável `x` a ter o valor 1. Em qualquer dos casos, a execução continua na linha 5[1].

Como a instrução na linha 2 só é executada se $\mathtt{x} < 0$, diz-se que $\mathtt{x} < 0$ é a sua guarda, normalmente representada por G. De igual modo, a guarda da instrução na linha 4 é $0 \leq \mathtt{x}$. A guarda da instrução alternativa após o `else` está implícita, podendo ser obtida por negação da guarda do `if`: $\neg(\mathtt{x} < 0)$ é equivalente a $0 \leq \mathtt{x}$. Assim, numa instrução de selecção pode-se sempre inverter a ordem das instruções alternativas desde que se negue a condição. Assim,

```
if(x < 0)
    // G₁ ≡ x < 0
    x = 0;
else
    // G₂ ≡ 0 ≤ x
    x = 1;
```

é equivalente a

[1]Se existirem instruções `return`, `break`, `continue` ou `goto` nas instruções controladas por um `if`, o fluxo normal de execução pode ser alterado de tal modo que a execução não continua na instrução imediatamente após esse `if`.

```
if(0 <= x)
    // G₂ ≡ 0 ≤ x
    x = 1;
else
    // G₁ ≡ x < 0
    x = 0;
```

O fluxo de execução de uma instrução de selecção genérica

```
if(C)
    // G₁ ≡ C
    instrução₁
else
    // G₂ ≡ ¬C
    instrução₂
```

é representado no diagrama de actividade da Figura 4.1.

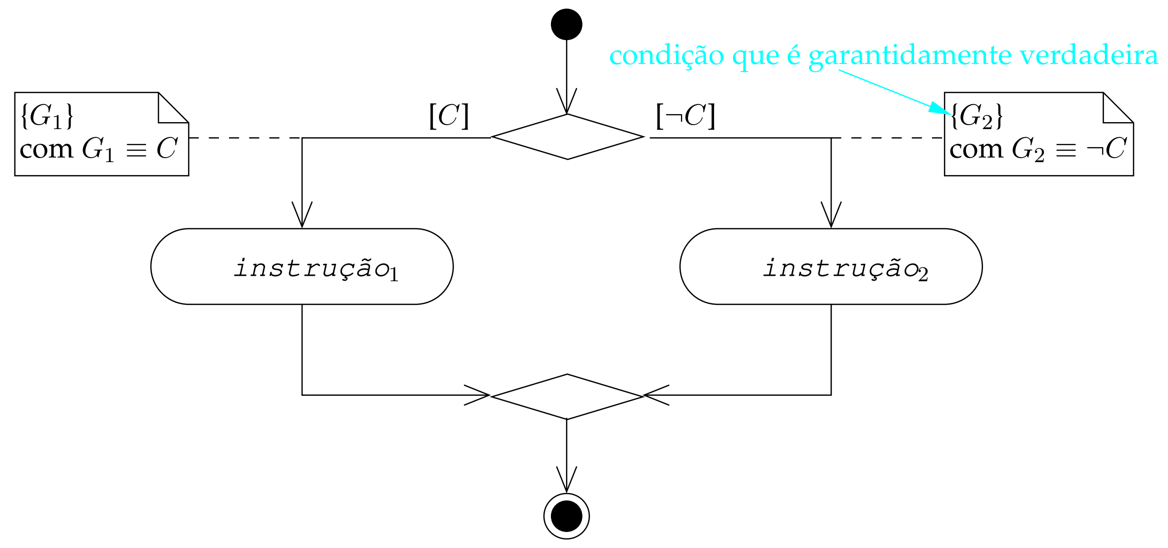


Figura 4.1: Diagrama de actividade de uma instrução de selecção genérica. Uma condição que tem de ser sempre verdadeira coloca-se num comentário entre chavetas.

Em certos casos pretende-se apenas executar uma dada instrução se determinada condição se verificar, não se desejando fazer nada caso a condição seja falsa. Nestes casos pode-se omitir o `else` e a instrução que se lhe segue. Por exemplo, no troço de programa

```
if(x < 0)  // 1
    x = 0; // 2
...        // 3
```

se `x` for inicialmente menor do que zero, então é executada a instrução condicionada na linha 2, passando o valor de `x` a ser zero, sendo em seguida executada a instrução na linha 3. No caso contrário a execução passa directamente para a linha 3.

A este tipo restrito de instrução de selecção também se chama *instrução condicional*. Uma instrução condicional é sempre equivalente a uma instrução de selecção em que a instrução após o `else` é uma instrução nula (a instrução nula corresponde em C++ a um simples terminador `;`). Ou seja, o exemplo anterior é equivalente a:

```
if(x < 0)  // 1
    x = 0; // 2a
else       // 2b
    ;      // 2c
...        // 3
```

O fluxo de execução de uma instrução de condicional genérica

```
if(C)
    instrução
```

é representado no diagrama de actividade da Figura 4.2.

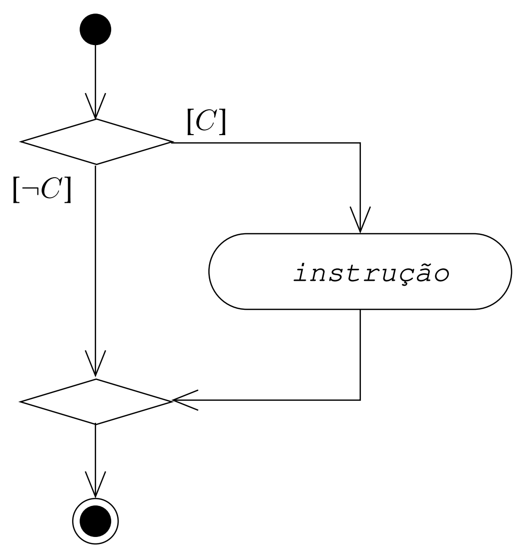


Figura 4.2: Diagrama de actividade de uma instrução condicional genérica.

Em muitos casos é necessário executar condicional ou alternativamente não uma instrução simples mas sim uma sequência de instruções. Para o conseguir, agrupam-se essas instruções num bloco de instruções ou instrução composta, colocando-as entre chavetas `{}`. Por exemplo, o código que se segue ordena os valores guardados nas variáveis `x` e `y` de tal modo que `x` termine com um valor menor ou igual ao de `y`:

```
int x, y;
...
if(y < x) {
    int const auxiliar = x;
    x = y;
    y = auxiliar;
}
```

4.1.2 Instruções de selecção encadeadas

Caso seja necessário, podem-se encadear instruções de selecção umas nas outras. Isso acontece quando se pretende seleccionar uma entre mais do que duas instruções alternativas. Por exemplo, para verificar qual a posição do valor de uma variável `a` relativamente a um intervalo [`mínimo` `máximo`], pode-se usar

```
int a, mínimo, máximo;
cin >> mínimo >> máximo >> a;

if(a < mínimo)
    cout << a << " menor que mínimo." << endl;
else {
    if(a <= máximo)
        cout << a << " entre mínimo e máximo." << endl;
    else
        cout << a << " maior que máximo." << endl;
}
```

Sendo as instruções de selecção instruções por si só, o código acima pode-se escrever sem recurso às chavetas, ou seja,

```
int a, mínimo, máximo;
cin >> mínimo >> máximo >> a;

if(a < mínimo)
    cout << a << " menor que mínimo." << endl;
else
    if(a <= máximo)
        cout << a << " entre mínimo e máximo." << endl;
    else
        cout << a << " maior que máximo." << endl;
```

Em casos como este, em que se encadeiam `if else` sucessivos, é comum usar uma indentação que deixa mais claro que existem mais do que duas alternativas,

```
int a, mínimo, máximo;
cin >> mínimo >> máximo >> a;

if(a < mínimo)
    cout << a << " menor que mínimo." << endl;
else if(a <= máximo)
    cout << a << " entre mínimo e máximo." << endl;
else
    cout << a << " maior que máximo." << endl;
```

Guardas em instruções alternativas encadeadas

Podem-se colocar como comentários no exemplo anterior as guardas de cada instrução alternativa. Estas guardas reflectem as circunstâncias nas quais as instruções alternativas respectivas são executadas

```
int a, mínimo, máximo;
cin >> mínimo >> máximo >> a;

if(a < mínimo)
    // $G_1 \equiv \texttt{a} < \texttt{mínimo}$.
    cout << a << " menor que mínimo." << endl;
else if(a <= máximo)
    // $G_2 \equiv \texttt{mínimo} \leq \texttt{a} \leq \texttt{máximo}$.
    cout << a << " entre mínimo e máximo." << endl;
else
    // $G_3 \equiv \texttt{mínimo} \leq \texttt{a} \wedge \texttt{máximo} < \texttt{a}$.
    cout << a << " maior que máximo." << endl;
```

Fica claro que as guardas não correspondem directamente à condição indicada no `if` respectivo, com excepção da primeira.

As n guardas de uma instrução de selecção, construída com $n - 1$ instruções `if` encadeadas, podem ser obtidas a partir das condições de cada um dos `if` como se segue:

```
if($C_1$)
    // $G_1 \equiv C_1$.
    ...
else if($C_2$)
    // $G_2 \equiv \neg G_1 \wedge C_2$  (ou, $\neg C_1 \wedge C_2$).
    ...
else if($C_3$)
    // $G_3 \equiv \neg G_1 \wedge \neg G_2 \wedge C_3$  (ou, $\neg C_1 \wedge \neg C_2 \wedge C_3$).
    ...
...
```

```
else if(C_{n-1})
      // G_{n-1} ≡ ¬G_1 ∧ ··· ∧ ¬G_{n-2} ∧ C_{n-1}  (ou, ¬C_1 ∧ ··· ∧ ¬C_{n-2} ∧ C_{n-1}).
      ...
else
      // G_n ≡ ¬G_1 ∧ ··· ∧ ¬G_{n-1}  (ou, ¬C_1 ∧ ··· ∧ ¬C_{n-1}).
      ...
```

Ou seja, as guardas das instruções alternativas são obtidas por conjunção da condição do `if` respectivo com a negação das guardas das instruções alternativas (ou das condições de todos os `if`) anteriores na sequência, como seria de esperar. Uma instrução de selecção encadeada é pois equivalente a uma instrução de selecção com mais do que duas instruções alternativas, como se mostra na Figura 4.3.

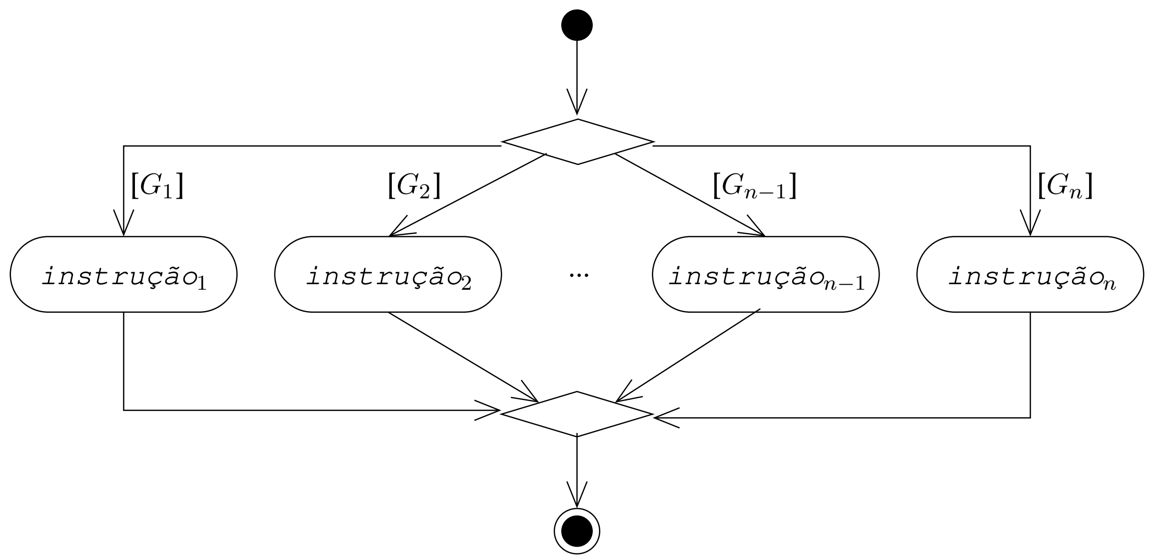


Figura 4.3: Diagrama de actividade de uma instrução de múltipla (sem correspondente directo na linguagem C++) equivalente a uma instrução de selecção encadeada. As guardas presumem-se mutuamente exclusivas.

Influência de pré-condições

A última guarda do exemplo dado anteriormente parece ser redundante: se `a` é maior que `máximo` não é forçosamente maior que `mínimo`? Então porque não é a guarda simplesmente `máximo` $<$ `a`? Acontece que nada garante que `mínimo` $\leq$ `máximo`! Se se introduzir essa condição como pré-condição das instruções de selecção encadeadas, então essa condição verifica-se imediatamente antes de cada uma das instruções alternativas, pelo que as guardas podem ser simplificadas e tomar a sua forma mais intuitiva

```
// PC ≡ mínimo ≤ máximo
if(a < mínimo)
```

```
        // G₁ ≡ a < mínimo.
        cout << a << " menor que mínimo." << endl;
    else if(a <= máximo)
        // G₂ ≡ mínimo ≤ a ≤ máximo.
        cout << a << " entre mínimo e máximo." << endl;
    else
        // G₃ ≡ máximo < a.
        cout << a << " maior que máximo." << endl;
```

4.1.3 Problemas comuns

A sintaxe das instruções `if` e `if else` do C++ presta-se a ambiguidades. No código

```
if(m == 0)
    if(n == 0)
        cout << "m e n são zero." << endl;
else
    cout << "m não é zero." << endl;
```

o `else` não diz respeito ao primeiro `if`. Ao contrário do que a indentação do código sugere, o `else` diz respeito ao segundo `if`. Em caso de dúvida, um `else` pertence ao `if` mais próximo (e acima...) dentro mesmo bloco de instruções e que não tenha já o respectivo `else`. Para corrigir o exemplo anterior é necessário construir uma instrução composta, que neste caso consiste de uma única instrução de selecção

```
if(m == 0) {
    if(n == 0)
        cout << "m e n são zero." << endl;
} else
    cout << "m não é zero." << endl;
```

É conveniente usar blocos de instruções de uma forma liberal, pois construções como a que se apresentou podem dar origem a erros muito difíceis de detectar e corrigir. Os compiladores de boa qualidade, no entanto, avisam o programador da presença de semelhantes (aparentes) ambiguidades.

Um outro erro frequente corresponde a colocar um terminador `;` logo após a condição do `if` ou logo após o `else`. Por exemplo, a intenção do programador do troço de código

```
if(x < 0);
    x = 0;
```

era provavelmente que se mantivesse o valor de `x` excepto quando este fosse negativo. Mas a interpretação feita pelo compilador (e a correcta dada a sintaxe da linguagem C++) é

```
if(x < 0)
    ; // instrução nula: não faz nada.
x = 0;
```

ou seja, `x` terminará sempre com o valor zero! Este tipo de erro, não sendo muito comum, é ainda mais difícil de detectar do que o da (suposta) ambiguidade da pertença de um `else`: os olhos do programador, habituados que estão à presença de `;` no final de cada linha, recusam-se a detectar o erro.

4.2 Asserções

Antes de se passar ao desenvolvimento de instruções de selecção, é importante fazer uma pequena digressão para introduzir um pouco mais formalmente o conceito de asserção.

Chama-se asserção a um predicado (ver Secção A.1) escrito normalmente na forma de comentário antes ou depois de uma instrução de um programa. As asserções correspondem a afirmações acerca das variáveis do programa que se sabe serem verdadeiras antes da instrução seguinte à asserção e depois da instrução anterior à asserção. Uma asserção pode sempre ser vista como pré-condição PC da instrução seguinte e condição objectivo CO da instrução anterior. As asserções podem também incluir afirmações acerca de variáveis matemáticas, que não pertencem ao programa.

Nas asserções cada variável pertence a um determinado conjunto. Para as variáveis C++, esse conjunto é determinado pelo tipo da variável indicado na sua definição (e.g., `int x;` significa que `x` pertence ao conjunto dos inteiros entre -2^{n-1} e $2^{n-1}-1$, se os `int` tiverem n *bits*). Para as variáveis matemáticas, esse conjunto deveria, em rigor, ser indicado explicitamente. Neste texto, no entanto, admite-se que as variáveis matemáticas pertencem ao conjunto dos inteiros, salvo onde for explicitamente indicado outro conjunto ou onde o conjunto seja fácil de inferir pelo contexto da asserção. Nas asserções é também normal assumir que as variável C++ não têm limitações (e.g., admite-se que uma variável `int` pode guardar qualquer inteiro). Embora isso não seja rigoroso, permite resolver com maior facilidade um grande número de problemas sem que seja necessário introduzir demasiados pormenores nas demonstrações.

Em cada ponto de um programa existe um determinado conjunto de variáveis, cada uma das quais pode tomar determinados valores, consoante o seu tipo. Ao conjunto de todos os possíveis valores de todas as variáveis existentes num dado ponto de um programa chama-se o *espaço de estados* do programa nesse ponto. Ao conjunto dos valores das variáveis existentes num determinado ponto do programa num determinado instante de tempo chama-se o *estado* de um programa. Assim, o estado de um programa é um elemento do espaço de estados.

As asserções fazem afirmações acerca do estado do programa num determinado ponto. Podem ser mais fortes, por exemplo se afirmarem que uma dada variável toma o valor 1, ou mais fracas, por exemplo se afirmarem que uma dada variável toma um valor positivo.

4.2.1 Dedução de asserções

Suponha-se o seguinte troço de programa

```
++n;
```

onde se admite que a variável `n` tem inicialmente um valor não-negativo. Como adicionar asserções a este troço de programa? Em primeiro lugar escreve-se a asserção que corresponde à assunção de que `n` guarda inicialmente um valor não-negativo:

```
// 0 ≤ n
++n;
```

Como se pode verificar, as asserções são colocadas no código na forma de comentários.

A asserção que segue uma instrução, a sua condição objectivo, é tipicamente obtida pela semântica da instrução e pela respectiva pré-condição. Ou seja, dada a PC, a instrução implica a veracidade da respectiva CO. No caso acima é óbvio que

```
// 0 ≤ n.
++n;
// 1 ≤ n.
```

ou seja, se `n` era maior ou igual a zero antes da incrementação, depois da incrementação será forçosamente maior ou igual a um.

A demonstração informal da correcção de um pedaço de código pode, portanto, ser feita recorrendo a asserções. Suponha-se que se pretende demonstrar que o código

```
int const t = x;
x = y;
y = t;
```

troca os valores de duas variáveis `x` e `y` do tipo `int`. Começa-se por escrever as duas principais asserções: a pré-condição e a condição objectivo da sequência completa de instruções. Neste caso a escrita destas asserções é complicada pelo facto de a CO ter de ser referir aos valores das variáveis `x` e `y` *antes das instruções*. Para resolver este problema, considere-se que as variáveis matemáticas x e y representam os valores iniciais das variáveis C++ `x` e `y` (repare-se bem na diferença de tipo de letra usado para variáveis matemáticas e para variáveis do programa C++). Então a CO pode ser escrita simplesmente como

$$\mathtt{x} = y \wedge \mathtt{y} = x$$

e a PC como

$$\mathtt{x} = x \wedge \mathtt{y} = y$$

donde o código com as asserções iniciais e finais é

```
// x = x ∧ y = y.
int const t = x;
x = y;
y = t;
// x = y ∧ y = x.
```

A demonstração de correcção pode ser feita deduzindo as asserções intermédias:

```
// x = x ∧ y = y.
int const t = x;
// x = x ∧ y = y ∧ t = x.
x = y;
// x = y ∧ y = y ∧ t = x.
y = t;
// x = y ∧ y = x ∧ t = x ⇒ x = y ∧ y = x.
```

A demonstração de correcção pode ser feita também partindo dos objectivos. Para cada instrução, começando na última, determina-se quais as condições mínimas a impor às variáveis antes da instrução, i.e., determina-se qual a PC mais fraca a impor à instrução em causa, de modo a que, depois dessa instrução, a sua CO seja verdadeira. Antes do o fazer, porém, é necessário introduzir mais alguns conceitos.

4.2.2 Predicados mais fortes e mais fracos

Diz-se que um predicado P é mais fraco do que outro Q se Q implicar P, ou seja, se o conjunto dos valores que tornam o predicado P verdadeiro contém o conjunto dos valores que tornam o predicado Q verdadeiro. Por exemplo, se $P \equiv 0 < x$ e $Q \equiv x = 1$, então P é mais fraco do que Q, pois o conjunto $\{1\}$ está contido no conjunto dos positivos $\{x : 0 < x\}$. O mais fraco de todos os possíveis predicados é aquele que é sempre verdadeiro, pois o conjunto dos valores que o verificam é o conjunto universal. Logo, o mais fraco de todos os possíveis predicados é $\mathcal{V}$. Por razões óbvias, o mais forte de todos os possíveis predicados é $\mathcal{F}$, sendo vazio o conjunto dos valores que o verificam.

4.2.3 Dedução da pré-condição mais fraca de uma atribuição

É possível estabelecer uma relação entre as asserções que é possível escrever antes e depois de uma instrução de atribuição. Ou seja, é possível relacionar duma forma algébrica PC e CO em

```
// PC
x = expressão;
// CO
```

A relação é mais de estabelecer partindo da condição objectivo CO. É que a PC mais fraca que, depois da atribuição, conduz à CO, pode ser obtida substituindo por `expressão` todas as ocorrências da variável `x` em CO. Esta dedução da PC mais fraca só pode ser feita se a expressão cujo valor se atribui a `x` não tiver efeitos laterais, i.e., se não implicar a alteração de nenhuma variável do programa (ver Secção 2.7.8). Se a expressão tiver efeitos laterais mas apesar de tudo for bem comportada, é possível decompô-la numa sequência de instruções e aplicar a dedução a cada uma delas.

Por exemplo, suponha-se que se pretendia saber qual a PC mais fraca para a qual a instrução de atribuição `x = -x;` conduz à CO $0 \leq x$. Aplicando o método descrito conclui-se que

```
// 0 ≤ −x, ou seja, x ≤ 0.
x = -x;
// 0 ≤ x.
```

Ou seja, para que a inversão do sinal de `x` conduza a um valor não-negativo, o menos que tem de se exigir é que o valor de `x` seja inicialmente não-positivo.

Voltando ao exemplo da troca de valores de duas variáveis

```
int const t = x;
x = y;
y = t;
// x = y ∧ y = x.
```

e aplicando a técnica proposta obtém-se sucessivamente

```
int const t = x;
x = y;
// x = y ∧ t = x.
y = t;
// x = y ∧ y = x.

int const t = x;
// y = y ∧ t = x.
x = y;
// x = y ∧ t = x.
y = t;
// x = y ∧ y = x.

// y = y ∧ x = x.
int const t = x;
// y = y ∧ t = x.
x = y;
// x = y ∧ t = x.
y = t;
// x = y ∧ y = x.
```

tendo-se recuperado a PC inicial.

Em geral este método não conduz exactamente à PC escrita inicialmente. Suponha-se que se pretendia demonstrar que

```
// x < 0.
x = -x;
// 0 ≤ x.
```

Usando o método anterior conclui-se que:

```
// x < 0.
// 0 ≤ −x, ou seja, x ≤ 0.
x = -x;
// 0 ≤ x.
```

Mas como $x < 0$ implica que $x \leq 0$, conclui-se que o código está correcto.

Em geral, portanto, quando se escrevem duas asserções em sequência, a primeira inserção implica a segunda, ou seja,

```
// A1.
// A2.
```

só pode acontecer se $A_1 \Rightarrow A_2$. Se as duas asserções surgirem separadas por uma instrução, então se a primeira asserção se verificar antes da instrução a segunda asserção verificar-se-á depois da instrução ser executada.

4.2.4 Asserções em instruções de selecção

Suponha-se de novo o troço de código que pretende ordenar os valores das variáveis `x` e `y` (que se presume serem do tipo `int`) de modo que `x` ≤ `y`,

```
if(y < x) {
    int const t = x;
    x = y;
    y = t;
}
```

Qual a CO e qual a PC? Considerem-se x e y os valores das variáveis `x` e `y` antes da instrução de selecção. Então a PC e a CO podem ser escritas

```
// PC ≡ x = x ∧ y = y.
if(y < x) {
    int const t = x;
    x = y;
    y = t;
}
// CO ≡ x ≤ y ∧ ((x = x ∧ y = y) ∨ (x = y ∧ y = x)).
```

Ou seja, o problema fica resolvido quando as variáveis `x` e `y` mantêm ou trocam os seus valores iniciais e `x` termina com um valor não-superior ao de `y`.

Determinar uma CO não é fácil. A PC e a CO, em conjunto, constituem a especificação formal do problema. A sua escrita obriga à compreensão profunda do problema a resolver, e daí a dificuldade.

Será que a instrução condicional conduz forçosamente da PC à CO? É necessário demonstrá-lo.

Partindo da pré-condição

É conveniente começar por converter a instrução condicional na instrução de selecção equivalente e, simultaneamente, explicitar as guardas das instruções alternativas:

```
// PC ≡ x = x ∧ y = y.
if(y < x) {
    // G1 ≡ y < x.
    int const t = x;
    x = y;
    y = t;
} else
    // G2 ≡ x ≤ y.
    ; // instrução nula!
```

A PC, sendo verdadeira antes da instrução de selecção, também o será imediatamente antes de qualquer das instruções alternativas, pelo que se pode escrever

```
// PC ≡ x = x ∧ y = y.
if(y < x) {
    // y < x ∧ x = x ∧ y = y.
    int const t = x;
    x = y;
    y = t;
} else
    // x ≤ y ∧ x = x ∧ y = y.
    ; // instrução nula!
```

Pode-se agora deduzir as asserções válidas após cada instrução alternativa:

```
// PC ≡ x = x ∧ y = y.
if(y < x) {
    // y < x ∧ x = x ∧ y = y.
    int const t = x;
    // y < x ∧ x = x ∧ y = y ∧ y < t ∧ t = x.
    x = y;
    // y = y ∧ y < t ∧ t = x ∧ x = y ∧ x < t.
    y = t;
    // t = x ∧ x = y ∧ x < t ∧ y = x ∧ x < y, que implica
    // x ≤ y ∧ x = y ∧ y = x.
} else
    // x ≤ y ∧ x = x ∧ y = y.
    ; // instrução nula!
    // x ≤ y ∧ x = x ∧ y = y, já que a instrução nula não afecta asserções.
```

Conclui-se que, depois da troca de valores entre `x` e `y` na primeira das instruções alternativas, `x` $<$ `y`, o que implica que `x` $\leq$ `y`. Eliminando as asserções intermédias, úteis apenas durante a demonstração,

```
// PC ≡ x = x ∧ y = y.
if(y < x) {
    int const t = x;
    x = y;
    y = t;
    // x ≤ y ∧ x = y ∧ y = x.
} else
    ; // instrução nula!
    // x ≤ y ∧ x = x ∧ y = y, já que a instrução nula não afecta asserções.
// Que asserção é válida aqui?
```

Falta agora deduzir a asserção válida depois da instrução de selecção completa. Esse ponto pode ser atingido depois de se ter passado por qualquer uma das instruções alternativas, pelo que uma asserção que é válida certamente é a disjunção das asserções deduzidas para cada uma das instruções alternativas:

```
// PC ≡ x = x ∧ y = y.
if(y < x) {
    int const t = x;
    x = y;
    y = t;
    // x ≤ y ∧ x = y ∧ y = x.
} else
    ; // instrução nula!
```

```
        // x ≤ y ∧ x = x ∧ y = y, já que a instrução nula não afecta asserções.
    // (x ≤ y ∧ x = x ∧ y = y) ∨ (x ≤ y ∧ x = y ∧ y = x), ou seja
    // x ≤ y ∧ ((x = x ∧ y = y) ∨ (x = y ∧ y = x)).
```

que é exactamente a CO que se pretendia demonstrar válida.

Partindo da condição objectivo

Neste caso começa por se observar que a CO tem de ser válida no final de qualquer das instruções alternativas para que o possa ser no final da instrução de selecção:

```
if(y < x) {
    int const t = x;
    x = y;
    y = t;
    // x ≤ y ∧ ((x = x ∧ y = y) ∨ (x = y ∧ y = x)).
} else
    ; // instrução nula!
    // x ≤ y ∧ ((x = x ∧ y = y) ∨ (x = y ∧ y = x)).
// CO ≡ x ≤ y ∧ ((x = x ∧ y = y) ∨ (x = y ∧ y = x)).
```

Depois vão-se determinando sucessivamente as pré-condições mais fracas de cada instrução (do fim para o início) usando as regras descritas acima para as atribuições:

```
if(y < x) {
    // y ≤ x ∧ ((y = x ∧ x = y) ∨ (y = y ∧ x = x)).
    int const t = x;
    // y ≤ t ∧ ((y = x ∧ t = y) ∨ (y = y ∧ t = x)).
    x = y;
    // x ≤ t ∧ ((x = x ∧ t = y) ∨ (x = y ∧ t = x)).
    y = t;
    // x ≤ y ∧ ((x = x ∧ y = y) ∨ (x = y ∧ y = x)).
} else
    // x ≤ y ∧ ((x = x ∧ y = y) ∨ (x = y ∧ y = x)).
    ; // instrução nula!
    // x ≤ y ∧ ((x = x ∧ y = y) ∨ (x = y ∧ y = x)).
// CO ≡ x ≤ y ∧ ((x = x ∧ y = y) ∨ (x = y ∧ y = x)).
```

Eliminando as asserções intermédias obtém-se:

```
if(y < x) {
    // y ≤ x ∧ ((y = x ∧ x = y) ∨ (y = y ∧ x = x)).
    int const t = x;
    x = y;
```

```
        y = t;
    } else
        // x ≤ y ∧ ((x = x ∧ y = y) ∨ (x = y ∧ y = x)).
        ; // instrução nula!
    // CO ≡ x ≤ y ∧ ((x = x ∧ y = y) ∨ (x = y ∧ y = x)).
```

Basta agora verificar se a PC em conjunção com cada uma das guardas implica a respectiva asserção deduzida. Ou seja, sabendo o que se sabe desde o início, a pré-condição PC, e sabendo que a primeira instrução alternativa será executada, e portanto a guarda G_1, será que a pré-condição mais fraca dessa instrução alternativa se verifica? E o mesmo para a segunda instrução alternativa? Resumindo, tem de se verificar se:

1. $PC \wedge G_1 \Rightarrow \mathtt{y} \leq \mathtt{x} \wedge ((\mathtt{y} = x \wedge \mathtt{x} = y) \vee (\mathtt{y} = y \wedge \mathtt{x} = x))$ e
2. $PC \wedge G_2 \Rightarrow \mathtt{x} \leq \mathtt{y} \wedge ((\mathtt{x} = x \wedge \mathtt{y} = y) \vee (\mathtt{x} = y \wedge \mathtt{y} = x))$ são implicações verdadeiras.

É fácil verificar que o são de facto.

Resumo

Em geral, para demonstrar a correcção de uma instrução de selecção com n alternativas, ou seja, com n instruções alternativas

```
    // PC
    if(C₁)
        // G₁ ≡ C₁.
        instrução₁
    else if(C₂)
        // G₂ ≡ ¬G₁ ∧ C₂  (ou, ¬C₁ ∧ C₂).
        instrução₂
    else if(C₃)
        // G₃ ≡ ¬G₁ ∧ ¬G₂ ∧ C₃  (ou, ¬C₁ ∧ ¬C₂ ∧ C₃).
        instrução₃
    ...
    else if(Cₙ₋₁)
        // Gₙ₋₁ ≡ ¬G₁ ∧ ··· ∧ ¬Gₙ₋₂ ∧ Cₙ₋₁  (ou, ¬C₁ ∧ ··· ∧ ¬Cₙ₋₂ ∧ Cₙ₋₁).
        instruçãoₙ₋₁
    else
        // Gₙ ≡ ¬G₁ ∧ ··· ∧ ¬Gₙ₋₁  (ou, ¬C₁ ∧ ··· ∧ ¬Cₙ₋₁).
        instruçãoₙ
    // CO
```

seguem-se os seguintes passos:

Demonstração directa Partindo da PC:

1. Para cada instrução alternativa `instrução`$_i$ (com $i = 1 \ldots n$), deduz-se a respectiva CO_i admitindo que a pré-condição PC da instrução de selecção e a guarda G_i da instrução alternativa são ambas verdadeiras. Ou seja, deduz-se CO_i tal que:

   ```
   // PC ∧ G_i
   instrução_i
   // CO_i
   ```

2. Demonstra-se que $(CO_1 \vee CO_2 \vee \cdots \vee CO_n) \Rightarrow CO$. Ou, o que é o mesmo, que $(CO_1 \Rightarrow CO) \wedge (CO_2 \Rightarrow CO) \wedge \cdots \wedge (CO_n \Rightarrow CO)$.

Demonstração inversa Partindo da CO:

1. Para cada instrução alternativa `instrução`$_i$ (com $i = 1 \ldots n$), determina-se a pré-condição mais fraca PC_i que leva forçosamente à CO da instrução de selecção. Ou seja, determina-se a PC_i mais fraca tal que:

   ```
   // PC_i
   instrução_i
   // CO
   ```

2. Demonstra-se que $PC \wedge G_i \Rightarrow PC_i$ para $i = 1 \ldots n$.

Não é necessário verificar se pelo menos uma guarda é sempre verdadeira, porque, por construção da instrução de selecção, esta termina sempre com um `else`. Se isso não acontecer, é necessário fazer a demonstração para a instrução de selecção equivalente em que todos os `if` têm o respectivo `else`, o que pode implicar introduzir uma instrução alternativa nula.

4.3 Desenvolvimento de instruções de selecção

As secções anteriores apresentaram a noção de asserção e sua relação com as instruções de selecção. Falta agora ver como esse formalismo pode ser usado para desenvolver programas.

Regresse-se ao problema inicial, da escrita de uma função para calcular o valor absoluto de um valor inteiro. O objectivo é, portanto, preencher o corpo da função

```
/** Devolve o valor absoluto do argumento.
    @pre  PC ≡ V (ou seja, sem restrições).
    @post CO ≡ absoluto = |x|, ou seja,
              0 ≤ absoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    ...
}
```

onde se usou o predicado $\mathcal{V}$ (verdadeiro) para indicar que a função não tem pré-condição.

Para simplificar o desenvolvimento, pode-se começar o corpo pela definição de uma variável local para guardar o resultado e terminar com a devolução do seu valor, o que permite escrever as asserções principais da função em termos do valor desta variável[2]:

```
/** Devolve o valor absoluto do argumento.
    @pre   PC ≡ V (ou seja, sem restrições).
    @post  CO ≡ absoluto = |x|, ou seja,
               0eqabsoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    // PC ≡ V (ou seja, sem restrições).
    int r;
    ...
    // CO ≡ r = |x|, ou seja, 0 ≤ r ∧ (r = −x ∨ r = x).

    assert(0 <= r and (r == -x or r == x));

    return r;
}
```

Antes de começar o desenvolvimento, é necessário perceber se para a resolução do problema é necessário recorrer a uma instrução de selecção. Neste caso é óbvio que sim, pois tem de se discriminar entre valores negativos e positivos (e talvez nulos) de x.

O desenvolvimento usado será baseado nos objectivos. Este é um princípio importante da programação [8] *a programação deve ser orientada pelos objectivos*. É certo que a pré-condição afecta a solução de qualquer problema, mas os problemas são essencialmente determinados pela condição objectivo. De resto, com se pode verificar depois de alguma prática, mais inspiração para a resolução de um problema pode ser obtida por análise da condição objectivo do que por análise da pré-condição. Por exemplo, é comum não haver qualquer pré-condição na especificação de um problema, donde nesses casos só a condição objectivo poderá ser usada como fonte de inspiração.

4.3.1 Escolha das instruções alternativas

O primeiro passo do desenvolvimento corresponde a identificar possíveis instruções alternativas que pareçam poder levar à veracidade da CO. É fácil verificar que há duas possíveis

[2] A semântica de uma instrução de retorno é muito semelhante à de uma instrução de atribuição. A pré-condição mais fraca pode ser obtida por substituição, na CO da função,do nome da função pela expressão usada na instrução de retorno. Assim, a pré-condição mais fraca da instrução `return r;` que leva à condição objectivo

$$CO \equiv \texttt{absoluto} = |x|, \text{ ou seja, } 0eq\texttt{absoluto} \wedge (\texttt{absoluto} = -\texttt{x} \vee \texttt{absoluto} = \texttt{x}).$$

é

$$CO \equiv \texttt{r} = |x|, \text{ ou seja, } 0 \leq \texttt{r} \wedge (\texttt{r} = -\texttt{x} \vee \texttt{r} = \texttt{x}).$$

No entanto, a instrução de retorno difere da instrução de atribuição pelo numa coisa: a instrução de retorno termina a execução da função.

instruções nessas condições: `r = -x;` e `r = x;`. Estas possíveis instruções podem ser obtidas por simples observação da CO.

4.3.2 Determinação das pré-condições mais fracas

O segundo passo corresponde a verificar em que circunstâncias estas instruções alternativas levam à veracidade da CO. Ou seja, quais as pré-condições PC_i mais fracas que garantem que, depois da respectiva instrução i, a condição CO é verdadeira. Comece-se pela primeira instrução:

```
r = -x;
// CO ≡ 0 ≤ r ∧ (r = −x ∨ r = x).
```

Usando a regra da substituição discutida na Secção 4.2.3, chega-se a

```
// CO ≡ 0 ≤ −x ∧ (−x = −x ∨ −x = x), ou seja, x ≤ 0 ∧ (V ∨ x = 0), ou seja, x ≤ 0.
r = -x;
// CO ≡ 0 ≤ r ∧ (r = −x ∨ r = x).
```

Logo, a primeira instrução, `r = -x;`, só conduz aos resultados pretendidos desde que x tenha inicialmente um valor não-positivo, i.e., $PC_1 \equiv \mathtt{x} \leq 0$.

A mesma verificação pode ser feita para a segunda instrução

```
// CO ≡ 0 ≤ x ∧ (x = −x ∨ x = x), ou seja, 0 ≤ x ∧ (x = 0 ∨ V), ou seja, 0 ≤ x.
r = x;
// CO ≡ 0 ≤ r ∧ (r = −x ∨ r = x).
```

Logo, a segunda instrução, `r = x;`, só conduz aos resultados pretendidos desde que x tenha inicialmente um valor não-negativo, i.e., $PC_2 \equiv 0 \leq \mathtt{x}$.

O corpo da função pode-se agora escrever

```
/** Devolve o valor absoluto do argumento.
    @pre   PC ≡ V (ou seja, sem restrições).
    @post  CO ≡ absoluto = |x|, ou seja,
               0eqabsoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    // PC ≡ V (ou seja, sem restrições).
    int r;
    if(C1)
        // G1
        // PC1 ≡ x ≤ 0.
        r = -x;
```

```
    else
        // G₂
        // PC₂ ≡ 0 ≤ x.
        r = x;
    // CO ≡ r = |x|, ou seja, 0 ≤ r ∧ (r = −x ∨ r = x).

    assert(0 <= r and (r == -x or r == x));

    return r;
}
```

4.3.3 Determinação das guardas

O terceiro passo corresponde a determinar as guardas G_i de cada uma das instruções alternativas. De acordo com o que se viu anteriormente, para que a instrução de selecção resolva o problema, é necessário que $PC \wedge G_i \Rightarrow PC_i$. Só assim se garante que, sendo a guarda G_i verdadeira, a instrução alternativa i conduz à condição objectivo desejada. Neste caso PC é sempre $\mathcal{V}$, pelo que dizer que $PC \wedge G_i \Rightarrow PC_i$ é o mesmo que dizer que $G_i \Rightarrow PC_i$, e portanto a forma mais simples de escolher as guardas é fazer simplesmente $G_i = PC_i$. Ou seja,

```
/** Devolve o valor absoluto do argumento.
    @pre  PC ≡ V (ou seja, sem restrições).
    @post CO ≡ absoluto = |x|, ou seja,
              0eqabsoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    // PC ≡ V (ou seja, sem restrições).
    int r;
    if(C₁)
        // G₁ ≡ x ≤ 0.
        r = -x;
    else
        // G₂ ≡ 0eqx.
        r = x;
    // CO ≡ r = |x|, ou seja, 0eqr ∧ (r = −x ∨ r = x).

    assert(0 <= r and (r == -x or r == x));

    return r;
}
```

4.3.4 Verificação das guardas

O quarto passo corresponde a verificar se a pré-condição PC da instrução de selecção implica a veracidade de pelo menos uma das guardas G_i das instruções alternativas. Se isso não acontecer, significa que pode haver casos para os quais nenhuma das guardas seja verdadeira. Se isso acontecer o problema ainda não está resolvido, sendo necessário determinar instruções alternativas adicionais e respectivas guardas até todos os possíveis casos estarem cobertos.

Neste caso a PC não impõe qualquer restrição, ou seja $PC \equiv \mathcal{V}$. Logo, tem de se verificar se $\mathcal{V} \Rightarrow (G_1 \vee G_2)$. Neste caso $G_1 \vee G_2$ é $\mathtt{x} \leq 0 \vee 0 \leq \mathtt{x}$, ou seja, $\mathcal{V}$. Como $\mathcal{V} \Rightarrow \mathcal{V}$, o problema está quase resolvido.

4.3.5 Escolha das condições

No quinto e último passo determinam-se as condições das instruções de selecção encadeadas de modo a obter as guardas entretanto determinadas. De acordo com o que se viu na Secção 4.2.4, $G_1 = C_1$, pelo que a função fica

```
/** Devolve o valor absoluto do argumento.
    @pre    PC ≡ V  (ou seja, sem restrições).
    @post   CO ≡ absoluto = |x|, ou seja,
                 0eqabsoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    // PC ≡ V  (ou seja, sem restrições).
    int r;
    if(x <= 0)
        // G1 ≡ x ≤ 0.
        r = -x;
    else
        // G2 ≡ 0 < x.
        r = x;
    // CO ≡ r = |x|, ou seja, 0eqr ∧ (r = −x ∨ r = x).

    assert(0 <= r and (r == -x or r == x));

    return r;
}
```

A segunda guarda foi alterada, pois de acordo com a Secção 4.2.4 $G_2 = \neg G_1 = 0 < \mathtt{x}$. Esta guarda é mais forte que a guarda originalmente determinada, pelo que a alteração não traz qualquer problema. Na realidade o que aconteceu foi que a semântica da instrução `if` do C++ forçou à escolha de qual das instruções alternativas lida com o caso $\mathtt{x} = 0$.

Finalmente podem-se eliminar as asserções intermédias:

```
/** Devolve o valor absoluto do argumento.
    @pre    PC ≡ V  (ou seja, sem restrições).
    @post   CO ≡ absoluto = |x|, ou seja,
                 0eqabsoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
```

```
{
    int r;
    if(x <= 0)
        // G₁ ≡ x ≤ 0.
        r = -x;
    else
        // G₂ ≡ 0 < x.
        r = x;

    assert(0 <= r and (r == -x or r == x));

    return r;
}
```

4.3.6 Alterando a solução

A solução obtida pode ser simplificada se se observar que, depois de terminada a instrução de selecção, a função se limita a devolver o valor guardado em `r` por uma das duas instruções alternativas: é possível eliminar essa variável e devolver imediatamente o valor apropriado:

```
/** Devolve o valor absoluto do argumento.
    @pre  PC ≡ V (ou seja, sem restrições).
    @post CO ≡ absoluto = |x|, ou seja,
          0eqabsoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    if(x <= 0)
        // G₁ ≡ x ≤ 0.
        return -x;
    else
        // G₂ ≡ 0 < x.
        return x;
}
```

Por outro lado, se a primeira instrução alternativa (imediatamente abaixo do `if`) termina com uma instrução `return`, então não é necessária uma instrução de selecção, bastando uma instrução condicional:

```
/** Devolve o valor absoluto do argumento.
    @pre  PC ≡ V (ou seja, sem restrições).
    @post CO ≡ absoluto = |x|, ou seja,
          0eqabsoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    if(x <= 0)
```

```
        return -x;
    return x;
}
```

Este último formato não é forçosamente mais claro ou preferível ao anterior, mas é muito comum encontrá-lo em programas escritos em C++. É uma expressão idiomática do C++. Uma desvantagem destes formatos é que não permitem usar instruções de asserção para verificar a validade da condição objectivo.

4.3.7 Metodologia

Existem (pelo menos) dois métodos semi-formais para o desenvolvimento de instruções de selecção. O primeiro foi usado para o desenvolvimento na secção anterior, e parte dos objectivos:

1. Determina-se n instruções `instrução`$_i$, com $i = 1 \ldots n$, que pareçam poder levar à veracidade da CO. Tipicamente estas instruções são obtidas por análise da CO.

2. Determina-se as pré-condições mais fracas PC_i de cada uma dessas instruções de modo que conduzam à CO.

3. Determina-se uma guarda G_i para cada alternativa i de modo que $PC \wedge G_i \Rightarrow PC_i$. Se o problema não tiver pré-condição (ou melhor, se $PC \equiv \mathcal{V}$), então pode-se fazer $G_i = PC_i$. Note-se que uma guarda $G_i = \mathcal{F}$ resolve sempre o problema, embora seja tão forte que nunca leve à execução da instrução respectiva! Por isso *deve-se sempre escolher as guardas o mais fracas possível*[3].

4. Verifica-se se, em todas as circunstâncias, pelo menos uma das guardas encontradas é verdadeira. Isto é, verifica-se se $PC \Rightarrow (G_1 \vee \cdots \vee G_n)$. Se isso não acontecer, é necessário acrescentar mais instruções alternativas às encontradas no ponto 1. Para o fazer, identifica-se que casos ficaram por cobrir analisando as guardas já encontradas e a PC.

No segundo método tenta-se primeiro determinar as guardas e só depois se desenvolve as respectivas instruções alternativas:

1. Determina-se os n casos possíveis interessantes, restritos àqueles que verificam a PC, que cobrem todas as hipóteses. Cada caso possível corresponde a uma guarda G_i. A verificação da PC tem de implicar a verificação de pelo menos uma das guardas, ou seja, $PC \Rightarrow (G_1 \vee G_n)$.

2. Para cada alternativa i, encontra-se uma instrução `instrução`$_i$ tal que

   ```
   // PC ∧ G_i
   instrução_i
   // CO_i
   // CO
   ```

[3] Excepto, naturalmente, quando se reconhece que as guardas se sobrepõem. Nesse caso pode ser vantajoso fortalecer as guardas de modo a minimizar as sobreposições, desde que isso não conduza a guardas mais complicadas: a simplicidade é uma enorme vantagem.

ou seja, tal que, se a pré-condição PC e a guarda G_i forem verdadeiras, então a instrução leve forçosamente à veracidade da condição objectivo CO.

Em qualquer dos casos, depois de encontradas as guardas e as respectivas instruções alternativas, é necessário escrever a instrução de selecção com o número de instruções `if` apropriado (para n instruções alternativas são necessárias $n-1$ instruções `if` encadeadas) e escolher as condições C_i apropriadas de modo a obter as guardas pretendidas. Pode-se começar por fazer $C_i = G_i$, com $i = 1 \cdots n-1$, e depois identificar sobreposições e simplificar as condições C_i. Muitas vezes as guardas encontrada contêm sobreposições (e.g., $0 \leq$ `x` e `x` ≤ 0 sobrepõem-se no valor 0) que podem ser eliminadas ao escolher as condições de cada `if`.

4.3.8 Discussão

As vantagens das metodologias informais apresentadas face a abordagens mais ou menos *ad-hoc* do desenvolvimento são pelo menos:

1. Forçam à especificação rigorosa do problema, e portanto à sua compreensão profunda.
2. O desenvolvimento é acompanhado da demonstração de correcção, o que reduz consideravelmente a probabilidade de erros.
3. Não são descurados aparentes pormenores que, na realidade, podem dar origem a erros graves e difíceis de corrigir.

É claro que a experiência do programador e a sua maior ou menor inventiva muitas vezes levem a desenvolvimentos "ao sabor da pena". Não é forçosamente errado, desde que feito em consciência. Recomenda-se, nesses casos, que se tente fazer *a posteriori* uma demonstração de correcção (e não um teste!) para garantir que a inspiração funcionou...

Para se verificar na prática a importância da especificação cuidada do problema, tente-se resolver o seguinte problema:

Escreva uma função que, dados dois argumentos do tipo `int`*, devolva o booleano verdadeiro se o primeiro for múltiplo do segundo e falso no caso contrário.*

Neste ponto o leitor deve parar de ler e tentar resolver o problema.

$$\vdots$$
$$\vdots$$

A abordagem típica do programador é considerar que um inteiro n é múltiplo de outro inteiro m se o resto da divisão de n por m for zero, e passar directo ao código:

```
bool éMúltiplo(int const m, int const n)
{
    if(m % n == 0)
```

```
        return true;
    else
        return false;
}
```

Implementada a função, o programador fornece-a à sua equipa para utilização. Depois de algumas semanas de utilização, o programa em desenvolvimento, na fase de testes, aborta com uma mensagem (ambiente Linux):

```
Floating exception (core dumped)
```

Depois de umas horas no depurador, conclui-se que, se o segundo argumento da função for zero, a função tenta uma divisão por zero, com a consequente paragem do programa.

Neste ponto o leitor deve parar de ler e tentar resolver o problema.

⋮

⋮

Chamado o programador original da função, este observa que não há múltiplos de zero, e portanto tipicamente corrige a função para

```
bool éMúltiplo(int const m, int const n)
{
    if(n != 0)
        if(m % n == 0)
            return true;
        else
            return false;
    else
        return false;
}
```

Corrigida a função e integrada de novo no programa em desenvolvimento, e repetidos os testes, não se encontra qualquer erro e o desenvolvimento continua. Finalmente o programa é fornecido ao cliente. O cliente utiliza o programa durante meses até que detecta um problema estranho, incompreensível. Como tem um contrato de manutenção com a empresa que desenvolveu o programa, comunica-lhes o problema. A equipa de manutenção, da qual o programador original não faz parte, depois de algumas horas de execução do programa em modo depuração e de tentar reproduzir as condições em que o erro ocorre (que lhe foram fornecidas de uma forma parcelar pelo cliente), acaba por detectar o problema: a função devolve `false` quando lhe são passados dois argumentos zero! Mas zero é múltiplo de zero! Rogando pragas ao programador original da função, esta é corrigida para:

```
bool éMúltiplo(int const m, int const n)
{
    if(n != 0)
        if(m % n == 0)
            return true;
        else
            return false;
    else
        if(m == 0)
            return true;
        else
            return false;
}
```

O código é testado e verifica-se que os erros estão corrigidos.

Depois, o programador olha para o código e acha-o demasiado complicado: para quê as instruções de selecção se uma simples instrução de retorno basta?

Neste ponto o leitor deve parar de ler e tentar resolver o problema com uma única instrução de retorno.

⋮

⋮

Basta devolver o resultado de uma expressão booleana que reflicta os casos em que `m` é múltiplo de `n`:

```
bool éMúltiplo(int const m, int const n)
{
    return (m % n == 0 and n != 0) or (m == 0 and n == 0);
}
```

Como a alteração é meramente cosmética, o programador não volta a testar e o programa corrigido é fornecido ao cliente. Poucas horas depois de ser posto ao serviço o programa aborta. O cliente recorre de novo aos serviços de manutenção, desta vez furioso. A equipa de manutenção verifica rapidamente que a execução da função leva a uma divisão por zero como originalmente. Desta vez a correcção do problema é simples: basta inverter os operandos da primeira conjunção:

```
bool éMúltiplo(int const m, int const n)
{
    return (n != 0 and m % n == 0) or (m == 0 and n == 0);
}
```

Esta simples troca corrige o problema porque nos operadores lógicos o cálculo é atalhado, i.e., o operando esquerdo é calculado em primeiro lugar e, se o resultado ficar imediatamente determinado (ou seja, se o primeiro operando numa conjunção for falso ou se o primeiro operando numa disjunção for verdadeiro) o operando direito não chega a ser calculado (ver Secção 2.7.3).

A moral desta estória é que "quanto mais depressa, mais devagar"... O programador deve evitar as soluções rápidas, pouco pensadas e menos verificadas.

Ah! E faltou dizer que há uma solução ainda mais simples:

```
bool éMúltiplo(int const m, int const n)
{
    return (n != 0 and m % n == 0) or m == 0;
}
```

Talvez tivesse sido boa ideia ter começado por especificar a função. Se tal tivesse acontecido ter-se-ia evitado os erros e ter-se-ia chagado imediatamente à solução mais simples...

4.3.9 Outro exemplo de desenvolvimento

Apresenta-se brevemente o desenvolvimento de uma outra instrução alternativa, um pouco mais complexa. Neste caso pretende-se escrever um procedimento de interface

```
void limita(int& x, int const mín, int const máx)
```

que limite o valor de `x` ao intervalo $[\mathtt{mín}, \mathtt{máx}]$, onde se assume que $\mathtt{mín} \leq \mathtt{máx}$. I.e., se o valor de `x` for inferior a `mín`, então deve ser alterado para `mín`, se for superior a `máx`, deve ser alterado para `máx`, e se pertencer ao intervalo, deve ser deixado com o valor original.

Como habitualmente, começa por se escrever a especificação do procedimento, onde x é uma variável matemática usada para representar o valor inicial da variável de programa `x`:

`/**` Limita `x` ao intervalo $[\mathtt{mín}, \mathtt{máx}]$.
@pre $PC \equiv \mathtt{mín} \leq \mathtt{máx} \land \mathtt{x} = x$.
@post $CO \equiv (\mathtt{x} = \mathtt{mín} \land x \leq \mathtt{mín}) \lor (\mathtt{x} = \mathtt{máx} \land \mathtt{máx} \leq x) \lor$
$(\mathtt{x} = x \land \mathtt{mín} \leq x \leq \mathtt{máx})$. `*/`

```
void limita(int& x, int const mín, int const máx)
{
}
```

A observação da condição objectivo conduz imediatamente às seguintes possíveis instruções alternativas:

1. `;` `//` para manter `x` com o valor inicial x.
2. `x = mín;`

3. `x = máx;`

As pré-condições mais fracas para que se verifique a condição objectivo do procedimento depois de cada uma das instruções são

$$
\begin{array}{rcl}
PC_1 & \equiv & (\mathtt{x} = x \land \mathtt{mín} \leq x \leq \mathtt{máx}) \lor (\mathtt{x} = \mathtt{mín} \land x \leq \mathtt{mín}) \lor (\mathtt{x} = \mathtt{máx} \land \mathtt{máx} \leq x) \\
PC_2 & \equiv & (\mathtt{mín} = x \land \mathtt{mín} \leq x \leq \mathtt{máx}) \lor (\mathtt{mín} = \mathtt{mín} \land x \leq \mathtt{mín}) \lor (\mathtt{mín} = \mathtt{máx} \land \mathtt{máx} \leq x) \\
& \equiv & (\mathtt{mín} = x \land \mathtt{mín} \leq x \leq \mathtt{máx}) \lor x \leq \mathtt{mín} \lor (\mathtt{mín} = \mathtt{máx} \land \mathtt{máx} \leq x) \\
PC_3 & \equiv & (\mathtt{máx} = x \land \mathtt{mín} \leq x \leq \mathtt{máx}) \lor (\mathtt{máx} = \mathtt{mín} \land x \leq \mathtt{mín}) \lor (\mathtt{máx} = \mathtt{máx} \land \mathtt{máx} \leq x) \\
& \equiv & (\mathtt{máx} = x \land \mathtt{mín} \leq x \leq \mathtt{máx}) \lor (\mathtt{máx} = \mathtt{mín} \land x \leq \mathtt{mín}) \lor \mathtt{máx} \leq x
\end{array}
$$

A determinação das guardas faz-se de modo a que $PC \land G_i \Rightarrow PC_i$.

No primeiro caso tem-se da pré-condição PC que $\mathtt{x} = x$, pelo que a guarda mais fraca é

$$
\begin{array}{rcl}
G_1 & \equiv & (\mathtt{mín} \leq \mathtt{x} \land \mathtt{x} \leq \mathtt{máx}) \lor \mathtt{x} = \mathtt{mín} \lor \mathtt{x} = \mathtt{máx} \\
& \equiv & \mathtt{mín} \leq \mathtt{x} \land \mathtt{x} \leq \mathtt{máx}
\end{array}
$$

pois os casos $\mathtt{x} = \mathtt{mín}$ e $\mathtt{x} = \mathtt{máx}$ são cobertos pela primeira conjunção dado que a pré-condição PC garante que $\mathtt{mín} \leq \mathtt{máx}$.

No segundo caso, pelas mesmas razões, tem-se que

$$
\begin{array}{rcl}
G_2 & \equiv & (\mathtt{mín} = \mathtt{x} \land \mathtt{mín} \leq \mathtt{x} \leq \mathtt{máx}) \lor \mathtt{x} \leq \mathtt{mín} \lor (\mathtt{mín} = \mathtt{máx} \land \mathtt{máx} \leq \mathtt{x}) \\
& \equiv & \mathtt{mín} = \mathtt{x} \lor \mathtt{x} \leq \mathtt{mín} \lor (\mathtt{mín} = \mathtt{máx} \land \mathtt{máx} \leq \mathtt{x}) \\
& \equiv & \mathtt{x} \leq \mathtt{mín} \lor (\mathtt{mín} = \mathtt{máx} \land \mathtt{máx} \leq \mathtt{x})
\end{array}
$$

pois da pré-condição PC sabe-se que $\mathtt{mín} \leq \mathtt{máx}$, logo se $\mathtt{x} = \mathtt{mín}$ também será $\mathtt{x} \leq \mathtt{máx}$.

Da mesma forma se obtém

$$G_3 \equiv (\mathtt{máx} = \mathtt{mín} \land \mathtt{x} \leq \mathtt{mín}) \lor \mathtt{máx} \leq \mathtt{x}$$

É evidente que há sempre pelo menos uma destas guardas válidas quaisquer que sejam os valores dos argumentos verificando a PC. Logo, não são necessárias quaisquer outras instruções alternativas.

Ou seja, a estrutura da instrução de selecção é (trocando a ordem das instruções de modo a que a instrução nula fique em último lugar):

```
if(C₁)
    // G₂ ≡ x ≤ mín ∨ (mín = máx ∧ máx ≤ x).
    x = mín;
else if(C₂)
    // G₃ ≡ (máx = mín ∧ x ≤ mín) ∨ máx ≤ x.
    x = máx;
else
    // G₁ ≡ mín ≤ x ∧ x ≤ máx.
    ;
```

A observação das guardas demonstra que elas são redundantes para algumas combinações de valores. Por exemplo, se `mín` $=$ `máx` qualquer das guardas G_2 e G_3 se verifica. É fácil verificar, portanto, que as guardas podem ser reforçadas para:

```
if(C₁)
    // G₂ ≡ x ≤ mín.
    x = mín;
else if(C₂)
    // G₃ ≡ máx ≤ x.
    x = máx;
else
    // G₁ ≡ mín ≤ x ∧ x ≤ máx.
    ;
```

de modo a que, caso `mín` $=$ `máx`, seja válida apenas uma das guardas (excepto, claro, quando `x` $=$ `mín` $=$ `máx`, em que se verificam de novo as duas guardas). De igual modo se podem eliminar as redundâncias no caso de `x` ter como valor um dos extremos do intervalo, pois nesse caso a guarda G_1 "dá conta do recado":

```
if(C₁)
    // G₂ ≡ x < mín.
    x = mín;
else if(C₂)
    // G₃ ≡ máx < x.
    x = máx;
else
    // G₁ ≡ mín ≤ x ∧ x ≤ máx.
    ;
```

As condições das instruções `if` são:

```
if(x < mín)
    // G₂ ≡ x < mín.
    x = mín;
else if(máx < x)
    // G₃ ≡ máx < x.
    x = máx;
else
    // G₁ ≡ mín ≤ x ∧ x ≤ máx.
    ;
```

Finalmente, pode-se eliminar o último `else`, pelo que o procedimento fica, já equipado com as instruções de asserção:

```
/** Limita x ao intervalo [mín, máx].
    @pre PC ≡ mín ≤ máx ∧ x = x.
    @post CO ≡ (x = mín ∧ x ≤ mín) ∨ (x = máx ∧ máx ≤ x)∨
              (x = x ∧ mín ≤ x ≤ máx). */
void limita(int& x, int const mín, int const máx)
{
    assert(mín <= máx);

    if(x < mín)
        x = mín;
    else if(máx < x)
        x = máx;

    assert(mín <= x and x <= máx);
}
```

4.4 Variantes das instruções de selecção

4.4.1 O operador ? :

Seja a função que calcula o valor absoluto de um número desenvolvida nas secções anteriores:

```
/** Devolve o valor absoluto do argumento.
    @pre PC ≡ V (ou seja, sem restrições).
    @post CO ≡ absoluto = |x|, ou seja,
              0eqabsoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    if(x <= 0)
        return -x;
    return x;
}
```

Será possível simplificá-la mais? Sim. A linguagem C++ fornece um operador ternário (com três operandos), que permite escrever a solução como[4]

```
/** Devolve o valor absoluto do argumento.
    @pre PC ≡ V (ou seja, sem restrições).
    @post CO ≡ absoluto = |x|, ou seja,
              0eqabsoluto ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
```

[4] O leitor mais atento notou que se alterou a instrução que lida com o caso em que $x = 0$. É que trocar o sinal de zero é uma simples perda de tempo...

```
    return x < 0 ? -x : x;
}
```

Este operador `? :`, também conhecido por *se aritmético*, tem a seguinte sintaxe:

```
condição ? expressão1 : expressão2
```

O resultado do operador é o resultado da expressão *`expressão1`*, se *`condição`* for verdadeira (nesse caso *`expressão2`* não chega a ser calculada), ou o resultado da expressão *`expressão2`*, se `condição` for falsa (nesse caso *`expressão1`* não chega a ser calculada).

4.4.2 A instrução **`switch`**

Suponha-se que se pretende escrever um procedimento que, dado um inteiro entre 0 e 9, o escreve no ecrã por extenso e em português (escrevendo "erro" se o inteiro for inválido). Os nomes dos dígitos decimais em português, como em qualquer outra língua natural, não obedecem a qualquer regra lógica. Assim, o procedimento terá de lidar com cada um dos 10 casos separadamente. Usando a instrução de selecção encadeada:

```
/** Escreve no ecrã, por extenso, um dígito inteiro entre 0 e 9.
    @pre  PC ≡ 0 ≤ dígito < 10.
    @post CO ≡ ecrã contém, para além do que continha originalmente, o dígito
               dado pelo inteiro dígito. */
void escreveDígitoPorExtenso(int const dígito)
{
    assert(0 <= dígito and dígito < 10);

    if(dígito == 0)
        cout << "zero";
    else if(dígito == 1)
        cout << "um";
    else if(dígito == 2)
        cout << "dois";
    else if(dígito == 3)
        cout << "três";
    else if(dígito == 4)
        cout << "quatro";
    else if(dígito == 5)
        cout << "cinco";
    else if(dígito == 6)
        cout << "seis";
    else if(dígito == 7)
        cout << "sete";
    else if(dígito == 8)
        cout << "oito";
```

```
    else
        cout << "nove";
}
```

Existe uma solução mais usual para este problema e que faz uso da instrução de selecção `switch`. Quando é necessário comparar uma variável com um número discreto de diferentes valores, e executar uma acção diferente em cada um dos casos, deve-se usar esta instrução. Esta instrução permite clarificar a solução do problema apresentado:

```
/** Escreve no ecrã, por extenso, um dígito inteiro entre 0 e 9.
    @pre  PC ≡ 0 ≤ dígito < 10.
    @post CO ≡ ecrã contém, para além do que continha originalmente, o dígito
               dado pelo inteiro dígito. */
void escreveDígitoPorExtenso(int const dígito)
{
    assert(0 <= dígito and dígito < 10);

    switch(dígito) {
      case 0:
        cout << "zero";
        break;

      case 1:
        cout << "um";
        break;

      case 2:
        cout << "dois";
        break;

      case 3:
        cout << "três";
        break;

      case 4:
        cout << "quatro";
        break;

      case 5:
        cout << "cinco";
        break;

      case 6:
        cout << "seis";
        break;
```

```
        case 7:
          cout << "sete";
          break;

        case 8:
          cout << "oito";
          break;

        case 9:
          cout << "nove";
}
```

Esta instrução não permite a especificação de gamas de valores nem de desigualdades: construções como `case 1..10:` ou `case < 10:` são inválidas. Assim, é possível usar como expressão de controlo do `switch` (i.e., a expressão que se coloca entre parênteses após a palavra-chave `switch`) apenas expressões de um dos tipos inteiros ou de um tipo enumerado (ver Capítulo 6), devendo as constantes colocadas nos casos a diferenciar ser do mesmo tipo.

É possível agrupar vários casos ou alternativas:

```
switch(valor) {
  case 1:
  case 2:
  case 3:
    cout << "1, 2 ou 3";
    break;
  ...
}
```

Isto acontece porque a construção `case` n`:` apenas indica qual o ponto de entrada nas instruções que compõem o `switch` quando a sua expressão de controlo tem valor n. A execução do corpo do `switch` (o bloco de instruções entre `{}`) só termina quando for atingida a chaveta final ou quando for executada uma instrução de `break`. Terminado o `switch`, a execução continua sequencialmente após a chaveta final. A consequência deste "agulhamento" do fluxo de instrução é que, se no exemplo anterior se eliminarem os `break`

```
/** Escreve no ecrã, por extenso, um dígito inteiro entre 0 e 9.
    @pre  PC ≡ 0 ≤ dígito < 10.
    @post CO ≡ ecrã contém, para além do que continha originalmente, o dígito
              dado pelo inteiro dígito. */
void escreveDígitoPorExtenso(int const dígito)
{
    // Código errado!
    switch(dígito) {
      case 0:
        cout << "zero";
```

```
        case 1:
          cout << "um";

        case 2:
          cout << "dois";

        case 3:
          cout << "três";

        case 4:
          cout << "quatro";

        case 5:
          cout << "cinco";

        case 6:
          cout << "seis";

        case 7:
          cout << "sete";

        case 8:
          cout << "oito";

        case 9:
          cout << "nove";
    }
```

uma chamada `escreveDígitoPorExtenso(7)` resulta em:

```
seteoitonove
```

que não é de todo o que se pretendia!

Pelas razões que se indicaram atrás, não é possível usar a instrução `switch` (pelo menos de uma forma elegante) como alternativa às instruções de selecção em:

```
/** Escreve no ecrã a ordem de grandeza de um valor.
    @pre  PC ≡ 0 ≤ v < 10000.
    @post CO ≡ ecrã contém, para além do que continha originalmente, a
               ordem de grandeza de v. */
void escreveGrandeza(double v)
{
    assert(0 <= v and v < 10000);
```

```
    if(v < 10)
        cout << "unidades";
    else if(v < 100)
        cout << "dezenas";
    else if(v < 1000)
        cout << "centenas";
    else
        cout << "milhares";
}
```

A ordem pela qual se faz as comparações em instruções de selecção encadeadas pode ser muito relevante em termos do tempo de execução do programa, embora seja irrelevante no que diz respeito aos resultados. Suponha-se que os valores passados como argumento a `escreveGrandeza()` são equiprováveis, i.e., é tão provável ser passado um número como qualquer outro. Nesse caso consegue-se demonstrar que, em média, são necessárias 2,989 comparações ao executar o procedimento e que, invertendo a ordem das instruções alternativas para

```
/** Escreve no ecrã a ordem de grandeza de um valor.
    @pre  PC ≡ 0 ≤ v < 10000.
    @post CO ≡ ecrã contém, para além do que continha originalmente, a
              ordem de grandeza de v. */
void escreveGrandeza(double v)
{
    assert(0 <= v and v < 10000);

    if(1000 <= x)
        cout << "milhares";
    else if(100 <= x)
        cout << "centenas";
    else if(10 <= x)
        cout << "dezenas";
    else
        cout << "unidades";
}
```

são necessárias 1,11 comparações[5]!

No caso da instrução de selecção `switch`, desde que todos os conjuntos de instruções tenham o respectivo `break`, a ordem dos casos é irrelevante.

4.5 Instruções de iteração

Na maioria dos programas há conjuntos de operações que é necessário repetir várias vezes. Para controlar o fluxo do programa de modo a que um conjunto de instruções sejam repetidos

[5] Demonstre-o!

condicionalmente (em ciclos) *usam*-se as *instruções de iteração* ou repetição, que serão estudadas até ao final deste capítulo.

O conhecimento da sintaxe e da semântica das instruções de iteração do C++ não é suficiente para o desenvolvimento de bom código que as utilize. Por um lado, o desenvolvimento de bom código obriga à demonstração formal ou informal do seu correcto funcionamento. Por outro lado, o desenvolvimento de ciclos não é uma tarefa fácil, sobretudo para o principiante. Assim, nas próximas secções apresenta-se um método de demonstração da correcção de ciclos e faz-se uma pequena introdução à metodologia de Dijkstra, que fundamenta e disciplina a construção de ciclos.

4.5.1 A instrução de iteração `while`

O `while` é a mais importante das instruções de iteração. A sua sintaxe é simples:

```
while(expressão_booleana)
    instrução_controlada
```

A execução da instrução `while` consiste na execução repetida da instrução *`instrução_controlada`* enquanto *`expressão_booleana`* tiver o valor lógico verdadeiro. A *`expressão_booleana`* é sempre calculada antes da instrução *`instrução_controlada`*. Assim, se a expressão for inicialmente falsa, a instrução *`instrução_controlada`* não chega a ser executada, passando o fluxo de execução para a instrução subsequente ao `while`. A instrução *`instrução_controlada`* pode consistir em qualquer instrução do C++, e.g., um bloco de instruções ou um outro ciclo.

À expressão booleana de controlo do `while` chama-se a *guarda* do ciclo (representada muitas vezes por G), enquanto à instrução controlada se chama *passo*. Assim, o passo é repetido enquanto a guarda for verdadeira. Ou seja

```
while(G)
    passo
```

A execução da instrução de iteração `while` pode ser representada pelo diagrama de actividade na Figura 4.4.

Exemplo

O procedimento que se segue escreve no ecrã uma linha com todos os números inteiros de zero a `n` (*exclusive*), sendo `n` o seu único parâmetro (não se colocaram instruções de asserção para simplificar):

```
/** Escreve inteiros de 0 a n (exclusive) no ecrã.
    @pre PC ≡ 0 ≤ n.
    @post CO ≡ ecrã contém, para além do que continha originalmente, os
              inteiros entre 0 e n exclusive, em representação decimal. */
```

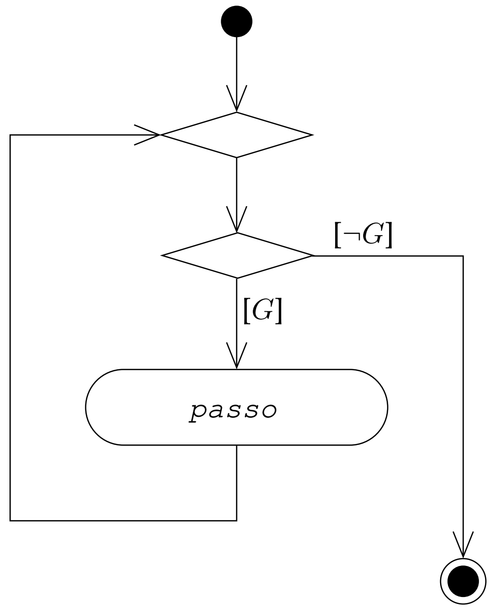


Figura 4.4: Diagrama de actividade da instrução de iteração `while`.

```
void escreveInteiros(int const n) // 1
{                                 // 2
    int i = 0;                    // 3
    while(i != n) {               // 4
        cout << i << ' ';         // 5
        ++i;                      // 6
    }                             // 7
    cout << endl;                 // 8
}                                 // 9
```

O diagrama de actividade correspondente encontra-se na Figura 4.5.

Seguindo o diagrama, começa por se construir a variável `i` com valor inicial 0 (linha 3) e em seguida executa-se o ciclo `while` que:

1. Verifica se a guarda $\mathtt{i} \neq \mathtt{n}$ é verdadeira (linha 4).
2. Se a guarda for verdadeira, executa as instruções nas linhas 5 e 6, voltando depois a verificar a guarda (volta a 1.).
3. Se a guarda for falsa, o ciclo termina.

Depois de terminado o ciclo, escreve-se um fim-de-linha (linha 8) e o procedimento termina.

4.5.2 Variantes do ciclo `while`: `for` e `do while`

A maior parte dos ciclos usando a instrução `while` têm a forma

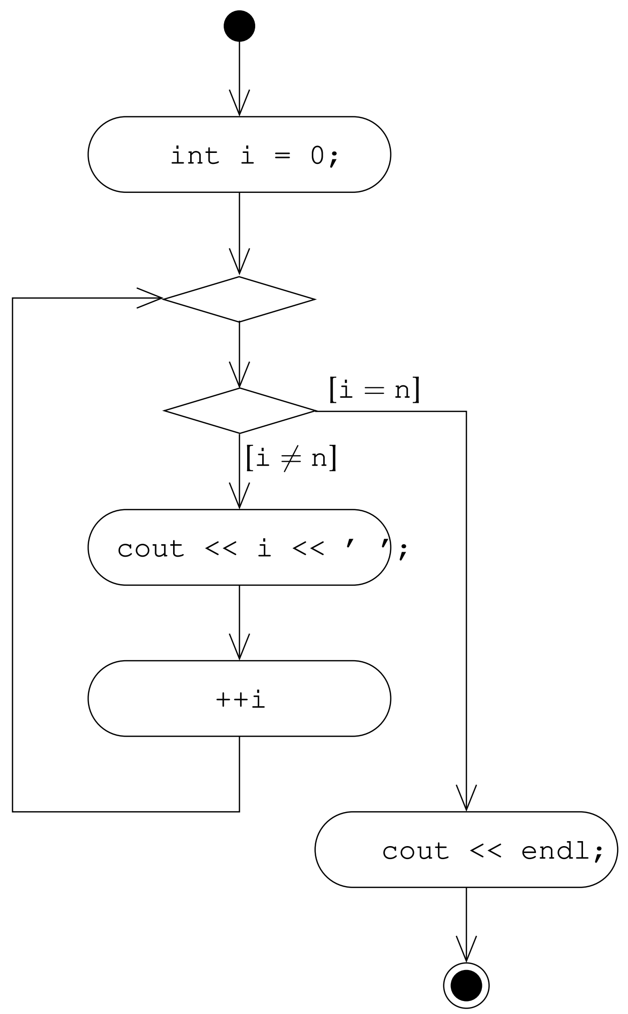


Figura 4.5: Diagrama de actividade do procedimento `escreveInteiros()`.

```
inic
while(G) {
    acção
    prog
}
```

onde `inic` são as instruções de *inicialização* que preparam as variáveis envolvidas no ciclo, `prog` são instruções correspondentes ao chamado *progresso* do ciclo que garantem que o ciclo termina em tempo finito, e `acção` são instruções correspondentes à chamada *acção* do ciclo. Nestes ciclos, portanto, o passo subdivide-se em acção e progresso, ver Figura 4.6.

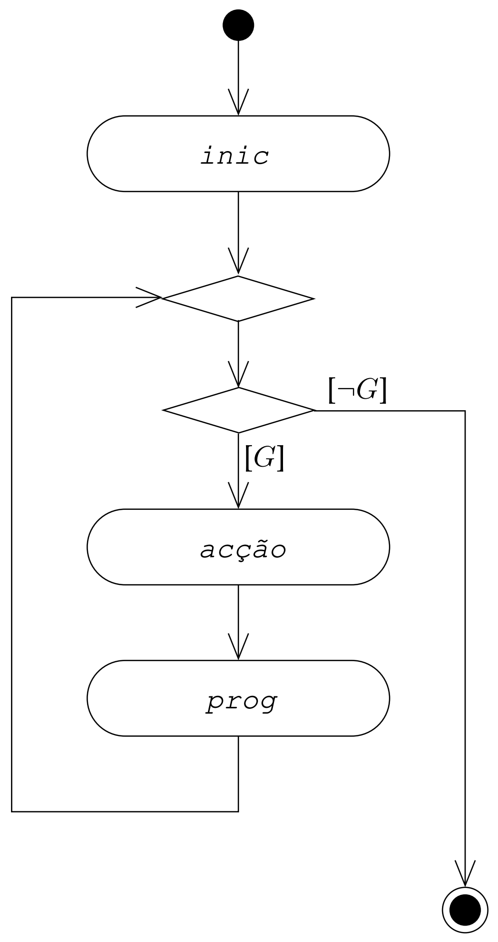


Figura 4.6: Diagrama de actividade de um `ciclo` típico.

Existe uma outra instrução de iteração, a instrução `for`, que permite abreviar este tipo de ciclos:

```
for(inic; G; prog)
    acção
```

em que `inic` e `prog` têm de ser expressões, embora `inic` possa definir também uma variável locais.

O procedimento `escreveInteiros()` já desenvolvido pode ser reescrito como

```
/** Escreve inteiros de 0 a n (exclusive) no ecrã.
    @pre PC ≡ 0 ≤ n.
    @post CO ≡ ecrã contém, para além do que continha originalmente, os
              inteiros entre 0 e n exclusive, em representação decimal. */
void escreveInteiros(int const n) // 1
{                                 // 2
    for(int i = 0; i != n; ++i)   // 3
        cout << i << ' ';         // 4
    cout << endl;                 // 5
}                                 // 6
```

Ao converter o `while` num `for` aproveitou-se para restringir ainda mais a visibilidade da variável `i`. Esta variável, estando definida no *cabeçalho* do `for`, só é visível nessa instrução (linhas 3 e 4). Relembra-se que é de toda a conveniência que a visibilidade das variáveis seja sempre o mais restrita possível à zona onde de facto são necessárias.

Qualquer das expressões no cabeçalho de uma instrução `for` (`inic`, G e `prog`) pode ser omitida. A omissão da guarda G é equivalente à utilização de uma guarda sempre verdadeira, gerando por isso um ciclo infinito, ou laço (*loop*). Ou seja, escrever

```
for(inic; ; prog)
    acção
```

é o mesmo que escrever

```
for(inic; true; prog)
    acção
```

ou mesmo

```
inic;
while(true) {
    acção
    prog
}
```

cujo diagrama de actividade se vê na Figura 4.7.

No entanto, os ciclos infinitos só são verdadeiramente úteis se o seu passo contiver alguma instrução `return` ou `break` (ver Secção 4.5.4) que obrigue o ciclo a terminar alguma vez (nesse caso, naturalmente, deixam de ser infinitos...).

Um outro tipo de ciclo corresponde à instrução `do while`. A sintaxe desta instrução é:

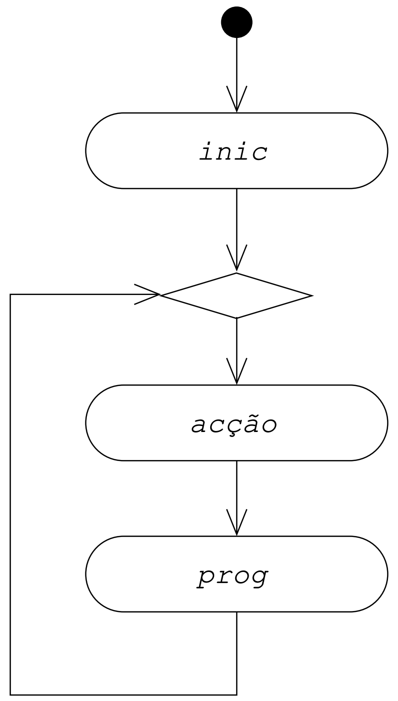


Figura 4.7: Diagrama de actividade de um laço.

```
do
    instrução_controlada
while(expressão_booleana);
```

A execução da instrução `do while` consiste na execução repetida de *`instrução_controlada`* enquanto *`expressão_booleana`* tiver o valor verdadeiro. Mas, ao contrário do que se passa no caso da instrução `while`, *`expressão_booleana`* é sempre calculada e verificada *depois* da instrução *`instrução_controlada`*. Quando o resultado é falso o fluxo de execução passa para a instrução subsequente ao `do while`. Note-se que *`instrução_controlada`* pode consistir em qualquer instrução do C++, e.g., um bloco de instruções.

Tal como no caso da instrução `while`, à instrução controlada pela instrução `do while` chama-se *passo* e à condição que a controla chama-se *guarda*. A execução da instrução de iteração `do while` pode ser representada pelo diagrama de actividade na Figura 4.8.

Ao contrário do que se passa com a instrução `while`, durante a execução de uma instrução `do while` o passo é executado sempre pelo menos uma vez.

Exemplo com `for`

O ciclo `for` é usado frequentemente para repetições ou contagens simples. Por exemplo, se se pretender escrever no ecrã os inteiros de 0 a 9 (um por linha), pode-se usar o seguinte código:

```
for(int i = 0; i != 10; ++i)
    cout << i << endl;
```

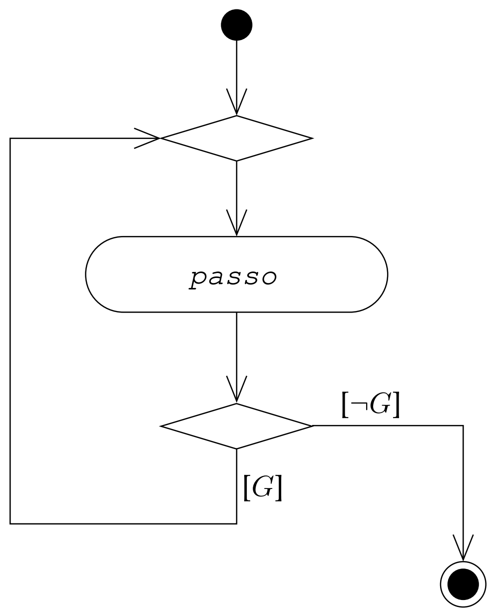


Figura 4.8: Diagrama de actividade da instrução de iteração `do while`.

Se se pretender escrever no ecrã uma linha com 10 asteriscos, pode-se usar o seguinte código:

```
for(int i = 0; i != 10; ++i)
    cout << '*';
cout << endl; // para terminar a linha.
```

É importante observar que, em C++, é típico começar as contagens em zero e usar como guarda o total de repetições a efectuar. Nos ciclos acima a variável `i` toma todos os valores entre 0 e 10 *inclusive*, embora para `i` $= 10$ a acção do ciclo não chegue a ser executada. Alterando o primeiro ciclo de modo a que a definição da variável `i` seja feita fora do ciclo e o seu valor no final do ciclo mostrado no ecrã

```
int i = 0;
for(; i != 10; ++i)
    cout << i << endl;
cout << "O valor final é " << i << '.' << endl;
```

então a execução deste troço de código resultaria em

```
0
1
2
3
4
5
```

```
6
7
8
9
O valor final é 10.
```

Exemplo com `do while`

É muito comum usar-se o ciclo `do while` quando se pretende validar uma entrada de dados por um utilizador do programa.

Suponha-se que se pretende que o utilizador introduza um número inteiro entre 0 e 100 *inclusive*). Se se pretender obrigá-lo à repetição da entrada de dados até que introduza um valor nas condições indicadas, pode-se usar o seguinte código:

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre   PC ≡ V
    @post  CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    cout << "Introduza valor (0 a 100): ";
    cin >> n;
    while(n < 0 or 100 < n) {
        cout << "Valor incorrecto! << endl;
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
    }

    assert(0 <= n and n <= 100);
}
```

A instrução `cin >> n` e o pedido de um valor aparecem duas vezes, o que não parece uma solução muito elegante. Isto deve-se a que, antes de entrar no ciclo para a primeira iteração, tem de fazer sentido verificar se `n` está ou não dentro dos limites impostos, ou seja, a variável `n` tem de ter um valor que não seja arbitrário. A alternativa usando o ciclo `do while` seria:

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre   PC ≡ V
    @post  CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    do {
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
        if(n < 0 or 100 < n)
```

```
            cout << "Valor incorrecto!" << endl;
    } while(n < 0 or 100 < n);

    assert(0 <= n and n <= 100);
}
```

Este ciclo executa o bloco de instruções controlado pelo menos uma vez, dado que a guarda só é avaliada depois da primeira execução. Assim, só é necessária uma instrução `cin >> n` e um pedido de valor. A contrapartida é a necessidade da instrução alternativa `if` dentro do ciclo, com a consequente repetição da guarda... Mais tarde se verão formas alternativas de escrever este ciclo.

Em geral os ciclos `while` e `for` são suficientes, sendo muito raras as ocasiões em que a utilização do ciclo `do while` resulta em código realmente mais claro. No entanto, é má ideia resolver o problema atribuindo um valor inicial à variável `n` que garanta que a guarda é inicialmente verdadeira, de modo a conseguir utilizar o ciclo `while`:

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre PC ≡ V
    @post CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    // Truque sujo: inicialização para a guarda ser inicialmente verdadeira. Má ideia!
    n = -1;
    while(n < 0 or 100 < n) {
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
        if(n < 0 or 100 < n)
            cout << "Valor incorrecto!" << endl;
    }

    assert(0 <= n and n <= 100);
}
```

Quando se tiver de inicializar uma variável de modo a que o passo de um ciclo seja executado pelo menos uma vez, é melhor recorrer a um ciclo `do while`.

Equivalências entre instruções de iteração

É sempre possível converter um ciclo de modo a usar qualquer das instruções de iteração, como indicado na Tabela 4.1. No entanto, a maior parte dos problemas resolvem-se de um modo mais óbvio e mais legível com uma destas instruções do que com as outras.

Tabela 4.1: Equivalências entre ciclos. Estas equivalências não são verdadeiras se as instruções controladas incluírem as instruções `return`, `break`, `continue`, ou `goto` (ver Secção 4.5.4). Há diferenças também quanto ao âmbito das variáveis definidas nestas instruções.

{ *inic* while(G) { *acção* *prog* } }	é equivalente a	for(*inic*; G; *prog*) *acção*
{ *inic* while(G) *passo*	é equivalente a	for(*inic*; G;) *passo*
inic while(G) *passo*	é equivalente a	// Má ideia... inic if(G) do passo while(G);
do *passo* while(G)	é equivalente a	{ *passo* while(G) *passo*

4.5.3 Exemplo simples

A título de exemplo de utilização simultânea de instruções de iteração e de selecção e de instruções de iteração encadeadas, segue-se um programa que escreve no ecrã um triângulo rectângulo de asteriscos com o interior vazio:

```
#include <iostream>

using namespace std;

/** Escreve um triângulo oco de asteriscos com a altura passada como argumento.
    @pre  PC ≡ 0 ≤ altura.
    @post CO ≡ ecrã contém, para além do que continha originalmente,
               altura linhas adicionais representando um triângulo
               rectângulo oco de asteriscos. */
void escreveTriânguloOco(int const altura)
{
    assert(0 <= altura);

    for(int i = 0; i != altura; ++i) {
        for(int j = 0; j != i + 1; ++j)
            if(j == 0 or j == i or i == altura - 1)
                cout << '*';
            else
                cout << ' ';
        cout << endl;
    }
}

int main()
{
    cout << "Introduza a altura do triângulo: ";
    int altura;
    cin >> altura;
    escreveTriânguloOco(altura);
}
```

Sugere-se que o leitor faça o traçado deste programa e que o compile e execute em modo de depuração para compreender bem os dois ciclos encadeados.

4.5.4 `return`, `break`, `continue`, e `goto` em ciclos

Se um ciclo estiver dentro de uma rotina e se pretender retornar (sair da rotina) a meio do ciclo, então pode-se usar a instrução de retorno. Por exemplo, se se pretender escrever uma função que devolva verdadeiro caso o seu parâmetro (inteiro) seja primo e falso no caso contrário, ver-se-á mais tarde que uma possibilidade é (ver Secção 4.7.5):

```
/** Devolve V se n for um número primo e F no caso contrário.
    @pre  PC ≡ 0 ≤ n.
    @post CO ≡ éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
    assert(0 <= n);

    if(n <= 1)
        return false;
    for(int i = 2; i != n; ++i)
        if(n % i == 0)
            // Se se encontrou um divisor ≥ 2 e < n, então n não é primo e pode-se
            // retornar imediatamente:
            return false;
    return true;
}
```

Este tipo de terminação abrupta de ciclos pode ser muito útil, mas também pode contribuir para tornar difícil a compreensão dos programas. Deve portanto ser usado com precaução e parcimónia e apenas em rotinas muito curtas. Noutros casos torna-se preferível usar técnicas de reforço das guardas do ciclo (ver também Secção 4.7.5).

As instruções de `break`, `continue`, e `goto` oferecem outras formas de alterar o funcionamento normal dos ciclos. A sintaxe da última encontra-se em qualquer livro sobre o C++ (e.g., [12]), e não será explicada aqui, desaconselhando-se vivamente a sua utilização.

A instrução `break` serve para terminar abruptamente a instrução `while`, `for`, `do while`, ou `switch` mais interior dentro da qual se encontre. Ou seja, se existirem duas dessas instruções encadeadas, uma instrução `break` termina apenas a instrução interior. A execução continua na instrução subsequente à instrução interrompida.

A instrução `continue` é semelhante à instrução `break`, mas serve para começar antecipadamente a próxima iteração do ciclo (apenas no caso das instruções `while`, `for` e `do while`). Desaconselha-se vivamente a utilização desta instrução.

À instrução `break` aplica-se a recomendação feita quanto à utilização da instrução `return` dentro de ciclos: usar pouco e com cuidado. No entanto, ver-se-á no próximo exemplo que uma utilização adequada das instruções `return` e `break` pode conduzir código simples e elegante.

Exemplo

Considerem-se de novo os dois ciclos alternativos para validar uma entrada de dados por um utilizador:

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
```

```
    @post  CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    cout << "Introduza valor (0 a 100): ";
    cin >> n;
    while(n < 0 or 100 < n) {
        cout << "Valor incorrecto!" << endl;
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
    }

    assert(0 <= n and n <= 100);
}
```

e

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre   PC ≡ V
    @post  CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    do {
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
        if(n < 0 or 100 < n)
            cout << "Valor incorrecto!" << endl;
    } while(n < 0 or 100 < n);

    assert(0 <= n and n <= 100);
}
```

Nenhuma é completamente satisfatória. A primeira porque obriga à repetição da instrução de leitura do valor, que portanto aparece antes da instrução `while` e no seu passo. A segunda porque obriga a uma instrução de selecção cuja guarda é idêntica à guarda do ciclo. O problema está em que teste da guarda deveria ser feito não antes do passo (como na instrução `while`), nem depois do passo (como na instrução `do while`), mas dentro do passo! Ou seja, negando a guarda da instrução de selecção e usando uma instrução `break`,

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre   PC ≡ V
    @post  CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    do {
        cout << "Introduza valor (0 a 100): ";
```

```
        cin >> n;
        if(0 <= n and n <= 100)
            break;
        cout << "Valor incorrecto!" << endl;
    } while(n < 0 or 100 < n);

    assert(0 <= n and n <= 100);
}
```

que, como a guarda do ciclo é sempre verdadeira quando é verificada (quando `n` é válido o ciclo termina na instrução `break`, sem se chegar a testar a guarda), é equivalente a

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    do {
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
        if(0 <= n and n <= 100)
            break;
        cout << "Valor incorrecto!" << endl;
    } while(true);

    assert(0 <= n and n <= 100);
}
```

que é mais comum escrever como

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    while(true) {
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
        if(0 <= n and n <= 100)
            break;
        cout << "Valor incorrecto!" << endl;
    }

    assert(0 <= n and n <= 100);

    return n;
}
```

A Figura 4.9 mostra o diagrama de actividade da função.

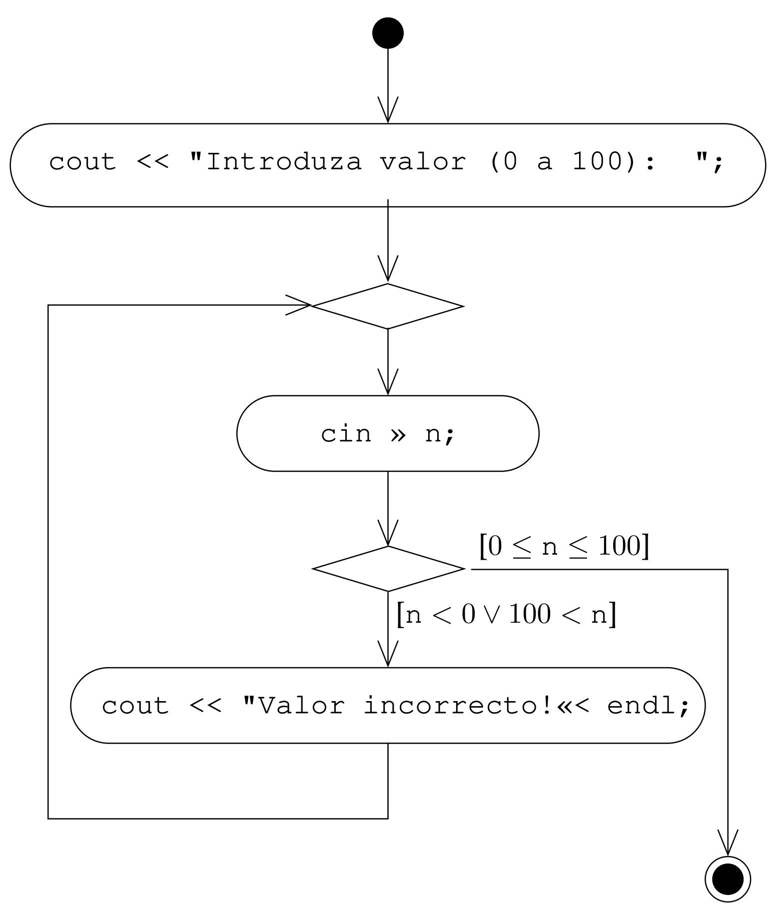


Figura 4.9: Diagrama de actividade da função `inteiroPedidoDe0a100()`.

Os ciclos `do while(true)`, `for(;;)`, e `while(true)` são ciclos infinitos ou laços, que só terminam com recurso a uma instrução `break` ou `return`. Não há nada de errado em ciclos desta forma, desde que recorram a uma e uma só instrução de terminação do ciclo. Respeitando esta restrição, um laço pode ser analisado e a sua correcção demonstrada recorrendo à noção de invariante, como usual.

Uma última versão do ciclo poderia ser escrita recorrendo a uma instrução de retorno, desde que se transformasse o procedimento numa função:

```
/** Devolve um inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ inteiroPedidoDe0a100 ≤ 100. */
int inteiroPedidoDe0a100()
{
    while(true) {
        cout << "Introduza valor (0 a 100): ";
        int n;
```

```
        cin >> n;
        if(0 <= n and n <= 100) {
            assert(0 <= n and n <= 100);
            return n;
        }
        cout << "Valor incorrecto!" << endl;
    }
}
```

Note-se que se passou a definição da variável `n` para o mais perto possível da sua primeira utilização[6].

Esta última versão, no entanto, não é muito recomendável, pois a função tem efeitos laterais: o canal `cin` sofre alterações durante a sua execução. Em geral é má ideia desenvolver mistos de funções e procedimentos, i.e., funções com efeitos laterais ou, o que é o mesmo, procedimentos que devolvem valores.

Nesta altura é conveniente fazer uma pequena digressão e explicar como se pode no procedimento `lêInteiroPedidoDe0a100()` lidar não apenas com erros do utilizador quanto ao valor do inteiro introduzido, mas também com erros mais graves, como a introdução de uma letra em vez de uma sequência de dígitos.

Uma versão "à prova de bala"

Que acontece no procedimento `lêInteiroPedidoDe0a100()` quando o utilizador introduz dados errados, i.e., dados que não podem ser interpretados como um valor inteiro? Infelizmente, o ciclo torna-se infinito, repetindo a mensagem

[6]Noutras linguagens existe uma instrução `loop` para este efeito, que pode ser simulada em C++ recorrendo ao pré-processador (ver Secção 9.2.1):

```
#define loop while(true)
```

Depois desta definição o ciclo pode-se escrever:

```
/** Devolve um inteiro entre 0 e 100 pedido ao utilizador.
    @pre PC ≡ V
    @post CO ≡ 0 ≤ inteiroPedidoDe0a100 ≤ 100. */
int inteiroPedidoDe0a100()
{
    loop {
        cout << "Introduza valor (0 a 100): ";
        int n;
        cin >> n;
        if(0 <= n and n <= 100) {
            assert(0 <= n and n <= 100);
            return n;
        }
        cout << "Valor incorrecto!" << endl;
    }
}
```

```
Introduza valor (0 a 100): Valor incorrecto!
```

eternamente. Isso deve-se ao facto de o canal de entrada `cin` de onde se faz a extracção do valor inteiro, ficar em estado de erro. Uma característica interessante de um canal em estado de erro é que qualquer tentativa de extracção subsequente falhará! Assim, para resolver o problema é necessário limpar explicitamente essa condição de erro usando a instrução

```
cin.clear();
```

que corresponde à invocação de uma operação `clear()` do tipo `istream`, ao qual `cin` pertence (o significado de "operação" neste contexto será visto no Capítulo 7).

Assim, o código original pode ser modificado para

```
/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre   PC ≡ V
    @post  CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    while(true) {
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
        if(not cin) {
            cout << "Isso não é um inteiro!" << endl;
            cin.clear();
        } else if(n < 0 or 100 < n)
            cout << "Valor incorrecto!" << endl;
        else
            break;
    }

    assert(0 <= n and n <= 100);

    return n;
}
```

onde se tirou partido do facto de um canal poder ser interpretado como um valor booleano, correspondendo o valor falso a um canal em estado de erro.

O problema do código acima é que... fica tudo na mesma... Isto acontece porque o caractere erróneo detectado pela operação de extracção mantém-se no canal `cin` apesar de se ter limpo a condição de erro, logo a próxima extracção tem de forçosamente falhar, tal como a primeira, e assim sucessivamente. A solução passa por remover os caracteres erróneos do canal `cin` sempre que se detectar um erro. A melhor forma de o fazer é eliminar toda a linha introduzida pelo utilizador: é mais simples e mais intuitivo para o utilizador do programa.

Para eliminar toda a linha, basta extrair do canal caracteres até se encontrar o caractere que representa o fim-de-linha, i.e., `'\n'`. Para que a extracção seja feita para cada caractere individualmente, coisa que não acontece com o operador `>>`, que por omissão ignora todos os espaços em branco que encontra[7], usa-se a operação `get()` da classe `istream`:

```
/** Extrai todos os caracteres do canal cin até encontrar o fim-de-linha.
    @pre PC ≡ V.
    @post CO ≡Todos os caracteres no canal cin até ao primeiro fim-de-linha
              inclusiva foram extraídos. */
void ignoraLinha()
{
    char caractere;
    do
        cin.get(caractere);
    while(cin and caractere != '\n')
}

/** Lê inteiro entre 0 e 100 pedido ao utilizador.
    @pre PC ≡ V
    @post CO ≡ 0 ≤ n ≤ 100. */
void lêInteiroPedidoDe0a100(int& n)
{
    while(true) {
        cout << "Introduza valor (0 a 100): ";
        cin >> n;
        if(not cin) {
            cout << "Isso não é um inteiro!" << endl;
            cin.clear();
            ignoraLinha();
        } else if(n < 0 or 100 < n)
            cout << "Valor incorrecto!" << endl;
        else
            break;
    }

    assert(0 <= n and n <= 100);

    return n;
}
```

4.5.5 Problemas comuns

De longe o problema mais comum ao escrever ciclos é o "falhanço por um" (*off by one*). Por exemplo, quando se desenvolve um ciclo `for` para escrever 10 asteriscos no ecrã, é comum

[7] Espaços em branco são os caracteres espaço, tabulador, tabulador vertical e fim-de-linha.

errar a guarda do ciclo e escrever

```
for(int i = 0; i <= 10; ++i)
    cout << '*';
```

que escreve um asterisco a mais. É necessário ter cuidado com a guarda dos ciclos, pois estes erros são comuns e muito difíceis de detectar. Na Secção 4.7 estudar-se-ão metodologias de desenvolvimento de ciclos que minimizam grandemente a probabilidade de estes erros ocorrerem.

Outro problema comum corresponde a colocar um `;` após o cabeçalho de um ciclo. Por exemplo:

```
int i = 0;
while(i != 10); // Isto é uma instrução nula!
{
    cout << '*';
    ++i;
}
```

Neste caso o ciclo nunca termina, pois o passo do `while` é a instrução nula, que naturalmente não afecta o valor de `i`. Um caso pior ocorre quando se usa um ciclo `for`:

```
for(int i = 0; i != 10; ++i);
    cout << '*';
```

Este caso é mais grave porque o ciclo termina[8]. Mas, ao contrário do que o programador pretendia, este pedaço de código escreve apenas um asterisco, e não 10!

4.6 Asserções com quantificadores

A especificação de de um problema sem quaisquer ambiguidades é, como se viu, o primeiro passo a realizar para a sua solução. A especificação de um problema faz-se tipicamente indicando a pré-condição (PC) e a condição objectivo (CO). A pré-condição é um predicado acerca das variáveis do problema que se assume ser verdadeiro no início. A condição objectivo é um predicado que se pretende que seja verdadeiro depois de resolvido o problema, i.e., depois de executado o troço de código que o resolve. A pré-condição e a condição objectivo não passam, como se viu antes, de asserções ou afirmações feitas acerca das variáveis de um programa, i.e., acerca do seu estado, sendo o estado de um programa dado pelo valor das suas variáveis em determinado instante de tempo. Assim, a pré-condição estabelece limites ao estado do programa *imediatamente antes* de começar a sua resolução e a condição objectivo estabelece limites ao estado do programa *imediatamente depois* de terminar a sua resolução.

[8]Se não terminar é mais fácil perceber que alguma coisa está errada no código!

A escrita de asserções para problemas um pouco mais complicados que os vistos até aqui requer a utilização de quantificadores. Quantificadores são formas matemáticas abreviadas de escrever expressões que envolvem a repetição de uma dada operação. Exemplos são os quantificadores aritméticos somatório e produto, e os quantificadores lógicos universal ("qualquer que seja") e existencial ("existe pelo menos um").

Apresentam-se aqui algumas notas sobre os quantificadores que são mais úteis para a construção de asserções acerca do estado dos programas. A notação utilizada encontra-se resumida no Apêndice A.

Os quantificadores terão sempre a forma

$$(\mathbf{X}\, v\, :\, \text{predicado}(v)\, :\, \text{expressão}(v))$$

onde

X indica a operação realizada: **S** para a soma (somatório), **P** para o produto ("produtório" ou "piatório"), **Q** para a conjunção ("qualquer que seja"), e **E** para a disjunção ("existe pelo menos um").

v é uma variável muda, que tem significado apenas dentro do quantificador, e que se assume normalmente pertencer ao conjunto dos inteiros. Se pertencer a outro conjunto tal pode ser indicado explicitamente. Por exemplo:

$$\left(\mathbf{Q}\, x \in \mathbb{R}\, :\, \sqrt{2} \leq |x|\, :\, 0 \leq x^2 - 2\right).$$

Um quantificador pode possuir mais do que uma variável muda. Por exemplo:

$$(\mathbf{S}\, i, j\, :\, 0 \leq i < m \wedge 0 \leq j < n\, :\, f(i, j))$$

$\text{predicado}(v)$ é um predicado envolvendo a variável muda v e que define implicitamente o conjunto de valores de v para os quais deve ser realizada a operação.

$\text{expressão}(v)$ é uma expressão envolvendo a variável muda v e que deve ter um resultado aritmético no caso dos quantificadores aritméticos e lógico no caso dos quantificadores lógicos. I.e., no caso dos quantificadores lógicos "qualquer que seja" e "existe pelo menos um", essa expressão deve ser também um predicado.

4.6.1 Somas

O quantificador soma corresponde ao usual somatório. Na notação utilizada,

$$(\mathbf{S}\, j\, :\, m \leq j < n\, :\, \text{expressão}(j))$$

tem exactamente o mesmo significado que o somatório de $\text{expressão}(j)$ com j variando entre m e $n - 1$ *inclusive*. Numa notação mais clássica escrever-se-ia

$$\sum_{j=m}^{n-1} \text{expressão}(j).$$

Por exemplo, é um facto conhecido que o somatório dos primeiros n inteiros não-negativos é $\frac{n(n-1)}{2}$, ou seja,

$$(\mathbf{S}\, j \,:\, 0 \leq j < n \,:\, j) = \frac{n(n-1)}{2} \text{ se } 0 < n.$$

Que acontece se $n \leq 0$? Neste caso é evidente que a variável muda j não pode tomar quaisquer valores, e portanto o resultado é a soma de zero termos. A soma de zero termos é zero, por ser 0 o elemento neutro da soma. Logo,

$$(\mathbf{S}\, j \,:\, 0 \leq j < n \,:\, j) = 0 \text{ se } n \leq 0.$$

Em geral pode dizer que

$$(\mathbf{S}\, j \,:\, m \leq j < n \,:\, \text{expressão}(j)) = 0 \text{ se } n \leq m.$$

Se se pretender desenvolver uma função que calcule a soma dos `n` primeiros números ímpares positivos (sendo `n` um parâmetro da função), pode-se começar por escrever o seu cabeçalho bem como a pré-condição e a condição objectivo, como habitual:

```
/** Devolve a soma dos primeiros n ímpares positivos.
    @pre PC ≡ 0 ≤ n.
    @post CO ≡ somaÍmpares = (S j : 0 ≤ j < n : 2j + 1) */
int somaÍmpares(int const n)
{
    ...
}
```

Mais uma vez fez-se o parâmetro da função constante de modo a deixar claro que a função não lhe altera o valor.

4.6.2 Produtos

O quantificador produto corresponde ao produto de factores por vezes conhecido como "produtório" ou mesmo "piatório". Na notação utilizada,

$$(\mathbf{P}\, j \,:\, 0 \leq j < n \,:\, \text{expressão}(j))$$

tem exactamente o mesmo significado que o usual produto de $\text{expressão}(j)$ com j variando entre m e $n-1$ *inclusive*. Numa notação mais clássica escrever-se-ia

$$\prod_{j=m}^{n-1} \text{expressão}(j).$$

Por exemplo, a definição de factorial é

$$n! = (\mathbf{P}\, j \,:\, 1 \leq j < n+1 \,:\, j)\,.$$

O produto de zero termos é um, por ser 1 o elemento neutro da multiplicação. Ou seja,

$$(\mathbf{P}\, j\, :\, m \leq j < n\, :\, \text{expressão}(j)) = 1 \text{ se } n \leq m.$$

Se se pretender desenvolver uma função que calcule o factorial de `n` (sendo `n` um parâmetro da função), pode-se começar por escrever o seu cabeçalho bem como a pré-condição e a condição objectivo:

```
/** Devolve o factorial de n.
    @pre  PC ≡ 0 ≤ n.
    @post CO ≡ factorial = n! = (P j : 1 ≤ j < n + 1 : j). */
int factorial(int const n)
{
    ...
}
```

4.6.3 Conjunções e o quantificador universal

O quantificador universal corresponde à conjunção (e) de vários predicados usualmente conhecida por "qualquer que seja"[9]. Na notação utilizada,

$$(\mathbf{Q}\, j\, :\, m \leq j < n\, :\, \text{predicado}(j))$$

tem exactamente o mesmo significado que a conjunção dos predicados $\text{predicado}(j)$ com j variando entre m e $n-1$ *inclusive*. Numa notação mais clássica escrever-se-ia

$$\bigwedge_{j=m}^{n-1} \text{predicado}(j)$$

ou ainda

$$\forall m \leq j < n\, :\, \text{predicado}(j).$$

A conjunção de zero predicados tem valor verdadeiro, por ser $\mathcal{V}$ o elemento neutro da conjunção. Ou seja[10],

$$(\mathbf{Q}\, j\, :\, m \leq j < n\, :\, \text{predicado}(j)) = \mathcal{V} \text{ se } n \leq m.$$

Por exemplo, a definição do predicado $\text{primo}(n)$ que tem valor $\mathcal{V}$ se n é primo e $\mathcal{F}$ no caso contrário, é

$$\text{primo}(n) = \begin{cases} (\mathbf{Q}\, j\, :\, 2 \leq j < n\, :\, n \div j \neq 0) & \text{se } 2 \leq n, \text{ e} \\ \mathcal{F} & \text{se } 0 \leq n < 2, \end{cases}$$

[9] Se não é claro para si que o quantificador universal corresponde a uma sequência de conjunções, pense no significado de escrever "todos os humanos têm cabeça". A tradução para linguagem mais matemática seria "qualquer que seja h pertencente ao conjunto dos humanos, h tem cabeça". Como o conjunto dos humanos é finito, pode-se escrever por extenso, listando todos os possíveis humanos: "o António tem cabeça **e** o Sampaio tem cabeça **e** ...". Isto é, a conjunção de todas as afirmações.

[10] A afirmação "todos os marcianos têm cabeça" é verdadeira, pois não existem marcianos. Esta propriedade é menos intuitiva que no caso dos quantificadores soma e produto, mas é importante.

ou seja,

$$\mathrm{primo}(n) = ((\mathbf{Q}\, j : 2 \leq j < n : n \div j \neq 0) \wedge 2 \leq n) \text{ para } 0 \leq n,$$

sendo $\div$ a operação resto da divisão inteira[11].

Se se pretender desenvolver uma função que devolva o valor lógico verdadeiro quando `n` (sendo `n` um parâmetro da função) é um número primo, e falso no caso contrário, pode-se começar por escrever o seu cabeçalho bem como a pré-condição e a condição objectivo:

```
/** Devolve V se n for um número primo e F no caso contrário.
    @pre  PC ≡ 0 ≤ n.
    @post CO ≡ éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
    ...
}
```

4.6.4 Disjunções e o quantificador existencial

O quantificador existencial corresponde à disjunção (ou) de vários predicados usualmente conhecida por "existe pelo menos um"[12]. Na notação utilizada,

$$(\mathbf{E}\, j : m \leq j < n : \mathrm{predicado}(j))$$

tem exactamente o mesmo significado que a disjunção dos predicados $\mathrm{predicado}(j)$ com j variando entre m e $n-1$ *inclusive* e pode-se ler como "existe um valor j entre m e n *exclusive* tal que $\mathrm{predicado}(i)$ é verdadeiro". Numa notação mais clássica escrever-se-ia

$$\bigvee_{j=m}^{n-1} \mathrm{predicado}(j)$$

ou ainda

$$\exists m \leq j < n : \mathrm{predicado}(j).$$

A disjunção de zero predicados tem valor falso, por ser $\mathcal{F}$ o elemento neutro da disjunção. Ou seja[13],

$$(\mathbf{E}\, j : m \leq j < n : \mathrm{predicado}(j)) = \mathcal{F} \text{ se } n \leq m.$$

Este quantificador está estreitamente relacionado com o quantificador universal. É sempre verdade que

$$\neg\left(\mathbf{Q}\, j : m \leq j < n : \mathrm{predicado}(j)\right) = \left(\mathbf{E}\, j : m \leq j < n : \neg\mathrm{predicado}(j)\right),$$

[11] Como o operador `%` em C++.

[12] Se não é claro para si que o quantificador existencial corresponde a uma sequência de disjunções, pense no significado de escrever "existe pelo menos um humano com cabeça". A tradução para linguagem mais matemática seria "existe pelo menos um h pertencente ao conjunto dos humanos tal que h tem cabeça". Como o conjunto dos humanos é finito, pode-se escrever por extenso, listando todos os possíveis humanos: "o Zé tem cabeça **ou** o Sampaio tem cabeça **ou** ...". Isto é, a disjunção de todas as afirmações.

[13] A afirmação "existe pelo menos um marciano com cabeça" é falsa, pois não existem marcianos.

ou seja, se não é verdade que para qualquer j o predicado $\mathrm{predicado}(j)$ é verdadeiro, então existe pelo menos um j para o qual o predicado $\mathrm{predicado}(j)$ é falso. Aplicado à definição de número primo acima, tem-se

$$\begin{aligned}\neg\mathrm{primo}(n) &= \neg\left((\mathbf{Q}\,j\,:\,2\leq j<n\,:\,n\div j\neq 0)\wedge 2\leq n\right)\\ &= \neg(\mathbf{Q}\,j\,:\,2\leq j<n\,:\,n\div j\neq 0)\vee n<2\\ &= (\mathbf{E}\,j\,:\,2\leq j<n\,:\,n\div j=0)\vee n<2\end{aligned}$$

para $0 \leq n$. I.e., um inteiro não-negativo não é primo se for inferior a dois ou se for divisível por algum inteiro superior a 1 e menor que ele próprio.

Se se pretender desenvolver uma função que devolva o valor lógico verdadeiro quando existir um número primo entre `m` e `n` *exclusive* (sendo `m` e `n` parâmetros da função, com `m` não-negativo), e falso no caso contrário, pode-se começar por escrever o seu cabeçalho bem como a pré-condição e a condição objectivo:

```
/** Devolve verdadeiro se só se existir um número primo no intervalo [m, n[.
    @pre PC ≡ 0 ≤ m.
    @post CO ≡ existePrimoNoIntervalo = (E j : m ≤ j < n : primo(j)). */
bool existePrimoNoIntervalo(int const m, int const n)
{
    ...
}
```

4.6.5 Contagens

O quantificador de contagem

$$(\mathbf{N}\,j\,:\,m\leq j<n\,:\,\mathrm{predicado}(j))$$

tem como valor (inteiro) o número de predicados $\mathrm{predicados}(j)$ verdadeiros para j variando entre m e $n-1$ *inclusive*. Por exemplo,

$$(\mathbf{N}\,j\,:\,1\leq j<10\,:\,\mathrm{primo}(j))=4,$$

ou seja, "existem quatro primos entre 1 e 10 *exclusive*", é uma afirmação verdadeira.

Este quantificador é extremamente útil quando é necessário especificar condições em que o número de ordem é fundamental. Se se pretender desenvolver uma função que devolva o `n`-ésimo número primo (sendo `n` parâmetro da função), pode-se começar por escrever o seu cabeçalho bem como a pré-condição e a condição objectivo:

```
/** Devolve o n-ésimo número primo.
    @pre PC ≡ 1 ≤ n.
    @post CO ≡ primo(primo) ∧ (N j : 2 ≤ j < primo : primo(j)) = n − 1.
int primo(int const n)
{
    ...
}
```

A condição objectivo afirma que o valor devolvido é um primo e que existem exactamente $\mathtt{n} - 1$ primos de valor inferior.

4.6.6 O resto da divisão

Apresentou-se atrás o operador resto da definição inteira $\div$ sem que se tenha definido formalmente. A definição pode ser feita à custa de alguns dos quantificadores apresentados: $m \div n$, com $0 \leq m$ e $0 < n$[14], é o único elemento do conjunto $\{0 \leq r < n : (\mathbf{E}\, q : 0 \leq q : m = qn + r)\}$. Claro que a definição está incompleta enquanto não se demonstrar que de facto o conjunto tem um único elemento, ou seja que, quando $0 \leq m$ e $0 < n$, se tem

$$(\mathbf{N}\, r : 0 \leq r < n : (\mathbf{E}\, q : 0 \leq q : m = qn + r)) = 1.$$

Deixa-se a demonstração como exercício para o leitor.

4.7 Desenvolvimento de ciclos

O desenvolvimento de programas usando ciclos é simultaneamente uma arte [10] e uma ciência [8]. Embora a intuição seja muito importante, muitas vezes é importante usar metodologias mais ou menos formais de desenvolvimento de ciclos que permitam garantir simultaneamente a sua correcção. Embora esta matéria seja formalizada na disciplina de Computação e Algoritmia, apresentam-se aqui os conceitos básicos da metodologia de Dijkstra para o desenvolvimento de ciclos. Para uma apresentação mais completa consultar [8].

Suponha-se que se pretende desenvolver uma função para calcular a potência $\mathtt{n}$ de $\mathtt{x}$, isto é, uma função que, sendo $\mathtt{x}$ e $\mathtt{n}$ os seus dois parâmetros, devolva $\mathtt{x}^{\mathtt{n}}$. A sua estrutura básica é

```
double potência(double const x, int const n)
{
    ...
}
```

É claro que $\mathtt{x}^{\mathtt{n}} = \mathtt{x} \times \mathtt{x} \times \cdots \times \mathtt{x}$, ou seja, $\mathtt{x}^{\mathtt{n}}$ pode ser obtido por multiplicação repetida de $\mathtt{x}$. Assim, uma solução passa por usar um ciclo que no seu passo faça cada uma dessas multiplicações:

[14] A definição de resto pode ser generalizada para englobar valores negativos de m e n. Em qualquer dos casos tem de existir um quociente q tal que $m = qn + r$. Mas o intervalo onde r se encontra varia:

$$\begin{array}{rll}
0 \leq r < n & \text{se} & 0 \leq m \wedge 0 < n \text{ (neste caso } 0 \leq q) \\
0 \leq r < -n & \text{se} & 0 \leq m \wedge n < 0 \text{(neste caso } q \leq 0) \\
-n < r \leq 0 & \text{se} & m \leq 0 \wedge 0 < n \text{(neste caso } q \leq 0) \\
n < r \leq 0 & \text{se} & m \leq 0 \wedge n < 0 \text{(neste caso } 0 \leq q)
\end{array}$$

```
double potência(double const x, int const n)
{
    int i = 1;    // usada para contar o número de x já incluídos no produto.
    double r = x; // usada para acumular os produtos de x.
    while(i <= n) {
        r *= x; // o mesmo que r = r * x;
        ++i;    // o mesmo que i = i + 1;
    }
    return r;
}
```

Será que o ciclo está correcto? Fazendo um traçado da função admitindo que é chamada com os argumentos 5,0 e 2, verifica-se facilmente que devolve 125,0 e não 25,0! Isso significa que é feita uma multiplicação a mais. Observando com atenção a guarda da instrução de iteração, conclui-se que esta não deveria deixar o contador `i` atingir o valor de `n`. Ou seja, a guarda deveria ser $\texttt{i} < \texttt{n}$ e não $\texttt{i} \leq \texttt{n}$. Corrigindo a função:

```
double potência(double const x, int const n)
{
    int i = 1;
    double r = x;
    while(i < n) {
        r *= x;
        ++i;
    }
    return r;
}
```

Estará o ciclo definitivamente correcto? Fazendo traçados da função admitindo que é chamada com argumentos 5 e 2 e com 5 e 3, facilmente se verifica que devolve respectivamente 25 e 125, pelo que aparenta estar correcta. Mas estará correcta para todos os valores? E se os argumentos forem 5 e 0? Nesse caso a função devolve 5 em vez do valor correcto, que é $5^0 = 1$! Observando com atenção a inicialização do contador e do acumulador dos produtos, conclui-se que estes deveriam ser inicializados com 0 e 1, respectivamente. Corrigindo a função:

```
double potência(double const x, int const n)
{
    int i = 0;
    double r = 1.0;
    while(i < n) {
        r *= x;
        ++i;
    }
    return r;
}
```

Neste momento a função parece estar correcta. Mas que acontece se o expoente for negativo? Fazendo o traçado da função admitindo que é chamada com os argumentos 5 e -1, facilmente se verifica que devolve 1 em vez de $5^{-1} = 0{,}2$!

Os vários problemas que surgiram ao longo desde desenvolvimento atribulado deveram-se a que:

1. O problema não foi bem especificado através da escrita da pré-condição e da condição objectivo. Em particular a pré-condição deveria estabelecer claramente se a função sabe lidar com expoentes negativos ou não.
2. O desenvolvimento foi feito "ao sabor da pena", de uma forma pouco disciplinada.

Retrocedendo um pouco na resolução do problema de escrever a função `potência()`, é fundamental, pelo que se viu, começar por especificar o problema sem ambiguidades. Para isso formalizam-se a pré-condição e a condição objectivo da função. Para simplificar, suponha-se que a função só deve garantir bom funcionamento para expoentes não-negativos:

```
/** Devolve a potência n de x.
    @pre PC ≡ 0 ≤ n.
    @post CO ≡ potência = x^n. */
double potência(double const x, int const n)
{
    int i = 0;
    double r = 1.0;
    while(i < n) {
        r *= x;
        ++i;
    }
    return r;
}
```

É importante verificar agora o que acontece se o programador consumidor da função se enganar e, violando o contrato expresso pela pré-condição e pela condição objectivo, a invocar com um expoente negativo. Como se viu, a função simplesmente devolve um valor errado: 1. Isso acontece porque, sendo `n` negativo e `i` inicializado com 0, a guarda é inicialmente falsa, não sendo o passo do ciclo executado nenhuma vez. É possível enfraquecer a guarda, de modo a que seja falsa em menos circunstâncias e de tal forma que o contrato da função não se modifique: basta alterar a guarda de $\mathtt{i} < \mathtt{n}$ para $\mathtt{i} \neq \mathtt{n}$. É óbvio que para valores do expoente não-negativos as duas guardas são equivalentes. Mas para expoentes negativos a nova guarda leva a um ciclo infinito! O contador `i` vai crescendo a partir de zero, afastando-se irremediavelmente do valor negativo de `n`.

Qual das guardas será preferível? A primeira, que em caso de engano por parte do programador consumidor da função devolve um valor errado, ou a segunda, que nesse caso entra num ciclo infinito, não chegando a terminar? A verdade é que é preferível a segunda, pois o programador consumidor mais facilmente se apercebe do erro e o corrige em tempo útil[15]. Com a primeira guarda o problema pode só ser detectado demasiado tarde, quando os resultados errados já causaram danos irremediáveis. Assim, a nova versão da função é

```
/** Devolve a potência n de x.
    @pre PC ≡ 0 ≤ n.
    @post CO ≡ potência = xⁿ. */
double potência(double const x, int const n)
{
    int i = 0;
    double r = 1.0;
    while(i != n) {
        r *= x;
        ++i;
    }
    return r;
}
```

onde a guarda é consideravelmente mais fraca do que anteriormente.

Claro está que o ideal é explicitar também a verificação da validade da pré-condição:

```
/** Devolve a potência n de x.
    @pre PC ≡ 0 ≤ n.
    @post CO ≡ potência = xⁿ. */
double potência(double const x, int const n)
{
    assert(0 <= n);

    int i = 0;
    double r = 1.0;
    while(i != n) {
        r *= x;
        ++i;
    }
    return r;
}
```

[15]Note-se que na realidade o ciclo não é infinito, pois os `int` são limitados e incrementações sucessivas levarão o valor do contador ao limite superior dos `int`. O que acontece depois depende do compilador, sistema operativo e máquina em que o programa foi compilado e executado. Normalmente o que acontece é que uma incrementação feita ao contador quando o seu valor já atingiu o limite superior dos `int` leva o valor a "dar a volta" aos inteiros, passando para o limite inferior dos `int` (ver Secção 2.3). Incrementações posteriores levarão o contador até zero, pelo que em rigor o ciclo não é infinito... Mas demora muito tempo a executar, pelo menos, o que mantém a validade do argumento usado para justificar a fraqueza das guardas.

Isto resolve o primeiro problema, pois agora o problema está bem especificado através de uma pré-condição e uma condição objectivo. E o segundo problema, do desenvolvimento indisciplinado?

É possível certamente desenvolver código "ao sabor da pena", mas é importante que seja desenvolvido correctamente. Significa isto que, para ciclos mais simples ou conhecidos, o programador desenvolve-os rapidamente, sem grandes preocupações. Mas para ciclos mais complicados o programador deve ter especial atenção à sua correcção. Das duas uma, ou os desenvolve primeiro e depois demonstra a sua correcção, ou usa uma metodologia de desenvolvimento que garanta a sua correcção[16]. As próximas secções lidam com estes dois problemas: o da demonstração de correcção de ciclos e o de metodologias de desenvolvimento de ciclos. Antes de avançar, porém, é fundamental apresentar a noção de invariante um ciclo.

4.7.1 Noção de invariante

Considerando de novo a função desenvolvida, assinalem-se com números todos as transições entre instruções da função:

```
/** Devolve a potência n de x.
    @pre PC ≡ 0 ≤ n.
    @post CO ≡ potência = x^n. */
double potência(double const x, int const n)
{
    assert(0 <= n);

    int i = 0;
    double r = 1.0;
```

1: Depois da inicialização do ciclo, i.e., depois da inicialização das variáveis nele envolvidas.

```
    while(i != n) {
```

2: Depois de se verificar que a guarda é verdadeira.

```
        r *= x;
```

3: Depois de acumular mais uma multiplicação de `x` em `r`.

```
        ++i;
```

4: Depois de incrementar o contador do número de `x` já incluídos no produto `r`.

```
    }
```

[16]Em alguns casos a utilização desta metodologia não é prática, uma vez que requer um arsenal considerável de modelos: é o caso de ciclos que envolvam leituras do teclado e/ou escritas no ecrã. Nesses casos a metodologia continua aplicável, como é óbvio, embora seja vulgar que as asserções sejam escritas com menos formalidade.

5: Depois de se verificar que a guarda é falsa, imediatamente antes do retorno.

```
        return r;
    }
```

Suponha-se que a função é invocada com os argumentos 3 e 4. Isto é, suponha-se que quando a função começa a ser executada as constantes `x` e `n` têm os valores 3 e 4. Faça-se um traçado da execução da função anotando o valor das suas variáveis em cada uma das transições assinaladas. Obtém-se a seguinte tabela:

Transição	`i`	`r`	**Comentários**
1	0	1	Como $0 \neq 4$, o passo do ciclo será executado, passando-se à transição 2.
2	0	1	
3	0	3	
4	1	3	Como $1 \neq 4$, o passo do ciclo será executado, passando-se à transição 2.
2	1	3	
3	1	9	
4	2	9	Como $2 \neq 4$, o passo do ciclo será executado, passando-se à transição 2.
2	2	9	
3	2	27	
4	3	27	Como $3 \neq 4$, o passo do ciclo será executado, passando-se à transição 2.
2	3	27	
3	3	81	
4	4	81	Como $4 = 4$, o ciclo termina, passando-se à transição 5.
5	4	81	

É fácil verificar que há uma condição relacionando `x`, `r`, `n` e `i` que se verifica em todos as transições do ciclo com excepção da transição 3, i.e., que se verifica *depois da inicialização*, *antes do passo*, *depois do passo*, e no *final do ciclo*. A relação é $\mathtt{r} = \mathtt{x}^{\mathtt{i}} \wedge 0 \leq \mathtt{i} \leq \mathtt{n}$. Esta relação diz algo razoavelmente óbvio: em cada instante (excepto no Ponto 3) o acumulador possui a potência `i` do valor em `x`, tendo `i` um valor entre 0 e `n`. Esta condição, por ser verdadeira ao longo de todo o ciclo (excepto a meio do passo, no Ponto 3), diz-se uma *invariante* do ciclo.

Representando o ciclo na forma de um diagrama de actividade e colocando as asserções que se sabe serem verdadeiras em cada transição do diagrama, é mais fácil compreender a noção de invariante, como se vê na Figura 4.10.

As condições invariantes são centrais na demonstração da correcção de ciclos e durante o desenvolvimento disciplinado de ciclos, como se verá nas próximas secções.

Em geral, para um ciclo da forma

```
inic
while(G)
    passo
```

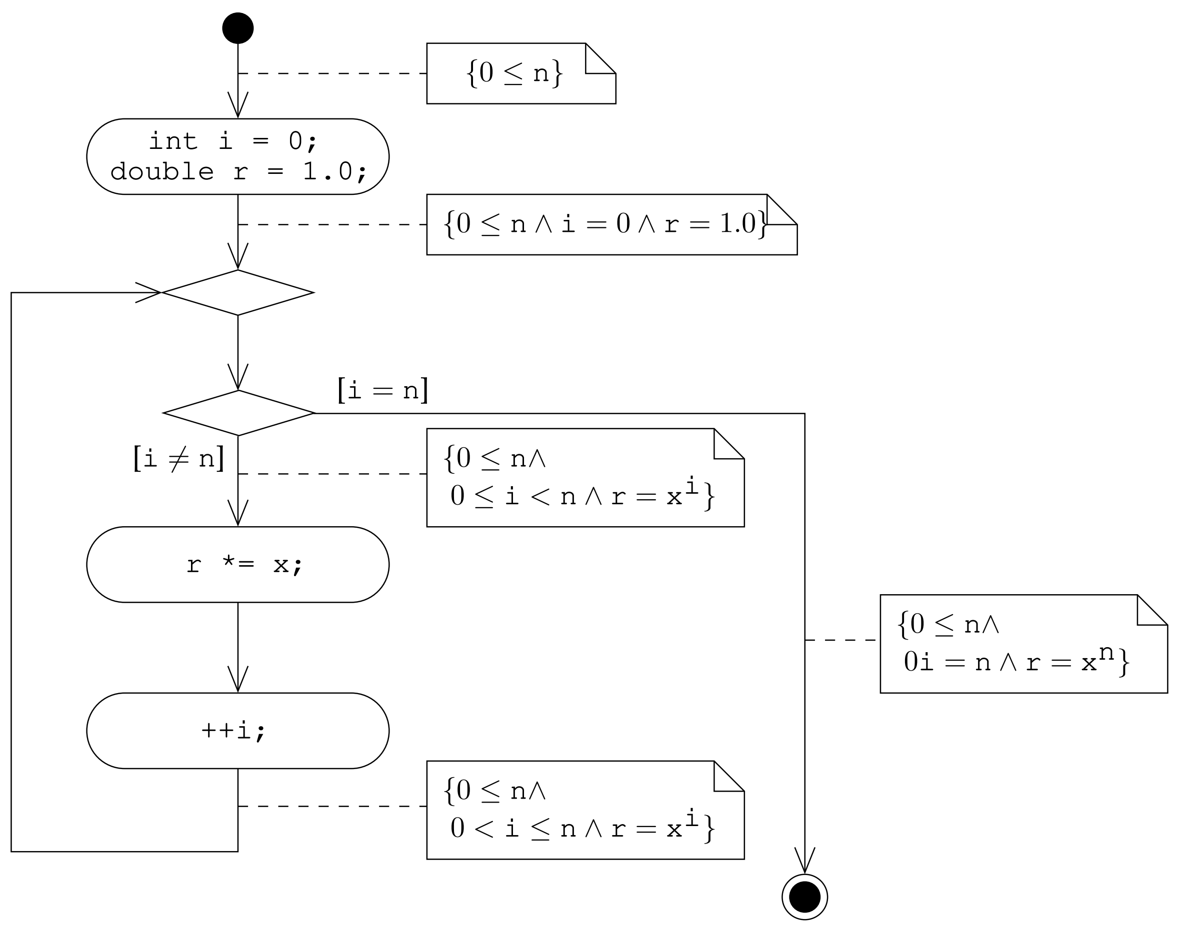


Figura 4.10: Diagrama de actividade do ciclo para cálculo da potência mostrando as asserções nas transições entre instruções (actividades).

uma condição diz-se invariante (CI) se

1. for verdadeira logo após a inicialização do ciclo (`inic`),
2. for verdadeira imediatamente antes do passo (`passo`),
3. for verdadeira imediatamente após o passo e
4. for verdadeira depois de terminado o ciclo.

O diagrama de actividade de um ciclo genérico pode-se ver na Figura 4.11.

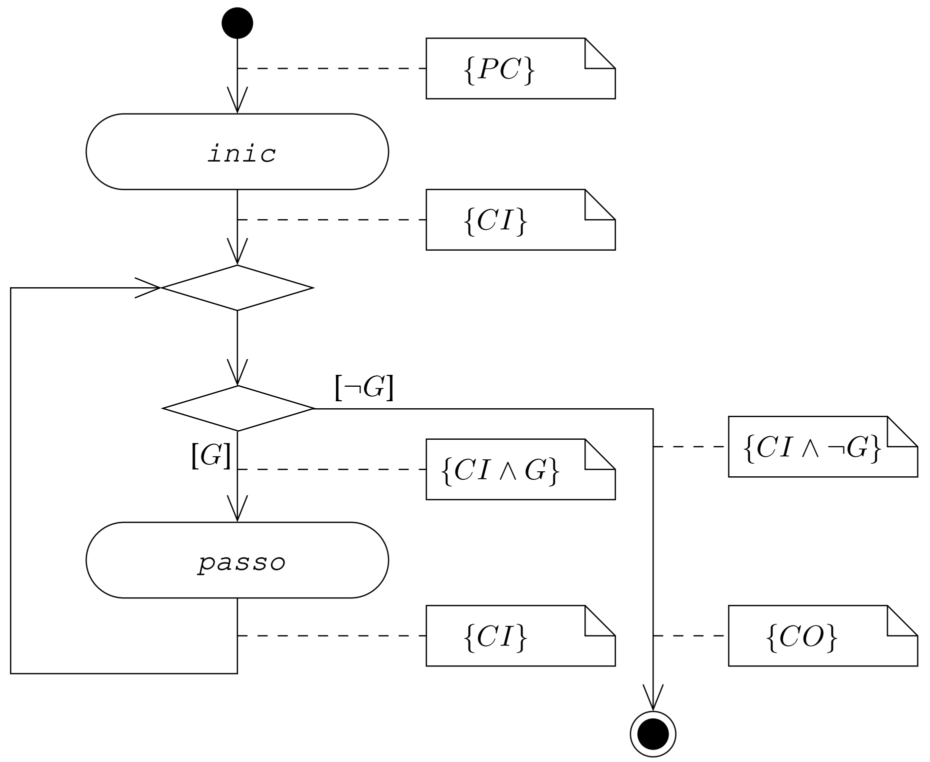


Figura 4.11: Diagrama de actividade de um ciclo genérico. Só as instruções podem alterar o estado do programa e portanto validar ou invalidar asserções.

Incluíram-se algumas asserções adicionais no diagrama. Antes da inicialização assume-se que a pré-condição (PC) do ciclo se verifica, e no final do ciclo pretende-se que se verifique a sua condição objectivo (CO). Além disso, depois da decisão do ciclo, em que se verifica a veracidade da guarda, sabe-se que a guarda (G) é verdadeira antes de executar o passo e falsa (ou seja, verdadeira a sua negação $\neg G$) depois de terminado o ciclo. Ou seja, antes do passo sabe-se que $CI \wedge G$ é uma asserção verdadeira e no final do ciclo sabe-se que $CI \wedge \neg G$ é também uma asserção verdadeira.

No caso da função `potência()`, a condição objectivo do ciclo é diferente da condição objectivo da função, pois refere-se à variável `r`, que será posteriormente devolvida pela função. I.e.:

```
/** Devolve a potência n de x.
    @pre 0 ≤ n.
    @post potência = xⁿ. */
double potência(double const x, int const n)
{
    // PC ≡ 0 ≤ n.
    assert(0 <= n);

    int i = 0;
    double r = 1.0;
    // CI ≡ r = xⁱ ∧ 0 ≤ i ≤ n.
    while(i != n) {
        r *= x;
        ++i;
    }
    // CO ≡ r = xⁿ.
    return r;
}
```

onde se aproveitou para documentar a condição invariante do ciclo.

Demonstração de invariância

Um passo fundamental na demonstração da correcção de um ciclo é a demonstração de invariância de uma dada asserção CI.

Para que a asserção CI possa ser invariante tem de ser verdadeira depois da inicialização (ver Figura 4.11). Assumindo que a pré-condição do ciclo se verifica, então a inicialização *inic* tem de conduzir forçosamente à veracidade da asserção CI. É necessário portanto demonstrar que:

```
// PC
inic
// CI
```

Para o exemplo do cálculo da potência isso é uma tarefa simples. Nesse caso tem-se:

```
// PC ≡ 0 ≤ n.
int i = 0;
double r = 1.0;
// CI ≡ r = xⁱ ∧ 0 ≤ i ≤ n.
```

A demonstração pode ser feita substituindo em CI os valores iniciais de `r` e `i`:

$$\begin{aligned} CI &\equiv \mathtt{r} = \mathtt{x}^{\mathtt{i}} \wedge 0 \leq \mathtt{i} \leq \mathtt{n} \\ &\equiv 1 = \mathtt{x}^0 \wedge 0 \leq 0 \leq \mathtt{n} \\ &\equiv \mathcal{V} \wedge 0 \leq \mathtt{n} \\ &\equiv 0 \leq \mathtt{n}, \end{aligned}$$

que é garantidamente verdadeira dada a PC, ou seja

$$CI \equiv \mathcal{V}$$

Neste momento demonstrou-se a veracidade da asserção CI depois da inicialização. Falta demonstrar que é verdadeira antes do passo, depois do passo, e no fim do ciclo. A demonstração pode ser feita por indução. Supõe-se que a asserção CI é verdadeira antes do passo numa qualquer iteração do ciclo e demonstra-se que é também verdadeira depois do passo nessa mesma iteração. Como antes do passo a guarda é forçosamente verdadeira, pois de outra forma o ciclo teria terminado, é necessário demonstrar que:

```
// CI ∧ G
passo
// CI
```

Se esta demonstração for possível, então por indução a asserção CI será verdadeira ao longo de todo o ciclo, i.e., antes e depois do passo em qualquer iteração do ciclo e no final do ciclo. Isso pode ser visto claramente observando o diagrama na Figura 4.11. Como se demonstrou que a asserção CI é verdadeira depois da inicialização, então também será verdadeira depois de verificada a guarda pela primeira vez, visto que a verificação da guarda não altera qualquer variável do programa[17]. Ou seja, se a guarda for verdadeira, a asserção CI será verdadeira antes do passo na primeira iteração do ciclo e, se a guarda for falsa, a asserção CI será verdadeira no final do ciclo. Se a guarda for verdadeira, como se demonstrou que a veracidade de CI antes do passo numa qualquer iteração implica a sua veracidade depois do passo na mesma iteração, conclui-se que, quando se for verificar de novo a guarda do ciclo (ver diagrama) a asserção CI é verdadeira. Pode-se repetir o argumento para a segunda iteração do ciclo e assim sucessivamente.

Para o caso dado, é necessário demonstrar que:

```
// CI ∧ G ≡ r = x^i ∧ 0 ≤ i ≤ n ∧ i ≠ n ≡ r = x^i ∧ 0 ≤ i < n.
r *= x;
++i;
// CI ≡ r = x^i ∧ 0 ≤ i ≤ n.
```

A demonstração pode ser feita verificando qual a pré-condição mais fraca de cada uma das atribuições, do fim para o princípio (ver Secção 4.2.3). Obtém-se:

```
// r = x^(i+1) ∧ 0 ≤ i + 1 ≤ n, ou seja,
// r = x^i × x ∧ −1 ≤ i ≤ n − 1, ou seja,
// r = x^i × x ∧ −1 ≤ i < n.
++i;    // o mesmo que i = i + 1;
// CI ≡ r = x^i ∧ 0 ≤ i ≤ n.
```

[17]Admite-se aqui que a guarda é uma expressão booleana *sem efeitos laterais*.

ou seja, para que a asserção CI se verifique depois da incrementação de `i` é necessário que se verifique a asserção

$$\mathtt{r} = \mathtt{x}^{\mathtt{i}} \times \mathtt{x} \wedge -1 \leq \mathtt{i} < \mathtt{n}$$

antes da incrementação. Aplicando a mesma técnica à atribuição anterior tem-se que:

```
// r × x = x^i × x ∧ −1 ≤ i < n.
r *= x; // o mesmo que r = r * x;
// r = x^i × x ∧ −1 ≤ i < n.
```

Falta portanto demonstrar que

```
// CI ∧ G ≡ r = x^i ∧ 0 ≤ i < n implica
// r × x = x^i × x ∧ −1 ≤ i < n.
```

mas isso verifica-se por mera observação, pois $0 \leq \mathtt{i} \Rightarrow -1 \leq \mathtt{i}$ e $\mathtt{r} = \mathtt{x}^{\mathtt{i}} \Rightarrow \mathtt{r} \times \mathtt{x} = \mathtt{x}^{\mathtt{i}} \times \mathtt{x}$

Em resumo, para provar a invariância de uma asserção CI é necessário:

1. Mostrar que, admitindo a veracidade da pré-condição antes da inicialização, a asserção CI é verdadeira depois da inicialização *inic*. Ou seja:

   ```
   // PC
   inic
   // CI
   ```

2. Mostrar que, se CI for verdadeira no início do passo numa qualquer iteração do ciclo, então também o será depois do passo nessa mesma iteração. Ou seja:

   ```
   // CI ∧ G
   passo
   // CI
   ```

   sendo a demonstração feita, comummente, do fim para o princípio, deduzindo as pré-condições mais fracas de cada instrução do passo.

4.7.2 Correcção de ciclos

Viu-se que a demonstração da correcção de ciclos é muito importante. Mas como fazê-la? Há que demonstrar dois factos: que o ciclo quando termina garante que a condição objectivo se verifica e que o ciclo termina sempre ao fim de um número finito de iterações. Diz-se que se demonstrou a correcção parcial de um ciclo se se demonstrou que a condição objectivo se verifica quando o ciclo termina, assumindo que a pré-condição se verifica no seu início. Diz-se que se demonstrou a correcção total de um ciclo se, para além disso, se demonstrou que o ciclo termina sempre ao fim de um número finito de iterações.

Correcção parcial

A determinação de uma condição invariante CI apropriada para a demonstração é muito importante. É que, como a CI é verdadeira no final do ciclo e a guarda G é falsa, para demonstrar a correcção parcial do ciclo basta mostrar que

$$CI \wedge \neg G \Rightarrow CO.$$

No caso da função `potência()` essa demonstração é simples:

$$\begin{array}{rcl} CI \wedge \neg G & \equiv & \mathtt{r} = \mathtt{x}^{\mathtt{i}} \wedge 0 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i} = \mathtt{n} \\ & \equiv & \mathtt{r} = \mathtt{x}^{\mathtt{i}} \wedge \mathtt{i} = \mathtt{n} \\ & \Rightarrow & \mathtt{r} = \mathtt{x}^{\mathtt{n}} \equiv CO \end{array}$$

Em resumo, para provar a correcção parcial de um ciclo é necessário

1. encontrar uma asserção CI apropriada (a asserção $\mathcal{V}$ é trivialmente invariante de qualquer ciclo e não ajuda nada na demonstração de correcção parcial: é necessário que a CI seja "rica em informação"),
2. demonstrar que essa asserção CI é de facto uma invariante do ciclo (ver secção anterior) e
3. demonstrar que $CI \wedge \neg G \Rightarrow CO$.

Correcção total

A correcção parcial é insuficiente. Suponha-se a seguinte versão da função `potência()`:

```
/** Devolve a potência n de x.
    @pre 0 ≤ n.
    @post potência = x^n. */
double potência(double const x, int const n)
{
    // PC ≡ 0 ≤ n.
    assert(0 <= n);

    int i = 0;
    double r = 1.0;
    // CI ≡ r = x^i ∧ 0 ≤ i ≤ n.
    while(i != n)
        ; // Instrução nula!
    // CO ≡ r = x^n.
    return r;
}
```

É fácil demonstrar a correcção parcial deste ciclo[18]! Mas este ciclo não termina nunca excepto quando `n` é 0. De acordo com a definição dada na Secção 1.3, este ciclo não implementa um algoritmo, pois não verifica a propriedade da finitude.

A demonstração formal de terminação de um ciclo ao fim de um número finito de iterações faz-se usando o conceito de *função de limitação* (*bound function*) [8], que não será abordado neste texto. Neste contexto será suficiente a demonstração informal desse facto. Por exemplo, na versão original da função `potência()`

```
/** Devolve a potência n de x.
    @pre 0 ≤ n.
    @post potência = xⁿ. */
double potência(double const x, int const n)
{
    // PC ≡ 0 ≤ n.
    assert(0 <= n);

    int i = 0;
    double r = 1.0;
    // CI ≡ r = xⁱ ∧ 0 ≤ i ≤ n.
    while(i != n) {
        r *= x;
        ++i;
    }
    // CO ≡ r = xⁿ.
    return r;
}
```

é evidente que, como a variável `i` começa com o valor zero e é incrementada de uma unidade ao fim de cada passo, fatalmente tem de atingir o valor de `n` (que, pela pré-condição, é não-negativo). Em particular é fácil verificar que o número exacto de iterações necessário para isso acontecer é exactamente `n`.

O passo é, tipicamente, dividido em duas partes: a acção e o progresso. Os ciclos têm tipicamente a seguinte forma:

```
inic
while(G) {
    acção
    prog
}
```

onde o progresso *prog* corresponde ao conjunto de instruções que garante a terminação do ciclo e a acção *acção* corresponde ao conjunto de instruções que garante a invariância de CI apesar de se ter realizado o progresso.

No caso do ciclo da função `potência()` a acção e o progresso são:

[18] Porquê?

```
r *= x; // acção
++i;    // progresso
```

Resumo

Para demonstrar a correcção total de um ciclo:

```
// PC
inic
while(G) {
    acção
    prog
}
// CO
```

é necessário:

1. encontrar uma asserção CI apropriada;
2. demonstrar que essa asserção CI é de facto uma invariante do ciclo:
   (a) mostrar que, admitindo a veracidade de PC antes da inicialização, a asserção CI é verdadeira depois da inicialização *inic*, ou seja:
   ```
   // PC
   inic
   // CI
   ```
   (b) mostrar que, se CI for verdadeira no início do passo numa qualquer iteração do ciclo, então também o será depois do passo nessa mesma iteração, ou seja:
   ```
   // CI ∧ G
   passo
   // CI
   ```
3. demonstrar que $CI \wedge \neg G \Rightarrow CO$; e
4. demonstrar que o ciclo termina sempre ao fim de um número finito de iterações, para o que se raciocina normalmente em termos da inicialização *inic*, da guarda G e do progresso *prog*.

4.7.3 Melhorando a função `potência()`

Ao se especificar sem ambiguidades o problema que a função `potência()` deveria resolver fez-se uma simplificação: admitiu-se que se deveriam apenas considerar expoentes não-negativos. Como relaxar esta exigência? Acontece que, se o expoente tomar valores negativos, então a base da potência não pode ser nula, sob pena de o resultado ser infinitamente grande. Então, a nova especificação será

```
/** Devolve a potência n de x.
    @pre 0 ≤ n ∨ x ≠ 0.
    @post potência = x^n. */
double potência(double const x, int const n)
{
    ...
}
```

É fácil verificar que a resolução do problema exige o tratamento de dois casos distintos, resolúveis através de uma instrução de selecção, ou, mais simplesmente, através do operador `? :`:

```
/** Devolve a potência n de x.
    @pre 0 ≤ n ∨ x ≠ 0.
    @post potência = x^n. */
double potência(double const x, int const n)
{
    assert(0 <= n or x != 0.0);

    int const exp = n < 0 ? -n : n;
    // PC ≡ 0 ≤ exp.
    int i = 0;
    double r = 1.0;
    // CI ≡ r = x^i ∧ 0 ≤ i ≤ exp.
    while(i != exp) {
        r *= x;
        ++i;
    }
    // CO ≡ r = x^exp.
    return n < 0 ? 1.0 / r : r;
}
```

ou ainda, convertendo para a instrução de iteração `for` e eliminando os comentários,

```
/** Devolve a potência n de x.
    @pre PC ≡ 0 ≤ n ∨ x ≠ 0.
    @post CO ≡ potência = x^n. */
double potência(double const x, int const n)
{
    assert(0 <= n or x != 0.0);

    int const exp = n < 0 ? -n : n;
    double r = 1.0;
    for(int i = 0; i != exp; ++i)
        r *= x;
    return n < 0 ? 1.0 / r : r;
}
```

Um outra alternativa é, sabendo que $x^n = \frac{1}{x^{-n}}$, usar recursividade:

```
/** Devolve a potência n de x.
    @pre PC ≡ 0 ≤ n ∨ x ≠ 0.
    @post CO ≡ potência = xⁿ. */
double potência(double const x, int const n)
{
    assert(0 <= n or x != 0.0);

    if(n < 0)
        return 1.0 / potência(x, -n);
    double r = 1.0;
    for(int i = 0; i != n; ++i)
        r *= x;
    return r;
}
```

4.7.4 Metodologia de Dijkstra

A programação deve ser uma actividade orientada pelos objectivos. A metodologia de desenvolvimento de ciclos de Dijkstra [8][5] tenta integrar o desenvolvimento do código com a demonstração da sua correcção, começando naturalmente por prescrever olhar com atenção para a condição objectivo do ciclo. O primeiro passo do desenvolvimento de um ciclo é a determinação de possíveis condições invariantes por observação da condição objectivo, seguindo-se-lhe a determinação da guarda, da inicialização e do passo, normalmente decomposto na acção e no progresso:

1. Especificar o problema sem margem para ambiguidades: definir a pré-condição PC e a condição objectivo CO.

2. Tentar perceber se um ciclo é, de facto, necessário. Este passo é muitas vezes descurado, com consequências infelizes, como se verá.

3. Olhando para a condição objectivo, determinar uma condição invariante CI interessante para o ciclo. A condição invariante escolhida é uma versão enfraquecida da condição objectivo CO.

4. Escolher uma guarda G tal que $CI \wedge \neg G \Rightarrow CO$.

5. Escolher uma inicialização `inic` de modo a garantir a veracidade da condição invariante logo no início do ciclo (assumindo, claro está, que a sua pré-condição PC se verifica).

6. Escolher o passo do ciclo de modo a que a condição invariante se mantenha verdadeira (i.e., de modo a garantir que é de facto invariante). É fundamental que a escolha do passo garanta a terminação do ciclo. Para isso o passo é usualmente dividido num progresso `prog` e numa acção `acção`, sendo o progresso que garante a terminação do ciclo e a acção que garante que, apesar do progresso, a condição invariante CI é de facto invariante.

As próximas secções detalham cada um destes passos para dois exemplos de funções a desenvolver.

Especificação do problema

A pré-condição PC e a condição objectivo CO devem indicar de um modo rigoroso e sem ambiguidade (tanto quanto possível) quais os possíveis estados do programa no início de um pedaço de código e quais os possíveis estados no seu final. Podem-se usar pré-condições e condições objectivo para qualquer pedaço de código, desde uma simples instrução, até um ciclo ou mesmo uma rotina. Pretende-se aqui exemplificar o desenvolvimento de ciclos através da escrita de duas funções. Estas funções têm cada uma uma pré-condição e um condição objectivo, que não são em geral iguais à pré-condição e à condição objectivo do ou dos ciclos que, presumivelmente, elas contêm, embora estejam naturalmente relacionadas.

Função `somaDosQuadrados()` Pretende-se escrever uma função `int somaDosQuadrados(int const n)` que devolva a soma dos quadrados dos primeiros `n` inteiros não-negativos. A sua estrutura pode ser:

```
/** Devolve a soma dos quadrados dos primeiros n inteiros não-negativos.
    @pre  PC ≡ 0 ≤ n.
    @post somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(0 <= n);

    int soma_dos_quadrados = ...;
    ...
    // CO ≡ soma_ dos_ quadrados = (S j : 0 ≤ j < n : j²).
    return soma_dos_quadrados;
}
```

Repare-se que existem duas condições objectivo diferentes. A primeira faz parte do contracto da função (ver Secção 3.2.4) e indica que valor a função deve devolver. A segunda, que é a que vai ser usada no desenvolvimento de ora em diante, indica que valor deve ter a variável `soma_dos_quadrados` antes da instrução de retorno.

Função `raizInteira()` Pretende-se escrever uma função `int raizInteira(int x)` que, assumindo que `x` é não-negativo, devolva o maior inteiro menor ou igual à raiz quadrada de `x`. Por exemplo, se `x` for 4 devolve 2, que é a raiz exacta de 4, se for 3 devolve 1, pois é o maior inteiro menor ou igual a $\sqrt{3} \approx 1{,}73$ e se for 5 devolve 2, pois é o maior inteiro menor ou igual a $\sqrt{5} \approx 2{,}24$. A estrutura da função pode ser:

```
/** Devolve a melhor aproximação inteira por defeito da raiz quadrada de x.
```

```
    @pre   PC ≡ 0eqx.
    @post 0 ≤ raizInteira ≤ √x < raizInteira + 1, ou seja,
          0 ≤ raizInteira ∧ 0 ≤ raizInteira² ≤ x < (raizInteira + 1)². */
int raizInteira(int const x)
{
    assert(0 <= x);

    int r = ...;
    ...
    // CO ≡ 0 ≤ r ∧ r² ≤ x < (r + 1)².

    assert(0 <= r and r * r <= x and x < (r + 1) * (r + 1));

    return r;
}
```

Mais uma vez existem duas condições objectivo diferentes. A primeira faz parte do contracto da função e a segunda, que é a que vai ser usada no desenvolvimento de ora em diante, indica que valor deve ter a variável `r` antes da instrução de retorno. É a relação $\mathtt{x} < (\mathtt{r}+1)^2$ que garante que o valor devolvido é o *maior* inteiro menor ou igual a $\sqrt{\mathtt{x}}$.

Determinando se um ciclo é necessário

A discussão deste passo será feita mais tarde, por motivos que ficarão claros a seu tempo. Para já admite-se que sim, que ambos os problemas devem ser resolvidos usando ciclos.

Determinação da condição invariante e da guarda

A escolha da condição invariante CI do ciclo faz-se sempre olhando para a sua condição objectivo CO. Esta escolha é a fase mais difícil no desenvolvimento de um ciclo. Não existem panaceias para esta dificuldade: é necessário usar de intuição, arte, engenho, experiência, analogias com casos semelhantes, etc. Mas existe uma metodologia, desenvolvida por Edsger Dijkstra [5], que funciona bem para um grande número de casos. A ideia subjacente à metodologia é que a condição invariante se deve obter por enfraquecimento da condição objectivo, ou seja, $CO \Rightarrow CI$. A ideia é que, depois de terminado o ciclo, a condição invariante reforçada pela negação da guarda $\neg G$ (o ciclo termina forçosamente com a guarda falsa) garante que foi atingida a condição objectivo, ou seja, $CI \wedge \neg G \Rightarrow CO$.

Por enfraquecimento da condição objectivo CO entende-se a obtenção de uma condição invariante CI tal que, de entre todas as combinações de valores das variáveis que a tornam verdadeira, contenha todas as combinações de valores de variáveis que tornam verdadeira a CO. Ou seja, o conjunto dos estados do programa que verificam CI deve conter o conjunto dos estados do programa que verificam a CO, que é o mesmo que dizer $CO \Rightarrow CI$.

Apresentar-se-ão aqui apenas dois dos vários possíveis métodos para obter a condição invariante por enfraquecimento da condição objectivo:

1. Substituir uma das constantes presentes em CO por uma variável de programa com limites apropriados, obtendo-se assim a CI. A maior parte das vezes substitui-se uma constante que seja limite de um quantificador. A negação da guarda $\neg G$ é depois escolhida de modo a que $CI \wedge \neg G \Rightarrow CO$, o que normalmente é conseguido escolhendo para $\neg G$ o predicado que afirma que a nova variável tem exactamente o valor da constante que substituiu.

2. Se CO corresponder à conjunção de vários termos, escolher parte deles para constituírem a condição invariante CI e outra parte para constituírem a negação da guarda $\neg G$. Por exemplo, se $CO \equiv C_1 \wedge \cdots \wedge C_m$, então pode-se escolher por exemplo $CI \equiv C_1 \wedge \cdots \wedge C_{m-1}$ e $\neg G \equiv C_m$. Esta selecção conduz, por um lado, a uma condição invariante que é obviamente uma versão enfraquecida da condição objectivo, e por outro lado à verificação trivial da implicação $CI \wedge \neg G \Rightarrow CO$, pois neste caso $CI \wedge \neg G = CO$. A este método chama-se *factorização da condição objectivo*.

Muitas vezes só ao se desenvolver o passo do ciclo se verifica que a condição invariante escolhida não é apropriada. Nesse caso deve-se voltar atrás e procurar uma nova condição invariante mais apropriada.

Substituição de uma constante por uma variável Muitas vezes é possível obter uma condição invariante para o ciclo substituindo uma constante presente na condição objectivo por uma variável de programa introduzida para o efeito. A "constante" pode corresponder a uma variável que não seja suposto ser alterada pelo ciclo, i.e., a uma variável que seja constante apenas do ponto de vista lógico.

Voltando à função `somaDosQuadrados()`, é evidente que o corpo da função pode consistir num ciclo cuja condição objectivo já foi apresentada:

$$CO \equiv \texttt{soma\_ dos\_ quadrados} = \left(\mathbf{S}\, j\, :\, 0 \leq j < \texttt{n}\, :\, j^2\right).$$

A condição invariante do ciclo pode ser obtida substituindo a constante `n` por uma nova variável de programa `i`, com limites apropriados, ficando portanto:

$$CI \equiv \texttt{soma\_ dos\_ quadrados} = \left(\mathbf{S}\, j\, :\, 0 \leq j < \texttt{i}\, :\, j^2\right) \wedge 0 \leq \texttt{i} \leq \texttt{n}.$$

Como se escolheram os limites da nova variável? Simples: um dos limites (normalmente o inferior) é o primeiro valor de `i` para o qual o quantificador não tem nenhum termo (não há qualquer valor de j que verifique $0 \leq j < 0$) e o outro limite (normalmente o superior) é a constante substituída.

Esta condição invariante tem um significado claro: a variável `soma_dos_quadrados` contém desde o início ao fim do ciclo a soma dos primeiros `i` inteiros não-negativos e a variável `i` varia entre 0 e `n`.

Claro está que a nova variável tem de ser definida na função:

```
/** Devolve a soma dos quadrados dos primeiros n inteiros não-negativos.
    @pre PC ≡ 0 ≤ n.
    @post somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(0 <= n);

    int soma_dos_quadrados = ...;
    int i = ...;
    ...
    // CO ≡ soma_dos_quadrados = (S j : 0 ≤ j < n : j²).
    return soma_dos_quadrados;
}
```

Em rigor, a condição invariante escolhida não corresponde simplesmente a uma versão enfraquecida da condição objectivo. Na verdade, a introdução de uma nova variável é feita em duas etapas, que normalmente se omitem.

A primeira etapa obriga a uma reformulação da condição objectivo de modo a reflectir a existência da nova variável, que aumenta a dimensão do espaço de estados do programa: substitui-se a constante pela nova variável, mas força-se também a nova variável a tomar o valor da constante que substituiu, acrescentando para tal uma conjunção à condição objectivo

$$CO' \equiv \texttt{soma\_dos\_quadrados} = \left(\mathbf{S}\, j : 0 \leq j < \texttt{i} : j^2\right) \wedge \texttt{i} = \texttt{n}$$

de modo a garantir que $CO' \Rightarrow CO$.

A segunda etapa corresponde à obtenção da condição invariante por enfraquecimento de CO': o termo que fixa o valor da nova variável é relaxado. A condição invariante obtida é de facto mais fraca que a condição objectivo reformulada CO', pois aceita mais possíveis valores para a variável `soma_dos_quadrados` do que CO', que só aceita o resultado final pretendido[19]. Os valores aceites para essa variável pela condição invariante correspondem a valores intermédios durante a execução do ciclo. Neste caso correspondem a todas as possíveis somas de quadrados de inteiros positivos desde a soma com zero termos até à soma com os `n` termos pretendidos.

A escolha da guarda é simples: para negação da guarda $\neg G$ escolhe-se a conjunção que se acrescentou à condição objectivo para se obter CO'

$$\neg G \equiv \texttt{i} = \texttt{n},$$

[19] Na realidade o enfraquecimento pode ser feito usando a factorização! Para isso basta um pequeno truque na reformulação da condição objectivo de modo a incluir um termo extra:

$$CO' \equiv \texttt{soma\_dos\_quadrados} = \left(\mathbf{S}\, j : 0 \leq j < \texttt{i} : j^2\right) \wedge 0 \leq \texttt{i} \leq \texttt{n} \wedge \texttt{i} = \texttt{n}.$$

Este novo termo não faz qualquer diferença efectiva em CO', mas permite aplicar facilmente a factorização da condição objectivo:

$$CO' \equiv \overbrace{\texttt{soma\_dos\_quadrados} = \left(\mathbf{S}\, j : 0 \leq j < \texttt{i} : j^2\right) \wedge 0 \leq \texttt{i} \leq \texttt{n}}^{CI} \wedge \overbrace{\texttt{i} = \texttt{n}}^{\neg G}.$$

ou seja,

$$G \equiv \texttt{i} \neq \texttt{n}.$$

Dessa forma tem-se forçosamente que $(CI \wedge \neg G) = CO' \Rightarrow CO$, como pretendido. A função neste momento é

```
/** Devolve a soma dos quadrados dos primeiros n inteiros não-negativos.
    @pre  PC ≡ 0 ≤ n.
    @post somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(0 <= n);

    int soma_dos_quadrados = ...;
    int i = ...;
    // CI ≡ soma_ dos_ quadrados = (S j : 0 ≤ j < i : j²) ∧ 0 ≤ i ≤ n.
    while(i != n) {
        passo
    }
    // CO ≡ soma_ dos_ quadrados = (S j : 0 ≤ j < n : j²).
    return soma_dos_quadrados;
}
```

É importante que, pelas razões que se viram mais atrás, a guarda escolhida seja o mais fraca possível. Neste caso pode-se-ia reforçar a guarda relaxando a sua negação para $\neg G \equiv \texttt{n} \leq \texttt{i}$, que corresponde à guarda $G \equiv \texttt{i} < \texttt{n}$ muito mais forte e infelizmente tão comum...

A escolha da guarda pode também ser feita observando que no final do ciclo a guarda será forçosamente falsa e a condição invariante verdadeira (i.e., que $CI \wedge \neg G$), e que se pretende, nessa altura, que a condição objectivo do ciclo seja verdadeira. Ou seja, sabe-se que no final do ciclo

$$\texttt{soma\_ dos\_ quadrados} = \left(\mathbf{S}\, j\, :\, 0 \leq j < \texttt{i}\, :\, j^2\right) \wedge 0 \leq \texttt{i} \leq \texttt{n} \wedge \neg G$$

e pretende-se que

$$CO \equiv \texttt{soma\_ dos\_ quadrados} = \left(\mathbf{S}\, j\, :\, 0 \leq j < \texttt{n}\, :\, j^2\right).$$

A escolha mais simples da guarda que garante a verificação da condição objectivo é $\neg G \equiv \texttt{i} = \texttt{n}$, ou seja, $G \equiv \texttt{i} \neq \texttt{n}$. Ou seja, esta guarda quando for falsa garante que $\texttt{i} = \texttt{n}$, pelo que sendo a condição invariante verdadeira, também a condição objectivo o será.

Mais uma vez, só se pode confirmar se a escolha da condição invariante foi apropriada depois de completado o desenvolvimento do ciclo. Se o não tiver sido, há que voltar a este passo e tentar de novo. Isso não será necessário neste caso, como se verá.

Factorização da condição objectivo Quando a condição objectivo é composta pela conjunção de várias condições, pode-se muitas vezes utilizar parte delas como negação da guarda $\neg G$ e a parte restante como condição invariante CI.

Voltando à função `raizInteira()`, é evidente que o corpo da função pode consistir num ciclo cuja condição objectivo já foi apresentada:

$$\begin{aligned} CO &\equiv 0 \leq \mathtt{r} \wedge \mathtt{r}^2 \leq \mathtt{x} < (\mathtt{r}+1)^2 \\ &\equiv 0 \leq \mathtt{r} \wedge \mathtt{r}^2 \leq \mathtt{x} \wedge \mathtt{x} < (\mathtt{r}+1)^2 \end{aligned}$$

Escolhendo para negação da guarda o termo $\mathtt{x} < (\mathtt{r}+1)^2$, ou seja, escolhendo

$$G \equiv (\mathtt{r}+1)^2 \leq \mathtt{x}$$

obtém-se para a condição invariante os termos restantes

$$CI \equiv 0 \leq \mathtt{r} \wedge \mathtt{r}^2 \leq \mathtt{x}.$$

que é o mesmo que

$$CI \equiv 0 \leq \mathtt{r} \wedge \mathtt{r} \leq \sqrt{\mathtt{x}}.$$

Esta escolha faz com que $CI \wedge \neg G$ seja igual à condição objectivo CO, pelo que *se o ciclo terminar* termina com o valor correcto.

A condição invariante escolhida tem um significado claro: desde o início ao fim do ciclo que a variável `r` não excede a raiz quadrada de `x`. Além disso, a condição invariante é, mais uma vez, uma versão enfraquecida da condição objectivo, visto que admite um maior número de possíveis valores para a variável `r` do que a condição objectivo, que só admite o resultado final pretendido. A função neste momento é

```
/** Devolve a melhor aproximação inteira por defeito da raiz quadrada de x.
    @pre  PC ≡ 0eqx.
    @post 0 ≤ raizInteira ≤ √x < raizInteira + 1, ou seja,
          0 ≤ raizInteira ∧ 0 ≤ raizInteira² ≤ x < (raizInteira + 1)². */
int raizInteira(int const x)
{
    assert(0 <= x);

    int r = ...;
    // CI ≡ 0 ≤ r ∧ r² ≤ x.
    while((r + 1) * (r + 1) <= x) {
        passo
    }
    // CO ≡ 0 ≤ r ∧ r² ≤ x < (r + 1)².

    assert(0 <= r and r * r <= x and x < (r + 1) * (r + 1));

    return r;
}
```

Escolha da inicialização

A inicialização de um ciclo é feita normalmente de modo a, assumindo a veracidade da pré-condição PC, garantir a verificação da condição invariante CI da forma mais simples possível.

Função `somaDosQuadrados()` A condição invariante já determinada é

$$CI \equiv \texttt{soma\_ dos\_ quadrados} = \left(\mathbf{S}\, j\, :\, 0 \le j < \texttt{i}\, :\, j^2\right) \wedge 0 \le \texttt{i} \le \texttt{n}.$$

pelo que a forma mais simples de se inicializar o ciclo é com as instruções

```
int soma_dos_quadrados = 0;
int i = 0;
```

pois nesse caso, por simples substituição,

$$\begin{aligned} CI &\equiv 0 = \left(\mathbf{S}\, j\, :\, 0 \le j < 0\, :\, j^2\right) \wedge 0 \le 0 \le \texttt{n} \\ &\equiv 0 = 0 \wedge 0 \le \texttt{n} \\ &\equiv 0 \le \texttt{n} \end{aligned}$$

que é verdadeira desde que a pré-condição $PC \equiv 0 \le \texttt{n}$ o seja. A função neste momento é

```
/** Devolve a soma dos quadrados dos primeiros n inteiros não-negativos.
    @pre PC ≡ 0 ≤ n.
    @post somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(0 <= n);

    int soma_dos_quadrados = 0;
    int i = 0;
    // CI ≡ soma_ dos_ quadrados = (S j : 0 ≤ j < i : j²) ∧ 0 ≤ i ≤ n.
    while(i != n) {
        passo
    }
    // CO ≡ soma_ dos_ quadrados = (S j : 0 ≤ j < n : j²).
    return soma_dos_quadrados;
}
```

Função `raizInteira()` A condição invariante já determinada é

$$CI \equiv 0 \le \texttt{r} \wedge \texttt{r}^2 \le \texttt{x}.$$

pelo que a forma mais simples de se inicializar o ciclo é com a instrução

```
int r = 0;
```

pois nesse caso, por simples substituição,

$$\begin{aligned} CI &\equiv 0 \leq 0 \wedge 0 \leq \mathtt{x} \\ &\equiv 0 \leq \mathtt{x} \end{aligned}$$

que é verdadeira desde que a pré-condição $PC \equiv 0eq\mathtt{x}$ o seja. A função neste momento é

```
/** Devolve a melhor aproximação inteira por defeito da raiz quadrada de x.
    @pre   PC ≡ 0eqx.
    @post 0 ≤ raizInteira ≤ √x < raizInteira + 1, ou seja,
          0 ≤ raizInteira ∧ 0 ≤ raizInteira² ≤ x < (raizInteira + 1)². */
int raizInteira(int const x)
{
    assert(0 <= n);

    int r = 0;
    // CI ≡ 0 ≤ r ∧ r² ≤ x.
    while((r + 1) * (r + 1) <= x) {
        passo
    }
    // CO ≡ 0 ≤ r ∧ r² ≤ x < (r + 1)².

    assert(0 <= r and r * r <= x and x < (r + 1) * (r + 1));

    return r;
}
```

Determinação do passo: progresso e acção

A construção de um ciclo fica completa depois de determinado o passo, normalmente constituído pela acção `acção` e pelo progresso `prog`. O progresso é escolhido de modo a garantir que o ciclo termina, i.e., de modo a garantir que ao fim de um número finito de repetições do passo a guarda G se torna falsa. A acção é escolhida de modo a garantir a invariância de CI *apesar do progresso*.

Função `somaDosQuadrados()` A guarda já determinada é

$$G \equiv \mathtt{i} \neq \mathtt{n}$$

pelo que o progresso mais simples é a simples incrementação de `i`. Este progresso garante que o ciclo termina (i.e., que a guarda se torna falsa) ao fim de no máximo `n` iterações, uma vez que `i` foi inicializada com 0. Ou seja, o passo do ciclo é

// $CI \wedge G \equiv$ `soma_ dos_ quadrados` $= (\mathbf{S}\, j : 0 \leq j < \mathtt{i} : j^2) \wedge 0 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i} \neq$ `n`, ou seja,
// `soma_ dos_ quadrados` $= (\mathbf{S}\, j : 0 \leq j < \mathtt{i} : j^2) \wedge 0 \leq \mathtt{i} < \mathtt{n}$
acção
`++i;`
// $CI \equiv$ `soma_ dos_ quadrados` $= (\mathbf{S}\, j : 0 \leq j < \mathtt{i} : j^2) \wedge 0 \leq \mathtt{i} \leq \mathtt{n}$.

onde se assume a condição invariante como verdadeira antes da acção, se sabe que a guarda é verdadeira nesse mesmo lugar (de outra forma o ciclo teria terminado) e se pretende escolher uma acção de modo a que a condição invariante seja verdadeira também *depois do passo*, ou melhor, *apesar do progresso*. Usando a semântica da operação de atribuição, pode-se deduzir a condição mais fraca antes do progresso de modo a que condição invariante seja verdadeira no final do passo:

// $CI \wedge G \equiv$ `soma_ dos_ quadrados` $= (\mathbf{S}\, j : 0 \leq j < \mathtt{i} : j^2) \wedge 0 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i} \neq$ `n`, ou seja,
// `soma_ dos_ quadrados` $= (\mathbf{S}\, j : 0 \leq j < \mathtt{i} : j^2) \wedge 0 \leq \mathtt{i} < \mathtt{n}$
acção
// `soma_ dos_ quadrados` $= (\mathbf{S}\, j : 0 \leq j < \mathtt{i}+1 : j^2) \wedge 0 \leq \mathtt{i}+1 \leq \mathtt{n}$, ou seja,
// `soma_ dos_ quadrados` $= (\mathbf{S}\, j : 0 \leq j < \mathtt{i}+1 : j^2) \wedge -1 \leq \mathtt{i} < \mathtt{n}$.
`++i;` // o mesmo que `i = i + 1;`
// $CI \equiv$ `soma_ dos_ quadrados` $= (\mathbf{S}\, j : 0 \leq j < \mathtt{i} : j^2) \wedge 0 \leq \mathtt{i} \leq \mathtt{n}$.

Falta pois determinar uma acção tal que

// `soma_ dos_ quadrados` $= (\mathbf{S}\, j : 0 \leq j < \mathtt{i} : j^2) \wedge 0 \leq \mathtt{i} < \mathtt{n}$
acção
// `soma_ dos_ quadrados` $= (\mathbf{S}\, j : 0 \leq j < \mathtt{i}+1 : j^2) \wedge -1 \leq \mathtt{i} < \mathtt{n}$.

Para simplificar a determinação da acção, pode-se fortalecer um pouco a sua condição objectivo forçando `i` a ser maior do que zero, o que permite extrair o último termo do somatório

// `soma_ dos_ quadrados` $= (\mathbf{S}\, j : 0 \leq j < \mathtt{i} : j^2) \wedge 0 \leq \mathtt{i} < \mathtt{n}$
acção
// `soma_ dos_ quadrados` $= (\mathbf{S}\, j : 0 \leq j < \mathtt{i} : j^2) + \mathtt{i}^2 \wedge 0 \leq \mathtt{i} < \mathtt{n}$
// `soma_ dos_ quadrados` $= (\mathbf{S}\, j : 0 \leq j < \mathtt{i}+1 : j^2) \wedge -1 \leq \mathtt{i} < \mathtt{n}$

A acção mais simples limita-se a acrescentar $\mathtt{i}^2$ ao valor da variável `soma_dos_quadrados`:

```
soma_dos_quadrados += i * i;
```

pelo que o o ciclo e a função completos são

```
/** Devolve a soma dos quadrados dos primeiros n inteiros não-negativos.
    @pre $PC \equiv 0 \leq$ n.
    @post $CO \equiv$ somaDosQuadrados $= (\mathbf{S}\, j : 0 \leq j <$ n $: j^2)$. */
int somaDosQuadrados(int const n)
{
    assert(0 <= n);

    int soma_dos_quadrados = 0;
    int i = 0;
    // $CI \equiv$ soma_ dos_ quadrados $= (\mathbf{S}\, j : 0 \leq j <$ i $: j^2) \wedge 0 \leq$ i $\leq$ n.
    while(i != n) {
        soma_dos_quadrados += i * i;
        ++i;
    }
    return soma_dos_quadrados;
}
```

ou, substituindo pelo ciclo com a instrução for equivalente,

```
/** Devolve a soma dos quadrados dos primeiros n inteiros não-negativos.
    @pre $PC \equiv 0 \leq$ n.
    @post $CO \equiv$ somaDosQuadrados $= (\mathbf{S}\, j : 0 \leq j <$ n $: j^2)$. */
int somaDosQuadrados(int const n)
{
    assert(0 <= n);

    int soma_dos_quadrados = 0;
    // $CI \equiv$ soma_ dos_ quadrados $= (\mathbf{S}\, j : 0 \leq j <$ i $: j^2) \wedge 0 \leq$ i $\leq$ n.
    for(int i = 0; i != n; ++i)
        soma_dos_quadrados += i * i;
    return soma_dos_quadrados;
}
```

Nas versões completas da função incluiu-se um comentário com a condição invariante CI. Este é um comentário normal, e não de documentação. Os comentários de documentação, após `///` ou entre `/** */`, servem para documentar a interface da função ou procedimento (ou outras entidades), i.e., para dizer claramente o que essas entidades fazem, para que servem ou como se comportam, sempre de um ponto de vista externo. Os comentários normais, pelo contrário, servem para que o leitor do saiba como o código, neste caso a função, funciona. Sendo a condição invariante a mais importante das asserções associadas a um ciclo, é naturalmente candidata a figurar num comentário na versão final das funções ou procedimentos.

Função `raizInteira()` A guarda já determinada é

$$G \equiv (\mathtt{r} + 1)^2 \leq \mathtt{x}$$

pelo que o progresso mais simples é a simples incrementação de `r`. Este progresso garante que o ciclo termina (i.e., que a guarda se torna falsa) ao fim de um número finito de iterações (quantas?), uma vez que `r` foi inicializada com 0. Ou seja, o passo do ciclo é

```
// CI ∧ G ≡ 0 ≤ r ∧ r² ≤ x ∧ (r + 1)² ≤ x, ou seja,
// 0 ≤ r ∧ (r + 1)² ≤ x, porque r² ≤ (r + 1)² sempre que 0 ≤ r.
acção
++r;
// CI ≡ 0 ≤ r ∧ r² ≤ x.
```

onde se assume a condição invariante como verdadeira antes da acção, se sabe que a guarda é verdadeira nesse mesmo lugar (de outra forma o ciclo teria terminado) e se pretende escolher uma acção de modo a que a condição invariante seja verdadeira também *depois do passo*, ou melhor, *apesar do progresso*. Usando a semântica da operação de atribuição, pode-se deduzir a condição mais fraca antes do progresso de modo a que condição invariante seja verdadeira no final do passo:

```
// 0 ≤ r ∧ (r + 1)² ≤ x.
acção
// 0 ≤ r + 1 ∧ (r + 1)² ≤ x, ou seja, −1 ≤ r ∧ (r + 1)² ≤ x.
++r; // o mesmo que r = r + 1;
// CI ≡ 0 ≤ r ∧ r² ≤ x.
```

Como $0 \leq \mathtt{r} \Rightarrow -1 \leq \mathtt{r}$, então é evidente que neste caso não é necessária qualquer acção, pelo que o ciclo e a função completos são

```
/** Devolve a melhor aproximação inteira por defeito da raiz quadrada de x.
    @pre   PC ≡ 0eqx.
    @post CO ≡ 0 ≤ raizInteira ≤ √x < raizInteira + 1, ou seja,
          0 ≤ raizInteira ∧ 0 ≤ raizInteira² ≤ x < (raizInteira + 1)². */
int raizInteira(int const x)
{
    assert(0 <= x);

    int r = 0;
    // CI ≡ 0 ≤ r ∧ r² ≤ x.
    while((r + 1) * (r + 1) <= x)
        ++r;

    assert(0 <= r and r * r <= x and x < (r + 1) * (r + 1));

    return r;
}
```

A guarda do ciclo pode ser simplificada se se adiantar a variável `r` de uma unidade

```
/** Devolve a melhor aproximação inteira por defeito da raiz quadrada de x.
    @pre  PC ≡ 0eqx.
    @post CO ≡ 0 ≤ raizInteira ≤ √x < raizInteira + 1, ou seja,
              0 ≤ raizInteira ∧ 0 ≤ raizInteira² ≤ x <
(raizInteira + 1)². */
int raizInteira(int const x)
{
    assert(0 <= x);

    int r = 1;
    // CI ≡ 1 ≤ r ∧ (r − 1)² ≤ x.
    while(r * r <= x)
        ++r;

    assert(1 <= r and (r - 1) * (r - 1) <= x and x < r * r);

    return r - 1;
}
```

Determinando se um ciclo é necessário

Deixou-se para o fim a discussão deste passo, que em rigor deveria ter tido lugar logo após a especificação do problema. É que é essa a tendência natural do programador principiante... Considere-se de novo a função `int somaDosQuadrados(int const n)`. Será mesmo necessário um ciclo? Acontece que a soma dos quadrados dos primeiros n inteiros não-negativos, pode ser expressa simplesmente por[20]

$$\frac{n(n-1)(2n-1)}{6}.$$

Não é necessário qualquer ciclo na função, que pode ser implementada simplesmente como[21]:

```
/** Devolve a soma dos quadrados dos primeiros n inteiros não-negativos.
    @pre  PC ≡ 0 ≤ n.
    @post CO ≡ somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(0 <= n);
```

[20]A demonstração faz-se utilizando a propriedade telescópica dos somatórios

$$\sum_{j=0}^{n-1}\left(f(j)-f(j-1)\right)=f(n-1)-f(-1)$$

com $f(j)=j^3$.

[21]Porquê devolver `n * (n - 1) / 2 * (2 * n - 1) / 3` e não `n * (n - 1) * (2 * n - 1) / 6`? Lembre-se das limitações dos inteiros em C++.

```
    return n * (n - 1) / 2 * (2 * n - 1) / 3;
}
```

4.7.5 Um exemplo

Pretende-se desenvolver uma função que devolva verdadeiro se o valor do seu argumento inteiro não-negativo n for primo e falso no caso contrário. Um número inteiro não-negativo é primo se for apenas é divisível por 1 e por si mesmo. O inteiro 1 é classificado como não-primo, o mesmo acontecendo com o inteiro 0.

O primeiro passo da resolução do problema é a sua especificação, ou seja, a escrita da estrutura da função, incluindo a pré-condição e a condição objectivo:

```
/** Devolve 𝒱 se n for um número primo e ℱ no caso contrário.
    @pre  PC ≡ 0 ≤ n.
    @post CO ≡ éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
    assert(0 <= n);

    ...
}
```

Como abordar este problema? Em primeiro lugar, é conveniente verificar que os valores 0 e 1 são casos particulares. O primeiro porque é divisível por todos os inteiros positivos e portanto não pode ser primo. O segundo porque, apesar de só ser divisível por 1 e por si próprio, não é considerado, por convenção, um número primo. Estes casos particulares podem ser tratados com recurso a uma simples instrução condicional, pelo que se pode reescrever a função como

```
/** Devolve 𝒱 se n for um número primo e ℱ no caso contrário.
    @pre  0 ≤ n.
    @post éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
    assert(0 <= n);

    if(n == 0 || n == 1)
        return false;

    // PC ≡ 2 ≤ n.
    bool é_primo = ...;
    ...
    // CO ≡ é_ primo = (Q j : 2 ≤ j < n : n ÷ j ≠ 0).
    return é_primo;
}
```

uma vez que, se a guarda da instrução condicional for falsa, e admitindo que a pré-condição da função $0 \le$ n é verdadeira, então forçosamente $2 \le$ n depois da instrução condicional. Por outro lado, a condição objectivo antes da instrução de retorno no final da função pode ser simplificada dada a nova pré-condição, mais forte que a da função. Aproveitou-se para introduzir uma variável booleana que, no final da função, deverá conter o valor lógico apropriado a devolver pela função.

Assim, o problema resume-se a escrever o código que garante que CO se verifica sempre que PC se verificar. Um ciclo parece ser apropriado para resolver este problema, pois para verificar se um número n é primo pode-se ir verificando se é divisível por algum inteiro entre 2 e $n-1$.

A condição invariante do ciclo pode ser obtida substituindo o limite superior da conjunção (a constante n) por uma variável i criada para o efeito, e com limites apropriados. Obtém-se a seguinte estrutura para o ciclo

```
// PC ≡ 2 ≤ n.
bool é_primo = ...;
int i = ...;
// CI ≡ é_primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(G) {
    passo
}
// CO ≡ é_primo = (Q j : 2 ≤ j < n : n ÷ j ≠ 0).
```

O que significa a condição invariante? Simplesmente que a variável é_primo tem valor lógico verdadeiro se e só se não existirem divisores de n superiores a 1 e inferiores a i, variando i entre 2 e n. Ou melhor, significa que, num dado passo do ciclo, já se testaram todos os potenciais divisores de n entre 2 e i *exclusive*.

A escolha da guarda é muito simples: quando o ciclo terminar $CI \wedge \neg G$ deve implicar CO. Isso consegue-se simplesmente fazendo

$$\neg G \equiv \texttt{i} = \texttt{n},$$

ou seja,

$$G \equiv \texttt{i} \neq \texttt{n}.$$

Quanto à inicialização, também é simples, pois basta atribuir 2 a i para que a conjunção não tenha qualquer termo e portanto tenha valor lógico verdadeiro. Assim, a inicialização é:

```
bool é_primo = true;
int i = 2;
```

pelo que o ciclo fica

```
// PC ≡ 2 ≤ n.
bool é_primo = true;
int i = 2;
```

```
// CI ≡ é_primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(i != n) {
    passo
}
// CO ≡ é_primo = (Q j : 2 ≤ j < n : n ÷ j ≠ 0).
```

Que progresso utilizar? A forma mais simples de garantir a terminação do ciclo é simplesmente incrementar `i`. Dessa forma, como `i` começa com valor 2 e $2 \leq \mathtt{n}$pela pré-condição, a guarda torna-se falsa, e o ciclo termina, ao fim de exactamente $\mathtt{n} - 2$ iterações do ciclo. Assim, o ciclo fica

```
// PC ≡ 2 ≤ n.
bool é_primo = true;
int i = 2;
// CI ≡ é_primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(i != n) {
    acção
    ++i;
}
// CO ≡ é_primo = (Q j : 2 ≤ j < n : n ÷ j ≠ 0).
```

Finalmente, falta determinar a acção a executar para garantir a veracidade da condição invariante apesar do progresso realizado. No início do passo admite-se que a condição é verdadeira e sabe-se que a guarda o é também:

$$CI \wedge G \equiv \texttt{é\_ primo} = (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i} \neq \mathtt{n},$$

ou seja,

$$CI \wedge G \equiv \texttt{é\_ primo} = (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n}.$$

Por outro lado, no final do passo deseja-se que a condição invariante seja verdadeira. Logo, o passo com as respectivas asserções é:

```
// Aqui admite-se que CI∧G ≡ é_primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0)∧2 ≤ i < n.
acção
++i;
// Aqui pretende-se que CI ≡ é_primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

Antes de determinar a acção, é conveniente verificar qual a pré-condição mais fraca do progresso que é necessário impor para que, depois dele, se verifique a condição invariante do ciclo. Usando a transformação de variáveis correspondente à atribuição `i = i + 1` (equivalente a `++i`), chega-se a:

```
// CI ∧ G ≡ é_primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i < n.
acção
// é_primo = (Q j : 2 ≤ j < i + 1 : n ÷ j ≠ 0) ∧ 2 ≤ i + 1 ≤ n, ou seja,
// é_primo = (Q j : 2 ≤ j < i + 1 : n ÷ j ≠ 0) ∧ 1 ≤ i < n.
++i;
// CI ≡ é_primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

Por outro lado, se $2 \leq$ i, pode-se extrair o último termo da conjunção. Assim, reforçando a pré-condição do progresso,

```
// CI ∧ G ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i < n.
acção
// é_ primo = ((Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ n ÷ i ≠ 0) ∧ 2 ≤ i < n.
// é_ primo = (Q j : 2 ≤ j < i + 1 : n ÷ j ≠ 0) ∧ 1 ≤ i < n.
++i;
// CI ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

Assim sendo, a acção deve ser tal que

```
// é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i < n.
acção
// é_ primo = ((Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ n ÷ i ≠ 0) ∧ 2 ≤ i < n.
```

É evidente, então, que a acção pode ser simplesmente:

```
é_primo = (é_primo and n % i != 0);
```

onde os parênteses são dispensáveis dadas as regras de precedência dos operadores em C++ (ver Secção 2.7.7).

A correcção da acção determinada pode ser verificada facilmente calculando a respectiva pré-condição mais fraca

```
// (é_ primo ∧ n ÷ i ≠ 0) = ((Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ n ÷ i ≠ 0) ∧ 2 ≤ i < n.
é_primo = (é_primo and n % i != 0);
// é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ n ÷ i ≠ 0 ∧ 2 ≤ i < n.
```

e observando que $CI \wedge G$ leva forçosamente à sua verificação:

$$
\begin{array}{ll}
& \text{é\_ primo} = (\mathbf{Q}\, j : 2 \leq j < \mathtt{i} : \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n} \\
\Rightarrow & (\text{é\_ primo} \wedge \mathtt{n} \div \mathtt{i} \neq 0) = ((\mathbf{Q}\, j : 2 \leq j < \mathtt{i} : \mathtt{n} \div j \neq 0) \wedge \mathtt{n} \div \mathtt{i} \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n}.
\end{array}
$$

Escrevendo agora o ciclo completo tem-se:

```
// PC ≡ 2 ≤ n.
bool é_primo = true;
int i = 2;
// CI ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(i != n) {
    é_primo = é_primo and n % i != 0;
    ++i;
}
// CO ≡ é_ primo = (Q j : 2 ≤ j < n : n ÷ j ≠ 0).
```

Reforço da guarda

A observação atenta deste ciclo revela que a variável `é_primo`, se alguma vez se tornar falsa, nunca mais deixará de o ser. Tal deve-se a que $\mathcal{F}$ é o elemento neutro da conjunção. Assim, é evidente que o ciclo poderia terminar antecipadamente, logo que essa variável tomasse o valor $\mathcal{F}$: o ciclo só deveria continuar enquanto $G \equiv \texttt{é\_ primo} \wedge \texttt{i} \neq \texttt{n}$. Será que esta nova guarda, reforçada com uma conjunção, é mesmo apropriada?

Em primeiro lugar é necessário demonstrar que, quando o ciclo termina, se atinge a condição objectivo. Ou seja, será que $CI \wedge \neg G \Rightarrow CO$? Neste caso tem-se

$$CI \wedge \neg G \equiv \texttt{é\_ primo} = (\mathbf{Q}\, j\,:\, 2 \leq j < \texttt{i}\,:\,\texttt{n} \div j \neq 0) \wedge 2 \leq \texttt{i} \leq \texttt{n} \wedge (\neg \texttt{é\_ primo} \vee \texttt{i} = \texttt{n}).$$

Considere-se o último termo da conjunção, ou seja, a disjunção $\neg\texttt{é\_ primo} \vee \texttt{i} = \texttt{n}$. Quando o ciclo termina pelo menos um dos termos da disjunção é verdadeiro.

Suponha-se que $\neg\texttt{é\_ primo} = \mathcal{V}$. Então, como $\texttt{é\_ primo} = (\mathbf{Q}\, j\,:\, 2 \leq j < \texttt{i}\,:\,\texttt{n} \div j \neq 0)$ também é verdadeira no final do ciclo, tem-se que $(\mathbf{Q}\, j\,:\, 2 \leq j < \texttt{i}\,:\,\texttt{n} \div j \neq 0)$ é falsa. Mas isso implica que $(\mathbf{Q}\, j\,:\, 2 \leq j < \texttt{n}\,:\,\texttt{n} \div j \neq 0)$, pois se não é verdade que não há divisores de `n` entre 2 e `i` *exclusive*, então também não é verdade que não os haja entre 2 e `n` *exclusive*, visto que $\texttt{i} \leq \texttt{n}$. Ou seja, a condição objectivo

$$\begin{aligned} CO &\equiv \texttt{é\_ primo} = (\mathbf{Q}\, j\,:\, 2 \leq j < \texttt{n}\,:\,\texttt{n} \div j \neq 0) \\ CO &\equiv \mathcal{F} = \mathcal{F} \\ CO &\equiv \mathcal{V} \end{aligned}$$

é verdadeira!

O outro caso, se $\texttt{i} = \texttt{n}$, é idêntico ao caso sem reforço da guarda. Logo, a condição objectivo é de facto atingida em qualquer dos casos[22].

Quanto ao passo (acção e progresso), é evidente que o reforço da guarda não altera a sua validade. No entanto, é instrutivo determinar de novo a acção do passo. A acção deve garantir a veracidade da condição invariante apesar do progresso. No início do passo admite-se que a condição é verdadeira e sabe-se que a guarda o é também:

$$\begin{aligned} CI \wedge G &\equiv \texttt{é\_ primo} = (\mathbf{Q}\, j\,:\, 2 \leq j < \texttt{i}\,:\,\texttt{n} \div j \neq 0) \wedge 2 \leq \texttt{i} \leq \texttt{n} \wedge \texttt{é\_ primo} \wedge \texttt{i} \neq \texttt{n} \\ &\equiv (\mathbf{Q}\, j\,:\, 2 \leq j < \texttt{i}\,:\,\texttt{n} \div j \neq 0) \wedge 2 \leq \texttt{i} < \texttt{n} \wedge \texttt{é\_ primo} \end{aligned}$$

Por outro lado, no final do passo deseja-se que a condição invariante seja verdadeira. Logo, o passo com as respectivas asserções é:

```
// Aqui admite-se que CI∧G ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0)∧2 ≤ i < n∧é_ primo.
acção
++i;
// Aqui pretende-se que CI ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

[22]Se não lhe pareceu claro lembre-se que $(A \vee B) \Rightarrow C$ é o mesmo que $A \Rightarrow C \wedge B \Rightarrow C$ e que $a \wedge (B \vee C)$ é o mesmo que $(A \wedge B) \vee (A \wedge C)$.

Antes de determinar a acção, é conveniente verificar qual a pré-condição mais fraca do progresso que é necessário impor para que, depois dele, se verifique a condição invariante do ciclo. Usando a transformação de variáveis correspondente à atribuição `i = i + 1` (equivalente a `++i`), chega-se a:

```
// CI ∧ G ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i < n ∧ é_ primo.
acção
// é_ primo = (Q j : 2 ≤ j < i + 1 : n ÷ j ≠ 0) ∧ 1 ≤ i < n.
++i;
// CI ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

Por outro lado, se $2 \leq$ `i`, pode-se extrair o último termo da conjunção. Assim, reforçando a pré-condição do progresso,

```
// CI ∧ G ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i < n ∧ é_ primo.
acção
// é_ primo = ((Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ n ÷ i ≠ 0) ∧ 2 ≤ i < n.
// é_ primo = (Q j : 2 ≤ j < i + 1 : n ÷ j ≠ 0) ∧ 1 ≤ i < n.
++i;
// CI ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

Assim sendo, a acção deve ser tal que

```
// (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i < n ∧ é_ primo.
acção
// é_ primo = ((Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ n ÷ i ≠ 0) ∧ 2 ≤ i < n.
```

É evidente, então, que a acção pode ser simplesmente:

```
é_primo = n % i != 0;
```

A acção simplificou-se relativamente à acção no ciclo com a guarda original. A alteração da acção pode ser percebida observando que, sendo a guarda sempre verdadeira no início do passo do ciclo, a variável `é_primo` tem aí sempre o valor verdadeiro, pelo que a acção antiga tinha uma conjunção com a veracidade. Como $\mathcal{V} \wedge P$ é o mesmo que P, pois $\mathcal{V}$ é o elemento neutro da conjunção, a conjunção pode ser eliminada.

A correcção da acção determinada pode ser verificada facilmente calculando a respectiva pré-condição mais fraca:

```
// (n ÷ i ≠ 0) = ((Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ n ÷ i ≠ 0) ∧ 2 ≤ i < n.
é_primo = n % i != 0;
// é_ primo = ((Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ n ÷ i ≠ 0) ∧ 2 ≤ i < n.
```

e observando que $CI \wedge G$ leva forçosamente à sua verificação:

$$\begin{aligned} & (\mathbf{Q}\, j : 2 \leq j < \mathtt{i} : \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n} \wedge \mathtt{é\_ primo} \\ \Rightarrow \quad & (\mathtt{n} \div \mathtt{i} \neq 0) = ((\mathbf{Q}\, j : 2 \leq j < \mathtt{i} : \mathtt{n} \div j \neq 0) \wedge \mathtt{n} \div \mathtt{i} \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n}. \end{aligned}$$

Escrevendo agora o ciclo completo tem-se:

```
// PC ≡ 2 ≤ n.
bool é_primo = true;
int i = 2;
// CI ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(é_primo and i != n) {
    é_primo = n % i != 0;
    ++i;
}
// CO ≡ é_ primo = (Q j : 2 ≤ j < n : n ÷ j ≠ 0).
```

Versão final

Convertendo para a instrução `for` e inserindo o ciclo na função tem-se:

```
/** Devolve V se n for um número primo e F no caso contrário.
    @pre   PC ≡ 0 ≤ n.
    @post  CO ≡ éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
    assert(0 <= n);

    if(n == 0 || n == 1)
        return false;

    bool é_primo = true;
    // CI ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
    for(int i = 2; é_primo and i != n; ++i)
        é_primo = n % i != 0;

    return é_primo;
}
```

A função inclui desde o início uma instrução de asserção que verifica a veracidade da pré-condição. E quanto à condição objectivo? A condição objectivo envolve um quantificador, pelo que não é possível exprimi-la na forma de uma expressão booleana em C++, excepto recorrendo a funções auxiliares. Como resolver o problema? Uma hipótese passa por reflectir na condição objectivo apenas os termos da condição objectivo que não envolvem quantificadores:

```
assert(not é_primo or 2 <= n);
```

Recorde-se que $A \Rightarrow B$ é o mesmo que $\neg A \vee B$. A expressão da asserção verifica portanto se é_ primo $\Rightarrow 2 \leq$ n.

Esta asserção é pouco útil, pois deixa passar como primos ou não primos todos os números não primos superiores a 2... Pode-se fazer melhor. Todos os inteiros positivos podem ser escritos na forma $6k - l$, com $l = 0, \ldots, 5$ e k inteiro positivo. É imediato que os inteiros das formas $6k$, $6k - 2$, $6k - 3$ e $6k - 4$ não podem ser primos (com excepção de 2 e 3). Ou seja, os números primos ou são o 2 ou o 3, ou são sempre de uma das formas $6k - 1$ ou $6k - 5$. Assim, pode-se reforçar um pouco a asserção final para:

```
assert(((n != 2 and n != 3) or é_primo) and
       (not é_primo or n == 2 or n == 3 or
        (2 <= n and ((n + 1) % 6 == 0 or (n + 5) % 6 == 0))));
```

que, expressa em notação matemática fica

$$\begin{gathered}((\mathrm{n} = 2 \vee \mathrm{n} = 3) \Rightarrow \text{é\_ primo}) \wedge \\ (\text{é\_ primo} \Rightarrow (\mathrm{n} = 2\mathrm{n} = 3 \vee (2 \leq \mathrm{n} \wedge ((\mathrm{n}+1) \div 6 = 0 \vee (\mathrm{n}+5) \div 6 = 0))))\end{gathered}$$

A função fica então:

```
/** Devolve 𝒱 se n for um número primo e ℱ no caso contrário.
    @pre   PC ≡ 0 ≤ n.
    @post  CO ≡ éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
    assert(0 <= n);

    if(n == 0 || n == 1)
        return false;

    bool é_primo = true;
    // CI ≡ é_ primo = (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
    for(int i = 2; é_primo and i != n; ++i)
        é_primo = n % i != 0;

    assert(n == 2 or n == 3 or
           (5 <= n and ((n - 1) % 6 == 0 or (n - 5) % 6 == 0)));

    assert(((n != 2 and n != 3) or é_primo) and
           (not é_primo or n == 2 or n == 3 or
            (2 <= n and ((n + 1) % 6 == 0 or (n + 5) % 6 == 0))));

    return é_primo;
}
```

Desta forma continuam a não se detectar muitos possíveis erros, mas passaram a detectar-se bastante mais do que inicialmente.

Finalmente, uma observação atenta da função revela que ainda pode ser simplificada (do ponto de vista da sua escrita) para

```
/** Devolve 𝒱 se n for um número primo e ℱ no caso contrário.
    @pre  PC ≡ 0 ≤ n.
    @post CO ≡ éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
    assert(0 <= n);

    if(n == 0 || n == 1)
        return false;

    // CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
    for(int i = 2; i != n; ++i)
        if(n % i != 0)
            return false;

    return true;
}
```

Este último formato é muito comum em programas escritos em C ou C++. A demonstração da correcção de ciclos incluindo saídas antecipadas no seu passo é um pouco mais complicada e será abordada mais tarde.

Outra abordagem

Como se viu, o valor $2 \leq$ n é primo se nenhum inteiro entre 2 e n *exclusive* o dividir. Mas pode-se pensar neste problema de outra forma mais interessante. Considere-se o conjunto $\mathbf{D}$ de todos os possíveis divisores de n. Claramente o próprio n é sempre membro deste conjunto, n $\in \mathbf{D}$. Se n for primo, o conjunto tem apenas como elemento o próprio n, i.e., $\mathbf{C} = \{\mathrm{n}\}$. Se n não for primo existirão outros divisores no conjunto, forçosamente inferiores ao próprio n. Considere-se o mínimo desse conjunto. Se n for primo, o mínimo é o próprio n. Se n não for primo, o mínimo é forçosamente diferente de n. Ou seja, a afirmação "n é primo" tem o mesmo valor lógico que "o mais pequeno dos divisores não-unitários de n é o n".

Regresse-se à estrutura da função:

```
/** Devolve 𝒱 se n for um número primo e ℱ no caso contrário.
    @pre  0 ≤ n.
    @post éPrimo = (min {2 ≤ j ≤ n : n ÷ j = 0} = n ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
```

```
    assert(0 <= n);

    if(n == 0 || n == 1)
        return false;

    // PC ≡ 2 ≤ n.
    ...
    return ...;
    // CO ≡ éPrimo = min {2 ≤ j ≤ n : n ÷ j = 0} = n.
}
```

Introduza-se uma variável `i` que se assume conter o mais pequeno dos divisores não-unitários de `n` no final da função. Nesse caso a função deverá devolver o valor lógico de `i` $=$ `n`. Então pode-se reescrever a função:

```
/** Devolve V se n for um número primo e F no caso contrário.
    @pre 0 ≤ n.
    @post éPrimo = (min {2 ≤ j ≤ n : n ÷ j = 0} = n ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
    assert(0 <= n);


    if(n == 0 || n == 1)
        return false;

    // PC ≡ 2 ≤ n.
    int i = ...;
    ...
    // CO ≡ i = min {2 ≤ j ≤ n : n ÷ j = 0}.
    return i == n;
    // éPrimo = (min {2 ≤ j ≤ n : n ÷ j = 0} = n).
}
```

O problema reduz-se pois a escrever um ciclo que, dada a pré-condição PC, garanta que no seu final se verifique a nova CO. A condição objectivo pode ser reescrita numa forma mais simpática. Se `i` for o menor dos divisores de `n` entre 2 e `n` *inclusive*, então

1. `i` está entre 2 e `n` *inclusive*.
2. `i` tem de ser divisor de `n`, i.e., $\mathtt{n} \div \mathtt{i} = 0$, e
3. nenhum outro inteiro inferior a `i` e superior ou igual a 2 é divisor de `n`, i.e., $(\mathbf{Q}\, j : 2 \leq j < \mathtt{i} : \mathtt{n} \div j \neq 0)$.

Traduzindo para notação matemática:

$$CO \equiv \mathtt{n} \div \mathtt{i} = 0 \wedge (\mathbf{Q}\, j : 2 \leq j < \mathtt{i} : \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} \leq \mathtt{n}.$$

Como a nova condição objectivo é uma conjunção, pode-se tentar obter a condição invariante e a negação da guarda do ciclo por factorização. A escolha mais evidente faz do primeiro termo a guarda e do segundo a condição invariante

$$CO \equiv \overbrace{\mathtt{n} \div \mathtt{i} = 0}^{\neg G} \wedge \overbrace{(\mathbf{Q}\, j \,:\, 2 \leq j < \mathtt{i} \,:\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} \leq \mathtt{n}}^{CI},$$

conduzindo ao seguinte ciclo

```
// PC ≡ 2 ≤ n.
int i = ...;
// CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(n % i != 0) {
    passo
}
// CO ≡ n ÷ i = 0 ∧ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

A escolha da inicialização é simples. Como a conjunção de zero termos é verdadeira, basta fazer

```
int i = 2;
```

pelo que o ciclo fica

```
// PC ≡ 2 ≤ n.
int i = 2;
// CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(n % i != 0) {
    passo
}
// CO ≡ n ÷ i = 0 ∧ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

Mais uma vez o progresso mais simples é a incrementação de `i`

```
// PC ≡ 2 ≤ n.
int i = 2;
// CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(n % i != 0) {
    acção
    ++i;
}
// CO ≡ n ÷ i = 0 ∧ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

Com este progresso o ciclo termina na pior das hipóteses com $\mathtt{i} = \mathtt{n}$. Que acção usar? Antes da acção sabe-se que a guarda é verdadeira e admite-se que a condição invariante o é também. Depois do progresso pretende-se que a condição invariante seja verdadeira:

```
// CI ∧ G ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n ∧ n ÷ i ≠ 0.
acção
++i;
// CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

Como habitualmente, começa por se determinar a pré-condição mais fraca do progresso que garante a verificação da condição invariante no final do passo:

```
// CI ∧ G ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n ∧ n ÷ i ≠ 0,
// que, como n ÷ i ≠ 0 implica que i ≠ n, é o mesmo que
// (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ n ÷ i ≠ 0 ∧ 2 ≤ i < n.
acção
// (Q j : 2 ≤ j < i + 1 : n ÷ j ≠ 0) ∧ 2 ≤ i + 1 ≤ n, ou seja,
// (Q j : 2 ≤ j < i + 1 : n ÷ j ≠ 0) ∧ 1 ≤ i < n.
++i;
// CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

Fortalecendo a pré-condição do progresso de modo garantir que $2 \leq$ `i`, pode-se extrair o último termo da conjunção:

```
// (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ n ÷ i ≠ 0 ∧ 2 ≤ i < n.
acção
// (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ n ÷ i ≠ 0 ∧ 2 ≤ i < n, ou seja,
// (Q j : 2 ≤ j < i + 1 : n ÷ j ≠ 0) ∧ 1 ≤ i < n.
++i;
// CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

pelo que a acção pode ser a instrução nula! O ciclo fica pois

```
// PC ≡ 2 ≤ n.
int i = 2;
// CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
while(n % i != 0)
    ++i;
// CO ≡ n ÷ i = 0 ∧ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
```

e a função completa é

```
/** Devolve V se n for um número primo e F no caso contrário.
    @pre 0 ≤ n.
    @post éPrimo = (min {2 ≤ j ≤ n : n ÷ j = 0} = n ∧ 2 ≤ n). */
bool éPrimo(int const n)
{
```

```
    assert(0 <= n);

    if(n == 0 || n == 1)
        return false;

    int i = 2;
    // CI ≡ (Q j : 2 ≤ j < i : n ÷ j ≠ 0) ∧ 2 ≤ i ≤ n.
    while(n % i != 0)
        ++i;

    assert(((n != 2 and n != 3) or (i == n)) and
           (i != n or n == 2 or n == 3 or
            (2 <= n and ((n + 1) % 6 == 0 or (n + 5) % 6 == 0))));

    return i == n;
}
```

A instrução de asserção para verificação da condição objectivo foi obtida por simples adaptação da obtida na secção anterior.

Discussão

Há inúmeras soluções para cada problema. Neste caso começou-se por uma solução simples mas ineficiente, aumentou-se a eficiência recorrendo ao reforço da guarda e finalmente, usando uma forma diferente de expressar a condição objectivo, obteve-se um ciclo mais eficiente que os iniciais, visto que durante o ciclo é necessário fazer apenas uma comparação e uma incrementação.

Mas há muitas outras soluções para este mesmo problema, e mais eficientes. Recomenda-se que o leitor tente resolver este problema depois de aprender sobre matrizes, no Capítulo 5. Experimente procurar informação sobre um velho algoritmo chamado a "peneira de Eratóstenes".

4.7.6 Outro exemplo

Suponha-se que se pretende desenvolver um procedimento que, dados dois inteiros como argumento, um o dividendo e outro o divisor, sendo o dividendo não-negativo e o divisor positivo, calcule o quociente e o resto da divisão inteira do dividendo pelo divisor e os guarde em variáveis externas ao procedimento (usando passagem de argumentos por referência). Para que o problema tenha algum interesse, não se pode recorrer aos operadores `*`, `/` e `%` do C++, nem tão pouco aos operadores de deslocamento *bit*-a-*bit*. A estrutura do procedimento é (ver Secção 4.6.6)

```
/** Coloca em q e r respectivamente o quociente e o resto da divisão
    inteira de dividendo por divisor.
```

```
    @pre    PC ≡ 0 ≤ dividendo ∧ 0 < divisor.
    @post CO ≡ 0 ≤ r < divisor ∧ dividendo = q × divisor + r.
void divide(int const dividendo, int const divisor,
            int& q, int& r)
{
    assert(0 <= dividendo and 0 < divisor);

    ...

    assert(0 <= r and r < divisor and
           dividendo = q * divisor + r);
}
```

A condição objectivo pode ser vista como uma definição da divisão inteira. Não só o quociente `q` multiplicado pelo divisor e somado do resto `r` tem de resultar no dividendo, como o resto tem de ser não-negativo e menor que o divisor (senão não estaria completa a divisão!), existindo apenas uma solução nestas circunstâncias.

Como é evidente a divisão tem de ser conseguida através de um ciclo. Qual será a sua condição invariante? Neste caso, como a condição objectivo é a conjunção de três termos

$$CO \equiv 0 \leq \texttt{r} \wedge \texttt{r} < \texttt{divisor} \wedge \texttt{dividendo} = \texttt{q} \times \texttt{divisor} + \texttt{r},$$

a solução passa por obter a condição invariante e a negação da guarda por factorização da condição objectivo.

Mas quais das proposições usar para $\neg G$ e quais usar para CI? Um pouco de experimentação e alguns falhanços levam a que se perceba que a negação da guarda deve corresponder ao segundo termo da conjunção, ou seja, reordenando os termos da conjunção,

$$CO \equiv \overbrace{\texttt{r} < \texttt{divisor}}^{\neg G} \wedge \overbrace{\texttt{dividendo} = \texttt{q} \times \texttt{divisor} + \texttt{r} \wedge 0 \leq \texttt{r}}^{CI},$$

Dada esta escolha, a forma mais simples de inicializar o ciclo é fazendo

```
r = dividendo;
q = 0;
```

pois substituindo os valores na condição invariante obtém-se

$$CI \equiv \texttt{dividendo} = 0 \times \texttt{divisor} + \texttt{dividendo} \wedge 0 \leq \texttt{dividendo},$$

que é verdadeira dado que a pré-condição garante que dividendo é não-negativo.

O ciclo terá portanto a seguinte forma:

```
r = dividendo;
q = 0;
// CI ≡ dividendo = q × divisor + r ∧ 0 ≤ r.
while(divisor <= r) {
    passo
}
```

O progresso deve ser tal que a guarda se torne falsa ao fim de um número finito de iterações do ciclo. Neste caso é claro que basta ir reduzindo sempre o valor do resto para que isso aconteça. A forma mais simples de o fazer é decrementá-lo:

```
--r;
```

Para que CI seja de facto invariante, há que escolher uma acção tal que:

```
// Aqui admite-se que:
// CI ∧ G ≡ dividendo = q × divisor + r ∧ 0 ≤ r ∧ divisor ≤ r, ou seja,
// dividendo = q × divisor + r ∧ divisor ≤ r.
acção
--r;
// Aqui pretende-se que CI ≡ dividendo = q × divisor + r ∧ 0 ≤ r.
```

Antes de determinar a acção, verifica-se qual a pré-condição mais fraca do progresso que leva à veracidade da condição invariante no final do passo:

```
// dividendo = q × divisor + r − 1 ∧ 0 ≤ r − 1.
--r; // o mesmo que r = r - 1;
// CI ≡ dividendo = q × divisor + r ∧ 0 ≤ r.
```

A acção deve portanto ser tal que

```
// dividendo = q × divisor + r ∧ divisor ≤ r.
acção
// dividendo = q × divisor + r − 1 ∧ 1 ≤ r.
```

As únicas variáveis livres no procedimento são `r` e `q`, pelo que se o progresso fez evoluir o valor de `r`, então a acção deverá actuar sobre `q`. Deverá pois ter o formato:

```
q = expressão;
```

Calculando a pré-condição mais fraca da acção:

```
// dividendo = expressão × divisor + r − 1 ∧ 1 ≤ r.
q = expressão;
// dividendo = q × divisor + r − 1 ∧ 1 ≤ r.
```

a expressão deve ser tal que

// $\texttt{dividendo} = \texttt{q} \times \texttt{divisor} + \texttt{r} \wedge \texttt{divisor} \leq \texttt{r}$.
// $\texttt{dividendo} = \textit{expressão} \times \texttt{divisor} + \texttt{r} - 1 \wedge 1 \leq \texttt{r}$.

É evidente que a primeira das asserções só implica a segunda se `divisor` $= 1$. Como se pretende que o ciclo funcione para qualquer divisor positivo, houve algo que falhou. Uma observação mais cuidada do passo leva a compreender que o progresso não pode fazer o resto decrescer de 1 em 1, mas de `divisor` em `divisor`! Se assim for, o passo é

// $CI \wedge G \equiv \texttt{dividendo} = \texttt{q} \times \texttt{divisor} + \texttt{r} \wedge \texttt{divisor} \leq \texttt{r}$.
acção
```
r -= divisor;
```
// $CI \equiv \texttt{dividendo} = \texttt{q} \times \texttt{divisor} + \texttt{r} \wedge 0 \leq \texttt{r}$.

e calculando de novo a pré-condição mais fraca do progresso

// $\texttt{dividendo} = \texttt{q} \times \texttt{divisor} + \texttt{r} - \texttt{divisor} \wedge 0 \leq \texttt{r} - \texttt{divisor}$, ou seja,
// $\texttt{dividendo} = (\texttt{q} - 1) \times \texttt{divisor} + \texttt{r} \wedge \texttt{divisor} \leq \texttt{r}$.
`r -= divisor; //` o mesmo que `r = r - divisor;`
// $CI \equiv \texttt{dividendo} = \texttt{q} \times \texttt{divisor} + \texttt{r} \wedge 0 \leq \texttt{r}$.

A acção deve portanto ser tal que

// $\texttt{dividendo} = \texttt{q} \times \texttt{divisor} + \texttt{r} \wedge \texttt{divisor} \leq \texttt{r}$.
acção
// $\texttt{dividendo} = (\texttt{q} - 1) \times \texttt{divisor} + \texttt{r} \wedge \texttt{divisor} \leq \texttt{r}$.

É evidente então que a acção pode ser simplesmente:

```
++q;
```

Tal pode ser confirmado determinando a pré-condição mais fraca da acção

// $\texttt{dividendo} = (\texttt{q} + 1 - 1) \times \texttt{divisor} + \texttt{r} \wedge \texttt{divisor} \leq \texttt{r}$, ou seja,
// $\texttt{dividendo} = \texttt{q} \times \texttt{divisor} + \texttt{r} \wedge \texttt{divisor} \leq \texttt{r}$.
`++q; //` o mesmo que `q = q + 1;`
// $\texttt{dividendo} = (\texttt{q} - 1) \times \texttt{divisor} + \texttt{r} \wedge \texttt{divisor} \leq \texttt{r}$.

e observando que $CI \wedge G$ implica essa pré-condição. A implicação é trivial neste caso, pois as duas asserções são idênticas!

Transformando o ciclo num ciclo for e colocando no procedimento:

```
/** Coloca em q e r respectivamente o quociente e o resto da divisão
    inteira de dividendo por divisor.
    @pre  PC ≡ 0 ≤ dividendo ∧ 0 < divisor.
    @post CO ≡ 0 ≤ r < divisor ∧ dividendo = q × divisor + r.
void divide(int const dividendo, int const divisor,
            int& q, int& r)
{
    assert(0 <= dividendo and 0 < divisor);

    r = dividendo;
    q = 0;
    // CI ≡ dividendo = q × divisor + r ∧ 0 ≤ r.
    while(divisor <= r) {
        ++q;
        r -= divisor;
    }

    assert(0 <= r and r < divisor and
           dividendo = q * divisor + r);
}
```

que também é comum ver escrito em C++ como

```
/** Coloca em q e r respectivamente o quociente e o resto da divisão
    inteira de dividendo por divisor.
    @pre  PC ≡ 0 ≤ dividendo ∧ 0 < divisor.
    @post CO ≡ 0 ≤ r < divisor ∧ dividendo = q × divisor + r.
void divide(int const dividendo, int const divisor,
            int& q, int& r)
{
    // CI ≡ dividendo = q × divisor + r ∧ 0 ≤ r.
    for(r = dividendo, q = 0; divisor <= r; ++q, r -= divisor)
        ;

    assert(0 <= r and r < divisor and
           dividendo = q * divisor + r);
}
```

onde o progresso da instrução `for` contém o passo completo. Esta é uma expressão idiomática do C++ pouco clara, e por isso pouco recomendável, mas que ilustra a utilização de um novo operador: a vírgula usada para separar o progresso e a acção do ciclo é o operador de sequenciamento do C++. Esse operador garante que o primeiro operando é calculado antes do segundo operando, e tem como resultado o valor do segundo operando. Por exemplo, as instruções

```
int n = 0;
n = (1, 2, 3, 4);
```

colocam o valor 4 na variável `n`.

Podem-se desenvolver algoritmos mais eficientes para a divisão inteira se se considerarem progressos que façam diminuir o valor do resto mais depressa. Uma ideia interessante é subtrair ao resto não apenas o divisor, mas o divisor multiplicado por uma potência tão grande quanto possível de 2.

Capítulo 5

Matrizes, vectores e outros agregados

É muitas vezes conveniente para a resolução de problemas ter uma forma de guardar agregados de valores de um determinado tipo. Neste capítulo apresentam-se duas alternativas para a representação de agregados em C++: as matrizes e os vectores. As primeiras fazem parte da linguagem propriamente dita e são bastante primitivas e pouco flexíveis. No entanto, há muitos problemas para os quais são a solução mais indicada, além de que existe muito código C++ escrito que as utiliza, pelo que a sua aprendizagem é importante. Os segundos não fazem parte da linguagem. São fornecidos pela biblioteca padrão do C++, que fornece um conjunto vasto de ferramentas que, em conjunto com a linguagem propriamente dita, resultam num potente ambiente de programação. Os vectores são consideravelmente mais flexíveis e fáceis de usar que as matrizes, pelo que estas serão apresentadas primeiro. No entanto, as secções sobre matrizes de vectores têm a mesma estrutura, pelo que o leitor pode optar por ler as secções pela ordem inversa à da apresentação.

Considere-se o problema de calcular a média de três valores de vírgula flutuante dados e mostrar cada um desses valores dividido pela média calculada. Um possível programa para resolver este problema é:

```
#include <iostream>

using namespace std;

int main()
{
    // Leitura:
    cout << "Introduza três valores: ";
    double a, b, c;
    cin >> a >> b >> c;

    // Cálculo da média:
    double const média = (a + b + c) / 3.0;

    // Divisão pela média:
```

```
        a /= média;
        b /= média;
        c /= média;

        // Escrita do resultado:
        cout << a << ' ' << b << ' ' << c << endl;
    }
```

O programa é simples e resolve bem o problema em causa. Mas será facilmente generalizável? E se se pretendesse dividir, já não três, mas 100 ou mesmo 1000 valores pela sua média? A abordagem seguida no programa acima seria no mínimo pouco elegante, já que seriam necessárias tantas variáveis quantos os valores a ler do teclado, e estas variáveis teriam de ser todas definidas explicitamente. Convém usar um *agregado* de variáveis.

5.1 Matrizes clássicas do C++

A generalização de este tipo de problemas faz-se recorrendo a uma das possíveis formas de agregado que são as matrizes clássicas do C++[1]. Uma matriz é um agregado de elementos do mesmo tipo que podem ser indexados (identificados) por números inteiros e que podem ser interpretados e usados como outra variável qualquer.

A conversão do programa acima para ler 1000 em vez de três valores pode ser feita facilmente recorrendo a este novo conceito:

```
    #include <iostream>

    using namespace std;

    int main()
    {
        int const número_de_valores = 1000;

        // Leitura:
        cout << "Introduza " << número_de_valores << " valores: ";

        // Definição da matriz com número_de_valores elementos do tipo double:
        double valores[número_de_valores];

        for(int i = 0; i != número_de_valores; ++i)
            cin >> valores[i]; // lê o i-ésimo valor do teclado.

        // Cálculo da média:
```

[1]Traduziu-se o inglês *array* por matriz, à falta de melhor alternativa. Isso pode criar algumas confusões se se usar uma biblioteca que defina o conceito matemático de matriz. Por isso a estas matrizes básicas se chama "matrizes clássicas do C++".

```
    double soma = 0.0;
    for(int i = 0; i != número_de_valores; ++i)
        soma += valores[i]; // acrescenta o i-ésimo valor à soma.

    double const média = soma / número_de_valores;

    // Divisão pela média:
    for(int i = 0; i != número_de_valores; ++i)
        valores[i] /= média; // divide o i-ésimo valor pela média.

    // Escrita do resultado:
    for(int i = 0; i != número_de_valores; ++i)
        cout << valores[i] << ' '; // escreve o i-ésimo valor.
    cout << endl;
}
```

Utilizou-se uma constante para representar o número de valores a processar. Dessa forma, alterar o programa para processar não 1000 mas 10000 valores é trivial: basta alterar o valor da constante. A alternativa à utilização da constante seria usar o valor literal 1000 explicitamente onde quer que fosse necessário. Isso implicaria que, ao adaptar o programa para fazer a leitura de 1000 para 10000 valores, o programador recorreria provavelmente a uma substituição de 1000 por 10000 ao longo de todo o texto do programa, usando as funcionalidade do editor de texto. Essa substituição poderia ter consequências desastrosas se o valor literal 1000 fosse utilizado em algum local do programa num contexto diferente.

Recorda-se que o nome de uma constante atribui um significado ao valor por ela representado. Por exemplo, definir num programa de gestão de uma disciplina as constantes:

```
int const número_máximo_de_alunos_por_turma = 50;
int const peso_do_trabalho_na_nota_final = 50;
```

permite escrever o código sem qualquer utilização explícita do valor 50. A utilização explícita do valor 50 obrigaria a inferir o seu significado exacto (número máximo de alunos ou peso do trabalho na nota final) a partir do contexto, tarefa que nem sempre é fácil e que se torna mais difícil à medida que os programas se vão tornando mais extensos.

5.1.1 Definição de matrizes

Para utilizar uma matriz de variáveis é necessário defini-la[2]. A sintaxe da definição de matrizes é simples:

```
tipo nome[número_de_elementos];
```

[2]Na realidade, para se usar uma variável, é necessário que ela esteja declarada, podendo por vezes a sua definição estar noutro local, um pouco como no caso das funções e procedimentos. No Capítulo 9 se verá como e quando se podem declarar variáveis ou constantes sem as definir.

em que *tipo* é o tipo de cada elemento da matriz, *nome* é o nome da matriz, e *número_de_elementos* é o número de elementos ou dimensão da nova matriz. Por exemplo:

```
int mi[10];    // matriz com 10 int.
char mc[80];   // matriz com 80 char.
float mf[20];  // matriz com 20 float.
double md[3];  // matriz com 3 double.
```

O número de elementos de uma matriz tem de ser um valor conhecido durante a compilação do programa (i.e., um valor literal ou uma constante). Não é possível criar uma matriz usando um número de elementos variável[3]. Mas é possível, e em geral aconselhável, usar constantes em vez de valores literais para indicar a sua dimensão:

```
int n = 50;
int const m = 100;
int matriz_de_inteiros[m];    // ok, m é uma constante.
int matriz_de_inteiros[300];  // ok, 300 é um valor literal.
int matriz_de_inteiros[n];    // errado, n não é uma constante.
```

Nem todas as constantes têm um valor conhecido durante a compilação: a palavra-chave `const` limita-se a indicar ao compilador que o objecto a que diz respeito não poderá ser alterado. Por exemplo:

```
// Valor conhecido durante a compilação:
int const número_máximo_de_alunos = 10000;
int alunos[número_máximo_de_alunos]; // ok.

int número_de_alunos_lido;
cin >> número_de_alunos_lido;

// Valor desconhecido durante a compilação:
int const número_de_alunos = número_de_alunos_lido;
int alunos[número_de_alunos]; // erro!
```

5.1.2 Indexação de matrizes

Seja a definição

```
int m[10];
```

Para atribuir o valor 5 ao sétimo elemento da matriz pode-se escrever:

[3] Na realidade podem-se definir matrizes com um dimensão variável, desde que seja uma matriz *dinâmica*. Alternativamente pode-se usar o tipo genérico `vector` da biblioteca padrão do C++. Ambos os assuntos serão vistos mais tarde.

```
m[6] = 5;
```

Ao valor 6, colocado entre `[]`, chama-se *índice*. Ao escrever numa expressão o nome de uma matriz seguido de `[]` com um determinado índice está-se a efectuar uma *operação de indexação*, i.e., a aceder a um dado elemento da matriz indicando o seu índice.

Os índices das matrizes são sempre números inteiros e começam sempre em zero. Assim, o primeiro elemento da matriz `m` é `m[0]`, o segundo `m[1]`, e assim sucessivamente. Embora esta convenção pareça algo estranha a início, ao fim de algum tempo torna-se muito natural. Recorde-se os problemas de contagem de anos que decorreram de chamar ano 1 ao primeiro ano de vida de Cristo e a consequente polémica acerca de quando ocorre de facto a mudança de milénio[4]...

Como é óbvio, os índices usados para aceder às matrizes não necessitam de ser constantes: podem-se usar variáveis, como aliás se pode verificar no exemplo da leitura de 1000 valores apresentado atrás. Assim, o exemplo anterior pode-se escrever:

```
int a = 6;
m[a] = 5; // atribui o inteiro 5 ao 7º elemento da matriz.
```

Usando de novo a analogia das folhas de papel, uma matriz é como um bloco de notas em que as folhas são numeradas a partir de 0 (zero). Ao se escrever `m[6] = 5`, está-se a dizer algo como "substitua-se o valor que está escrito na folha 6 (a sétima) do bloco `m` pelo valor 5". Na Figura 5.1 mostra-se um diagrama com a representação gráfica da matriz `m` depois desta atribuição (por '?' representa-se um valor arbitrário, também conhecido por "lixo").

m: int[10]									
m[0]:	m[1]:	m[2]:	m[3]:	m[4]:	m[5]:	m[6]:	m[7]:	m[8]:	m[9]:
?	?	?	?	?	?	5	?	?	?

Figura 5.1: Matriz `m` definida por `int m[10];` depois das instruções `int a = 6;` e `m[a] = 5;`.

Porventura uma das maiores deficiências da linguagem C++ está no tratamento das matrizes, particularmente na indexação. Por exemplo, o código seguinte não resulta em qualquer erro de compilação nem tão pouco na terminação do programa com uma mensagem de erro, isto apesar de tentar atribuir valores a posições inexistentes da matriz (as posições com índices -1 e 4):

[4] Note-se que o primeiro ano antes do nascimento de Cristo é o ano -1: não há ano zero! O problema é mais frequente do que parece. Em Portugal, por exemplo, a numeração dos andares começa em R/C, ou zero, enquanto nos EUA começa em um (e que número terá o andar imediatamente abaixo?). Ainda no que diz respeito a datas e horas, o primeiro dia de cada mês é 1, mas a primeira hora do dia (e o primeiro minuto de cada hora) é 0, apesar de nos relógios analógicos a numeração começar em 1! E, já agora, porque se usam em português as expressões absurdas "de hoje a oito dias" e "de hoje a quinze dias" em vez de "daqui a sete dias" ou "daqui a 14 dias"? Será que "amanhã" é sinónimo de "de hoje a dois dias" e "hoje" o mesmo que "de hoje a um dia"?

```
int a = 0;
int m[4];
int b = 0;

m[-1] = 1; // erro! só se pode indexar de 0 a 3!
m[4] = 2;  // erro! só se pode indexar de 0 a 3!
cout << a << ' ' << b << endl;
```

O que aparece provavelmente escrito no ecrã é, dependendo do ambiente em que o programa foi compilado e executado,

```
2 1
```

ou

```
1 2
```

dependendo da arrumação que é dada às variáveis `a`, `m` e `b` na memória. O que acontece é que as variáveis são arrumadas na memória (neste caso na pilha ou *stack*) do computador por ordem de definição, pelo que as "folhas de papel" correspondentes a `a` e `b` seguem ou precedem (conforme a direcção de crescimento da pilha) as folhas do "bloco de notas" correspondente a `m`. Assim, a atribuição `m[-1] = 1;` coloca o valor 1 na folha que precede a folha 0 de `m` na memória, que é normalmente a folha de `b` (a pilha normalmente cresce "para baixo", como se verá na disciplina de Arquitectura de Computadores).

Esta é uma fonte muito frequente de erros no C++, pelo que se recomenda extremo cuidado na utilização de matrizes. Uma vez que muitos ciclos usam matrizes, este é mais um argumento a favor do desenvolvimento disciplinado de ciclos (Secção 4.7) e da utilização de asserções (Secção 3.2.19).

5.1.3 Inicialização de matrizes

Tal como para as variáveis simples, também as matrizes só são inicializadas implicitamente quando são estáticas[5] . Matrizes automáticas, i.e., locais a alguma rotina (e sem o qualificador `static`) não são inicializadas: os seus elementos contêm inicialmente valores arbitrários ("lixo"). Mas é possível inicializar explicitamente as matrizes. Para isso, colocam-se os valores com que se pretende inicializar os elementos da matriz entre `{}`, por ordem e separados por vírgulas. Por exemplo:

```
int m[4] = {10, 20, 0, 0};
```

[5]Na realidade as variáveis de tipos definidos pelo programador ou os elementos de matrizes com elementos de um tipo definido pelo programador (com excepção dos tipos enumerados) são sempre inicializadas, ainda que implicitamente (nesse caso são inicializadas com o construtor por omissão, ver Secção 7.17.5). Só variáveis automáticas ou elementos de matrizes automáticas de tipos básicos do C++ ou de tipos enumerados não são inicializadas implicitamente, contendo por isso lixo se não forem inicializadas explicitamente.

Podem-se especificar menos valores de inicialização do que o número de elementos da matriz. Nesse caso, os elementos por inicializar explicitamente são inicializados implicitamente com 0 (zero), ou melhor, com o valor por omissão correspondente ao seu tipo (o valor por omissão dos tipos aritméticos é zero, o dos booleanos é falso, o dos enumerados, ver Capítulo 6, é zero, mesmo que este valor não seja tomado por nenhum dos seus valores literais enumerados, e o das classes, ver Secção 7.17.5, é o valor construído pelo construtor por omissão). Ou seja,

```
int m[4] = {10, 20};
```

tem o mesmo efeito que a primeira inicialização. Pode-se aproveitar este comportamento algo obscuro para forçar a inicialização de uma matriz inteira com zeros:

```
int grande[100] = {}; // inicializa toda a matriz com 0 (zero).
```

Mas, como este comportamento é tudo menos óbvio, recomenda-se que se comentem bem tais utilizações, tal como foi feito aqui.

Quando a inicialização é explícita, pode-se omitir a dimensão da matriz, sendo esta inferida a partir do número de inicializadores. Por exemplo,

```
int m[] = {10, 20, 0, 0};
```

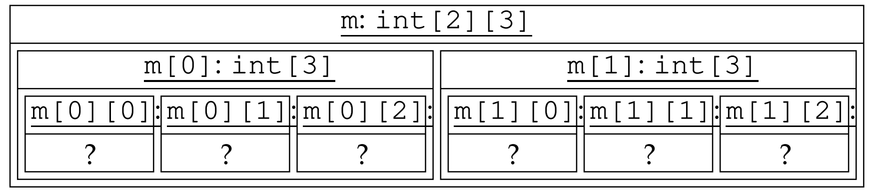
tem o mesmo efeito que as definições anteriores.

5.1.4 Matrizes multidimensionais

Em C++ não existe o conceito de matrizes multidimensionais. Mas a sintaxe de definição de matrizes permite a definição de matrizes cujos elementos são outras matrizes, o que acaba por ter quase o mesmo efeito prático. Assim, a definição:

```
int m[2][3];
```

é interpretada como significando `int (m[2])[3]`, o que significa "`m` é uma matriz com dois elementos, cada um dos quais é uma matriz com três elementos inteiros". Ou seja, graficamente:



Embora sejam na realidade matrizes de matrizes, é usual interpretarem-se como matrizes multidimensionais (de resto, será esse o termo usado daqui em diante). Por exemplo, a matriz m acima é interpretada como:

A indexação destas matrizes faz-se usando tantos índices quantas as "matrizes dentro de matrizes" (incluindo a exterior), ou seja, tantos índices quantas as dimensões da matriz. Para `m` conforme definida acima:

```
m[1][2] = 4;          // atribui 4 ao elemento na linha 1, coluna 2 da matriz.
int i = 0, j = 0;
m[i][j] = m[1][2]; // atribui 4 à posição (0,0) da matriz.
```

A inicialização destas matrizes pode ser feita como indicado, tomando em conta, no entanto, que cada elemento da matriz é por sua vez uma matriz e por isso necessita de ser inicializado da mesma forma. Por exemplo, a inicialização:

```
int m[2][3] = { {1, 2, 3}, {4} };
```

leva a que o troço de programa[6]

```
for(int i = 0; i != 2; ++i) {
    for(int j = 0; j != 3; ++j)
        cout << setw(2) << m[i][j];
   cout << endl; }
```

escreva no ecrã

```
1 2 3
4 0 0
```

5.1.5 Matrizes constantes

Não existe em C++ a noção de matriz constante. Um efeito equivalente pode no entanto ser obtido indicando que os elementos da matriz são constantes. Por exemplo:

```
int const dias_no_mês_em_ano_normal[] = {
    31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};

int const dias_no_mês_em_ano_bissexto[] = {
    31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
};
```

5.1.6 Matrizes como parâmetros de rotinas

É possível usar matrizes como parâmetros de rotinas. O programa original pode ser modularizado do seguinte modo:

[6] O manipulador `setw()` serve para indicar quantos caracteres devem ser usados para escrever o próximo valor inserido no canal. Por omissão são acrescentados espaços à esquerda para perfazer o número de caracteres indicado. Para usar este manipulador é necessário fazer `#include <iomanip>`.

```
#include <iostream>

using namespace std;

int const número_de_valores = 1000;

/** Preenche a matriz com valores lidos do teclado.
    @pre PC ≡ o canal de entrada (cin) contém número_de_valores
              números decimais.
    @post CO ≡ a matriz m contém os valores decimais que estavam em cin. */
void lê(double m[número_de_valores])
{
    for(int i = 0; i != número_de_valores; ++i)
        cin >> m[i]; // lê o i-ésimo elemento do teclado.
}

/** Devolve a média dos valores na matriz.
    @pre PC ≡ V.
    @post CO ≡ média = (S j : 0 ≤ j < número_de_valores : m[j]) / número_de_valores. */
double média(double const m[número_de_valores])
{
    double soma = 0.0;
    for(int i = 0; i != número_de_valores; ++i)
        soma += m[i]; // acrescenta o i-ésimo elemento à soma.
    return soma / número_de_valores;
}

/** Divide todos os elementos da matriz por divisor.
    @pre PC ≡ divisor ≠ 0 ∧ m = m.
    @post CO ≡ (Q j : 0 ≤ j < número_de_valores : m[j] = m[j] / divisor). */
void normaliza(double m[número_de_valores], double const divisor)
{
    assert(divisor != 0); // ver nota7 abaixo.

    for(int i = 0; i != número_de_valores; ++i)
        m[i] /= divisor; // divide o i-ésimo elemento.
}

/** Escreve todos os valores no ecrã.
    @pre PC ≡ V.
    @post CO ≡ o canal cout contém representações dos valores em m,
```

[7] Na pré-condição há um termo, $\texttt{m} = m$, em que ocorre uma variável matemática: m. Como esta variável não faz parte do programa C++, não é possível usar esse termo na instrução de asserção. De resto, essa variável é usada simplesmente para na condição objectivo significar o valor original da variável de programa m. Como se disse mais atrás, o mecanismo de instruções de asserção do C++ é algo primitivo, não havendo nenhuma forma de incluir referências ao valor original de uma variável.

```
                 por ordem. */
void escreve(double const m[número_de_valores])
{
    for(int i = 0; i != número_de_valores; ++i)
        cout << m[i] << ' '; // escreve o i-ésimo elemento.
}

int main()
{
    // Leitura:
    cout << "Introduza " << número_de_valores << " valores: ";
    double valores[número_de_valores];
    lê(valores);

    // Divisão pela média:
    normaliza(valores, média(valores));

    // Escrita do resultado:
    escreve(valores);
    cout << endl;
}
```

Passagens por referência

Para o leitor mais atento o programa acima tem, aparentemente, um erro. É que, nos procedimentos onde se alteram os valores da matriz (nomeadamente `lê()` e `normaliza()`), a matriz parece ser passada por valor e não por referência, visto que não existe na declaração dos parâmetros qualquer `&` (ver Secção 3.2.11). Acontece que as matrizes são sempre passadas por referência[8] e o `&` é, portanto, redundante[9]. Assim, os dois procedimentos referidos alteram de facto a matriz que lhes é passada como argumento.

Esta é uma característica desagradável das matrizes, pois introduz uma excepção à semântica das chamadas de rotinas. Esta característica mantém-se na linguagem apenas por razões de compatibilidade com código escrito na linguagem C.

Dimensão das matrizes parâmetro

Há uma outra característica das matrizes que pode causar alguma perplexidade. Se um parâmetro de uma rotina for uma matriz, então a sua dimensão é simplesmente ignorada pelo compilador. Assim, o compilador não verifica a dimensão das matrizes passadas como argumento, limitando-se a verificar o tipo dos seus elementos. Esta característica está relacionada com o

[8] A verdadeira explicação não é esta. A verdadeira explicação será apresentada quando se introduzir a noção de ponteiro no Capítulo 11.

[9] Poder-se-ia ter explicitado a referência escrevendo `void lê(double (&m)[número_de_valores])` (os `()` são fundamentais), muito embora isso não seja recomendado, pois sugere que na ausência do `&` a passagem se faz por valor, o que não é verdade.

facto de não se verificar a validade dos índices em operações de indexação[10], e com o facto de estarem proibidas a maior parte das operações sobre matrizes tratadas como um todo (ver Secção 5.1.7). Assim, `lêMatriz(double m[número_de_valores])` e `lêMatriz(double m[])` têm exactamente o mesmo significado[11].

Este facto pode ser usado a favor do programador (se ele tiver cuidado), pois permite-lhe escrever rotinas que operam sobre matrizes de dimensão arbitrária, desde que a dimensão das matrizes seja também passada como argumento[12]. Assim, na versão do programa que se segue todas as rotinas são razoavelmente genéricas, pois podem trabalhar com matrizes de dimensão arbitrária:

```
#include <iostream>

using namespace std;

/** Preenche os primeiros n elementos da matriz com n valores lidos do teclado.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m) ∧ o canal de entrada (cin)
             contém n números decimais[13].
    @post CO ≡ a matriz m contém os n valores decimais que estavam em cin. */
void lê(double m[], int const n)
{
    assert(0 <= n); // ver nota[14] abaixo.

    for(int i = 0; i != n; ++i)
```

[10] Na realidade as matrizes são sempre passadas na forma de um ponteiro para o primeiro elemento, como se verá no !!.

[11] Esta equivalência deve-se ao compilador ignorar a dimensão de uma matriz indicada como parâmetro de uma função ou procedimento. Isto não acontece se a passagem por referência for explicitada conforme indicado na nota Nota 9 na página 244: nesse caso a dimensão não é ignorada e o compilador verifica se o argumento respectivo tem tipo e dimensão compatíveis.

[12] Na realidade o valor passado como argumento não precisa de ser a dimensão real da matriz: basta que seja menor ou igual ao dimensão real da matriz. Isto permite processar apenas parte de uma matriz.

[13] Em expressões matemáticas, $\dim(m)$ significa a dimensão de uma matriz m. A mesma notação também se pode usar no caso dos vectores, que se estudarão em secções posteriores.

[14] Nesta instrução de asserção não é possível fazer melhor do que isto. Acontece que é impossível verificar se existem `n` valores decimais disponíveis para leitura no canal de entrada antes de os tentar extrair e, por outro lado, é característica desagradável da linguagem C++ não permitir saber a dimensão de matrizes usadas como parâmetro de uma função ou procedimento.

O primeiro problema poderia ser resolvido de duas formas. A primeira, mais correcta, passava por exigir ao utilizador a introdução dos números decimais pretendidos, insistindo com ele até a inserção ter sucesso. Neste caso a pré-condição seria simplificada. Alternativamente poder-se-ia usar uma asserção para saber se cada extracção teve sucesso. Como um canal, interpretado como um booleano, tem valor verdadeiro se e só se não houve qualquer erro de extracção, o ciclo poderia ser reescrito como:

```
for(int i = 0; i != n; ++i) {
    cin >> m[i]; // lê o i-ésimo elemento do teclado.
    assert(cin);
}
```

Um canal de entrada, depois de um erro de leitura, fica em estado de erro, falhando todas as extracções que se tentarem realizar. Um canal em estado de erro tem sempre o valor falso quando interpretado como um booleano.

```
        cin >> m[i]; // lê o i-ésimo elemento do teclado.
}

/** Devolve a média dos n primeiros elementos da matriz.
    @pre $PC \equiv 1 \leq n \wedge n \leq \dim(m)$.
    @post $CO \equiv$ média $= \frac{(\mathbf{S}\, j : 0 \leq j < n : m[j])}{n}$. */
double média(double const m[], int const n)
{
    assert(1 <= n);

    double soma = 0.0;
    for(int i = 0; i != n; ++i)
        soma += m[i]; // acrescenta o i-ésimo elemento à soma.
    return soma / n;
}

/** Divide os primeiros n elementos da matriz por divisor.
    @pre $PC \equiv 0 \leq n \wedge n \leq \dim(m) \wedge divisor \neq 0 \wedge m = m$.
    @post $CO \equiv \left(\mathbf{Q}\, j : 0 \leq j < n : m[j] = \frac{m[j]}{divisor}\right)$. */
void normaliza(double m[], int const n, double const divisor)
{
    assert(0 <= n and divisor != 0);

    for(int i = 0; i != n; ++i)
        m[i] /= divisor; // divide o i-ésimo elemento.
}

/** Escreve todos os valores no ecrã.
    @pre $PC \equiv 0 \leq n \wedge n \leq \dim(m)$.
    @post $CO \equiv$ o canal cout contém representações dos primeiros n
              elementos de m, por ordem. */
void escreve(double const m[], int const n)
{
    assert(0 <= n);

    for(int i = 0; i != n; ++i)
        cout << m[i] << ' '; // escreve o i-ésimo elemento.
}

int main()
{
    int const número_de_valores = 1000;

    // Leitura:
    cout << "Introduza " << número_de_valores << " valores: ";
    double valores[número_de_valores];
```

```
    lê(valores, número_de_valores);

    // Divisão pela média:
    normaliza(valores, número_de_valores,
              média(valores, número_de_valores));

    // Escrita do resultado:
    escreve(valores, número_de_valores);
    cout << endl;
}
```

O caso das matrizes multidimensionais é mais complicado: apenas é ignorada a primeira dimensão das matrizes multidimensionais definidas como parâmetros de rotinas. Assim, é impossível escrever

```
double determinante(double const m[][],
                    int const linhas, int const colunas)
{
    ...
}
```

com a esperança de definir uma função para calcular o determinante de uma matriz bidimensional de dimensões arbitrárias... É obrigatório indicar as dimensões de todas as matrizes excepto a mais "exterior":

```
double determinante(double const m[][10], int const linhas)
{
    ...
}
```

Ou seja, a flexibilidade neste caso resume-se a que a função pode trabalhar com um número arbitrário de linhas[15]...

5.1.7 Restrições na utilização de matrizes

Para além das particularidades já referidas relativamente à utilização de matrizes em C++, existem duas restrições que é importante conhecer:

Devolução Não é possível devolver matrizes (directamente) em funções. Esta restrição desagradável obriga à utilização de procedimentos para processamento de matrizes.

Operações Não são permitidas as atribuições ou comparações entre matrizes. Estas operações têm de ser realizadas através da aplicação sucessiva da operação em causa a cada um dos elementos da matriz. Pouco prático, em suma.

[15] Na realidade nem isso, porque o determinante de uma matriz só está definido para matrizes quadradas...

5.2 Vectores

Viu-se que as matrizes têm um conjunto de particularidades que as tornam pouco simpáticas de utilizar na prática. No entanto, a linguagem C++ traz associada a chamada biblioteca padrão (por estar disponível em qualquer ambiente de desenvolvimento que se preze) que disponibiliza um conjunto muito vasto de ferramentas (rotinas, variáveis e constantes, tipos, etc.), entre os quais um tipo genérico chamado `vector`. Este tipo genérico é uma excelente alternativa às matrizes e é apresentado brevemente nesta secção.

Pode-se resolver o problema de generalizar o programa apresentado no início deste capítulo recorrendo a outra das possíveis formas de agregado: os vectores. Ao contrário das matrizes, os vectores não fazem parte da linguagem C++. Os vectores são um tipo genérico, ou melhor uma classe C++ genérica, definido na biblioteca padrão do C++ e a que se acede colocando

```
#include <vector>
```

no início do programa. As noções de classe C++ e classe C++ genérica serão abordadas no Capítulo 7 e no Capítulo 13, respectivamente.

Um vector é um contentor de itens do mesmo tipo que podem ser indexados (identificados) por números inteiros e que podem ser interpretados e usados como outra variável qualquer. Ao contrário das matrizes, os vectores podem ser redimensionados sempre que necessário.

A conversão do programa inicial para ler 1000 em vez de três valores pode ser feita facilmente recorrendo ao tipo genérico `vector`:

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    int const número_de_valores = 1000;

    // Leitura:
    cout << "Introduza " << número_de_valores << " valores: ";

    // Definição de um vector com número_de_valores itens do tipo double:
    vector<double> valores(número_de_valores);

    for(int i = 0; i != número_de_valores; ++i)
        cin >> valores[i]; // lê o i-ésimo valor do teclado.

    // Cálculo da média:
    double soma = 0.0;
    for(int i = 0; i != número_de_valores; ++i)
```

```
        soma += valores[i]; // acrescenta o i-ésimo valor à soma.

    double const média = soma / número_de_valores;

    // Divisão pela média:
    for(int i = 0; i != número_de_valores; ++i)
        valores[i] /= média; // divide o i-ésimo valor pela média.

    // Escrita do resultado:
    for(int i = 0; i != número_de_valores; ++i)
        cout << valores[i] << ' '; // escreve o i-ésimo valor.
    cout << endl;
}
```

5.2.1 Definição de vectores

Para utilizar um vector é necessário defini-lo. A sintaxe da definição de vectores é simples:

```
vector<tipo> nome(número_de_itens);
```

em que *tipo* é o tipo de cada item do vector, *nome* é o nome do vector, e *número_de_itens* é o número de itens ou dimensão inicial do novo vector. É possível omitir a dimensão inicial do vector: nesse caso o vector inicialmente não terá qualquer elemento:

```
vector<tipo> nome;
```

Por exemplo:

```
vector<int>    vi(10); // vector com 10 int.
vector<char>   vc(80); // vector com 80 char.
vector<float>  vf(20); // vector com 20 float.
vector<double> vd;     // vector com zero double.
```

Um vector é uma variável como outra qualquer. Depois das definições acima pode-se dizer que `vd` é uma variável do tipo `vector<double>` (vector de `double`) que contém zero itens.

5.2.2 Indexação de vectores

Seja a definição

```
vector<int> v(10);
```

Para atribuir o valor 5 ao sétimo item do vector pode-se escrever:

```
v[6] = 5;
```

Ao valor 6, colocado entre `[]`, chama-se *índice*. Ao escrever numa expressão o nome de um vector seguido de `[]` com um determinado índice está-se a efectuar uma *operação de indexação*, i.e., a aceder a um dado item do vector indicando o seu índice.

Os índices dos vectores, como os das matrizes, são sempre números inteiros e começam sempre em zero. Assim, o primeiro item do vector `v` é `v[0]`, o segundo `v[1]`, e assim sucessivamente.

Os índices usados para aceder aos vectores não necessitam de ser constantes: podem-se usar variáveis, como aliás se pode verificar no exemplo da leitura de 1000 valores apresentado atrás. Assim, o exemplo anterior pode-se escrever:

```
int a = 6;
v[a] = 5; // atribui o inteiro 5 ao 7º item do vector.
```

Da mesma forma que acontece com as matrizes, também com os vectores as indexações não são verificadas. Por exemplo, o código seguinte não resulta em qualquer erro de compilação e provavelmente também não resulta na terminação do programa com uma mensagem de erro, isto apesar de tentar atribuir valores a posições inexistentes do vector:

```
vector<int> v(4);

v[-1] = 1; // erro! só se pode indexar de 0 a 3!
v[4] = 2;  // erro! só se pode indexar de 0 a 3!
```

Esta é uma fonte muito frequente de erros no C++, pelo que se recomenda extremo cuidado na indexação de vectores.

5.2.3 Inicialização de vectores

Os itens de um vector, ao contrário do que acontece com as matrizes, são sempre inicializados. A inicialização usada é a chamada inicialização por omissão, que no caso dos itens serem dos tipos básicos (`int`, `char`, `float`, etc.), é a inicialização com o valor zero. Por exemplo, depois da definição

```
vector<int> v(4);
```

todos os itens do vector `v` têm o valor zero.

Não é possível inicializar os itens de um vector como se inicializam os elementos de uma matriz. Mas pode-se especificar um valor com o qual se pretendem inicializar todos os itens do vector. Por exemplo, depois da definição

```
vector<int> v(4, 13);
```

todos os itens do vector `v` têm o valor 13.

5.2.4 Operações

Os vectores são variáveis como qualquer outras. A diferença principal está em suportarem a invocação das chamadas *operações*, um conceito que será visto em pormenor no Capítulo 7. Simplificando grosseiramente, as operações são rotinas associadas a um tipo (classe C++) que se invocam para uma determinada variável desse tipo. A forma de invocação é um pouco diferente da usada para as rotinas "normais". As operações invocam-se usando o operador `.` de selecção de membro (no Capítulo 7 se verá o que é exactamente um *membro*). Este operador tem dois operandos. O primeiro é a variável à qual a operação é aplicado. O segundo é a operação a aplicar, seguida dos respectivos argumentos, se existirem.

Uma operação extremamente útil do tipo genérico `vector` chama-se `size()` e permite saber a dimensão actual de um vector. Por exemplo:

```
vector<double> distâncias(10);

cout << "A dimensão é " << distâncias.size() << '.' << endl;
```

O valor devolvido pela operação `size()` pertence a um dos tipos inteiros do C++, mas a norma da linguagem não especifica qual... No entanto, é fornecido um sinónimo desse tipo para cada tipo de vector. Esse sinónimo chama-se `size_type` e pode ser acedido usando o operador `::` de resolução de âmbito em que o operando esquerdo é o tipo do vector:

```
vector<double>::size_type
```

Um ciclo para percorrer e mostrar todos os itens de um vector pode ser escrito fazendo uso deste tipo e da operação `size()`:

```
for(vector<double>::size_type i = 0; i != distâncias.size(); ++i)
    cout << distâncias[i] << endl;
```

Uma outra operação pode ser usada se se pretender apenas saber se um vector está ou não vazio, i.e., se se pretender saber se um vector tem dimensão zero. A operação chama-se `empty()` e devolve verdadeiro se o vector estiver vazio e falso no caso contrário.

5.2.5 Acesso aos itens de vectores

Viu-se que a indexação de vectores é insegura. O tipo genérico vector fornece uma operação chamada `at()` que permite aceder a um item dado o seu índice, tal como a operação de indexação, mas que garantidamente resulta num erro[16] no caso de o índice ser inválido. Esta operação tem como argumento o índice do item ao qual se pretende aceder e devolve esse mesmo item. Voltando ao exemplo original,

[16]Ou melhor, resulta no lançamento de uma *excepção*, ver !!.

```
vector<int> v(4);

v.at(-1) = 1; // erro! só se pode indexar de 0 a 3!
v.at(4) = 2;  // erro! só se pode indexar de 0 a 3!
```

este troço de código levaria à terminação abrupta[17] do programa em que ocorresse logo ao ser executada a primeira indexação errada.

Existem ainda duas operações que permitem aceder aos dois itens nas posições extremas do vector. A operação `front()` devolve o primeiro item do vector. A operação `back()` devolve o último item do vector. Por exemplo, o seguinte troço de código

```
vector<double> distâncias(10);

for(vector<double>::size_type i = 0; i != distâncias.size(); ++i)
    distâncias[i] = double(i); // converte i num double.

cout << "Primeiro: " << distâncias.front() << '.' << endl;
cout << "Último: " << distâncias.back() << '.' << endl;
```

escreve no ecrã

```
Primeiro: 0.
Último: 9.
```

5.2.6 Alteração da dimensão de um vector

Ao contrário do que acontece com as matrizes, os vectores podem ser redimensionados sempre que necessário. Para isso recorre-se à operação `resize()`. Esta operação recebe como primeiro argumento a nova dimensão pretendida para o vector e, opcionalmente, recebe como segundo argumento o valor com o qual são inicializados os possíveis novos itens criados pelo redimensionamento. Caso o segundo argumento não seja especificado, os possíveis novos itens são inicializados usando a inicialização por omissão. Por exemplo, o resultado de

```
vector<double> distâncias(3, 5.0);
distâncias.resize(6, 10.0);
distâncias.resize(9, 15.0);
distâncias.resize(8);
for(vector<double>::size_type i = 0; i != distâncias.size(); ++i)
    cout << distâncias[i] << endl;
```

é surgir no ecrã

---

[17] Isto, claro está, se ninguém capturar a excepção lançada, como se verá no !!.

```
5
5
5
10
10
10
15
15
```

Se a intenção for eliminar todos os itens do vector, reduzindo a sua dimensão a zero, pode-se usar a operação `clear()` como se segue:

```
distâncias.clear();
```

Os vectores não podem ter uma dimensão arbitrária. Para saber a dimensão máxima possível para um vector usar a operação `max_size()`:

```
cout << "Dimensão máxima do vector de distâncias é "
     << distâncias.max_size() << " itens." << endl;
```

Finalmente, como as operações que alteram a dimensão de um vector podem ser lentas, já que envolvem pedidos de memória ao sistema operativo, o tipo genérico fornece uma operação chamada `reserve()` para pré-reservar espaço para itens que se prevê venham a ser necessários no futuro. Esta operação recebe como argumento o número de itens que se prevê virem a ser necessários na pior das hipóteses e reserva espaço para eles. Enquanto a dimensão do vector não ultrapassar a reserva feita todas as operações têm eficiência garantida. A operação `capacity()` permite saber qual o número de itens para os quais há espaço reservado em cada momento. A secção seguinte demonstra a utilização desta operação.

5.2.7 Inserção e remoção de itens

As operações mais simples para inserção e remoção de itens referem-se ao extremo final dos vectores. Se se pretender acrescentar um item no final de um vector pode-se usar a operação `push_back()`, que recebe como argumento o valor com o qual inicializar o novo item. Se se pretender remover o último item de um vector pode-se usar a operação `pop_back()`. Por exemplo, em vez de escrever

```
vector<double> distâncias(10);

for(vector<double>::size_type i = 0; i != distâncias.size(); ++i)
    distâncias[i] = double(i); // converte i num double.
```

é possível escrever

```
vector<double> distâncias;

for(vector<double>::size_type i = 0; i != 10; ++i)
    distâncias.push_back(double(i));
```

A segunda versão, com `push_back()`, é particularmente vantajosa quando o número de itens a colocar no vector não é conhecido à partida.

Se a dimensão máxima do vector for conhecida à partida, pode ser útil começar por reservar espaço suficiente para os itens a colocar:

```
vector<double> distâncias;
distâncias.reserve(10);
/* Neste local o vector está vazio, mas tem espaço para crescer sem precisar de recor-
rer ao sistema operativo para requerer memória. */
for(vector<double>::size_type i = 0; i != 10; ++i)
    distâncias.push_back(double(i));
```

A operação `pop_back()` pode ser conjugada com a operação `empty()` para mostrar os itens de um vector pela ordem inversa e, simultaneamente, esvaziá-lo:

```
while(not distâncias.empty()) {
    cout << distâncias.back() << endl;
    distâncias.pop_back();
}
```

5.2.8 Vectores multidimensionais?

O tipo genérico vector não foi pensado para representar matrizes multidimensionais. No entanto, é possível definir vectores de vectores usando a seguinte sintaxe, apresentada sem mais explicações[18]:

```
vector<vector<tipo> > nome(linhas, vector<tipo>(colunas));
```

onde *`linhas`* é o número de linhas pretendido para a matriz e *`colunas`* o número de colunas. O processo não é elegante, mas funciona e pode ser estendido para mais dimensões:

```
vector<vector<vector<tipo> > >
    nome(planos, vector<vector<tipo>>(linhas,
                                      vector<tipo>(colunas)));
```

[18] Tem de se deixar um espaço entre símbolos > sucessivos para que não se confundam com o operador >>.

No entanto, estas variáveis não são verdadeiramente representações de matrizes, pois, por exemplo no primeiro caso, é possível alterar a dimensão das linhas da "matriz" independentemente umas das outras.

É possível simplificar as definições acima se o objectivo não for emular as matrizes mas simplesmente definir um vector de vectores. Por exemplo, o seguinte troço de programa

```
/* Definição de um vector de vectores com quatro itens inicialmente, cada um tendo ini-
cialmente dimensão nula. */
vector<vector<char> > v(4);

v[0].resize(1, 'a');
v[1].resize(2, 'b');
v[2].resize(3, 'c');
v[3].resize(4, 'd');

for(vector<vector<char> >::size_type i = 0; i != v.size(); ++i) {
    for(vector<char>::size_type j = 0; j != v[i].size(); ++j)
        cout << v[i][j];
    cout << endl;
}
```

quando executado resulta em

```
a
bb
ccc
dddd
```

5.2.9 Vectores constantes

Como para qualquer outro tipo, pode-se definir vectores constantes. A grande dificuldade está em inicializar um vector constante, visto que a inicialização ao estilo das matrizes não é permitida:

```
vector<char> const muitos_aa(10, 'a');
```

5.2.10 Vectores como parâmetros de rotinas

A passagem de vectores como argumento não difere em nada de outros tipos. Podem-se passar vectores por valor ou por referência. O programa original pode ser modularizado de modo a usar rotinas do seguinte modo:

```
#include <iostream>
#include <vector>

using namespace std;

/** Preenche o vector com valores lidos do teclado.
    @pre PC ≡ o canal de entrada (cin) contém v.size() números decimais.
    @post CO ≡ os itens do vector v contêm os v.size() valores decimais
               que estavam em cin. */
void lê(vector<double>& v)
{
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        cin >> v[i]; // lê o i-ésimo elemento do teclado.
}

/** Devolve a média dos itens do vector.
    @pre PC ≡ 1 ≤ v.size().
    @post CO ≡ média = (S j : 0 ≤ j < v.size() : v[j]) / v.size(). */
double média(vector<double> const v)
{
    assert(1 <= v.size());

    double soma = 0.0;
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        soma += v[i]; // acrescenta o i-ésimo elemento à soma.
    return soma / v.size();
}

/** Divide os itens do vector por divisor.
    @pre PC ≡ divisor ≠ 0 ∧ v = v.
    @post CO ≡ (Q j : 0 ≤ j < v.size() : v[j] = v[j] / divisor). */
void normaliza(vector<double>& v, double const divisor)
{
    assert(divisor != 0);

    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        v[i] /= divisor; // divide o i-ésimo elemento.
}

/** Escreve todos os valores do vector no ecrã.
    @pre PC ≡ V.
    @post CO ≡ o canal cout contém representações dos itens de v, por ordem. */
void escreve(vector<double> const v)
{
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        cout << v[i] << ' '; // escreve o i-ésimo elemento.
```

```
}

int main()
{
    int const número_de_valores = 1000;

    // Leitura:
    cout << "Introduza " << número_de_valores << " valores: ";
    vector<double> valores(número_de_valores);
    lê(valores);

    // Divisão pela média:
    normaliza(valores, média(valores));

    // Escrita do resultado:
    escreve(valores);
    cout << endl;
}
```

No entanto, é importante perceber que *a passagem de um vector por valor implica a cópia do vector inteiro*, o que pode ser extremamente ineficiente. O mesmo não acontece se a passagem se fizer por referência, mas seria desagradável passar a mensagem errada ao compilador e ao leitor do código, pois uma passagem por referência implica que a rotina em causa pode alterar o argumento. Para resolver o problema usa-se uma passagem de argumentos por referência constante.

5.2.11 Passagem de argumentos por referência constante

Considere-se de novo função para somar a média de todos os itens de um vector de inteiros:

```
double média(vector<double> const v)
{
    assert(1 <= v.size());

    double soma = 0.0;
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        soma += v[i]; // acrescenta o i-ésimo elemento à soma.
    return soma / v.size();
}
```

De acordo com o mecanismo de chamada de funções (descrito na Secção 3.2.11), e uma vez que o vector é passado por valor, é evidente que a chamada da função `média()` implica a construção de uma nova variável do tipo `vector<int>`, o parâmetro `v`, que é inicializada a partir do vector passado como argumento e que é, portanto, uma cópia desse mesmo argumento. Se o vector passado como argumento tiver muitos itens, como é o caso no programa acima, é evidente que esta cópia pode ser muito demorada, podendo mesmo em alguns casos tornar a utilização da função impossível na prática. Como resolver o problema? Se a passagem do vector fosse feita por referência, e não por valor, essa cópia não seria necessária. Assim, poder-se-ia aumentar a eficiência da chamada da função definindo-a como

```
double média(vector<double>& v) // má ideia!
{
    assert(1 <= v.size());

    double soma = 0.0;
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        soma += v[i]; // acrescenta o i-ésimo elemento à soma.
    return soma / v.size();
}
```

Esta nova versão da função tem uma desvantagem: na primeira versão, o consumidor da função e o compilador sabiam que o vector passado como argumento não poderia ser alterado pela função, já que esta trabalhava com uma cópia. Na nova versão essa garantia não é feita. O problema pode ser resolvido se se disser de algum modo que, apesar de o vector ser passado por referência, a função não está autorizada a alterá-lo. Isso consegue-se recorrendo de novo ao qualificador `const`:

```
double média(vector<double> const& v) // boa ideia!
{
    assert(1 <= v.size());

    double soma = 0.0;
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        soma += v[i]; // acrescenta o i-ésimo elemento à soma.
    return soma / v.size();
}
```

Pode-se agora converter o programa de modo a usar passagens por referência constante em todas as rotinas onde é útil fazê-lo:

```
#include <iostream>
#include <vector>

using namespace std;

/** Preenche o vector com valores lidos do teclado.
    @pre  PC ≡ o canal de entrada (cin) contém v.size() números decimais.
    @post CO ≡ os itens do vector v contêm os v.size() valores decimais
               que estavam em cin. */
void lê(vector<double>& v)
```

```
{
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        cin >> v[i]; // lê o i-ésimo elemento do teclado.
}

/** Devolve a média dos itens do vector.
    @pre $PC \equiv 1 \leq$ v.size().
    @post $CO \equiv$ média $= \frac{(\mathbf{S}\, j : 0 \leq j < \texttt{v.size()} : \texttt{v}[j])}{\texttt{v.size()}}$. */
double média(vector<double> const& v)
{
    assert(1 <= v.size());

    double soma = 0.0;
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        soma += v[i]; // acrescenta o i-ésimo elemento à soma.
    return soma / v.size();
}

/** Divide os itens do vector por divisor.
    @pre $PC \equiv$ divisor $\neq 0 \wedge$ v $= v$.
    @post $CO \equiv \left(\mathbf{Q}\, j : 0 \leq j <\right.$ v.size() : v$[j] = \left.\frac{v[j]}{\texttt{divisor}}\right)$. */
void normaliza(vector<double>& v, double const divisor)
{
    assert(divisor != 0);

    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        v[i] /= divisor; // divide o i-ésimo elemento.
}

/** Escreve todos os valores do vector no ecrã.
    @pre $PC \equiv \mathcal{V}$.
    @post $CO \equiv$ o canal cout contém representações dos itens de v, por ordem. */
void escreve(vector<double> const& v)
{
    for(vector<double>::size_type i = 0; i != v.size(); ++i)
        cout << v[i] << ' '; // escreve o i-ésimo elemento.
}

int main()
{
    int const número_de_valores = 1000;

    // Leitura:
    cout << "Introduza " << número_de_valores << " valores: ";
    vector<double> valores(número_de_valores);
    lê(valores);
```

```
    // Divisão pela média:
    normaliza(valores, média(valores));

    // Escrita do resultado:
    escreve(valores);
    cout << endl;
}
```

Compare-se esta versão do programa com a versão usando matrizes apresentada mais atrás.

As passagens por referência constante têm uma característica adicional que as distingue das passagens por referência simples: permitem passar qualquer constante (e.g., um valor literal) como argumento, o que não era possível no caso das passagens por referência simples. Por exemplo:

```
// Mau código. Bom para exemplos apenas...
int soma1(int& a, int& b)
{
    return a + b;
}

/* Não é grande ideia usar referências constantes para os tipos bási-
cos do C++ (não se poupa nada): */
int soma2(int const& a, int const& b)
{
    return a + b;
}

int main()
{
    int i = 1, j = 2, res;
    res = soma1(i, j); // válido.
    res = soma2(i, j); // válido.
    res = soma1(10, 20); // erro!
    res = soma2(10, 20); // válido! os parâmetros a e b
                         // tornam-se sinónimos de duas
                         // constantes temporárias inicializadas
                         // com 10 e 20.
}
```

A devolução de vectores em funções é possível, ao contrário do que acontece com as matrizes, embora não seja recomendável. A devolução de um vector implica também uma cópia: o valor devolvido é uma cópia do vector colocado na expressão de retorno. O problema pode, por vezes, ser resolvido também por intermédio de referências, como se verá na Secção 7.7.1.

5.2.12 Outras operações com vectores

Ao contrário do que acontece com as matrizes, é possível atribuir vectores e mesmo compará-los usando os operadores de igualdade e os operadores relacionais. A única particularidade merecedora de referência é o que se entende por um vector ser menor do que outro.

A comparação entre vectores com os operadores relacionais é feita usando a chamada *ordenação lexicográfica*. Esta ordenação é a mesma que é usada para colocar as palavras nos vulgares dicionários[19], e usa os seguintes critérios:

1. A comparação é feita da esquerda para a direita, começando portanto nos primeiros itens dos dois vectores em comparação, e prosseguindo sempre a par ao longo dos vectores.
2. A comparação termina assim que ocorrer uma de três condições:
   (a) os itens em comparação são diferentes: nesse caso o vector com o item mais pequeno é considerado menor que o outro.
   (b) um dos vectores não tem mais itens, sendo por isso mais curto que o outro: nesse caso o vector mais curto é considerado menor que o outro.
   (c) nenhum dos vectores tem mais itens: nesse caso os dois vectores têm o mesmo comprimento e os mesmos valores dos itens e são por isso considerados iguais.

Representando os vectores por tuplos:

- $() = ()$.
- $() < (10)$.
- $(1, 10, 20) < (2)$.
- $(1, 2, 3, 4) < (1, 3, 2, 4)$.
- $(1, 2) < (1, 2, 3)$.

O troço de código abaixo

```
vector<int> v1;

v1.push_back(1);
v1.push_back(2);

vector<int> v2;
```

[19] De acordo com [4]:
lexicográfico (cs). [De *lexicografia* + *-ico*^{2}.] ***Adj.*** Respeitante à lexicografia.
lexicografia (cs). [De *lexico-* + *-grafia*.] ***S.f.*** A ciência do lexicógrafo.
lexicógrafo (cs). [Do gr. *lexikográphos*.] ***S.m.*** Autor de dicionário ou de trabalho a respeito de palavras duma língua; dicionarista; lexicólogo. [Cf. *lexicografo*, do v. *lexicografar*.]

```
v2.push_back(1);
v2.push_back(2);
v2.push_back(3);

if(v1 < v2)
    cout << "v1 é mesmo menor que v2." << endl;
```

produz no ecrã

```
v1 é mesmo menor que v2.
```

como se pode verificar observando o último exemplo acima.

5.3 Algoritmos com matrizes e vectores

Esta secção apresenta o desenvolvimento pormenorizado de quatro funções usando ciclos e operando com matrizes e vectores e serve, por isso, de complemento aos exemplos de desenvolvimento de ciclos apresentados no capítulo anterior.

5.3.1 Soma dos elementos de uma matriz

O objectivo da função é calcular a soma dos valores dos primeiros `n` elementos de uma matriz `m`.

O primeiro passo da resolução do problema é a sua especificação, ou seja, a escrita da estrutura da função, incluindo a pré-condição e a condição objectivo:

```
/** Devolve a soma dos primeiros n elementos da matriz m.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ soma = (S j : 0 ≤ j < n : m[j]). */
int soma(int const m[], int const n)
{
    assert(0 <= n);

    int soma = ...;
    ...
    return soma;
}
```

Como é necessário no final devolver a soma dos elementos, define-se imediatamente uma variável `soma` para guardar esse valor. Usa-se uma variável com o mesmo nome da função para que a condição objectivo do ciclo e a condição objectivo da função sejam idênticas[20].

[20] O nome de uma variável local pode ser igual ao da função em que está definida. Nesse caso ocorre uma ocultação do nome da função (ver Secção 3.2.14). A única consequência desta ocultação é que para invocar recursivamente a função, se isso for necessário, é necessário usar o operador de resolução de âmbito `::`.

O passo seguinte consiste em, tendo-se percebido que a solução pode passar pela utilização de um ciclo, determinar uma condição invariante apropriada, o que se consegue usualmente por enfraquecimento da condição objectivo. Neste caso pode-se simplesmente substituir o limite superior do somatório (a constante `n`) por uma variável `i` inteira, limitada a um intervalo apropriado de valores. Assim, a estrutura do ciclo é

```
// PC ≡ 0 ≤ n ∧ n ≤ dim(m).
int soma = ...;
int i = ...;
// CI ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n.
while(G) {
    passo
}
// CO ≡ soma = (S j : 0 ≤ j < n : m[j]).
```

Identificada a condição invariante, é necessário escolher uma guarda tal que seja possível demonstrar que $CI \wedge \neg G \Rightarrow CO$. Ou seja, escolher uma guarda que, quando o ciclo terminar, conduza naturalmente à condição objectivo[21]. Neste caso é evidente que a escolha correcta para $\neg G$ é $\mathtt{i} = \mathtt{n}$, ou seja, $G \equiv \mathtt{i} \neq \mathtt{n}$:

```
// PC ≡ 0 ≤ n ∧ n ≤ dim(m).
int soma = ...;
int i = ...;
// CI ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n.
while(i != n) {
    passo
}
// CO ≡ soma = (S j : 0 ≤ j < n : m[j]).
```

De seguida deve-se garantir, por escolha apropriada das instruções das inicializações, que a condição invariante é verdadeira no início do ciclo. Como sempre, devem-se escolher as instruções mais simples que conduzem à veracidade da condição invariante. Neste caso é fácil verificar que se deve inicializar `i` com zero e `soma` também com zero (recorde-se de que a soma de zero elementos é, por definição, zero):

```
// PC ≡ 0 ≤ n ∧ n ≤ dim(m).
int soma = 0;
int i = 0;
// CI ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n.
while(i != n) {
    passo
}
// CO ≡ soma = (S j : 0 ≤ j < n : m[j]).
```

[21] A condição invariante é verdadeira, por construção, no início, durante, e no final do ciclo: por isso se chama condição invariante. Quando o ciclo termina, tem de se ter forçosamente que a guarda é falsa, ou seja, que $\neg G$ é verdadeira.

Como se pretende que o algoritmo termine, deve-se agora escolher um progresso que o garanta (o passo é normalmente dividido em duas partes, o progresso *prog* e a acção *acção*). Sendo `i` inicialmente zero, e sendo $0 \leq \mathtt{n}$ (pela pré-condição), é evidente que uma simples incrementação da variável `i` conduzirá à falsidade da guarda, e portanto à terminação do ciclo, ao fim de exactamente `n` iterações do ciclo:

```
// PC ≡ 0 ≤ n ∧ n ≤ dim(m).
int soma = 0;
int i = 0;
// CI ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n.
while(i != n) {
    acção
    ++i;
}
// CO ≡ soma = (S j : 0 ≤ j < n : m[j]).
```

Finalmente, é necessário construir uma acção que garanta a veracidade da condição invariante depois do passo e *apesar* do progresso entretanto realizado. Sabe-se que a condição invariante e a guarda são verdadeiras antes do passo, logo, é necessário encontrar uma acção tal que:

```
// CI ∧ G ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n ∧ i ≠ n, ou seja,
// soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i < n.
acção
++i;
// CI ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n.
```

Pode-se começar por verificar qual a pré-condição mais fraca do progresso que garante que a condição invariante é recuperada:

```
// soma = (S j : 0 ≤ j < i + 1 : m[j]) ∧ 0 ≤ i + 1 ≤ n, ou seja,
// soma = (S j : 0 ≤ j < i + 1 : m[j]) ∧ −1 ≤ i < n.
++i;
// CI ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n.
```

Se se admitir que $0 \leq \mathtt{i}$, então o último termo do somatório pode ser extraído:

```
// soma = (S j : 0 ≤ j < i : m[j]) + m[i] ∧ 0 ≤ i < n.
// soma = (S j : 0 ≤ j < i + 1 : m[j]) ∧ −1 ≤ i < n.
```

Conclui-se que a acção deverá ser escolhida de modo a que:

```
// soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i < n.
acção
// soma = (S j : 0 ≤ j < i : m[j]) + m[i] ∧ 0 ≤ i < n.
```

o que se consegue facilmente com a acção:

```
soma += m[i];
```

A função completa é então

```
/** Devolve a soma dos primeiros n elementos da matriz m.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ soma = (S j : 0 ≤ j < n : m[j]). */
int soma(int const m[], int const n)
{
    assert(0 <= n);

    int soma = 0;
    int i = 0;
    // CI ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n.
    while(i != n) {
        soma += m[i];
        ++i;
    }
    return soma;
}
```

Pode-se ainda converter o ciclo de modo a usar a instrução `for`:

```
/** Devolve a soma dos primeiros n elementos da matriz m.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ soma = (S j : 0 ≤ j < n : m[j]). */
int soma(int const m[], int const n)
{
    assert(0 <= n);

    int soma = 0;
    // CI ≡ soma = (S j : 0 ≤ j < i : m[j]) ∧ 0 ≤ i ≤ n.
    for(int i = 0; i != n; ++i)
        soma += m[i];
    return soma;
}
```

Manteve-se a condição invariante como um comentário antes do ciclo, pois é muito importante para a sua compreensão. A condição invariante no fundo reflecte como o ciclo (e a função) funcionam, ao contrário da pré-condição e da condição objectivo, que se referem àquilo que o ciclo (ou a função) faz. A pré-condição e a condição objectivo são úteis para o programador consumidor da função, enquanto a condição invariante é útil para o programador produtor e para o programador "assistência técnica", que pode precisar de verificar a correcta implementação do ciclo.

O ciclo desenvolvido corresponde a parte da função `média()` usada em exemplos anteriores.

5.3.2 Soma dos itens de um vector

O desenvolvimento no caso dos vectores é semelhante ao usado para as matrizes. A função resultante desse desenvolvimento é

```
/** Devolve a soma dos itens do vector v.
    @pre PC ≡ V.
    @post CO ≡ soma = (S j : 0 ≤ j < v.size() : v[j]). */
int soma(vector<int> const& v)
{
    int soma = 0;
    // CI ≡ soma = (S j : 0 ≤ j < i : v[j]) ∧ 0 ≤ i ≤ v.size().
    for(vector<int>::size_type i = 0; i != v.size(); ++i)
        soma += v[i];
    return soma;
}
```

5.3.3 Índice do maior elemento de uma matriz

O objectivo é construir uma função que devolva um dos índices do máximo valor contido nos primeiros `n` elementos de uma matriz `m`. Como não se especifica qual dos índices devolver caso existam vários elementos com o valor máximo, arbitra-se que a função deve devolver o primeiro desses índices. Assim, a estrutura da função é:

```
/** Devolve o índice do primeiro elemento com o máximo valor entre os primeiros
    n elementos da matriz m.
    @pre PC ≡ 1 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ 0 ≤ índiceDoPrimeiroMáximoDe < n∧
            (Q j : 0 ≤ j < n : m[j] ≤ m[índiceDoPrimeiroMáximoDe]) ∧
            (Q j : 0 ≤ j < índiceDoPrimeiroMáximoDe : m[j] < m[índiceDoPrimeiroMáximoDe]). */
int índiceDoPrimeiroMáximoDe(int const m[], int const n)
{
    assert(1 <= n);

    int i = ...;
    ...

    // Sem ciclos não se pode fazer muito melhor:
    assert(0 <= i < n and m[0] <= m[i]);

    return i;
}
```

A condição objectivo indica que o índice devolvido tem de pertencer à gama de índices válidos para a matriz, que o valor do elemento de `m` no índice devolvido tem de ser maior ou igual aos valores de todos os elementos da matriz (estas condições garantem que o índice devolvido é um dos índices do valor máximo na matriz) e que os valores dos elementos com índice menor do que o índice devolvido têm de ser estritamente menores que o valor da matriz no índice devolvido (ou seja, o índice devolvido é o primeiro dos índices dos elementos com valor máximo na matriz). A pré-condição, neste caso, impõe que `n` não pode ser zero, pois não tem sentido falar do máximo de um conjunto vazio, além de obrigar `n` a ser inferior ou igual à dimensão da matriz.

É evidente que a procura do primeiro máximo de uma matriz pode recorrer a um ciclo. A estrutura do ciclo é pois:

```
// PC ≡ 1 ≤ n ∧ n ≤ dim(m).
int i = ...;
while(G) {
    passo
}
// CO ≡ 0 ≤ i < n ∧ (Q j : 0 ≤ j < n : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]).
```

onde a condição objectivo do ciclo se refere à variável `i`, a ser devolvida pela função no seu final.

Os dois primeiros passos da construção de um ciclo, obtenção da condição invariante e da guarda, são, neste caso, idênticos aos do exemplo anterior: substituição da constante `n` por uma nova variável `k`:

```
// PC ≡ 1 ≤ n ∧ n ≤ dim(m).
int i = ...;
int k = ...;
// CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//      0 ≤ k ≤ n.
while(k != n) {
    passo
}
// CO ≡ 0 ≤ i < n ∧ (Q j : 0 ≤ j < n : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]).
```

A condição invariante indica que a variável `i` contém sempre o índice do primeiro elemento cujo valor é o máximo dos valores contidos nos `k` primeiros elementos da matriz.

A inicialização a usar também é simples, embora desta vez não se possa inicializar `k` com 0, pois não existe máximo de um conjunto vazio (não se poderia atribuir qualquer valor a `i`)! Assim, a solução é inicializar `k` com 1 e `i` com 0:

```
// PC ≡ 1 ≤ n ∧ n ≤ dim(m).
int i = 0;
int k = 1;
// CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//      0 ≤ k ≤ n.
```

```
while(k != n) {
    passo
}
// CO ≡ 0 ≤ i < n ∧ (Q j : 0 ≤ j < n : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]).
```

Analisando os termos da condição invariante um a um, verifica-se que:

1. $0 \leq \mathtt{i} < \mathtt{k}$ fica $0 \leq 0 < 1$ que é verdadeiro;
2. $(\mathbf{Q}\, j : 0 \leq j < \mathtt{k} : \mathtt{m}[j] \leq \mathtt{m}[\mathtt{i}])$ fica $(\mathbf{Q}\, j : 0 \leq j < 1 : \mathtt{m}[j] \leq \mathtt{m}[0])$, que é o mesmo que $\mathtt{m}[0] \leq \mathtt{m}[0]$, que é verdadeiro;
3. $(\mathbf{Q}\, j : 0 \leq j < \mathtt{i} : \mathtt{m}[j] < \mathtt{m}[\mathtt{i}])$ fica $(\mathbf{Q}\, j : 0 \leq j < 0 : \mathtt{m}[j] < \mathtt{m}[0])$ que, como existem zero termos no quantificador, tem valor verdadeiro por definição; e
4. $0 \leq \mathtt{k} \leq \mathtt{n}$ fica $0 \leq 1 \leq \mathtt{n}$, que é verdadeira desde que a pré-condição o seja, o que se admite acontecer;

isto é, a inicialização leva à veracidade da condição invariante, como se pretendia.

O passo seguinte é a determinação do progresso. Mais uma vez usa-se a simples incrementação de `k` em cada passo, que conduz forçosamente à terminação do ciclo:

```
// PC ≡ 1 ≤ n ∧ n ≤ dim(m).
int i = 0;
int k = 1;
// CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//      0 ≤ k ≤ n.
while(k != n) {
    acção
    ++k;
}
// CO ≡ 0 ≤ i < n ∧ (Q j : 0 ≤ j < n : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]).
```

A parte mais interessante deste exemplo é a determinação da acção a utilizar para manter a condição invariante verdadeira apesar do progresso. A acção tem de ser tal que

```
// CI∧G ≡ 0 ≤ i < k∧(Q j : 0 ≤ j < k : m[j] ≤ m[i])∧(Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//      0 ≤ k ≤ n ∧ k ≠ n, ou seja,
// 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//      0 ≤ k < n.
acção
++k;
// CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//      0 ≤ k ≤ n.
```

Mais uma vez começa-se por encontrar a pré-condição mais fraca do progresso:

```
// 0 ≤ i < k + 1 ∧ (Q j : 0 ≤ j < k + 1 : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//       0 ≤ k + 1 ≤ n, ou seja,
// 0 ≤ i ≤ k ∧ (Q j : 0 ≤ j < k + 1 : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//       −1 ≤ k < n.
++k;
// CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//       0 ≤ k ≤ n.
```

Se se admitir que $0 \leq$ `k`, então o último termo do primeiro quantificador universal pode ser extraído:

```
// 0 ≤ i ≤ k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ m[k] ≤ m[i]∧
//       (Q j : 0 ≤ j < i : m[j] < m[i]) ∧ 0 ≤ k < n.
// 0 ≤ i ≤ k ∧ (Q j : 0 ≤ j < k + 1 : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//       −1 ≤ k < n.
```

Conclui-se que a acção deverá ser escolhida de modo a que:

```
// 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//       0 ≤ k < n.
acção
// 0 ≤ i ≤ k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ m[k] ≤ m[i]∧
//       (Q j : 0 ≤ j < i : m[j] < m[i]) ∧ 0 ≤ k < n.
```

É claro que a acção deverá afectar apenas a variável `i`, pois a variável `k` é afectada pelo progresso. Mas como? Haverá alguma circunstância em que não seja necessária qualquer alteração da variável `i`, ou seja, em que a acção possa ser a instrução nula? Comparando termo a termo as asserções antes e depois da acção, conclui-se que isso só acontece se `m[k]` $\leq$ `m[i]`. Então a acção deve consistir numa instrução de selecção:

```
// 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//       0 ≤ k < n.
if(m[k] <= m[i])
    // G₁ ≡ m[k] ≤ m[i].
    ; // instrução nula!
else
    // G₂ ≡ m[i] < m[k].
    instrução₂

// 0 ≤ i ≤ k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ m[k] ≤ m[i]∧
//       (Q j : 0 ≤ j < i : m[j] < m[i]) ∧ 0 ≤ k < n.
```

Resta saber que instrução deve ser usada para resolver o problema no caso em que `m[i]` $<$ `m[k]`. Falta, pois, falta determinar uma $\texttt{instrução}_2$ tal que:

```
// 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//       0 ≤ k < n ∧ m[i] < m[k].
instrução₂
// 0 ≤ i ≤ k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ m[k] ≤ m[i]∧
//       (Q j : 0 ≤ j < i : m[j] < m[i]) ∧ 0 ≤ k < n.
```

Antes da instrução, `i` contém o índice do primeiro elemento contendo o maior dos valores dos elementos com índices entre 0 e `k` *exclusive*. Por outro lado, o elemento de índice `k` contém um valor superior ao valor do elemento de índice `i`. Logo, há um elemento entre 0 e `k` *inclusive* com um valor superior a todos os outros: o elemento de índice `k`. Assim, a variável `i` deverá tomar o valor `k`, de modo a continuar a ser o índice do elemento com maior valor entre os valores inspeccionados. A instrução a usar é portanto:

```
i = k;
```

A ideia é que, quando se atinge um elemento com valor maior do que aquele que se julgava até então ser o máximo, deve-se actualizar o índice do máximo. Para verificar que assim é, calcule-se a pré-condição mais fraca que conduz à asserção final pretendida:

```
// 0 ≤ k ≤ k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[k]) ∧ m[k] ≤ m[k]∧
//       (Q j : 0 ≤ j < k : m[j] < m[k]) ∧ 0 ≤ k < n, ou seja,
// (Q j : 0 ≤ j < k : m[j] < m[k]) ∧ 0 ≤ k < n.
i = k;
// 0 ≤ i ≤ k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ m[k] ≤ m[i]∧
//       (Q j : 0 ≤ j < i : m[j] < m[i]) ∧ 0 ≤ k < n.
```

Falta pois verificar se

```
// 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//       0 ≤ k < n ∧ m[i] < m[k]
// ⇒ (Q j : 0 ≤ j < k : m[j] < m[k]) ∧ 0 ≤ k < n.
```

Eliminando os termos que não são necessários para verificar a implicação,

```
// (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ 0 ≤ k < n ∧ m[i] < m[k]
// ⇒ (Q j : 0 ≤ j < k : m[j] < m[k]) ∧ 0 ≤ k < n.
```

é evidente que a implicação é verdadeira e, portanto, a atribuição `i = k;` resolve o problema. Assim, a acção do ciclo é a instrução de selecção

```
if(m[k] <= m[i])
    // G₁ ≡ m[k] ≤ m[i].
    ; // instrução nula!
else
    // G₂ ≡ m[i] < m[k].
    i = k;
```

que pode ser simplificada para uma instrução condicional mais simples

```
if(m[i] < m[k])
    i = k;
```

O ciclo completo fica

```
// PC ≡ 1 ≤ n ∧ n ≤ dim(m).
int i = 0;
int k = 1;
// CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//      0 ≤ k ≤ n.
while(k != n) {
    if(m[i] < m[k])
        i = k;
    ++k;
}
// CO ≡ 0 ≤ i < n ∧ (Q j : 0 ≤ j < n : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]).
```

que pode ser convertido para um ciclo `for`:

```
// PC ≡ 1 ≤ n ∧ n ≤ dim(m).
int i = 0;
// CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]) ∧
//      0 ≤ k ≤ n.
for(int k = 1; k != n; ++k)
    if(m[i] < m[k])
        i = k;
// CO ≡ 0 ≤ i < n ∧ (Q j : 0 ≤ j < n : m[j] ≤ m[i]) ∧ (Q j : 0 ≤ j < i : m[j] < m[i]).
```

A função completa é:

```
/** Devolve o índice do primeiro elemento com o máximo valor entre os primeiros
    n elementos da matriz m.
    @pre  PC ≡ 1 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ 0 ≤ índiceDoPrimeiroMáximoDe < n∧
              (Q j : 0 ≤ j < n : m[j] ≤ m[índiceDoPrimeiroMáximoDe]) ∧
              (Q j : 0 ≤ j < índiceDoPrimeiroMáximoDe : m[j] < m[índiceDoPrimeiroMáximoDe]). */
int índiceDoPrimeiroMáximoDe(int const m[], int const n)
{
    assert(1 <= n);

    int i = 0;
    // CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : m[j] ≤ m[i]) ∧
    //      (Q j : 0 ≤ j < i : m[j] < m[i]) ∧ 0 ≤ k ≤ n.
```

```
    for(int k = 1; k != n; ++k)
        if(m[i] < m[k])
            i = k;

    // Sem ciclos não se pode fazer muito melhor (amostragem em três locais):
    assert(0 <= i < n and
           m[0] <= m[i] and m[n / 2] <= m[i] and
           m[n - 1] <= m[i]);

    return i;
}
```

5.3.4 Índice do maior item de um vector

O desenvolvimento no caso dos vectores é semelhante ao usado para as matrizes. A função resultante desse desenvolvimento é

```
/** Devolve o índice do primeiro item com o máximo valor do vector v.
    @pre  PC ≡ 1 ≤ v.size().
    @post CO ≡ 0 ≤ índiceDoPrimeiroMáximoDe < v.size()∧
              (Q j : 0 ≤ j < v.size() : v[j] ≤ v[índiceDoPrimeiroMáximoDe])∧
              (Q j : 0 ≤ j < índiceDoPrimeiroMáximoDe : v[j] < v[índiceDoPrimeiroMáximoDe]). */
int índiceDoPrimeiroMáximo(vector<int> const& v)
{
    assert(1 <= v.size());

    int i = 0;
    // CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : v[j] ≤ v[i])∧
    //      (Q j : 0 ≤ j < i : v[j] < v[i]) ∧ 0 ≤ k ≤ v.size().
    for(vector<int>::size_type k = 1; k != v.size(); ++k)
        if(v[i] < v[k])
            i = k;

    // Sem ciclos não se pode fazer muito melhor (amostragem em três locais):
    assert(0 <= i < v.size() and
           m[0] <= m[i] and m[v.size() / 2] <= m[i] and
           m[v.size() - 1] <= m[i]);

    return i;
}
```

5.3.5 Elementos de uma matriz num intervalo

Pretende-se escrever uma função que devolva o valor lógico verdadeiro se e só se os valores dos `n` primeiros elementos de uma matriz `m` estiverem entre `mínimo` e `máximo` (*inclusive*).

Neste caso a estrutura da função e a sua especificação (i.e., a sua pré-condição e a sua condição objectivo) são mais fáceis de escrever:

```
/** Devolve verdadeiro se os primeiros n elementos da matriz m têm valores
    entre mínimo e máximo.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ estáEntre = (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo). */
bool estáEntre(int const m[], int const n,
               int const mínimo, int const máximo)
{
    assert(0 <= n);

    ...
}
```

Uma vez que esta função devolve um valor booleano, que apenas pode ser V ou F, vale a pena verificar em que circunstâncias cada uma das instruções de retorno

```
return false;
return true;
```

resolve o problema. Começa por se verificar a pré-condição mais fraca da primeira destas instruções:

```
// CO ≡ F = (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo), ou seja,
// (E j : 0 ≤ j < n : m[j] < mínimo ∨ máximo < m[j]).
return false;
// CO ≡ estáEntre = (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo).
```

Consequentemente, deve-se devolver falso se existir um elemento da matriz fora da gama pretendida.

Depois verifica-se a pré-condição mais fraca da segunda das instruções de retorno:

```
// CO ≡ V = (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo), ou seja,
// (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo).
return true;
// CO ≡ estáEntre = (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo).
```

Logo, deve-se devolver verdadeiro se todos os elementos da matriz estiverem na gama pretendida.

Que ciclo resolve o problema? Onde colocar, se é que é possível, estas instruções de retorno? Seja a condição invariante do ciclo:

$$CI \equiv (\mathbf{Q}\, j : 0 \leq j < \mathtt{i} : \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i} \leq \mathtt{n},$$

onde i é uma variável introduzida para o efeito. Esta condição invariante afirma que todos os elementos inspeccionados até ao momento (com índices inferiores a i) estão na gama pretendida. Esta condição invariante foi obtida intuitivamente, e não através da metodologia de Dijkstra. Em particular, esta condição invariante obriga o ciclo a terminar de uma forma pouco usual se se encontrar um elemento fora da gama pretendida, como se verá mais abaixo.

Se a guarda for

$$G \equiv \mathtt{i} \neq \mathtt{n},$$

então no final do ciclo tem-se $CI \wedge \neg G$, ou seja,

$$CI \wedge \neg G \equiv \left(\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}\right) \wedge 0 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i} = \mathtt{n},$$

que implica

$$\left(\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{n}\, :\, \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}\right).$$

Logo, a instrução

```
return true;
```

deve terminar a função.

Que inicialização usar? A forma mais simples de tornar verdadeira a condição invariante é inicializar i com 0, pois o quantificador "qualquer que seja" sem qualquer termo tem valor lógico verdadeiro. Pode-se agora acrescentar à função o ciclo parcialmente desenvolvido:

```
/** Devolve verdadeiro se os primeiros n elementos da matriz m têm valores
    entre mínimo e máximo.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ estáEntre = (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo). */
bool estáEntre(int const m[], int const n,
               int const mínimo, int const máximo)
{
    assert(0 <= n);

    int i = 0;
    // CI ≡ (Q j : 0 ≤ j < i : mínimo ≤ m[j] ≤ máximo) ∧ 0 ≤ i ≤ n.
    while(i != n) {
        passo
    }

    // Aqui recorreu-se a amostragem, mais uma vez:
    assert(n == 0 or // se n for zero a resposta correcta é true.
           (mínimo <= m[0] <= máximo and
           (mínimo <= m[n / 2] <= máximo and
           (mínimo <= m[n - 1] <= máximo));

    return true;
}
```

Que progresso usar? Pelas razões habituais, o progresso mais simples a usar é:

```
++i;
```

Resta determinar a acção de modo a que a condição invariante seja de facto invariante. Ou seja, é necessário garantir que

// $CI \wedge G \equiv (\mathbf{Q}\, j : 0 \leq j < \mathtt{i} : \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i} \neq \mathtt{n}$, ou seja,
// $(\mathbf{Q}\, j : 0 \leq j < \mathtt{i} : \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i} < \mathtt{n}$.
acção
`++i;`
// $CI \equiv (\mathbf{Q}\, j : 0 \leq j < \mathtt{i} : \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i} \leq \mathtt{n}$.

Verificando qual a pré-condição mais fraca que, depois do progresso, conduz à veracidade da condição invariante, conclui-se:

// $(\mathbf{Q}\, j : 0 \leq j < \mathtt{i}+1 : \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i}+1 \leq \mathtt{n}$, ou seja,
// $(\mathbf{Q}\, j : 0 \leq j < \mathtt{i}+1 : \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge -1 \leq \mathtt{i} < \mathtt{n}$.
`++i;`
// $CI \equiv (\mathbf{Q}\, j : 0 \leq j < \mathtt{i} : \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i} \leq \mathtt{n}$.

Se se admitir que $0 \leq \mathtt{i}$, então o último termo do quantificador universal pode ser extraído:

// $(\mathbf{Q}\, j : 0 \leq j < \mathtt{i} : \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge \mathtt{mínimo} \leq \mathtt{m}[\mathtt{i}] \leq \mathtt{máximo} \wedge 0 \leq \mathtt{i} < \mathtt{n}$.
// $(\mathbf{Q}\, j : 0 \leq j < \mathtt{i}+1 : \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge -1 \leq \mathtt{i} < \mathtt{n}$.

Conclui-se facilmente que, se $\mathtt{mínimo} \leq \mathtt{m}[\mathtt{i}] \leq \mathtt{máximo}$, então não é necessária qualquer acção para que a condição invariante se verifique depois do progresso. Isso significa que a acção consiste numa instrução de selecção:

// $(\mathbf{Q}\, j : 0 \leq j < \mathtt{i} : \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i} < \mathtt{n}$.
`if(mínimo <= m[i] and m[i] <= máximo)`
    // $G_1 \equiv \mathtt{mínimo} \leq \mathtt{m}[\mathtt{i}] \leq \mathtt{máximo}$.
    `; //` instrução nula!
`else`
    // $G_2 \equiv \mathtt{m}[\mathtt{i}] < \mathtt{mínimo} \vee \mathtt{máximo} < \mathtt{m}[\mathtt{i}]$.
$\mathtt{instrução}_2$
`++i;`
// $CI \equiv (\mathbf{Q}\, j : 0 \leq j < \mathtt{i} : \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i} \leq \mathtt{n}$.

Resta pois verificar que instrução deve ser usada na alternativa da instrução de selecção. Para isso começa por se verificar que, antes dessa instrução, se verificam simultaneamente a condição invariante a guarda e a segunda guarda da instrução de selecção, ou seja,

$$\begin{aligned} CI \wedge G \wedge G_2 \quad \equiv \quad & (\mathbf{Q}\, j : 0 \leq j < \mathtt{i} : \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i} < \mathtt{n} \wedge \\ & (\mathtt{m}[\mathtt{i}] < \mathtt{mínimo} \vee \mathtt{máximo} < \mathtt{m}[\mathtt{i}]) \,, \end{aligned}$$

Traduzindo para português vernáculo: "os primeiros `i` elementos da matriz estão dentro da gama pretendida mas o $\mathtt{i}+1$-ésimo (de índice `i`) não está". É claro portanto que

$$
\begin{aligned}
CI \wedge G \wedge G_2 \quad &\Rightarrow \quad (\mathbf{E}\, j : 0 \leq j < \mathtt{i}+1 : \mathtt{m}[j] < \mathtt{mínimo} \vee \mathtt{máximo} < \mathtt{m}[j]) \wedge 0 \leq \mathtt{i} < \mathtt{n} \\
&\Rightarrow \quad (\mathbf{E}\, j : 0 \leq j < \mathtt{n} : \mathtt{m}[j] < \mathtt{mínimo} \vee \mathtt{máximo} < \mathtt{m}[j])
\end{aligned}
$$

Ou seja, existe pelo menos um elemento com índice entre 0 e `i` *inclusive* que não está na gama pretendida e portanto o mesmo se passa para os elementos com índices entre 0 e `n` *exclusive*.

Conclui-se que a instrução alternativa da instrução de selecção deve ser

```
return false;
```

pois termina imediatamente a função (e portanto o ciclo), devolvendo o valor apropriado (ver pré-condição mais fraca desta instrução mais atrás).

A acção foi escolhida de tal forma que, ou termina o ciclo devolvendo o valor apropriado (falso), ou garante a validade da condição invariante apesar do progresso.

A função completa é:

```
/** Devolve verdadeiro se os primeiros n elementos da matriz m têm valores
    entre mínimo e máximo.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ estáEntre = (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo). */
bool estáEntre(int const m[], int const n,
               int const mínimo, int const máximo)
{
    assert(0 <= n);

    int i = 0;
    // CI ≡ (Q j : 0 ≤ j < i : mínimo ≤ m[j] ≤ máximo) ∧ 0 ≤ i ≤ n.
    while(i != n) {
        if(mínimo <= m[i] and m[i] <= máximo)
            ; // instrução nula!
        else
            return false;
        ++i;
    }

    // Aqui recorreu-se a amostragem, mais uma vez:
    assert(n == 0 or // se n for zero a resposta correcta é true.
           (mínimo <= m[0] <= máximo and
           (mínimo <= m[n / 2] <= máximo and
           (mínimo <= m[n - 1] <= máximo));

    return true;
}
```

Trocando as instruções alternativas da instrução de selecção (e convertendo-a numa instrução condicional) e convertendo o ciclo `while` num ciclo `for` obtém-se a versão final da função:

```
/** Devolve verdadeiro se os primeiros n elementos da matriz m têm valores
    entre mínimo e máximo.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ estáEntre = (Q j : 0 ≤ j < n : mínimo ≤ m[j] ≤ máximo). */
bool estáEntre(int const m[], int const n,
               int const mínimo, int const máximo)
{
    assert(0 <= n);

    // CI ≡ (Q j : 0 ≤ j < i : mínimo ≤ m[j] ≤ máximo) ∧ 0 ≤ i ≤ n.
    for(int i = 0; i != n; ++i)
        if(m[i] < mínimo or máximo < m[i])
            return false;

    // Aqui recorreu-se a amostragem, mais uma vez:
    assert(n == 0 or // se n for zero a resposta correcta é true.
           (mínimo <= m[0] <= máximo and
           (mínimo <= m[n / 2] <= máximo and
           (mínimo <= m[n - 1] <= máximo));

    return true;
}
```

5.3.6 Itens de um vector num intervalo

O desenvolvimento no caso dos vectores é semelhante ao usado para as matrizes. A função resultante desse desenvolvimento é

```
/** Devolve verdadeiro se os itens do vector v têm valores entre mínimo e máximo.
    @pre PC ≡ V.
    @post CO ≡ estáEntre = (Q j : 0 ≤ j < v.size() : mínimo ≤ v[j] ≤ máximo).
*/
bool estáEntre(vector<int> const& v,
               int const mínimo, int const máximo)
{
    // CI ≡ (Q j : 0 ≤ j < i : mínimo ≤ v[j] ≤ máximo) ∧ 0 ≤ i ≤ v.size().
    for(vector<int>::size_type i = 0; i != v.size(); ++i)
        if(v[i] < mínimo or máximo < v[i])
            return false;

    // Aqui recorreu-se a amostragem, mais uma vez:
    assert(n == 0 or // se n for zero a resposta correcta é true.
```

```
            (mínimo <= m[0] <= máximo and
            (mínimo <= m[n / 2] <= máximo and
            (mínimo <= m[n - 1] <= máximo));

        return true;
    }
```

5.3.7 Segundo elemento de uma matriz com um dado valor

O objectivo agora é encontrar o índice do segundo elemento com valor `k` nos primeiros `n` elementos de uma matriz `m`.

Neste caso a pré-condição é um pouco mais complicada do que para os exemplos anteriores, pois tem de se garantir que existem pelo menos dois elementos com o valor pretendido, o que, por si só, implica que a matriz tem de ter pelo menos dois elementos.

A condição objectivo é mais simples. Afirma que o índice a devolver deve corresponder a um elemento com valor `k` e que, no conjunto dos elementos com índice menor, existe apenas um elemento com valor `k` (diz ainda que o índice deve ser válido, neste caso maior do que 0, porque têm de existir pelo menos dois elementos com valores iguais até ao índice). Assim, a estrutura da função é:

```
/** Devolve o índice do segundo elemento com valor k nos primeiros n
    elementos da matriz m.
    @pre PC ≡ 2 ≤ n ∧ n ≤ dim(m) ∧ 2 ≤ (N j : 0 ≤ j < n : m[j] = k).
    @post CO ≡ (N j : 0 ≤ j < índiceDoSegundo : m[j] = k) = 1∧
              1 ≤ índiceDoSegundo < n ∧ m[índiceDoSegundo] = k. */
int índiceDoSegundo(int const m[], int const n, int const k)
{
    int i = ...;
    ...
    return i;
}
```

Para resolver este problema é necessário um ciclo, que pode ter a seguinte estrutura:

```
// PC ≡ 2 ≤ n ∧ n ≤ dim(m) ∧ 2 ≤ (N j : 0 ≤ j < n : m[j] = k).
int i = ...;
while(G) {
    passo
}
// CO ≡ (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ 1 ≤ i < n ∧ m[i] = k.
```

Neste caso não existe na condição objectivo do ciclo um quantificador onde se possa substituir (com facilidade) uma constante por uma variável. Assim, a determinação da condição objectivo pode ser tentada factorizando a condição objectivo, que é uma conjunção, em CI e $\neg G$.

Uma observação atenta das condições revela que a escolha apropriada é

$$CO \equiv \overbrace{(\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \land 1 \leq \mathtt{i} < \mathtt{n}}^{CI} \land \overbrace{\mathtt{m[i]} = \mathtt{k}}^{\neg G}$$

Que significa esta condição invariante? Simplesmente que, durante todo o ciclo, tem de se garantir que há um único elementos da matriz com valor `k` e com índice entre 0 e `i` *exclusive*.

O ciclo neste momento é

```
// PC ≡ 2 ≤ n ∧ n ≤ dim(m) ∧ 2 ≤ (N j : 0 ≤ j < n : m[j] = k).
int i = ...;
// CI ≡ (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ 1 ≤ i < n.
while(m[i] != k) {
    passo
}
// CO ≡ (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ 1 ≤ i < n ∧ m[i] = k.
```

Um problema com a escolha que se fez para a condição invariante é que, aparentemente, não é fácil fazer a inicialização: como escolher um valor para `i` tal que existe um elemento de valor `k` com índice inferior a `i`? Em vez de atacar imediatamente esse problema, adia-se o problema e assume-se que a inicialização está feita. O passo seguinte, portanto, é determinar o passo do ciclo. Antes do passo sabe-se que $CI \land G$, ou seja:

$$CI \land G \equiv (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \land 1 \leq \mathtt{i} < \mathtt{n} \land \mathtt{m[i]} \neq \mathtt{k}.$$

Mas

$$(\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \land \mathtt{m[i]} \neq \mathtt{k}$$

é o mesmo que

$$(\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i} + 1\, :\, \mathtt{m}[j] = \mathtt{k}) = 1,$$

pois, sendo $\mathtt{m[i]} \neq \mathtt{k}$, pode-se estender a gama de valores tomados por j sem afectar a contagem de afirmações verdadeiras:

$$CI \land G \equiv (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i} + 1\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \land 1 \leq \mathtt{i} < \mathtt{n}.$$

Atente-se bem na expressão acima. Será que pode ser verdadeira quando `i` atinge o seu maior valor possível de acordo com o segundo termo da conjunção, i.e., quando $\mathtt{i} = \mathtt{n} - 1$? Sendo $\mathtt{i} = \mathtt{n} - 1$, o primeiro termo da conjunção fica

$$(\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{n}\, :\, \mathtt{m}[j] = \mathtt{k}) = 1,$$

o que não pode acontecer, dada a pré-condição! Logo $\mathtt{i} \neq \mathtt{n} - 1$, e portanto

$$CI \land G \equiv (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i} + 1\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \land 1 \leq \mathtt{i} < \mathtt{n} - 1.$$

O passo tem de ser escolhido de modo a garantir a invariância da condição invariante, ou seja, de modo a garantir que

// $CI \wedge G \equiv (\mathbf{N}\, j : 0 \leq j < \mathtt{i} + 1 : \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n} - 1$.
passo
// $CI \equiv (\mathbf{N}\, j : 0 \leq j < \mathtt{i} : \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n}$.

Começa por se escolher um progresso apropriado. Qual a forma mais simples de garantir que a guarda se torna falsa ao fim de um número finito de passos? Simplesmente incrementando `i`. Se `i` atingisse alguma vez o valor `n` (índice para além do fim da matriz) sem que a guarda se tivesse alguma vez tornado falsa, isso significaria que a matriz não possuía pelo menos dois elementos com o valor `k`, o que violaria a pré-condição. Logo, o ciclo tem de terminar antes de `i` atingir `n`, ao fim de um número finito de passos, portanto. O passo do ciclo pode então ser escrito como:

// $CI \wedge G \equiv (\mathbf{N}\, j : 0 \leq j < \mathtt{i} + 1 : \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n} - 1$.
acção
`++i;`
// $CI \equiv (\mathbf{N}\, j : 0 \leq j < \mathtt{i} : \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n}$.

Determinando a pré-condição mais fraca do progresso que conduz à verificação da condição invariante no seu final,

// $(\mathbf{N}\, j : 0 \leq j < \mathtt{i} + 1 : \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} + 1 < \mathtt{n}$, ou seja,
// $(\mathbf{N}\, j : 0 \leq j < \mathtt{i} + 1 : \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 0 \leq \mathtt{i} < \mathtt{n} - 1$.
`++i;`
// $CI \equiv (\mathbf{N}\, j : 0 \leq j < \mathtt{i} : \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n}$.

Assim sendo, a acção terá de ser escolhida de modo a garantir que

// $CI \wedge G \equiv (\mathbf{N}\, j : 0 \leq j < \mathtt{i} + 1 : \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n} - 1$.
acção
// $(\mathbf{N}\, j : 0 \leq j < \mathtt{i} + 1 : \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 0 \leq \mathtt{i} < \mathtt{n} - 1$.

Mas isso consegue-se sem necessidade de qualquer acção, pois

$$1 \leq \mathtt{i} < \mathtt{n} - 1 \Rightarrow 0 \leq \mathtt{i} < \mathtt{n} - 1$$

O ciclo completo é

// $PC \equiv 2 \leq \mathtt{n} \wedge \mathtt{n} \leq \dim(\mathtt{m}) \wedge 2 \leq (\mathbf{N}\, j : 0 \leq j < \mathtt{n} : \mathtt{m}[j] = \mathtt{k})$.
`int i = ...;`
// $CI \equiv (\mathbf{N}\, j : 0 \leq j < \mathtt{i} : \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n}$.
```
while(m[i] != k)
    ++i
```
// $CO \equiv (\mathbf{N}\, j : 0 \leq j < \mathtt{i} : \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n} \wedge \mathtt{m}[\mathtt{i}] = \mathtt{k}$.

E a inicialização?

A inicialização do ciclo anterior é um problema por si só, com as mesmas pré-condições, mas com uma outra condição objectivo, igual à condição invariante do ciclo já desenvolvido. Isto é, o problema a resolver é:

```
// PC ≡ 2 ≤ n ∧ n ≤ dim(m) ∧ 2 ≤ (N j : 0 ≤ j < n : m[j] = k).
int i = ...;
inic
// CI ≡ (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ 1 ≤ i < n.
```

Pretende-se que `i` seja maior do que o índice da primeira ocorrência de `k` na matriz. A solução para este problema é mais simples se se reforçar a sua condição objectivo (que é a condição invariante do ciclo anterior) um pouco mais. Pode-se impor que `i` seja o índice imediatamente após a primeira ocorrência de `k`:

```
// PC ≡ 2 ≤ n ∧ n ≤ dim(m) ∧ 2 ≤ (N j : 0 ≤ j < n : m[j] = k).
int i = ...;
inic
// (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ m[i − 1] = k ∧ 1 ≤ i < n.
// CI ≡ (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ 1 ≤ i < n.
```

Pode-se simplificar ainda mais o problema se se terminar a inicialização com uma incrementação de `i` e se calcular a pré-condição mais fraca dessa incrementação:

```
// PC ≡ 2 ≤ n ∧ n ≤ dim(m) ∧ 2 ≤ (N j : 0 ≤ j < n : m[j] = k).
int i = ...;
inic
// (N j : 0 ≤ j < i + 1 : m[j] = k) = 1 ∧ m[i] = k ∧ 1 ≤ i + 1 < n, ou seja,
// (N j : 0 ≤ j < i + 1 : m[j] = k) = 1 ∧ m[i] = k ∧ 0 ≤ i < n − 1.
++i;
// (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ m[i − 1] = k ∧ 1 ≤ i < n.
```

Sendo $0 \leq \mathtt{i} < \mathtt{n} - 1$, então

$$(\mathbf{N}\, j : 0 \leq j < \mathtt{i} + 1 : \mathtt{m}[j] = \mathtt{k}) = 1 \wedge \mathtt{m}[\mathtt{i}] = \mathtt{k} \wedge 0 \leq \mathtt{i} < \mathtt{n} - 1$$

é o mesmo que

$$(\mathbf{N}\, j : 0 \leq j < \mathtt{i} : \mathtt{m}[j] = \mathtt{k}) = 0 \wedge \mathtt{m}[\mathtt{i}] = \mathtt{k} \wedge 0 \leq \mathtt{i} < \mathtt{n} - 1$$

ou ainda (ver Apêndice A)

$$(\mathbf{Q}\, j : 0 \leq j < \mathtt{i} : \mathtt{m}[j] \neq \mathtt{k}) \wedge \mathtt{m}[\mathtt{i}] = \mathtt{k} \wedge 0 \leq \mathtt{i} < \mathtt{n} - 1$$

pelo que o código de inicialização se pode escrever:

```
// PC ≡ 2 ≤ n ∧ n ≤ dim(m) ∧ 2 ≤ (N j : 0 ≤ j < n : m[j] = k).
int i = ...;
inic
// CO' ≡ (Q j : 0 ≤ j < i : m[j] ≠ k) ∧ m[i] = k ∧ 0 ≤ i < n − 1.
++i;
// (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ m[i − 1] = k ∧ 1 ≤ i < n.
```

onde CO' representa a condição objectivo do código de inicialização e não a condição objectivo do ciclo já desenvolvido.

A inicialização reduz-se portanto ao problema de encontrar o índice do primeiro elemento com valor `k`. Este índice é forçosamente inferior a $\mathtt{n} - 1$, pois a matriz, pela pré-condição, possui dois elementos com valor `k`. A solução deste problema passa pela construção de um outro ciclo e é relativamente simples, pelo que se apresenta a solução sem mais comentários (dica: factorize-se a condição objectivo):

```
// PC ≡ 2 ≤ n ∧ n ≤ dim(m) ∧ 2 ≤ (N j : 0 ≤ j < n : m[j] = k).
int i = 0;
// CI' ≡ (Q j : 0 ≤ j < i : m[j] ≠ k) ∧ 0 ≤ i < n − 1.
while(m[i] != k)
    ++i;
// CO' ≡ (Q j : 0 ≤ j < i : m[j] ≠ k) ∧ m[i] = k ∧ 0 ≤ i < n − 1.
++i;
// (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ m[i − 1] = k ∧ 1 ≤ i < n.
// CI ≡ (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ 1 ≤ i < n.
```

A função completa é:

```
/** Devolve o índice do segundo elemento com valor k nos primeiros n
    elementos da matriz m.
    @pre PC ≡ 2 ≤ n ∧ n ≤ dim(m) ∧ 2 ≤ (N j : 0 ≤ j < n : m[j] = k).
    @post CO ≡ (N j : 0 ≤ j < índiceDoSegundo : m[j] = k) = 1∧
              1 ≤ índiceDoSegundo < n ∧ m[índiceDoSegundo] = k. */
int índiceDoSegundo(int const m[], int const n, int const k)
{
    int i = 0;

    // CI' ≡ (Q j : 0 ≤ j < i : m[j] ≠ k) ∧ 0 ≤ i < n − 1.
    while(m[i] != k)
        ++i;
    // CO' ≡ (Q j : 0 ≤ j < i : m[j] ≠ k) ∧ m[i] = k ∧ 0 ≤ i < n − 1.

    ++i;

    // CI ≡ (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ 1 ≤ i < n.
    while(m[i] != k)
```

```
        ++i

    return i;
}
```

Pode-se obter código mais fácil de perceber se se começar por desenvolver uma função que devolva o primeiro elemento com valor `k` da matriz. Essa função usa um ciclo que é, na realidade o ciclo de inicialização usado acima:

```
/** Devolve o índice do primeiro elemento com valor k nos primeiros n
    elementos da matriz m.
    @pre PC ≡ 1 ≤ n ∧ n ≤ dim(m) ∧ 1 ≤ (N j : 0 ≤ j < n : m[j] = k).
    @post CO ≡ (Q j : 0 ≤ j < índiceDoPrimeiro : m[j] ≠ k) ∧
              0 ≤ índiceDoPrimeiro < n ∧ m[índiceDoPrimeiro] = k. */
int índiceDoPrimeiro(int const m[], int const n, int const k)
{

    int i = 0;

    // CI ≡ (Q j : 0 ≤ j < i : m[j] ≠ k) ∧ 0 ≤ i < n.
    while(m[i] != k)
        ++i;

    return i;
}

/** Devolve o índice do segundo elemento com valor k nos primeiros n
    elementos da matriz m.
    @pre PC ≡ 2 ≤ n ∧ n ≤ dim(m) ∧ 2 ≤ (N j : 0 ≤ j < n : m[j] = k).
    @post CO ≡ (N j : 0 ≤ j < índiceDoSegundo : m[j] = k) = 1∧
              1 ≤ índiceDoSegundo < n ∧ m[índiceDoSegundo] = k. */
int índiceDoSegundo(int const m[], int const n, int const k)
{

    int i = índiceDoPrimeiro(m, n, k) + 1;

    // CI ≡ (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ 1 ≤ i < n.
    while(m[i] != k)
        ++i

    return i;
}
```

Deixou-se propositadamente para o fim a escrita das instruções de asserção para verificação da pré-condição e da condição objectivo. No caso destas funções, quer a pré-condição quer a condição objectivo envolvem quantificadores. Será que, por isso, as instruções de asserção têm de ser mais fracas do que deveriam, verificando apenas parte do que deveriam? Na realidade não. É possível escrever-se uma função para contar o número de ocorrências de um valor nos primeiros elementos de uma matriz, e usá-la para substituir o quantificador de contagem:

```
/** Devolve o número de ocorrências do valor k nos primeiros n elementos
    da matriz m.
    @pre  PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ ocorrênciasDe = (N j : 0 ≤ j < n : m[j] = k). */
int ocorrênciasDe(int const m[], int const n, int const k)
{
    assert(0 <= n);

    int ocorrências = 0;

    // CI ≡ ocorrênciasDe = (N j : 0 ≤ j < i : m[j] = k) ∧ 0 ≤ i ≤ n.
    for(int i = 0; i != n; ++i)
        if(m[i] == k)
            ++ocorrências;

    return ocorrências;
}

/** Devolve o índice do primeiro elemento com valor k nos primeiros n
    elementos da matriz m.
    @pre  PC ≡ 1 ≤ n ∧ n ≤ dim(m) ∧ 1 ≤ (N j : 0 ≤ j < n : m[j] = k).
    @post CO ≡ (Q j : 0 ≤ j < índiceDoPrimeiro : m[j] ≠ k) ∧
               0 ≤ índiceDoPrimeiro < n ∧ m[índiceDoPrimeiro] = k. */
int índiceDoPrimeiro(int const m[], int const n, int const k)
{
    assert(1 <= n and 1 <= ocorrênciasDe(m, n, k));

    int i = 0;

    // CI ≡ (Q j : 0 ≤ j < i : m[j] ≠ k) ∧ 0 ≤ i < n.
    while(m[i] != k)
        ++i;

    assert(ocorrênciasDe(m, i, k) == 0 and
           0 <= i < n and m[i] = k);

    return i;
}

/** Devolve o índice do segundo elemento com valor k nos primeiros n
    elementos da matriz m.
    @pre  PC ≡ 2 ≤ n ∧ n ≤ dim(m) ∧ 2 ≤ (N j : 0 ≤ j < n : m[j] = k).
```

```
    @post CO ≡ (N j : 0 ≤ j < índiceDoSegundo : m[j] = k) = 1 ∧
              1 ≤ índiceDoSegundo < n ∧ m[índiceDoSegundo] = k. */
int índiceDoSegundo(int const m[], int const n, int const k)
{
    assert(2 <= n and 2 <= ocorrênciasDe(m, n, k));

    int i = índiceDoPrimeiro(m, n, k) + 1;

    // CI ≡ (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ 1 ≤ i < n.
    while(m[i] != k)
        ++i

    assert(ocorrênciasDe(m, i, k) == 1 and
           1 <= i < n and m[i] = k);

    return i;
}
```

Ao se especificar as funções acima, poder-se-ia ter decidido que, caso o número de ocorrências do valor `k` na matriz fosse inferior ao desejado, estas deveriam devolver o valor `n`, pois é sempre um índice inválido (os índices dos primeiros `n` elementos da matriz variam entre 0 e $\mathtt{n}-1$) sendo por isso um valor apropriado para indicar uma condição de erro. Nesse caso as funções poderiam ser escritas como[22]:

```
/** Devolve o índice do primeiro elemento com valor k nos primeiros n
    elementos da matriz m ou n se não existir.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ ((Q j : 0 ≤ j < índiceDoPrimeiro : m[j] ≠ k) ∧
              0 ≤ índiceDoPrimeiro < n ∧ m[índiceDoPrimeiro] = k) ∨
              ((Q j : 0 ≤ j < n : m[j] ≠ k) ∧ índiceDoPrimeiro = n). */
int índiceDoPrimeiro(int const m[], int const n, int const k)
{
    assert(0 <= n);

    int i = 0;
```

[22] As condições objectivo das duas funções não são, em rigor, correctas. O problema é que em

$$0 \leq \mathtt{índiceDoPrimeiro} < \mathtt{n} \wedge \mathtt{m}[\mathtt{índiceDoPrimeiro}] = \mathtt{k}$$

o segundo termo tem valor indefinido para $\mathtt{índiceDoPrimeiro} = \mathtt{n}$. Na realidade dever-se-ia usar uma conjunção especial, que tivesse valor falso desde que o primeiro termo tivesse valor falso *independentemente do segundo termo estar ou não definido*. Pode-se usar um símbolo especial para uma conjunção com estas características, por exemplo $\curlywedge$. De igual forma pode-se definir uma disjunção especial com valor verdadeiro se o primeiro termo for verdadeiro independentemente de o segundo termo estar ou não definido, por exemplo $\curlyvee$. Em [8] usam-se os nomes **cand** e **cor** com o mesmo objectivo. Em [11] chama-se-lhes "and if" e "or else". Estes operadores binários *não são comutativos*, ao contrário do que acontece com a disjunção e a conjunção usuais. Na linguagem C++, curiosamente, só existem as versões não-comutativas destes operadores.

```
    // CI ≡ (Q j : 0 ≤ j < i : m[j] ≠ k) ∧ 0 ≤ i ≤ n.
    while(i != n and m[i] != k)
        ++i;

    assert((ocorrênciasDe(m, i, k) == 0 and
            0 <= i < n and m[i] = k) or
           (ocorrênciasDe(m, n, k) == 0 and i == n));

    return i;
}

/** Devolve o índice do segundo elemento com valor k nos primeiros n
    elementos da matriz m ou n se não existir.
    @pre PC ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ ((N j : 0 ≤ j < índiceDoSegundo : m[j] = k) = 1∧
             1 ≤ índiceDoSegundo < n ∧ m[índiceDoSegundo] = k) ∨
             ((N j : 0 ≤ j < n : m[j] = k) < 2 ∧ índiceDoSegundo = n). */
int índiceDoSegundo(int const m[], int const n, int const k)
{
    assert(0 <= n);

    int i = índiceDoPrimeiro(m, n, k);

    if(i == n)
        return n;

    ++i;

    // CI ≡ (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ 1 ≤ i ≤ n.
    while(i != n and m[i] != k)
        ++i

    assert((ocorrênciasDe(m, i, k) == 1 and
            1 <= i < n and m[i] = k) or
           (ocorrênciasDe(m, n, k) < 2 and i == n));

    return i;
}
```

5.3.8 Segundo item de um vector com um dado valor

O desenvolvimento no caso dos vectores é semelhante ao usado para as matrizes. As funções resultantes desse desenvolvimento são

```
/** Devolve o número de ocorrências do valor k nos primeiros n elementos
```

```
    do vector v.
    @pre PC ≡ 0 ≤ n ∧ n ≤ v.size().
    @post CO ≡ ocorrênciasDe = (N j : 0 ≤ j < v.size() : v[j] = k). */
vector<int>::size_type ocorrênciasDe(vector<int> const& v,
                                     int const n, int const k)
{
    assert(0 <= n and n <= v.size());

    vector<int>::size_type ocorrências = 0;

    // CI ≡ ocorrênciasDe = (N j : 0 ≤ j < i : v[j] = k) ∧ 0 ≤ i ≤ v.size().
    for(vector<int>::size_type i = 0; i != v.size(); ++i)
        if(v[i] == k)
            ++ocorrências;

    return ocorrências;
}

/** Devolve o índice do primeiro item com valor k do vector v ou
    v.size() se não existir.
    @pre PC ≡ V.
    @post CO ≡ ((Q j : 0 ≤ j < índiceDoPrimeiro : v[j] ≠ k) ∧
              0 ≤ índiceDoPrimeiro < v.size() ∧ v[índiceDoPrimeiro] = k) ∨
              ((Q j : 0 ≤ j < v.size() : v[j] ≠ k) ∧ índiceDoPrimeiro = v.size()). */
vector<int>::size_type índiceDoPrimeiro(vector<int> const& v,
                                        int const k)
{
    vector<int>::size_type i = 0;

    // CI ≡ (Q j : 0 ≤ j < i : v[j] ≠ k) ∧ 0 ≤ i ≤ v.size().
    while(i != v.size() and v[i] != k)
        ++i;

    assert((ocorrênciasDe(v, i, k) == 0 and
            0 <= i < v.size() and v[i] = k) or
           (ocorrênciasDe(v, v.size(), k) == 0 and
            i == v.size()));

    return i;
}

/** Devolve o índice do segundo elemento com valor k do vector v ou
    v.size() se não existir.
    @pre PC ≡ V.
    @post CO ≡ ((N j : 0 ≤ j < índiceDoSegundo : v[j] = k) = 1∧
              1 ≤ índiceDoSegundo < v.size() ∧ v[índiceDoSegundo] = k) ∨
```

```
              ((N j : 0 ≤ j < v.size() : m[j] = k) < 2 ∧ índiceDoSegundo = v.size()). */
vector<int>::size_type índiceDoSegundo(vector<int> const& v,
                                       int const k)
{
    int i = índiceDoPrimeiro(m, k);

    if(i == v.size()
        return v.size();

    ++i;

    // CI ≡ (N j : 0 ≤ j < i : v[j] = k) = 1 ∧ 1 ≤ i ≤ v.size().
    while(i != v.size() and v[i] != k)
        ++i

    assert((ocorrênciasDe(v, i, k) == 1 and
            1 <= i < v.size() and v[i] = k) or
           (ocorrênciasDe(v, v.size(), k) < 2 and
            i == v.size()));

    return i;
}
```

5.4 Cadeias de caracteres

A maior parte da comunicação entre humanos faz-se usando a palavra, falada ou escrita. É natural, portanto, que o processamento de palavras escritas seja também parte importante da maior parte dos programas: a comunicação entre computador e humano suporta-se ainda fortemente na palavra escrita. Dos tipos básicos do C++ faz parte o tipo `char`, que é a base para todo este processamento. Falta, no entanto, forma de representar *cadeias ou sequências de caracteres*. O C++ herdou da linguagem C uma forma de representação de cadeias de caracteres peculiar. São as chamadas cadeias de caracteres clássicas, representadas por matrizes de caracteres. As cadeias de caracteres *clássicas* são matrizes de caracteres em que o fim da cadeia é assinalado usando um caractere especial: o chamado caractere nulo, de código 0, que não representa nenhum símbolo em nenhuma das possíveis tabelas de codificação de caracteres (ver exemplo no Apêndice G).

As cadeias de caracteres clássicas são talvez eficientes, mas são também seguramente inflexíveis e desagradáveis de utilizar, uma vez que sofrem de todos os inconvenientes das matrizes clássicas. É claro que uma possibilidade de representação alternativa para as cadeias de caracteres seria recorrendo ao tipo genérico `vector`: a classe `vector<char>` permite de facto representar sequências de caracteres arbitrárias. No entanto, a biblioteca padrão do C++ fornece uma classe, de nome `string`, que tem todas as capacidades de `vector<char>` mas acrescenta uma quantidade considerável de operações especializadas para lidar com cadeias de caracteres.

Como se fez mais atrás neste capítulo acerca das matrizes e dos vectores, começar-se-á por apresentar brevemente a representação mais primitiva de cadeias de caracteres, passando-se depois a uma descrição mais ou menos exaustiva da classe `string`.

5.4.1 Cadeias de caracteres clássicas

A representação mais primitiva de cadeias de caracteres em C++ usa matrizes de caracteres em que o final da cadeia é marcado através de um caractere especial, de código zero, e que pode ser explicitado através da sequência de escape `\0`. Por exemplo,

```
char nome[] = {'Z', 'a', 'c', 'a', 'r', 'i', 'a', 's', '\0'};
```

define uma cadeia de caracteres representando o nome "Zacarias". É importante perceber que uma matriz de caracteres só é uma cadeia de caracteres clássica se possuir o terminador `'\0'`. Assim,

```
char nome_matriz[] = {'Z', 'a', 'c', 'a', 'r', 'i', 'a', 's'};
```

é uma matriz de caracteres mas *não é* uma cadeia de caracteres clássica.

Para simplificar a inicialização de cadeias de caracteres, a linguagem permite colocar a sequência de caracteres do inicializador entre aspas e omitir o terminador, que é colocado automaticamente. Por exemplo:

```
char dia[] = "Sábado";
```

define uma cadeia com seis caracteres contendo "Sábado" e representada por uma matriz de dimensão sete: os seis caracteres da palavra "Sábado" e o caractere terminador.

Uma cadeia de caracteres não necessita de ocupar toda a matriz que lhe serve de suporte. Por exemplo,

```
char const janeiro[12] = "Janeiro";
```

define uma matriz de 12 caracteres constantes contendo uma cadeia de caracteres de comprimento sete, que ocupa exactamente oito elementos da matriz: os primeiros sete com os caracteres da cadeia e o oitavo com o terminador.

As cadeias de caracteres podem ser inseridas em canais, o que provoca a inserção de cada um dos seus caracteres (com excepção do terminador). Assim, dadas as definições acima, o código

```
cout << "O nome é " << nome << '.' << endl;
cout << "Hoje é " << dia << '.' << endl;
```

faz surgir no ecrã

```
O nome é Zacarias.
Hoje é Sábado.
```

Se se olhar atentamente o código acima, verificar-se-á que existe uma forma alternativa escrever cadeias de caracteres num programa: as chamadas cadeias de caracteres literais, que correspondem a uma sequência de caracteres envolvida em aspas, por exemplo `"O nome é"`. As cadeias de caracteres literais são uma peculiaridade da linguagem. Uma cadeia de caracteres literal:

1. É uma matriz de caracteres, como qualquer cadeia de caracteres clássica.
2. Os seus caracteres são constantes. O tipo de uma cadeia de caracteres literal é, por isso, `char const [`*`dimensão`*`]`, onde *`dimensão`* é o número de caracteres somado de um (para haver espaço para o terminador).
3. Não tem nome, ao contrário das variáveis usuais.
4. O seu âmbito está limitado à expressão em que é usada.
5. Tem permanência estática, existindo durante todo o programa, como se fosse global.

Para o demonstrar comece-se por um exemplo simples. É possível percorrer uma cadeia de caracteres usando um ciclo. Por exemplo:

```
char nome[] = "Zacarias";

for(int i = 0; nome[i] != '\0'; ++i)
    cout << nome[i];
cout << endl;
```

Este troço de código tem exactamente o mesmo resultado que

```
char nome[] = "Zacarias";

cout << nome << endl;
```

A sua particularidade é a guarda usada para o ciclo. É que, como a dimensão da cadeia não é conhecida à partida, uma vez que pode ocupar apenas uma parte da matriz, a forma mais segura de a percorrer é usar o terminador para verificar quando termina.

As mesmas ideias podem ser usadas para percorrer uma cadeia de caracteres literal, o que demonstra claramente que estas são também matrizes:

```
int comprimento = 0;
while("Isto é um teste."[comprimento] != '\0')
    ++comprimento;

cout << "O comprimento é " << comprimento << '.' << endl;
```

Este troço de programa escreve no ecrã o comprimento da cadeia "`Isto é um teste.`", i.e., escreve 16.

Uma cadeia de caracteres literal não pode ser dividida em várias linhas. Por exemplo, o código seguinte é ilegal:

```
cout << "Isto é uma frase comprida que levou à necessidade de a
dividir em três linhas.  Só que, infelizmente, de uma
forma ilegal." << endl;
```

No entanto, como cadeias de caracteres adjacentes no código são consideradas como uma e uma só cadeia de caracteres literal, pode-se resolver o problema acima de uma forma perfeitamente legal e legível:

```
cout << "Isto é uma frase comprida que levou à necessidade de a "
        "dividir em três linhas.  Desta vez de uma "
        "forma legal." << endl;
```

5.4.2 A classe `string`

As cadeias de caracteres clássicas devem ser utilizadas apenas onde indispensável: para especificar cadeias de caracteres literais, e.g., para compor frases a inserir no canal de saída `cout`, como tem vindo a ser feito até aqui. Para representar cadeias de caracteres deve-se usar a classe `string`, definida no ficheiro de interface com o mesmo nome. Ou seja, para usar esta classe deve-se colocar no topo dos programas a directiva

```
#include <string>
```

De este ponto em diante usar-se-ão as expressões "cadeia de caracteres" e "cadeia" para classificar qualquer variável ou constante do tipo `string`.

Como todas as descrições de ferramentas da biblioteca padrão feitas neste texto, a descrição da classe `string` que se segue não pretende de modo algum ser exaustiva. Para tirar partido de todas as possibilidades das ferramentas da biblioteca padrão é indispensável recorrer, por exemplo, a [12].

As cadeias de caracteres suportam todas as operações dos vectores descritas na Secção 5.2, com excepção das operações `push_back()`, `pop_back()`, `front()` e `back()`, pelo que serão apresentadas apenas as operações específicas das cadeias de caracteres.

Definição (construção) e inicialização

Para definir e inicializar uma cadeia de caracteres pode-se usar qualquer das formas seguintes:

```
string vazia;                               // cadeia vazia, sem caracteres.
string nome = "Zacarias Zebedeu Zagalo";    // a partir de cadeia clássica
                                            // (literal, neste caso).
string mesmo_nome = nome;                   // cópia a partir de cadeia.
string apelidos(nome, 9);                   // cópia a partir da posição 9.
string nome_do_meio(nome, 9, 7);            // cópia de 7 caracteres a
                                            // partir da posição 9.
string vinte_aa(20, 'a');                   // dimensão inicial 20,
                                            // tudo com 'a'.
```

Atribuição

Podem-se fazer atribuições entre cadeias de caracteres. Também se pode atribuir uma cadeia de caracteres clássica ou mesmo um simples caractere a uma cadeia de caracteres:

```
string nome1 = "Xisto Ximenes";
string nome2;
string nome3;

nome2 = nome1;       // atribuição entre cadeias.
nome1 = "Ana Anes"; // atribuição de cadeia clássica.
nome3 = 'X';         // atribuição de um só caractere (literal, neste caso).
```

Podem-se fazer atribuições mais complexas usando a operação `assign()`:

```
string nome = "Zacarias Zebedeu Zagalo";
string apelidos;
string nome_do_meio;
string vinte_aa;

apelidos.assign(nome, 9, string::npos); // só final de cadeia.
nome_do_meio.assign(nome, 9, 7);        // só parte de cadeia.
vinte_aa.assign(20, 'a');               // caractere 'a' repetido 20
                                        // vezes.
```

A constante `string::npos` é maior do que o comprimento de qualquer cadeia possível. Quando usada num local onde se deveria indicar um número de caracteres significa "todos os restantes". Daí que a variável apelidos passe a conter "Zebedeu Zagalo".

Dimensão e capacidade

As cadeias de caracteres suportam todas as operações relativas a dimensões e capacidade dos vectores, adicionadas das operações `length()` e `erase()`:

size() Devolve a dimensão actual da cadeia.

length() Sinónimo de `size()`, porque é mais usual falar-se em comprimento que em dimensão de uma cadeia.

max_size() Devolve a maior dimensão possível de uma cadeia.

resize(*n*, *v*) Altera a dimensão da cadeia para *n*. Tal como no caso dos vectores, o segundo argumento, o valor dos possíveis novos caracteres, é opcional (se não for especificado os novos caracteres serão o caractere nulo).

reserve(*n*) Reserva espaço para *n* caracteres na cadeia, de modo a evitar a ineficiência associada a aumentos sucessivos.

capacity() Devolve a capacidade actual da cadeia, que é a dimensão até à qual a cadeia pode crescer sem ter de requerer memória ao sistema operativo.

clear() Esvazia a cadeia.

erase() Sinónimo de `clear()`, porque é mais usual dizer-se apagar do que limpar uma cadeia.

empty() Indica se a cadeia está vazia.

Indexação

Os modos de indexação são equivalentes aos dos vectores. A indexação usando o operador de indexação `[]` é insegura, no sentido em que a validade dos índices não é verificada. Para realizar uma indexação segura utilizar a operação `at()`.

Acrescento

Existem várias formas elementares de acrescentar cadeias de caracteres:

```
string nome = "Zacarias";
string nome_do_meio = "Zebedeu";

nome += ' ';           // um só caractere.
nome += nome_do_meio; // uma cadeia.
nome += " Zagalo";     // uma cadeia clássica (literal, neste caso).
```

Para acrescentos mais complexos existe a operação `append()`:

```
string vinte_aa(10, 'a'); // só 10...
string contagem = "um dois ";
string dito = "não há duas sem três";

vinte_aa.append(10, 'a');      // caracteres repetidos (10 novos 'a').
contagem.append(dito, 17, 4); // quatro caracteres da posição 17 de dito.
```

Inserção

De igual forma existem várias versões da operação `insert()` para inserir material em lugares arbitrários de uma cadeia:

```
string contagem = "E vão  ,  e !";
string dito = "não há uma sem duas";
string duas = "duas";

contagem.insert(12, "três");     // cadeia clássica na posição 12.
contagem.insert(9, duas);        // cadeia na posição 9.
contagem.insert(7, dito, 7, 3);  // parte da cadeia dito na posição 7.
contagem.insert(6, 3, '.');      // caracteres repetidos na posição 6.
// contagem fica com "E vão ... uma, duas e três!".
```

O material é inserido *antes* da posição indicada.

Substituição

É possível, recorrendo às várias operações `replace()`, substituir pedaços de uma cadeia por outras sequências de caractere. Os argumentos são os mesmos que para as operações `insert()`, mas, para além de se indicar onde começa a zona a substituir, indica-se também quantos caracteres substituir. Por exemplo:

```
string texto = "O nome é $nome.";

texto.replace(9, 5, "Zacarias"); // substitui "$nome" por "Zacarias".
```

Procuras

Há muitas formas de procurar texto dentro de uma cadeia de caracteres. A primeira forma procura uma sequência de caracteres do início para o fim (operação `find()`) ou do fim para o início (operação `rfind()`) da cadeia. Estas operações têm como argumentos, em primeiro lugar, a sequência a procurar e opcionalmente, em segundo lugar, a posição a partir da qual iniciar a procura (se não se especificar este argumento, a procura começa no início ou no fim da cadeia, consoante a direcção de procura). A sequência a procurar pode ser uma cadeia de caracteres, uma cadeia de caracteres clássica ou um simples caractere. O valor devolvido é a posição onde a sequência foi encontrada ou, caso não o tenha sido, o valor `string::npos`. Por exemplo:

```
string texto = "O nome é $nome.";

// Podia também ser texto.find('$'):
string::size_type posição = texto.find("$nome");
```

```
if(posição != string::npos)
    // Substitui "$nome" por "Zacarias":
    texto.replace(posição, 5, "Zacarias");
```

A segunda forma serve para procurar caracteres de uma dado conjunto dentro de uma cadeia de caracteres. A procura é feita do início para o fim (operação `find_first_of()`) ou do fim para o início (operação `find_last_of()`) da cadeia. O conjunto de caracteres pode ser dado na forma de uma cadeia de caracteres, de uma cadeia de caracteres clássica ou de um simples caractere. Os argumentos são mais uma vez o conjunto de caracteres a procurar e a posição inicial de procura, que é opcional. O valor devolvido é mais uma vez a posição encontrada ou `string::npos` se nada foi encontrado. Por exemplo:

```
string texto = "O nome é $nome.";

// Podia também ser texto.find_first_of('$'):
string::size_type posição = texto.find_first_of("$%#", 3);
// Começa-se em 3 só para clarificar onde se especifica a posição inicial de procura.

if(posição != string::npos)
    // Substitui "$nome" por "Zacarias":
    texto.replace(posição, 5, "Zacarias");
```

Existem ainda as operações `find_first_not_of()` e `find_last_not_of()` se se quiser procurar caracteres que *não pertençam* a um dado conjunto de caracteres.

Concatenação

É possível concatenar cadeias usando o operador `+`. Por exemplo:

```
string um = "um";
string contagem = um + " dois" + ' ' + 't' + "rês";
```

Comparação

A comparação entre cadeias é feita usando os operadores de igualdade e relacionais. A ordem considerada para as cadeias é lexicográfica (ver Secção 5.2.12) e depende da ordenação dos caracteres na tabela de codificação usada, que por sua vez é baseada no respectivo código. Assim, consultando o Apêndice G, que contém a tabela de codificação Latin-1, usada por enquanto em Portugal, conclui-se que

$$
\begin{aligned}
\texttt{'A'} &< \texttt{'a'} \\
\texttt{'a'} &< \texttt{'á'} \\
\texttt{'Z'} &< \texttt{'a'} \\
\texttt{'z'} &< \texttt{'a'}
\end{aligned}
$$

onde infelizmente a ordenação relativa de maiúsculas e minúsculas e, sobretudo, dos caracteres acentuados, não respeita as regras habituais do português. Aqui entra-se pela terceira vez no problema das variações regionais de critérios[23]. Estes problemas resolvem-se recorrendo ao conceito de `locales`, que saem fora do âmbito deste texto. Assim, representando as cadeias por sequências de caracteres entre aspas:

- "leva" < "levada"
- "levada" < "levadiça"
- "órfão" > "orfeão" (infelizmente...)
- "fama" < "fome"
- etc.

Formas mais complexas de comparação recorrem à operação `compare()`, que devolve um valor positivo se a cadeia for maior que a passada como argumento, zero se for igual, e um valor negativo se for menor. Por exemplo:

```
string cadeia1 = "fama";
string cadeia2 = "fome";

int resultado = cadeia1.compare(cadeia2);

if(resultado == 0)
    cout << "São iguais!" << endl;
else if(resultado < 0)
    cout << "A primeira é a menor!" << endl;
else
    cout << "A segunda é a menor!" << endl;
```

Este código, para as cadeias indicadas, escreve:

```
A primeira é a menor!
```

[23]As duas primeiras não foram referidas explicitamente e dizem respeito ao formato dos valores decimais e de valores booleanos em operações de inserção e extracção. Seria desejável, por exemplo, que os números decimais aparecessem ou fossem lidos com vírgula em vez de ponto, e os valores booleanos por extenso na forma `verdadeiro` e `falso`, em vez de `true` e `false`.

Capítulo 6

Tipos enumerados

Além dos tipos de dados pré-definidos no C++, os chamados tipos básicos, podem-se criar tipos de dados adicionais. Essa é, aliás, uma das tarefas fundamentais da programação centrada nos dados. Para já, abordar-se-ão extensões mais simples aos tipos básicos: os tipos enumerados. No próximo capítulo falar-se-á de tipos de primeira categoria, usando classes C++

Uma variável de um tipo enumerado pode conter um número limitado de valores, que se enumeram na definição do tipo[1]. Por exemplo,

```
enum DiaDaSemana {
    segunda_feira,
    terça_feira,
    quarta_feira,
    quinta_feira,
    sexta_feira,
    sábado,
    domingo
};
```

define um tipo enumerado com sete valores possíveis, um para cada dia da semana. Convencionalmente no nome dos novos tipos todas as palavras começam por uma letra maiúscula e não se usa qualquer caractere para as separar.

O novo tipo utiliza-se como habitualmente. Pode-se, por exemplo, definir variáveis do novo tipo[2]:

```
DiaDaSemana dia = quarta_feira;
```

Pode-se atribuir atribuir à nova variável `dia` qualquer dos valores listados na definição do tipo enumerado `DiaDaSemana`:

[1]Esta afirmação é uma pequena mentira "piedosa". Na realidade os enumerados podem conter valores que não correspondem aos especificados na sua definição [12, página 77].

[2]Ao contrário do que se passa com as classes, os tipos enumerados sofrem do mesmo problema que os tipos básicos: variáveis automáticas de tipos enumerados sem inicialização explícita contêm lixo!

```
dia = terça_feira;
```

Cada um dos valores associados ao tipo `DiaDaSemana` (viz. `segunda_feira`, ..., `domingo`) é utilizado como se fosse um valor literal para esse tipo, tal como `10` é um valor literal do tipo `int` ou `'a'` é um valor literal do tipo `char`. Convencionalmente nestes valores literais as palavras estão em minúsculas e usa-se um sublinhado (_) para as separar[3].

Como se trata de um tipo definido pelo programador, não é possível, sem mais esforço, ler valores desse tipo do teclado ou escrevê-los no ecrã usando os métodos habituais (viz. os operadores de extracção e inserção em canais: >> e <<). Mais tarde ver-se-á como se pode "ensinar" o computador a extrair e inserir valores de tipos definidos pelo utilizador de, e em, canais.

Na maioria dos casos os tipos enumerados são usados para tornar mais claro o significado dos valores atribuídos a uma variável. Por exemplo, `segunda_feira` tem claramente mais significado que `0`. Na realidade, os valores de tipos enumerados são representados como inteiros atribuídos sucessivamente a partir de zero. Assim, `segunda_feira` tem representação interna 0, `terça_feira` tem representação 1, etc. De facto, se se tentar imprimir `segunda_feira` o resultado será surgir `0` no ecrã, que é a sua representação na forma de um inteiro. É possível associar inteiros arbitrários a cada um dos valores de uma enumeração, pelo que podem existir representações idênticas para valores com nome diferentes:

```
enum DiaDaSemana { // agora com nomes alternativos...
    primeiro = 0,  // inicialização redundante: o primeiro valor é sempre 0.
    segunda = primeiro,
    segunda_feira = segunda,
    terça,
    terça_feira = terça,
    quarta,
    quarta_feira = quarta,
    quinta,
    quinta_feira = quinta,
    sexta,
    sexta_feira = sexta,
    sábado,
    domingo,
    último = domingo,
};
```

Se um operando de um tipo enumerado ocorrer numa expressão, será geralmente convertido num inteiro. Essa conversão também se pode explicitar, escrevendo `int(segunda_feira)`, por exemplo. As conversões opostas também são possíveis, usando-se `DiaDaSemana(2)`, por exemplo, para obter `quarta_feira`. Na próxima secção ver-se-á como redefinir os operadores existentes na linguagem de modo a operarem sobre tipos enumerados sem surpresas

[3] Existe uma convenção alternativa em que os valores literais de tipos enumerados e os nomes das constantes se escrevem usando apenas maiúsculas com as palavras separadas por um sublinhado (e.g., `SEGUNDA_FEIRA`). Desaconselha-se o uso dessa convenção, pois confunde-se com a convenção de dar esse tipo de nomes a macros, que serão vistas no Capítulo 9.

desagradáveis para o programador (pense-se no que deve acontecer quando se incrementa uma variável do tipo `DiaDaSemana` que contém o valor `domingo`).

6.1 Sobrecarga de operadores

Da mesma forma que se podem sobrecarregar nomes de funções, i.e., dar o mesmo nome a funções que, tendo semanticamente o mesmo significado, operam com argumentos de tipos diferentes (ou em diferente número), também é possível sobrecarregar o significado dos operadores usuais do C++ de modo a que tenham um significado especial quando aplicados a tipos definidos pelo programador. Se se pretender, por exemplo, sobrecarregar o operador `++` prefixo (incrementação prefixa) para funcionar com o tipo `DiaDaSemana` definido acima, pode-se definir uma rotina[4] com uma sintaxe especial:

```
DiaDaSemana operator ++ (DiaDaSemana& dia)
{
    if(dia == último)
        dia = primeiro;
    else
        dia = DiaDaSemana(int(dia) + 1);

    return dia;
}
```

ou simplesmente

```
DiaDaSemana operator ++ (DiaDaSemana& dia)
{
    if(dia == último)
        return dia = primeiro;
    else
        return dia = DiaDaSemana(int(dia) + 1);
}
```

A única diferença relativamente à sintaxe habitual da definição de funções e procedimentos é que se substitui o habitual nome do procedimento pela palavra-chave `operator` seguida do operador a sobrecarregar[5]. Não é possível sobrecarregar todos os operadores, ver Tabela 6.1.

O operador foi construído de modo a que a incrementação de uma variável do tipo `DiaDaSemana` conduza sempre ao dia da semana subsequente. Utilizou-se `primeiro` e `último` e não `segunda_feira`

[4] Este operador não é, em rigor, nem um procedimento nem uma função. Não é um procedimento porque não se limita a incrementar: devolve o valor do argumento depois de incrementado. Não é uma função porque não se limita a devolver um valor: altera, incrementando, o seu argumento. É por esta razão que o operador `++` tem efeitos laterais, podendo a sua utilização descuidada conduzir a expressões mal comportadas, com os perigos que daí advêm (Secção 2.7.8).

[5] Em todo o rigor o operador de incrementação prefixa deveria devolver uma referência para um `DiaDaSemana`. Veja-se a Secção 7.7.1.

Tabela 6.1: Operadores que é possível sobrecarregar.

+	-	*	/	%	xor	bitand
bitor	compl	not	=	<	>	+=
-=	*=	/=	%=	^=	&=	|=
<<	>>	<<=	>>=	==	!=	<=
>=	and	or	++	--	->*	,
->	[]	()	new	new[]	delete	delete[]

e `domingo`, pois dessa forma pode-se mais tarde decidir que a semana começa ao `Domingo` sem ter de alterar o procedimento acima, alterando apenas a definição da enumeração.

Este tipo de sobrecarga, como é óbvio, só pode ser feito para novos tipos definidos pelo programador. Esta restrição evita redefinições abusivas do significado do operador `+` quando aplicado a tipos básicos como o `int`, por exemplo, que poderiam ter resultados trágicos.

Capítulo 7

Tipos abstractos de dados e classes C++

In this connection it might be worthwhile to point out that
the purpose of abstraction is not to be vague, but to create
a new semantic level in which one can be absolutely precise.

Edsger W Dijkstra, The Humble Programmer,
Communications of the ACM, 15(10), 1972

Quando se fala de uma linguagem de programação, não se fala apenas da linguagem em si, com o seu léxico, sintaxe, gramática e semântica. Fala-se também de um conjunto de ferramentas acessíveis ao programador que, não fazendo parte da linguagem propriamente dita, estão acessíveis em qualquer ambiente de desenvolvimento de programas. Ao conjunto dessas ferramentas adicionais que se encontra em todos os ambientes de desenvolvimento chama-se biblioteca padrão (*standard library*). Da biblioteca padrão do C++ fazem parte, por exemplo, os canais `cin` e `cout`, que permitem leituras do teclado e escritas para o ecrã, o tipo `string` e o tipo genérico `vector`. Em rigor, portanto, o programador tem à sua disposição não a linguagem em si, mas a linguagem equipada com a biblioteca padrão. Para o programador, no entanto, tudo funciona como se a linguagem em si incluísse essas ferramentas. Isto é, para o programador em C++ o que está acessível não é o C++, mas um "C++ ++" de que fazem parte todas as ferramentas da biblioteca padrão.

A tarefa de um programador é resolver problemas usando (pelo menos) um computador. Fá-lo através da escrita de programas numa linguagem de programação dada. Depois de especificado o problema com exactidão, o programador inteligente começa por procurar, na linguagem básica, na biblioteca padrão e noutras quaisquer bibliotecas disponíveis, ferramentas que resolvam o problema na totalidade ou pelo menos parcialmente: esta procura evita as perdas de tempo associadas ao reinventar da roda infelizmente ainda tão em voga[1]. Se não existirem

[1]Por outro lado, é importante notar que se pede muitas vezes ao *estudante* que reinvente a roda. Fazê-lo é parte fundamental do treino na resolução de problemas concretos. Convém, portanto, que o estudante se disponha a essa tarefa que fora do contexto da aprendizagem é inútil. Mas convém também que não se deixe viciar na resolução por si próprio de todos os pequenos problemas que já foram resolvidos milhares de vezes. É importante saber fazer um equilíbrio entre a curiosidade intelectual de resolver esses problemas e o pragmatismo de procurar um solução já pronta. Durante a vida académica, a balança deve pender fortemente no sentido da curiosidade intelectual. Finda a vida académica, o equilíbrio deve pender mais para o pragmatismo.

ferramentas disponíveis, então há que construí-las. Ao fazê-lo, o programador está a expandir mais uma vez a linguagem disponível, que passa a dispor de ferramentas adicionais (digamos que "incrementa" de novo a linguagem para "C++ ++ ++").

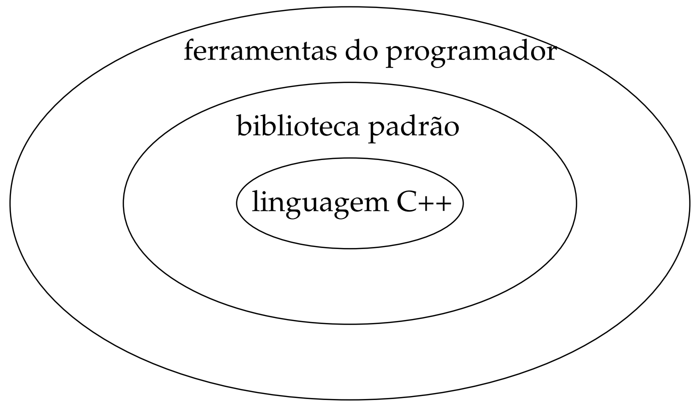


Figura 7.1: A biblioteca padrão do C++ e as ferramentas do programador como extensões à funcionalidade básica da linguagem C++.

Há essencialmente duas formas distintas de construir ferramentas adicionais para uma linguagem. A primeira passa por equipar a linguagem com operações adicionais, na forma de rotinas, mas usando os tipos existentes (`int`, `char`, `bool`, `double`, matrizes, etc.): é a chamada programação procedimental. A segunda passa por adicionar tipos à linguagem e engloba a programação centrada nos dados (ou programação baseada em objectos). Para que os novos tipos criados tenham algum interesse, é fundamental que tenham operações próprias, que têm de ser concretizadas pelo programador. Assim, a segunda forma de expandir a linguagem passa necessariamente pela primeira.

Neste capítulo ver-se-á a forma por excelência de acrescentar tipos, e respectivas operações, à linguagem. No capítulo anterior abordaram-se as simples e limitadas enumerações, neste ver-se-ão os tipos abstractos de dados, peça fundamental da programação centrada nos dados. A partir deste ponto, portanto, o ênfase será posto na construção de novos tipos. Neste capítulo construir-se-ão novos tipos relativamente simples e independentes uns dos outros. Quando se iniciar o estudo da programação orientada por objectos, em capítulos posteriores, ver-se-á como se podem desenhar classes e hierarquias de classes e quais as suas aplicações na resolução de problemas de maior escala.

7.1 De novo a soma de fracções

Na Secção 3.2.20 desenvolveu-se um pequeno programa para ler duas fracções do teclado e mostrar a sua soma. Neste capítulo desenvolver-se-á esse programa até construir uma pequena calculadora. Durante esse processo aproveitar-se-á para introduzir uma quantidade considerável de conceitos novos.

O programa apresentado na Secção 3.2.20 pode ser melhorado. Assim, apresenta-se abaixo uma versão melhorada nos seguintes aspectos:

- A noção de máximo divisor comum é facilmente generalizável a inteiros negativos ou nulos. O único caso complicado é o de $\mathrm{mdc}(0,0)$. Como é óbvio, todos os inteiros positivos são divisores comuns de zero, pelo que não existe este máximo divisor comum. No entanto, é de toda a conveniência estender a definição do máximo divisor comum, arbitrando o valor 1 como resultado de $\mathrm{mdc}(0,0)$. Ou seja, por definição $\mathrm{mdc}(0,0) = 1$. Assim, a função `mdc()` foi flexibilizada, tendo-se enfraquecido a respectiva pré-condição de modo a ser aceitar argumentos arbitrários. A utilidade da cobertura do caso $\mathrm{mdc}(0,0)$ será vista mais tarde.
- O enfraquecimento da pré-condição da função `mdc` permitiu enfraquecer também todas as restantes pré-condições, tornando o programa capaz de lidar com fracções com termos negativos.
- O ciclo usado na função `mdc()` foi optimizado, passando a usar-se um ciclo pouco ortodoxo, com duas possíveis saídas. Fica como exercício para o leitor demonstrar o seu correcto funcionamento e verificar a sua eficiência.
- Foram acrescentadas rotinas para a leitura e cálculo da soma de duas fracções.
- Nas rotinas lidando com fracções alterou-se o nome das variáveis para explicitar melhor aquilo que representam (e.g., `numerador` em vez de `n`).
- Para evitar código demasiado extenso para uma versão impressa deste texto, cada rotina é definida antes das rotinas que dela fazem uso, não se fazendo uma distinção clara entre declaração e definição. Mais tarde ser verá que esta não é forçosamente uma boa solução.
- Uma vez que a pré-condição e a condição objectivo são facilmente identificáveis pela sua localização na documentação das rotinas, após "@pre" e "@post" respectivamente, abandonou-se o hábito de nomear essas condições PC e CO.
- Protegeu-se de erros a leitura das fracções (ver Secção 7.14).

```
#include <iostream>
#include <cassert>

using namespace std;

/** Devolve o máximo divisor comum dos inteiros passados como argumento.
    @pre m = m ∧ n = n.
    @post mdc = { mdc(m, n)   m ≠ 0 ∨ n ≠ 0
                {     1       m = 0 ∧ n = 0 . */
int mdc(int m, int n)
{
    if(m == 0 and n == 0)
        return 1;

    if(m < 0)
        m = -m;
    if(n < 0)
```

```
        n = -n;

    while(true) {
        if(m == 0)
            return n;
        n = n % m;
        if(n == 0)
            return m;
        m = m % n;
    }
}
```

/** Reduz a fracção recebida como argumento.
@pre $\texttt{denominador} \neq 0 \land \texttt{denominador} = d \land \texttt{numerador} = n$.
@post $\texttt{denominador} \neq 0 \land \mathrm{mdc}(\texttt{numerador}, \texttt{denominador}) = 1 \land \frac{\texttt{numerador}}{\texttt{denominador}} = \frac{n}{d}$. */

```
void reduzFracção(int& numerador, int& denominador)
{
    assert(denominador != 0);

    int máximo_divisor_comum = mdc(numerador, denominador);

    numerador /= máximo_divisor_comum;
    denominador /= máximo_divisor_comum;

    assert(denominador != 0 and mdc(numerador, denominador) == 1);
}
```

/** Lê do teclado uma fracção, na forma de dois inteiros sucessivos.
@pre $\texttt{numerador} = n \land \texttt{denominador} = d$.
@post Se `cin` e `cin` tem dois inteiros n' e d' disponíveis para leitura, com $d' \neq 0$, então
$0 < denominador \land \mathrm{mdc}(\texttt{numerador}, \texttt{denominador}) = 1 \land \frac{\texttt{numerador}}{\texttt{denominador}} = \frac{n'}{d'} \land \texttt{cin}$,
senão
$\texttt{numerador} = n \land \texttt{denominador} = n \land \neg\texttt{cin}$. */

```
void lêFracção(int& numerador, int& denominador)
{
    int n, d;

    if(cin >> n >> d)
        if(d == 0)
            cin.setstate(ios_base::failbit);
        else {
            numerador = d < 0 ? -n : n;
            denominador = d < 0 ? -d : d;
```

```
            reduzFracção(numerador, denominador);

            assert(0 < denominador and mdc(numerador, denominador) == 1 and
                   numerador * d == n * denominador and cin);

            return;
        }

    assert(not cin);
}

/** Soma duas fracções.
    @pre denominador1 ≠ 0 ∧ denominador2 ≠ 0.
    @post numerador/denominador = numerador1/denominador1 + numerador2/denominador2 ∧
          denominador ≠ 0 ∧ mdc(numerador, denominador) = 1. */
void somaFracção(int& numerador, int& denominador,
                 int const numerador1, int const denominador1,
                 int const numerador2, int const denominador2)
{
    assert(denominador1 != 0 and denominador2 != 0);

    numerador = numerador1 * denominador2 + numerador2 * denominador1;
    denominador = denominador1 * denominador2;

    reduzFracção(numerador, denominador);

    assert(denominador != 0 and mdc(numerador, denominador) == 1);
}

/** Escreve uma fracção no ecrã no formato usual.
    @pre 𝒱.
    @post ¬cout ∨ cout contém n/d (ou simplesmente n se d = 1)
          sendo n e d os valores de numerador e denominador. */
void escreveFracção(int const numerador, int const denominador)
{
    cout << numerador;
    if(denominador != 1)
        cout << '/' << denominador;
}

int main()
{
    // Ler fracções:
    cout << "Introduza duas fracções (numerador denominador): ";
    int n1, d1, n2, d2;
```

```
        lêFracção(n1, d1);
        lêFracção(n2, d2);

        if(not cin) {
            cerr << "Opps!  A leitura das fracções falhou!" << endl;
            return 1;
        }

        // Calcular fracção soma reduzida:
        int n, d;
        somaFracção(n, d, n1, d1, n2, d2);

        // Escrever resultado:
        cout << "A soma de ";
        escreveFracção(n1, d1);
        cout << " com ";
        escreveFracção(n2, d2);
        cout << " é ";
        escreveFracção(n, d);
        cout << '.' << endl;
    }
```

A utilização de duas variáveis inteiras independentes para representar cada fracção não permite a definição de uma função para proceder à soma, visto que as funções em C++ podem devolver um único valor. De facto, a utilização de múltiplas variáveis independentes para representar um único valor torna o código complexo e difícil de perceber. O ideal seria poder reescrever o código da mesma forma que se escreveria se o seu objectivo fosse ler e somar inteiros, e não fracções. Sendo as fracções representações dos números racionais, pretende-se escrever o programa como se segue:

```
    ...

    int main()
    {
        cout << "Introduza duas fracções (numerador denominador): ";
        Racional r1, r2;
        cin >> r1 >> r2;

        if(not cin) {
            cerr << "Opps!  A leitura dos racionais falhou!" << endl;
            return 1;
        }

        Racional r = r1 + r2;

        cout << "A soma de " << r1 << " com " << r2 << " é "
```

```
            << r << '.' << endl;
    }
```

Este objectivo irá ser atingido ainda neste capítulo.

7.2 Tipos Abstractos de Dados e classes C++

Como representar cada número racional com uma variável apenas? É necessário definir um novo tipo que se comporte como qualquer outro tipo existente em C++. É necessário um TAD (Tipo Abstracto de Dados) ou tipo de primeira categoria[2]. Um TAD ou tipo de primeira categoria é um tipo definido pelo programador que se comporta como os tipos básicos, servindo para definir instâncias, i.e., variáveis ou constantes, que guardam valores sobre os quais se pode operar. A linguagem C++ proporciona uma ferramenta, as classes C++, que permite concretizar tipos de primeira categoria.

É importante notar aqui que o termo "classe" tem vários significados. Em capítulos posteriores falar-se-á de classes propriamente ditas, que servem para definir as características comuns de objectos dessa classe, e que se concretizam também usando as classes C++. Este capítulo, por outro lado, debruça-se sobre os TAD, que também se concretizam à custa de classes C++. Se se acrescentar que a fronteira entre TAD, cujo objectivo é definir instâncias, e as classes propriamente ditas, cujo objectivo é definir as características comuns de objectos independentes, percebe-se que é inevitável alguma confusão de nomenclatura. Assim, sempre que se falar simplesmente de classe, será na acepção de classe propriamente dita, enquanto que sempre que se falar do mecanismo da linguagem C++ que permite concretizar quer TAD quer classes propriamente ditas, usar-se-á sempre a expressão "classe C++"[3].

Assim:

TAD Tipo definido pelo utilizador que se comporta como qualquer tipo básico da linguagem. O seu objectivo é permitir a definição de instâncias que armazenam valores. O que distingue umas instâncias das outras é fundamentalmente o seu valor. Nos TAD o ênfase põe-se na igualdade, pelo que as cópias são comuns.

Classe propriamente dita Conceito mais complexo a estudar em capítulos posteriores. Representam as características comuns de objectos independentes. O seu objectivo é poder construir objectos independentes de cuja interacção e colaboração resulte o comportamento adequado do programa. O ênfase põe-se na identidade, e não na igualdade, pelo que as cópias são infrequentes, merecendo o nome de clonagens.

Classe C++ Ferramenta da linguagem que permite implementar quer TAD, quer classes propriamente ditas.

---

[2]Na realidade os tipos de primeira categoria são concretizações numa linguagem de programação de TAD, que são uma abstracção matemática. Como os TAD na sua acepção matemática estão fora (por enquanto) do âmbito deste texto, os dois termos usam-se aqui como sinónimos.

[3]Apesar do cuidado posto na redacção deste texto é provável que aqui e acolá ocorram violações a esta convenção. Espera-se que não sejam factor de distracção para o leitor.

7.2.1 Definição de TAD

É possível definir um novo tipo (um TAD) para representar números racionais (na forma de uma fracção), como se segue:

```
/** Representa números racionais. */
class Racional {
  public: // Isto é magia (por enquanto).
    int numerador;
    int denominador;
};
```

A sintaxe de definição de um TAD à custa de uma classe C++ é, portanto,

```
class nome_do_tipo {
    declaração_de_membros
};
```

sendo importante notar que este é um dos poucos locais onde a linguagem exige um terminador (`;`) depois de uma chaveta final[4].

A notação usada para representar a classe C++ `Racional` pode ser vista na Figura 7.2.

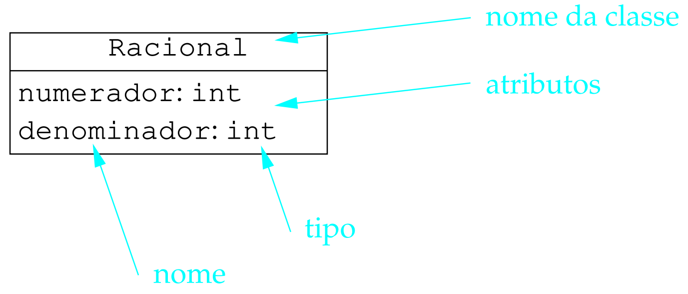


Figura 7.2: Notação usada para representar a classe C++ `Racional`.

A definição de uma classe C++ consiste na declaração dos seus *membros*. A definição da classe estabelece um modelo segundo o qual serão construídas as respectivas variáveis. No caso apresentado, as variáveis do tipo `Racional`, quando forem construídas, consistirão em dois membros: um numerador e um denominador do tipo `int`. Neste caso os membros são simples

[4] Ao contrário do que acontece na definição de rotinas e nos blocos de instruções em geral, o terminador é aqui imprescindível, pois a linguagem C++ permite a definição simultânea de um novo tipo de de variáveis desse tipo. Por exemplo:

```
class Racional {
    ...
} r; // Define o TAD Racional e uma variável r numa única instrução. Má ideia, mas possível.
```

Note-se que esta possibilidade deve ser evitada na prática.

variáveis, mas poderiam ser também constantes. Às variáveis e constantes membro de uma classe dá-se o nome de *atributos*.

Tal como as matrizes, as classes permitem guardar agregados de informação (ou seja, agregados de variáveis ou constantes, chamados elementos no caso das matrizes e membros no caso das classes), com a diferença de que, no caso das classes, essa informação pode ser de tipos diferentes.

As variáveis de um TAD definem-se como qualquer variável do C++:

```
TAD nome [= expressão];
```

ou

```
TAD nome[(expressão,...)];
```

Por exemplo:

```
Racional r1, r2;
```

define duas variáveis `r1` e `r2` não inicializadas, i.e., contendo lixo (mais tarde se verá como se podem evitar construções sem inicialização em TAD).

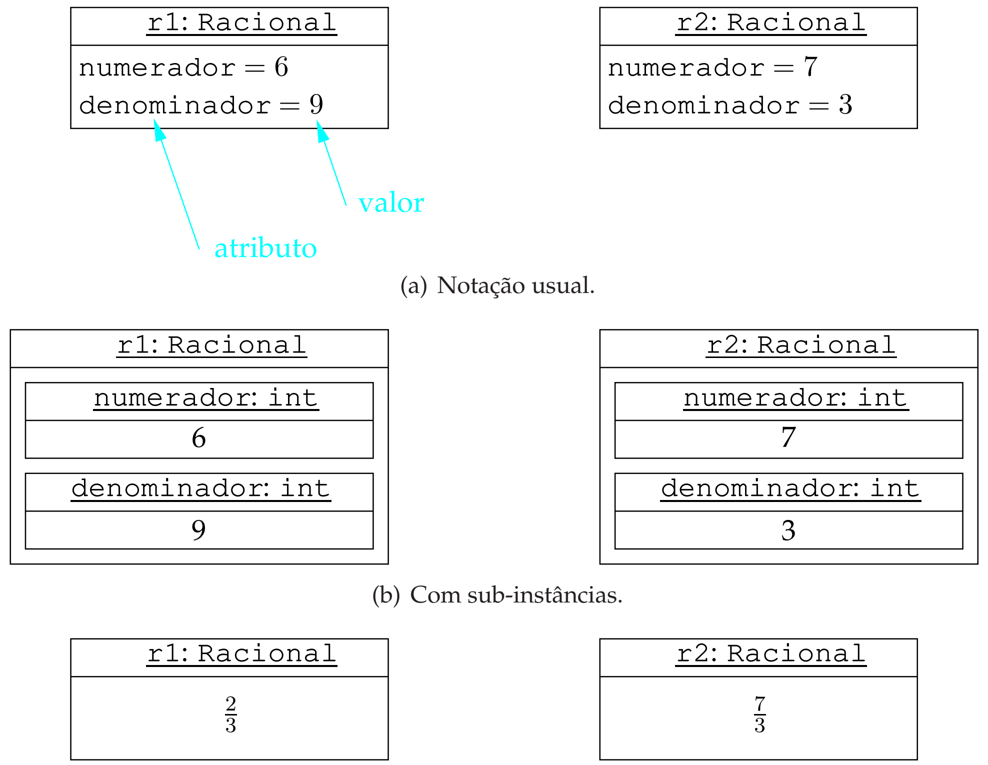
Para classes C++ que representem meros agregados de informação é possível inicializar cada membro da mesma forma como se inicializam os elementos de uma matriz clássica do C++:

```
Racional r1 = {6, 9};
Racional r2 = {7, 3};
```

Note-se, no entanto, que esta forma de inicialização deixará de ser possível (e desejável) quando se equipar a classe C++ com um construtor, como se verá mais à frente.

As instruções apresentadas constroem duas novas variáveis do tipo `Racional`, `r1` e `r2`, cada uma das quais com versões próprias dos atributos `numerador` e `denominador`. Às variáveis de um TAD também é comum chamar-se objectos e instâncias, embora em rigor o termo objecto deva ser reservado para as classes propriamente ditas, a estudar em capítulos posteriores.

Para todos os efeitos, os atributos da classe `Racional` funcionam como variáveis guardadas quer dentro da variável `r1`, quer dentro da variável `r2`. A notação usada para representar instâncias de uma classe é a que se pode ver na Figura 7.3, onde fica claro que os atributos são parte das instâncias da classe. Deve-se comparar a Figura 7.2 com a Figura 7.3, pois na primeira representa-se a classe `Racional` e na segunda as variáveis `r1` e `r2` dessa classe.



(a) Notação usual.

(b) Com sub-instâncias.

(c) Como TAD com valor lógico representado.

Figura 7.3: Notações usadas para representar instâncias da classe C++ `Racional`.

7.2.2 Acesso aos membros

O acesso aos membros de uma instância de uma classe C++ faz-se usando o operador de selecção de membro, colocando como primeiro operando a instância a cujo membro se pretende aceder, depois o símbolo `.` e finalmente o nome do membro pretendido:

```
instância.membro
```

Por exemplo,

```
Racional r1, r2;

r1.numerador = 6;
r1.denominador = 9;

r2.numerador = 7;
r2.denominador = 3;
```

constrói duas novas variáveis do tipo `Racional` e atribui valores aos respectivos atributos.

Os nomes dos membros de uma classe só têm visibilidade dentro dessa classe, pelo que poderia existir uma variável de nome `numerador` sem que isso causasse qualquer problema:

```
Racional r = {6, 9};

int numerador = 1000;
```

7.2.3 Alguma nomenclatura

Às instâncias, i.e., variáveis ou constantes, de uma classe C++ é comum chamar-se *objectos*, sendo essa a razão para as expressões "programação baseada em objectos" e "programação orientada para os objectos". No entanto, reservar-se-á o termo objecto para classes C++ que sejam concretizações de classes propriamente ditas, e não para classes C++ que sejam concretizações de TAD.

Às variáveis e constantes membro de uma classe C++ também se chama *atributos*.

Podem também existir rotinas membro de uma classe C++. A essas funções ou procedimentos chama-se *operações*. No contexto das classes propriamente ditas, em vez de se dizer "invocar uma operação para uma instância (de uma classe)" diz-se por vezes "enviar uma mensagem a um objecto".

Como se verá mais tarde, quer os atributos, quer as operações podem ser de instância ou de classe, consoante cada instância da classe C++ possua conceptualmente a sua própria cópia do membro em causa ou exista apenas uma cópia desse membro partilhada entre todas as instâncias da classe.

Todas as rotinas têm uma interface e uma implementação, e as rotinas membro não são excepção. Normalmente o termo operação é usado para designar a rotina membro do ponto de vista da sua interface, enquanto o termo *método* é usado para designar a implementação da rotina membro. Para já, a cada operação corresponde um e um só método, mas mais tarde se verá que é possível associar vários métodos à mesma operação.

Ao conjunto dos atributos e das operações de uma classe C++ chama-se *características*, embora, como se verá, o que caracteriza um TAD seja apenas a sua interface, que normalmente não inclui quaisquer atributos.

7.2.4 Operações suportadas pelas classes C++

Ao contrário do que se passa com as matrizes, as variáveis de uma classe C++ podem-se atribuir livremente entre si. O efeito de uma atribuição é o de copiar todos os atributos (de instância) entre as variáveis em causa. Da mesma forma, é possível construir uma instância de uma classe a partir de outra instância da mesma classe, ficando a primeira igual à segunda. Por exemplo:

```
Racional r1 = {6, 9};
Racional r2 = r1; // r2 construída igual a r1.

Racional r3;

r3 = r1;           // o valor de r1 é atribuído a r3, ficando as variáveis iguais.
```

Da mesma forma, estão bem definidas as devoluções e a passagem de argumentos por valor para valores de uma classe C++: as instâncias de um TAD concretizado por intermédio de uma classe C++ podem ser usadas exactamente da mesma forma que as instâncias dos tipos básicos. É possível, por isso, usar uma função, e não um procedimento, para calcular a soma de dois racionais no programa em desenvolvimento. Antes de o fazer, no entanto, far-se-á uma digressão sobre as formas de representação de número racionais.

7.3 Representação de racionais por fracções

Qualquer número racional pode ser representado por uma fracção, que é um par ordenado de números inteiros (n, d), em que n e d são os termos da fracção[5]. Ao segundo termo dá-se o nome de *denominador* (é o que dá o nome à fracção) e ao primeiro *numerador* (diz a quantas fracções nos referimos). Por exemplo, $(3, 4)$ significa "três quartos". Normalmente os racionais representam-se graficamente usando uma notação diferente da anterior: n/d ou $\frac{n}{d}$.

Uma fracção $\frac{n}{d}$ só representa um número racional se $d \neq 0$. Por outro lado, é importante saber se fracções diferentes podem representar o mesmo racional ou se, pelo contrário, fracções diferentes representam sempre racionais diferentes. A resposta à questão inversa é evidente:

[5] Há representações alternativas para as fracções, ver [1][7].

racionais diferentes têm forçosamente representações diferentes. Mas $\frac{-4}{2}$, $\frac{2}{-1}$ e $\frac{-2}{1}$ são fracções que correspondem a um único racional, e que, por acaso, também é um inteiro. Para se obter uma representação em fracções que seja única para cada racional, é necessário introduzir algumas restrições adicionais.

Em primeiro lugar, é necessário usar apenas o numerador ou o denominador para conter o sinal do número racional. Como já se impôs uma restrição ao denominador, viz. $d \neq 0$, é natural impor uma restrição adicional: d deve ser não-negativo. Assim, $0 < d$. Mas é necessária uma restrição adicional. Para que a representação seja única, é também necessário que n e d não tenham qualquer divisor comum diferente de 1, i.e., que $\mathrm{mdc}(n, d) = 1$. Uma fracção nestas condições diz-se em termos mínimos e dos seus termos diz-se que são *mutuamente primos*. Dos três exemplos acima ($\frac{-4}{2}$, $\frac{2}{-1}$ e $\frac{-2}{1}$), apenas a última fracção verifica todas as condições enunciadas, ou seja, tem denominador positivo e numerador e denominador são mutuamente primos. Uma fracção $\frac{n}{d}$ que verifique estas condições, i.e., $0 < d \wedge \mathrm{mdc}(n, d) = 1$, diz-se no *formato canónico*.

7.3.1 Operações aritméticas elementares

As operações aritméticas elementares (adição, subtracção, multiplicação, divisão, simétrico e identidade) estão bem definidas para os racionais (com excepção da divisão por 0, ou melhor, por $\frac{0}{1}$). Assim, em termos da representação dos racionais como fracções, o resultado das operações aritméticas elementares pode ser expresso como

$$
\begin{aligned}
\frac{n_1}{d_1} + \frac{n_2}{d_2} &= \frac{n_1 \times d_2 + n_2 \times d_1}{d_1 \times d_2}, \\
\frac{n_1}{d_1} - \frac{n_2}{d_2} &= \frac{n_1 \times d_2 - n_2 \times d_1}{d_1 \times d_2}, \\
\frac{n_1}{d_1} \times \frac{n_2}{d_2} &= \frac{n_1 \times n_2}{d_1 \times d_2}, \\
\frac{n_1}{d_1} / \frac{n_2}{d_2} &= \frac{\frac{n_1}{d_1}}{\frac{n_2}{d_2}} = \frac{n_1 \times d_2}{d_1 \times n_2} \text{ se } n_2 \neq 0, \\
-\frac{n}{d} &= \frac{-n}{d}, \text{ e} \\
+\frac{n}{d} &= \frac{n}{d}.
\end{aligned}
$$

7.3.2 Canonicidade do resultado

Tal como definidas, algumas destas operações sobre fracções não garantem que o resultado esteja no formato canónico, mesmo que as fracções que servem de operandos o estejam. Este problema é fácil de resolver, no entanto, pois dada uma fracção $\frac{n}{d}$ que não esteja forçosamente no formato canónico, pode-se dividir ambos os termos pelo seu máximo divisor comum para obter uma fracção equivalente em termos mínimos, $\frac{n/\mathrm{mdc}(n,d)}{d/\mathrm{mdc}(n,d)}$, e, se o denominador for negativo, pode-se multiplicar ambos os termos por -1 para obter uma fracção equivalente com o numerador positivo, $\frac{-n}{-d}$.

7.3.3 Aplicação à soma de fracções

Voltando à classe C++ definida,

```
/** Representa números racionais. */
class Racional {
  public: // Isto é magia (por enquanto).
    int numerador;
    int denominador;
};
```

é muito importante estar ciente das diferenças entre a concretização do conceito de racional e o conceito em si: os valores representáveis num `int` são limitados, o que significa que não é possível representar qualquer racional numa variável do tipo `Racional`, tal como não era possível representar qualquer inteiro numa variável do tipo `int`. Os problemas causados por esta diferença serão ignorados durante a maior parte deste capítulo, embora na Secção 7.13 sejam, senão resolvidos, pelo menos mitigados.

Um mero agregado de dois inteiros, mesmo com um nome sugestivo, não só tem pouco interesse, como poderia representar muitas coisas diferentes. Para que esse agregado possa ser considerado a concretização de um TAD, é necessário definir também as operações que o novo tipo suporta. Uma das operações a implementar é a soma. Pode-se implementar a soma actualizando o procedimento do programa original para a seguinte função:

```
/** Devolve a soma de dois racionais.
    @pre r1.denominador ≠ 0 ∧ r2.denominador ≠ 0.
    @post somaDe = r1 + r2∧
          somaDe.denominador ≠
0 ∧ mdc(somaDe.numerador, somaDe.denominador) = 1. */
Racional somaDe(Racional const r1, Racional const r2)
{
    assert(r1.denominador1 != 0 and r2.denominador2 != 0);

    Racional r;

    r.numerador = r1.numerador * r2.denominador +
        r2.numerador * r1.denominador;
    r.denominador = r1.denominador * r2.denominador;

    reduz(r);

    assert(r.denominador != 0 and mdc(r.numerador, r.denominador) == 1);

    return r;
}
```

onde `reduz()` é um procedimento para reduzir a fracção que representa o racional, i.e., uma adaptação do procedimento `reduzFracção()`.

O programa pode agora ser reescrito ser na íntegra para usar a nova classe C++, devendo-se ter o cuidado de colocar a definição da classe C++ `Racional` antes da sua primeira utilização no programa. Pode-se aproveitar para alterar os nomes das rotinas, onde o sufixo `Fracção` se torna desnecessário, dado o tipo dos respectivos parâmetros:

```
#include <iostream>
#include <cassert>

using namespace std;

/** Devolve o máximo divisor comum dos inteiros passados como argumento.
    @pre $m = m \wedge n = n$.
    @post $mdc = \begin{cases} \mathrm{mdc}(m,n) & m \neq 0 \vee n \neq 0 \\ 1 & m = 0 \wedge n = 0 \end{cases}$ . */
int mdc(int m, int n)
{
    ...
}
/** Representa números racionais. */
class Racional {
  public: // Isto é magia (por enquanto).
    int numerador;
    int denominador;
};

/** Reduz a fracção que representa o racional recebido como argumento.
    @pre $r.denominador \neq 0 \wedge r = r$.
    @post $r.denominador \neq 0 \wedge \mathrm{mdc}(r.numerador, r.denominador) = 1 \wedge r = r$. */
void reduz(Racional& r)
{
    assert(r.denominador != 0);

    int k = mdc(r.numerador, r.denominador);

    r.numerador /= k;
    r.denominador /= k;

    assert(r.denominador != 0 and mdc(r.numerador, r.denominador) == 1);
}

/** Lê do teclado um racional, na forma de dois inteiros sucessivos.
    @pre $r = r$.
    @post Se cin e cin tem dois inteiros $n'$ e $d'$ disponíveis para leitura, com $d' \neq$
$0$, então
```

```
            0 < r.denominador ∧ mdc(r.numerador, r.denominador) = 1∧
            r = n′/d′ ∧ cin,
        senão
            r = r ∧ ¬cin. */
void lê(Racional& r)
{
    int n, d;

    if(cin >> n >> d)
        if(d == 0)
            cin.setstate(ios_base::failbit);
        else {
            r.numerador = d < 0 ? -n : n;
            r.denominador = d < 0 ? -d : d;

            reduz(r);

            assert(0 < r.denominador and mdc(r.numerador, r. denominador) == 1 and
                   r.numerador * d == n * r.denominador and cin);

            return;
        }

    assert(not cin);
}

/** Devolve a soma de dois racionais.
    @pre r1.denominador ≠ 0 ∧ r2.denominador ≠ 0.
    @post somaDe = r1 + r2∧
          somaDe.denominador ≠
0 ∧ mdc(somaDe.numerador, somaDe.denominador) = 1. */
Racional somaDe(Racional const r1, Racional const r2)
{
    assert(r1.denominador != 0 and r2.denominador != 0);

    Racional r;

    r.numerador = r1.numerador * r2.denominador +
        r2.numerador * r1.denominador;
    r.denominador = r1.denominador * r2.denominador;

    reduz(r);

    assert(r.denominador != 0 and mdc(r.numerador, r.denominador) == 1);

    return r;
```

```
}

/** Escreve um racional no ecrã no formato de uma fracção.
    @pre V.
    @post ¬cout∨ cout contém n/d (ou simplesmente n se d = 1)
          sendo n/d a fracção canónica correspondente ao racional r. */
void escreve(Racional const r)
{
    cout << r.numerador;
    if(r.denominador != 1)
        cout << '/' << r.denominador;
}

int main()
{
    // Ler fracções:
    cout << "Introduza duas fracções (numerador denominador): ";
    Racional r1, r2;
    lê(r1);
    lê(r2);

    if(not cin) {
        cerr << "Opps!  A leitura das fracções dos racionais falhou!" << endl;
        return 1;
    }

    // Calcular racional soma:
    Racional r = somaDe(r1, r2);

    // Escrever resultado:
    cout << "A soma de ";
    escreve(r1);
    cout << " com ";
    escreve(r2);
    cout << " é ";
    escreve(r);
    cout << '.' << endl;
}
```

Ao escrever este pedaço de código o programador assumiu dois papeis: produtor e consumidor. Quando definiu a classe C++ `Racional` e a função `somaDe()`, que opera sobre variáveis dessa classe C++, fez o papel de produtor. Quando escreveu a função `main()`, assumiu o papel de consumidor.

7.3.4 Encapsulamento e categorias de acesso

O leitor mais atento terá reparado que o código acima tem pelo menos um problema: a classe `Racional` não tem qualquer mecanismo que impeça o programador de colocar 0 (zero) no denominador de uma fracção:

```
Racional r1;
r.numerador = 6;
r.denominador = 0;
```

ou

```
Racional r1 = {6, 0};
```

Isto é claramente indesejável, e tem como origem o facto do produtor ter tornado públicos os membros `numerador` e `denominador` da classe: é esse o significado do especificador de acesso `public`. De facto, os membros de uma classe podem pertencer a uma de três categorias de acesso: público, protegido e privado. Para já apenas se descreverão a primeira e a última.

Membros públicos, introduzidos pelo especificador de acesso `public:`, são acessíveis sem qualquer restrição. Membros privados, introduzidos pelo especificador de acesso `private:`, são acessíveis apenas por membros da mesma classe (ou, alternativamente, por funções "amigas" da classe, que serão vistas mais tarde). Fazendo uma analogia de uma classe com um clube, dir-se-ia que há certas partes de um clube que estão abertas ao público e outras que estão à disposição apenas dos seus membros.

O consumidor de um relógio ou de um micro-ondas assume que não precisa de conhecer o funcionamento interno desses aparelhos, podendo recorrer apenas a uma interface. Assim, o produtor desses aparelhos normalmente esconde o seu mecanismo numa caixa, deixando no exterior apenas a interface necessária para o consumidor. Também o produtor da classe C++ `Racional` deveria ter escondido os pormenores de implementação da classe C++ do consumidor final.

Podem-se resumir estas ideias num princípio básico da programação:

Princípio do encapsulamento: O produtor deve esconder do consumidor final tudo o que puder ser escondido. I.e., os pormenores de implementação devem ser escondidos, devendo-se fornecer interfaces limpas e simples para a manipulação das entidades fabricadas (aparelhos de cozinha, relógios, rotinas C++, classes C++, etc.).

Isso consegue-se, no caso das classes C++, usando o especificador de acesso `private:` para esconder os membros da classe:

```
/** Representa números racionais. */
class Racional {
  private:
    int numerador;
    int denominador;
};
```

Ao se classificar os membros `numerador` e `denominador` como privados não se impede o programador consumidor de, usando mecanismos mais ou menos obscuros e perversos, aceder ao seu valor. O facto de um membro ser privado não coloca barreiras muito fortes quanto ao seu acesso. Pode-se dizer que funciona como um aviso, esse sim forte, de que o programador consumidor não deve aceder a eles, para seu próprio bem (o produtor poderia, por exemplo, decidir alterar os nomes dos membros para `n` e `d`, com isso invalidando código que fizesse uso directo dos membros da classe). O compilador encarrega-se de gerar erros de compilação por cada acesso ilegal a membros privados de uma classe.

Assim, é claro que os membros privados de uma classe C++ fazem parte da sua implementação, enquanto os membros públicos fazem parte da sua interface.

Tornados os atributos da classe privados, torna-se impossível no procedimento `lê()` atribuir valores directamente aos seus membros. Da mesma forma, todas as outras rotinas deixam de poder aceder aos atributos da classe. A inicialização típica dos agregados, por exemplo

```
Racional r1 = {6, 9};
```

também deixa de ser possível.

Que fazer?

7.3.5 Rotinas membro: operações e métodos

Uma vez que a membros privados têm acesso quaisquer outros membros da classe, a solução passa por tornar as rotinas existentes membros da classe C++ `Racional`. Começar-se-á por tornar o procedimento `escreve()` membro da classe, i.e., por transformá-lo de simples rotina em operação do TAD em concretização:

```
...

/** Representa números racionais. */
class Racional {
  public:
    /** Escreve o racional no ecrã no formato de uma fracção.
          @pre *this = r.
          @post *this = r∧(¬cout∨ cout contém n/d (ou simplesmente n se d = 1)
                sendo n/d a fracção canónica correspondente ao racional *this). */
    void escreve(); // Declaração da rotina membro: operação.

  private:
    int numerador;
    int denominador;
};

// Definição da rotina membro: método.
void Racional::escreve()
```

```
{
    cout << numerador;
    if(denominador != 1)
        cout << '/' << denominador;
}

...
```

São de notar quatro pontos importantes:

1. Para o consumidor da classe C++ poder invocar a nova operação, é necessário que esta seja pública. Daí o especificador de acesso `public:`, que coloca a nova operação `escreve()` na interface da classe C++.

2. Qualquer operação ou rotina membro de uma classe C++ tem de ser declarada dentro da definição dessa classe e definida fora ou, alternativamente, definida (e portanto também declarada) dentro da definição da classe. Recorda-se que à implementação de uma operação se chama método, e que por isso todos os métodos fazem parte da implementação de uma classe C++[6].

3. A operação `escreve()` foi declarada sem qualquer parâmetro.

4. Há um pormenor na definição do método `escreve()` que é novo: o nome do método é precedido de `Racional::`. Esta notação serve para indicar que `escreve()` é um método correspondente a uma operação da classe `Racional`, e não uma rotina vulgar.

Onde irá a operação `Racional::escreve()` buscar o racional a imprimir? De onde vêem as variáveis `numerador` e `denominador` usadas no corpo do método `Racional::escreve()`?

Em primeiro lugar, recorde-se que o acesso aos membros de uma classe se faz usando o operador de selecção de membro. Ou seja,

```
instância.nome_do_membro
```

em que *`instância`* é uma qualquer instância da classe em causa. Esta notação é tão válida para atributos como para operações, pelo que a instrução para escrever a variável `r` no ecrã, no programa em desenvolvimento, deve passar a ser:

```
r.escreve();
```

O que acontece é que instância através da qual a operação `Racional::escreve()` é invocada está explícita na própria invocação, mas está implícita durante a execução do respectivo método! Mais, essa instância que está implícita durante a execução pode ser modificada pelo

[6] Em capítulos posteriores se verá que as classes propriamente ditas podem ter mais do que um método associado a cada operação.

método, pelo menos se for uma variável. Tudo funciona como se a instância usada para invocar a operação fosse passada automaticamente por referência.

Durante a execução do método `Racional::escreve()`, `numerador` e `denominador` referem-se aos atributos da instância através da qual a respectiva operação foi invocada. Assim, quando se adaptar o final do programa em desenvolvimento para

```
int main()
{
    ...

    // Escrever resultado:
    ...
    r1.escreve();
    ...
    r2.escreve();
    ...
    r.escreve();
    ...
}
```

durante a execução do método `Racional::escreve()` as variáveis `numerador` e `denominador` referir-se-ão sucessivamente aos correspondentes atributos de `r1`, `r2`, e `r`. À instância que está implícita durante a execução de um método chama-se naturalmente *instância implícita* (ou *variável implícita* se for uma variável, ou *constante implícita* se for uma constante), pelo que no exemplo anterior a instância implícita durante a execução do método começa por ser `r1`, depois é `r2` e finalmente é `r`.

É possível explicitar a instância implícita durante a execução de um método da classe, ou seja, a instância através da qual a respectiva operação foi invocada. Para isso usa-se a construção `*this`[7]. Esta construção usou-se na documentação da operação `escreve()`, nomeadamente no seu contrato, para deixar claro que a invocação da operação não afecta a instância implícita. Mais tarde se verá uma forma mais elegante de garantir a constância da instância implícita durante a execução de um método, i.e, uma forma de garantir que a instância implícita é tratada como uma constante implícita, mesmo que na realidade seja uma variável.

Resolvemos o problema do acesso aos atributos privados para o procedimento `escreve()`, transformando-o em procedimento membro da classe C++. É necessário fazer o mesmo para todas as outras rotinas que acedem directamente aos atributos:

```
#include <iostream>
#include <cassert>

using namespace std;

/** Devolve o máximo divisor comum dos inteiros passados como argumento.
```

[7]O significado do operador `*` ficará claro em capítulos posteriores.

```
    @pre m = m ∧ n = n.
    @post mdc = { mdc(m, n)   m ≠ 0 ∨ n ≠ 0
                {     1       m = 0 ∧ n = 0 . */
int mdc(int m, int n)
{
    ...
}
/** Representa números racionais. */
class Racional {
  public:
    /** Escreve o racional no ecrã no formato de uma fracção.
        @pre *this = r.
        @post *this = r∧(¬cout∨ cout contém n/d (ou simplesmente n se d = 1)
              sendo n/d a fracção canónica correspondente ao racional *this). */
    void escreve();

    /** Devolve a soma com o racional recebido como argumento.
        @pre denominador ≠ 0 ∧ r2.denominador ≠ 0 ∧ *this = r.
        @post *this = r ∧ somaCom = *this + r2 ∧ denominador ≠ 0∧
              somaCom.denominador ≠
0 ∧ mdc(somaCom.numerador, somaCom.denominador) = 1. */
    Racional somaCom(Racional const r2);

    /** Lê do teclado um novo valor para o racional, na forma de dois inteiros sucessi-
vos.
        @pre *this = r.
        @post Se cin e cin tem dois inteiros n′ e d′ disponíveis para lei-
tura, com d′ ≠ 0, então
                0 < denominador ∧ mdc(numerador, denominador) = 1∧
                *this = n′/d′ ∧ cin,
              senão
                *this = r ∧ ¬cin. */
    void lê();

  private:
    int numerador;
    int denominador;
    /** Reduz a fracção que representa o racional.
        @pre denominador ≠ 0 ∧ *this = r.
        @post denominador ≠ 0 ∧ mdc(numerador, denominador) = 1∧
              *this = r. */
    void reduz();
};

void Racional::escreve()
{
```

```
    cout << numerador;
    if(denominador != 1)
        cout << '/' << denominador;
}

Racional Racional::somaCom(Racional const r2)
{
    assert(denominador != 0 and r2.denominador != 0);

    Racional r;
    r.numerador = numerador * r2.denominador +
        r2.numerador * denominador;
    r.denominador = denominador * r2.denominador;

    r.reduz();

    assert(denominador != 0 and
              r.denominador != 0 and mdc(r.numerador, r.denominador) == 1);

    return r;
}

void Racional::lê()
{
    int n, d;

    if(cin >> n >> d)
        if(d == 0)
            cin.setstate(ios_base::failbit);
        else {
            numerador = d < 0 ? -n : n;
            denominador = d < 0 ? -d : d;

            reduz();

            assert(0 < denominador and mdc(numerador, denominador) == 1 and
                          numerador * d == n * denominador and cin);

            return;
        }

    assert(not cin);
}

void Racional::reduz()
{
```

```
    assert(denominador != 0);

    int k = mdc(numerador, denominador);

    numerador /= k;
    denominador /= k;

    assert(denominador != 0 and mdc(numerador, denominador) == 1);
}

int main()
{
    // Ler fracções:
    cout << "Introduza duas fracções (numerador denominador): ";
    Racional r1, r2;
    r1.lê();
    r2.lê();

    if(not cin) {
        cerr << "Opps!  A leitura dos racionais falhou!" << endl;
        return 1;
    }

    // Calcular racional soma:
    Racional r = r1.somaCom(r2);

    // Escrever resultado:
    cout << "A soma de ";
    r1.escreve();
    cout << " com ";
    r2.escreve();
    cout << " é ";
    r.escreve();
    cout << '.' << endl;
}
```

Na operação `Racional::somaCom()`, soma-se a instância implícita com o argumento passado à operação. No programa acima, por exemplo, a variável `r1` da função `main()` funciona como instância implícita durante a execução do método correspondente à operação `Racional::somaCom()` e `r2` funciona como argumento.

O procedimento `reduz()` foi transformado em operação *privada* da classe C++ que representa o TAD em desenvolvimento. Tomou-se tal decisão por não haver qualquer necessidade de o consumidor do TAD se preocupar directamente com a representação em fracção dos racionais. O consumidor do TAD limita-se a preocupar-se com o comportamento exterior do tipo. Pelo contrário, para o produtor da classe C++ a representação dos racionais é fundamental, pois é ele que tem de garantir que todas as operações cumprem o respectivo contrato.

A invocação da operação `Racional::reduz()` no método `Racional::lê()` é feita sem necessidade de usar a sintaxe usual para a invocação de operações, i.e., sem indicar explicitamente a instância através da qual (e para a qual) essa invocação é feita. Isso deve-se ao facto de se pretender fazer a invocação para a instância implícita. Seria possível explicitar essa instância,

```
(*this).reduz();
```

tal como de resto poderia ter sido feito para os atributos,

```
(*this).numerador = n;
```

mas isso conduziria apenas a código mais denso. Note-se que os parênteses em volta de `*this` são fundamentais, pois o operador de selecção de membro tem maior precedência que o operador unário `*` (ou seja, o operador "conteúdo de", a estudar mais tarde).

É também importante perceber-se que não existe qualquer vantagem em tornar a função `mdc()` membro na nova classe C++. Em primeiro lugar, pode haver necessidade de calcular o máximo divisor comum de outros inteiros que não o numerador e o denominador. Aliás, tal necessidade surgirá ainda durante este capítulo. Em segundo lugar porque o cálculo do máximo divisor comum poderá ser necessário em contextos que nada tenham a ver com números racionais.

Finalmente, a notação usada para calcular a soma

```
Racional r = r1.somaCom(r2);
```

é horrenda, sem dúvida alguma. Numa secção posterior se verá como sobrecarregar o operador + de modo a permitir escrever

```
Racional r = r1 + r2;
```

7.4 Classes C++ como módulos

Das discussões anteriores, nomeadamente sobre o princípio do encapsulamento e as categorias de acesso dos membros de uma classe, torna-se claro que as classes C++ são uma unidade de modularização. De facto, assim é. Aliás, as classes são a unidade de modularização por excelência na linguagem C++ e na programação baseada em (e orientada para) objectos.

Como qualquer módulo que se preze, as classes C++ distinguem claramente interface e implementação. A interface de uma classe C++ corresponde aos seus membros públicos. Usualmente a interface de uma classe C++ consiste num conjunto de operações e tipos públicos. A implementação de uma classe C++ consiste, pelo contrário, nos membros privados e na definição das respectivas operações, i.e., nos métodos da classe. Normalmente a implementação de uma classe C++ contém os atributos da classe, particularmente as variáveis membro, e operações utilitárias, necessárias apenas para o programador produtor da classe.

É de toda a conveniência que os atributos de uma classe C++ (e em especial as suas variáveis membro) sejam privados. Só dessa forma se garante que um consumidor da classe não pode, perversa ou acidentalmente, alterar os valores dos atributos de tal forma que um instância da classe C++ deixe de estar num estado válido. Este assunto será retomado com maior pormenor mais abaixo, quando se falar da chamada *CIC* (Condição Invariante de Classe).

As classes C++ possuem também um "manual de utilização", correspondente ao contrato entre o seu produtor e os seus consumidores. Esse contrato é normalmente expresso através de um comentário de documentação para a classe em si e dos comentários de documentação de todas os seus membros públicos.

7.4.1 Construtores

Suponha-se o código

```
Racional a;
a.numerador = 1;
a.denominador = 3;
```

ou

```
Racional a = {1, 3};
```

A partir do momento em que os atributos da classe passaram a ser privados ambas as formas de inicialização[8] deixaram de ser possíveis. Como resolver este problema?

Para os tipos básicos da linguagem, a inicialização faz-se usando uma de duas possíveis sintaxes:

```
int a = 10;
```

ou

```
int a(10);
```

Se realmente se pretende que a nova classe C++ `Racional` represente um tipo de primeira categoria, é importante fornecer uma forma de os racionais poderem se inicializados de uma forma semelhante. Por exemplo,

```
Racional r(1, 3); // Pretende-se que inicialize r com o racional 1/3.
```

[8]Na realidade no primeiro troço de código não se faz uma inicialização. As operações de atribuição alteram os valores dos atributos já inicializados (ou melhor, a atributos deixados por inicializar pelas regras absurdas importadas da linguagem C, e por isso contendo lixo).

ou mesmo

```
Racional r = 2; // Pretende-se que inicialize r com o racional 2/1.
Racional r(3);  // Pretende-se que inicialize r com o racional 3/1.
```

Por outro lado, deveria ser possível evitar o comportamento dos tipos básicos do C++ e eliminar completamente as instâncias por inicializar, fazendo com que à falta de uma inicialização explícita, os novos racionais fossem inicializados com o valor zero, (0, representado pela fracção $\frac{0}{1}$). Ou seja,

```
Racional r;
Racional r(0);
Racional r(0, 1);
```

deveriam ser instruções equivalentes.

Finalmente, deveria haver alguma forma de evitar a inicialização de racionais com valores impossíveis, nomeadamente com denominador nulo. I.e., a instrução

```
Racional r(3, 0);
```

deveria de alguma forma resultar num erro.

Quando se constrói uma instância de uma classe C++, é chamado um procedimento especial que se chama construtor da classe C++. Esse construtor é fornecido implicitamente pela linguagem e é um construtor por omissão, i.e., é um construtor que se pode invocar sem lhe passar quaisquer argumento[9]. O construtor por omissão fornecido implicitamente constrói cada um dos atributos da classe invocando o respectivo construtor por omissão. Neste caso, como os atributos são de tipos básicos da linguagem, não são inicializados durante a sua construção, ao contrário do que seria desejável, contendo por isso lixo. Para evitar o problema, deve ser o programador produtor a declarar explicitamente um ou mais construtores (e, já agora, defini-los com o comportamento pretendido), pois nesse caso o construtor por omissão deixa de ser fornecido implicitamente pela linguagem.

Uma vez que se pretende que os racionais sejam inicializados por omissão com zero, tem de se fornecer um construtor por omissão explicitamente que tenha esse efeito:

```
/** Representa números racionais. */
class Racional {
  public:
    /** Constrói racional com valor zero. Construtor por omissão.
        @pre V.
        @post *this = 0∧
              0 < denominador ∧ mdc(numerador, denominador) = 1. */
```

[9]Nem sempre a linguagem fornece um construtor por omissão implicitamente. Isso acontece quando a classe tem atributos que são constantes, referências, ou que não têm construtores por omissão, entre outros casos.

```
        Racional();

        ...

      private:

        ...

    };

    Racional::Racional()
        : numerador(0), denominador(1)
    {
        assert(0 < denominador and mdc(numerador, denominador) == 1);
    }
```

Os construtores são operações de uma classe C++, mas são muito especiais, quer por por razões semânticas, quer por razões sintácticas. Do ponto de vista semântico, o que os distingue dos outros operadores é o facto de não serem invocados através de variáveis da classe preexistentes. Pelo contrário, os construtores são invocados justamente para construir uma nova variável. Do ponto de vista sintáctico os construtores têm algumas particularidades. A primeira é que têm o mesmo nome que a própria classe. Os construtores são como que funções membro, pois têm como resultado uma nova variável da classe a que pertencem. No entanto, não só não se pode indicar qualquer tipo de devolução no seu cabeçalho, como no seu corpo não é permitido devolver qualquer valor, pois este age sobre uma instância implícita em construção.

Quando uma instância de uma classe é construída, por exemplo devido à definição de uma variável dessa classe, é invocado o construtor da classe compatível com os argumentos usados na inicialização. I.e., é possível que uma classe tenha vários construtores sobrecarregados, facto de que se tirará partido em breve. Os argumentos são passados aos construtores colocando-os entre parênteses na definição das instâncias. Por exemplo, as instruções

```
Racional r;
Racional r(0);
Racional r(0, 1);
```

deveriam todas construir uma nova variável racional com o valor zero, muito embora para já só a primeira instrução seja válida, pois a classe ainda não possui construtores com argumentos.

Note-se que as instruções

```
Racional r;
Racional r();
```

não são equivalentes! Esta irregularidade sintáctica do C++ deve-se ao facto de a segunda instrução ter uma interpretação alternativa: a de declarar uma função `r` que não tem parâmetros e devolve um valor `Racional`. Face a esta ambiguidade de interpretação, a linguagem optou por dar preferência à declaração de uma função...

Aquando da construção de uma instância de uma classe C++, um dos seus construtores é invocado. Antes mesmo de o seu corpo ser executado, no entanto, todos os atributos da classe são construídos. Se se pretender passar argumentos aos construtores dos atributos, então é obrigatória a utilização de listas de utilizadores, que se colocam na definição do construtor, entre o cabeçalho e o corpo, após o símbolo dois-pontos (`:`). Esta lista consiste no nome dos atributos pela mesma ordem pela qual estão definidos na classe C++, seguido cada um dos argumentos a passar ao respectivo construtor colocados entre parênteses. No caso da classe C++ `Racional`, pretende-se inicializar os atributos `numerador` e `denominador` respectivamente com os valores 0 e 1, pelo que a lista de inicializadores é

```
Racional::Racional()
    : numerador(0), denominador(1)
{
    ...
}
```

Uma vez que se pretendem mais duas formas de inicialização dos racionais, é necessário fornecer dois construtores adicionais. O primeiro constrói um racional a partir de um único inteiro, o que é quase tão simples como construir um racional com o valor zero. O segundo é um pouco mais complicado, pois, construindo um racional a partir do numerador e denominador de uma fracção, precisa de receber garantidamente um denominador não-nulo e tem de ter o cuidado de garantir que os seus atributos, `numerador` e `denominador`, estão no formato canónico das fracções:

```
/** Representa números racionais. */
class Racional {
  public:
    /** Constrói racional com valor zero. Construtor por omissão.
        @pre V.
        @post *this = 0∧
             0 < denominador ∧ mdc(numerador, denominador) = 1. */
    Racional();

    /** Constrói racional com valor inteiro.
        @pre V.
        @post *this = n∧
             0 < denominador ∧ mdc(numerador, denominador) = 1. */
    Racional(int const n);
    /** Constrói racional correspondente a n/d.
        @pre d ≠ 0.
        @post *this = n/d∧
             0 < denominador ∧ mdc(numerador, denominador) = 1. */
    Racional(int const n, int const d);
```

```
    ...

  private:

    ...

};

Racional::Racional()
    : numerador(0), denominador(1)
{
    assert(0 < denominador and mdc(numerador, denominador) == 1 and
           numerador == 0);
}

Racional::Racional(int const n)
    : numerador(n), denominador(1)
{
    assert(0 < denominador and mdc(numerador, denominador) == 1 and
           numerador == n * denominador);
}

Racional::Racional(int const n, int const d)
    : numerador(d < 0 ? -n : n),
      denominador(d < 0 ? -d : d)
{
    assert(d != 0);

    reduz();

    assert(0 < denominador and mdc(numerador, denominador) == 1 and
           numerador * d == n * denominador);
}

...
```

Uma observação atenta dos três construtores revela que os dois primeiros são quase iguais, enquanto o terceiro é mais complexo, pois necessita verificar o sinal do denominador recebido no parâmetro `d` e, além disso, tem de se preocupar com a redução dos termos da fracção. Assim, surge naturalmente a ideia de condensar os dois primeiros construtores num único, não se fazendo o mesmo relativamente ao último construtor (à custa do qual poderiam ser obtidos os dois primeiros), por razões de eficiência.

A condensação dos dois primeiros construtores num único faz-se recorrendo aos parâmetros com argumentos por omissão, vistos na Secção 3.6:

```
/** Representa números racionais. */
class Racional {
  public:
      /** Constrói racional com valor inteiro. Construtor por omissão.
          @pre V.
          @post *this = n∧
                0 < denominador ∧ mdc(numerador, denominador) = 1. */
    Racional(int const n = 0);
    /** Constrói racional correspondente a n/d.
          @pre d ≠ 0.
          @post *this = n/d∧
                0 < denominador ∧ mdc(numerador, denominador) = 1. */
    Racional(int const n, int const d);

    ...

  private:

    ...

};

Racional::Racional(int const n)
    : numerador(n), denominador(1)
{
    assert(0 < denominador and mdc(numerador, denominador) == 1 and
           numerador == n * denominador);
}

Racional::Racional(int const n, int const d)
    : numerador(d < 0 ? -n : n),
      denominador(d < 0 ? -d : d)
{
    assert(d != 0);

    reduz();

    assert(0 < denominador and mdc(numerador, denominador) == 1 and
           numerador * d == n * denominador);
}
...
```

A Figura 7.4 mostra a notação usada para representar a classe C++ `Racional` desenvolvida até aqui.

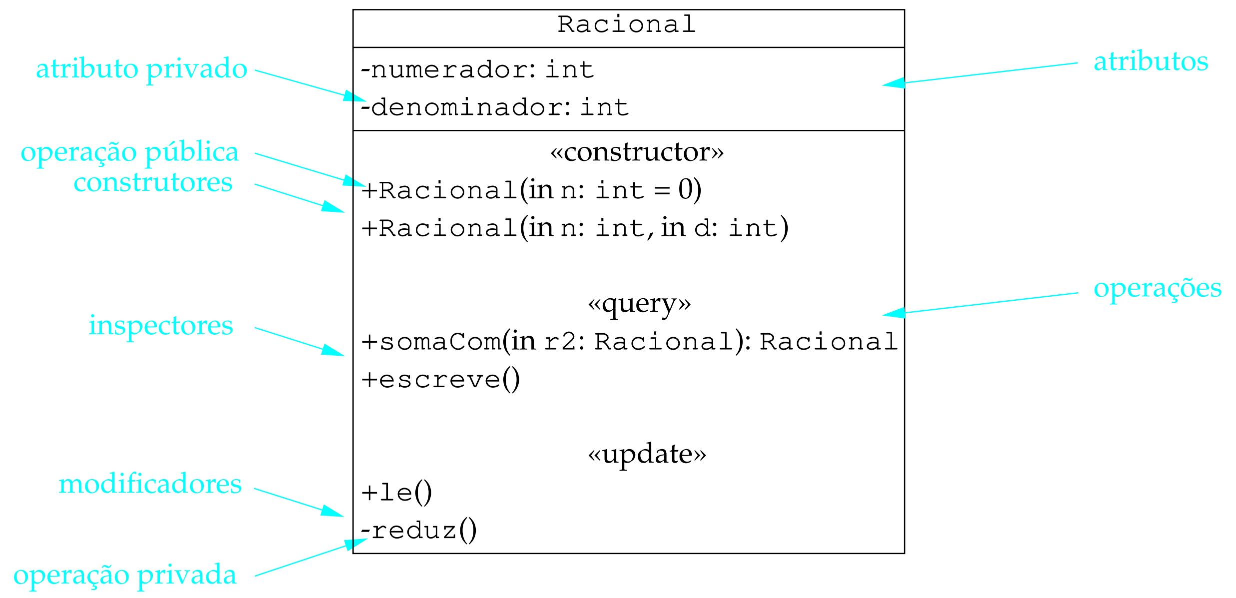


Figura 7.4: A classe C++ `Racional` agora também com operações. Note-se a utilização de + e - para indicar a características públicas e privadas da classe C++, respectivamente, e o termo "in" para indicar que o argumento da operação `Racional::somaCom()` é passado por valor, ou seja, apenas "para dentro" da operação.

7.4.2 Construtores por cópia

Viu-se que a linguagem fornece implicitamente um construtor por omissão para as classes, excepto quando estas declaram algum construtor explicitamente. Algo de semelhante se passa relativamente aos chamados *construtores por cópia*. Estes construtores são usados para construir uma instância de uma classe à custa de outra instância da mesma classe.

A linguagem fornece também implicitamente um construtor por cópia, desde que tal seja possível, para todas as classes C++ que não declarem explicitamente um construtor por cópia. O construtor por cópia fornecido implicitamente limita-se a invocar os construtores por cópia para construir os atributos da instância em construção à custa dos mesmos atributos na instância original, sendo a invocação realizada por ordem de definição dos atributos na definição da classe.

É possível, e muitas vezes desejável, declarar ou mesmo definir explicitamente um construtor por cópia para as classes. Este assunto será tratado com pormenor num capítulo posterior.

7.4.3 Condição invariante de classe

Na maior parte das classes C++ que concretizam um TAD, os atributos só estão num estado aceitável se verificarem um conjunto de restrições, expressos normalmente na forma de uma condição a que se dá o nome de *condição invariante de classe* ou CIC. A classe dos racionais possui uma condição invariante de classe que passa por exigir que os atributos `numerador` e `denominador` sejam o numerador e o denominador da fracção canónica representativa do racional correspondente, i.e.,

$$CIC \equiv 0 < \texttt{denominador} \land \mathrm{mdc}(\texttt{numerador}, \texttt{denominador}).$$

A vantagem da definição de uma condição invariante de classe é que todos os métodos correspondentes a operações públicas bem como todas as rotinas amigas da classe C++ (que fazem parte da interface da classe com o consumidor, Secção 7.15) poderem admitir que os atributos das variáveis da classe C++ com que trabalham verificam inicialmente a condição, o que normalmente os simplifica bastante. I.e., a condição invariante de classe pode ser vista como parte da pré-condição quer de métodos correspondentes a operações públicas, quer de rotinas amigas da classe C++. Claro que, para serem "bem comportadas", as rotinas, membro e não membro, também devem garantir que a condição se verifica para todas as variáveis da classe C++ criadas ou alteradas por essas rotinas. Ou seja, a condição invariante de classe para cada instância da classe criada ou alterada pelas mesmas rotinas pode ser vista também como parte da sua condição objectivo.

Tal como sucedia nos ciclos, em que durante a execução do passo a condição invariante muitas vezes não se verificava, embora se verificasse garantidamente antes e após o passo, também a condição invariante de classe pode não se verificar durante a execução dos métodos públicos ou das rotinas amigas da classe C++ em causa, embora se verifique garantidamente no seu início e no seu final. Durante os períodos em que a condição invariante de classe não é verdadeira, pode ser conveniente invocar alguma rotina auxiliar, que portanto terá de lidar com instâncias que não verificam a condição invariante de classe e que poderá também não garantir que a mesma condição se verifica para as instâncias por si criadas ou alteradas. Essas rotinas "mal comportadas" devem ser privadas, de modo a evitar utilizações erróneas por parte do consumidor final da classe C++ que coloquem alguma instância num estado inválido.

A definição de uma condição invariante de classe e a sua imposição à entrada e saída dos métodos públicos e de rotinas amigas de uma classe C++ não passa de um esforço inútil se as suas variáveis membro forem públicas, i.e., se o seu *estado* for alterável do exterior. Se o forem, o consumidor da classe C++ pode alterar o estado de uma variável da classe, por engano ou maliciosamente, invalidando a condição invariante de classe, com consequências potencialmente dramáticas no comportamento da classe C++ e no programa no seu todo. Essas consequências são normalmente graves, porque as rotinas que lidam com as variáveis membro da classe assumem que estas verificam a condição invariante de classe, não fazendo quaisquer garantias acerca do seu funcionamento quando ela não se verifica.

De todas as operações de uma classe C++, as mais importantes são porventura as operações construtoras[10]. São estas que garantem que as instâncias são criadas verificando imediatamente a condição invariante de classe. A sua importância pode ser vista na classe `Racional`, em que os construtores garantem, desde que as respectivas pré-condições sejam respeitadas, que a condição invariante da classe se verifica para as instâncias construídas.

Finalmente, é de notar que algumas classes C++ não têm condição invariante de classe. Tais classes C++ não são normalmente concretizações de nenhum TAD, sendo meros agregados de

[10]Note-se que construtor e operação construtora não significam forçosamente a mesma coisa. A noção de operação construtora é mais geral, e refere-se a qualquer operação que construa novas variáveis da classe C++. É claro que os construtores são operações construtoras, mas uma função membro pública que devolva um valor da classe C++ em causa também o é.

informação. É o caso, por exemplo, de um agregado que guarde nome e morada de utentes de um serviço qualquer. Essas classes C++ têm normalmente todas as suas variáveis membro públicas, e por isso usam normalmente a palavra-chave `struct` em vez de `class`. Note-se que estas palavras chave são quase equivalentes, pelo que a escolha de `class` ou `struct` é meramente convencional, escolhendo-se `class` para classes C++ que sejam concretizações de TAD ou classes propriamente ditas, e `struct` para classes C++ que sejam meros agregados de informação. A única diferença entre as palavras chave `struct` e `class` é que, com a primeira, todos os membros são públicos por omissão, enquanto com a segunda todos os membros são privados por omissão.

7.4.4 Porquê o formato canónico das fracções?

Qual a vantagem de manter todas as fracções que representam os racionais no seu formato canónico? I.e., qual a vantagem de impor

$$0 < \texttt{denominador} \wedge \mathrm{mdc}(\texttt{numerador}, \texttt{denominador})$$

como condição invariante de classe C++?

A verdade é que esta condição poderia ser consideravelmente relaxada: para o programador consumidor, a representação interna dos racionais é irrelevante, muito embora ele espere que a operação `escreve()` resulte numa representação canónica dos racionais. Logo, o problema poderia ser resolvido alterando apenas o método `escreve()`, de modo a reduzir a fracção, deixando o restante código de se preocupar com a questão. Ou seja, poder-se-ia relaxar a condição invariante de classe para

$$\texttt{denominador} \neq 0.$$

No entanto, a escolha de uma condição invariante de classe mais forte trará algumas vantagens.

A primeira vantagem tem a ver com a unicidade de representação garantida pela condição invariante de classe escolhida: a cada racional corresponde uma e uma só representação na forma de uma fracção canónica. Dessa forma é muito fácil comparar dois racionais: dois racionais são iguais se e só se as correspondentes fracções canónicas tiverem o mesmo numerador e o mesmo denominador.

A segunda vantagem tem a ver com as limitações dos tipos básicos do C++.

Sendo os valores do tipo `int` limitados em C++, como se viu no Capítulo 2, a utilização de uma representação em fracções não-canónicas põe alguns problemas graves de implementação. O primeiro tem a ver com a facilidade com que permite realizar algumas operações. Por exemplo, é muito fácil verificar a igualdade de dois racionais comparando simplesmente os seus numeradores e denominadores, coisa que só é possível fazer directamente se se garantir que as fracções que os representam estão no formato canónico. O segundo problema tem a ver com as limitações dos inteiros. Suponha-se o seguinte código:

```
int main()
{
    Racional x(50000, 50000), y(1, 50000);
    Racional z = x.soma(y);
    z.escreve();
    cout << endl;
}
```

No ecrã deveria aparecer

```
1/50000
```

Não se usando uma representação em fracções canónicas, ao se calcular o denominador do resultado, i.e., ao se multiplicar os dois denominadores, obtém-se $50000 \times 50000 = 2500000000$. Em máquinas em que os `int` têm 32 *bits*, esse valor não é representável, pelo que se obtém um valor errado (em Linux i386 obtém-se -1794967296), apesar de a fracção resultado ser perfeitamente representável! Este problema pode ser mitigado se se trabalhar sempre com fracções no formato canónico.

Mesmo assim, o problema não é totalmente resolvido. Suponha-se o seguinte código:

```
int main()
{
    Racional x(1, 50000), y(1, 50000);
    Racional z = x.soma(y);
    z.escreve();
    cout << endl;
}
```

No ecrã deveria aparecer

```
1/25000
```

mas ocorre exactamente o mesmo problema que anteriormente. É pois desejável não só usar uma representação canónica para os racionais, como também tentar garantir que os resultados de cálculos intermédios são tão pequenos quanto possível. Este assunto será retomado mais tarde (Secção 7.13).

7.4.5 Explicitação da condição invariante de classe

A condição invariante de classe é útil não apenas como uma ferramenta formal que permite verificar o correcto funcionamento de, por exemplo, um método. É útil como ferramenta de detecção de erros. Da mesma forma que é conveniente explicitar pré-condições e condições objectivo das rotinas através de instruções de asserção, também o é no caso da condição invariante de classe. A intenção é detectar as violações dessa condição durante a execução do programa e abortá-lo se alguma violação for detectada[11].

A condição invariante de classe é claramente uma noção de implementação: refere-se sempre aos atributos (que se presume serem privados) de uma classe. Uma das vantagens de se estabelecer esta distinção clara entre interface e implementação está em permitir alterações substanciais na implementação sem que a interface mude. De facto, é perfeitamente possível que o programador produtor mude substancialmente a implementação de uma classe C++ sem que isso traga qualquer problema para o programador consumidor, que se limita a usar a interface da classe C++. A mudança da implementação de uma classe implica normalmente uma alteração da condição invariante de classe, mas não do comportamento externo da classe. É por isso muito importante que pré-condição e condição objectivo de cada operação/método sejam claramente factorizadas em condições que dizem respeito apenas à implementação, e que devem corresponder à condição invariante de classe, e condições que digam respeito apenas ao comportamento externo da operação. Dito por outras palavras, apesar de do ponto de vista da implementação a condição invariante de classe fazer parte da pré-condição e da condição objectivo de todas as operações/métodos, como se disse na secção anterior, é preferível "pô-la em evidência", documentando-a claramente à parte das operações e métodos, e excluindo-a da documentação/contrato de cada operação. Ou seja, a condição invariante de classe fará parte do contrato de cada *método* (ponto de vista da implementação), mas não fará parte do contrato da correspondente operação (ponto de vista externo, da interface).

Quando a condição invariante de classe é violada, de quem é a culpa? Nesta altura já não deverão subsistir dúvidas: a culpa é do programador produtor da classe:

1. Violação da condição invariante de classe: culpa do programador produtor da classe.
2. Violação da pré-condição de uma operação: culpa do programador consumidor da classe.
3. Violação da condição objectivo de uma operação: culpa do programador produtor do respectivo método.

Como explicitar a condição invariante de classe? É apenas uma questão de usar instruções de asserção e comentários de documentação apropriados. Para simplificar, é conveniente definir uma operação privada da classe, chamada convencionalmente `cumpreInvariante()`, que devolve o valor lógico $\mathcal{V}$ se a condição invariante de classe se cumprir e falso no caso contrário.

```
/** Descrição da classe Classe.
    @invariant CIC.
class Classe {
  public:

    ...
```

[11] Mais tarde se verá que, dependendo da aplicação em desenvolvimento, abortar o programa em caso de erro de programação pode ou não ser apropriado.

```
        /** Descrição da operação operação().
              @pre PC.
              @post CO. */
        tipo operação(parâmetros);

    private:

        ...

        /** Descrição da operação operação_privada().
              @pre PC.
              @post CO. */
        tipo operação_privada(parâmetros);


        /** Indica se a condição invariante de classe (CIC) se verifica.
              @pre *this = v.
              @post cumpreInvariante = CIC ∧ *this = v. */
        bool cumpreInvariante();
};

...

// Implementação da operação operação(): método.
tipo Classe::operação(parâmetros)
{
    assert(cumpreInvariante() [and v.cumpreInvariante()]...);
    assert(PC);

    ... // Implementação.

    assert(cumpreInvariante() [and v.cumpreInvariante()]...);
    assert(CO);

    return ...;
}

...

bool Classe::cumpreInvariante()
{
    return CIC.
}

// Implementação da operação operação(): método.
tipo Classe::operação_privada(parâmetros)
```

```
{
    assert(PC);

    ... // Implementação.

    assert(CO);

    return ...;
}
```

São de notar os seguintes pontos importantes:

- A condição invariante de classe foi incluída na documentação da classe, que é parte da sua interface, apesar de antes se ter dito que esta condição era essencialmente uma questão de implementação. É de facto infeliz que assim seja, mas os programas que extraem automaticamente a documentação de uma classe (e.g., Doxygen) requerem este posicionamento[12].
- A documentação das operações não inclui a condição invariante de classe, visto que esta foi "posta em evidência", ficando na documentação da classe.
- A implementação das operações, i.e., o respectivo método, inclui instruções de asserção para verificar a condição invariante de classe para todas as instâncias da classe em jogo (que incluem a instância implícita[13], parâmetros, instâncias locais ao método, etc.), quer no início do método, quer no seu final.
- As instruções de asserção para verificar a veracidade da condição invariante de classe são anteriores quer à instrução de asserção para verificar a pré-condição da operação, quer à instrução de asserção para verificar a condição objectivo da operação. Esse posicionamento é importante, pois as verificações da pré-condição e da condição objectivo podem obrigar à invocação de outras operações públicas da classe, que por sua vez verificam a condição invariante de classe: se a ordem fosse outra, o erro surgiria durante a execução dessas outras operações.
- Separaram-se as instruções de asserção relativas a pré-condições, condições objectivo e condições invariantes de classe, de modo a ser mais óbvia a razão do erro no caso de o programa abortar.
- A função membro privada `cumpreInvariante()` não tem qualquer instrução de asserção. Isso deve-se ao facto de ter sempre pré-condição $\mathcal{V}$ e de poder operar sobre variáveis implícitas que não verificam a condição invariante de classe (como é óbvio, pois serve justamente para indicar se essa condição se verifica).

[12] Parece haver aqui uma contradição. Não será toda a documentação parte da interface? A resposta é simplesmente "não". Para uma classe, podem-se gerar três tipos de documentação. A primeira diz respeito de facto à interface, e inclui todos os membros públicos: é a documentação necessária ao programador consumidor. A segunda diz respeito à categoria de acesso `protected` e deixar-se-á para mais tarde. A terceira diz respeito à implementação, e inclui os membros de todas as categorias de acesso: é a documentação necessária ao programador produtor ou, pelo menos, à "assistência técnica", i.e., aos programadores que farão a manutenção do código existente. Assim, a condição invariante de classe deveria ser parte apenas da documentação de implementação.

[13] Excepto para operações de classe.

- Os métodos privados não têm instruções de asserção para a condição invariante de classe, pois podem ser invocados por outros métodos em instantes de tempo durante os quais as instâncias da classe (instância implícita, parâmetros, etc.) não verifiquem essa condição.

Aplicando estas ideias à classe `Racional` em desenvolvimento obtém-se:

```
#include <iostream>
#include <cassert>

using namespace std;

/** Devolve o máximo divisor comum dos inteiros passados como argumento.
    @pre m = m ∧ n = n.
    @post mdc = { mdc(m, n)   m ≠ 0 ∨ n ≠ 0
                { 1           m = 0 ∧ n = 0 . */
int mdc(int m, int n)
{
    ...
}
/** Representa números racionais.
    @invariant 0 < denominador ∧ mdc(numerador, denominador) = 1. */
class Racional {
  public:
    /** Constrói racional com valor inteiro. Construtor por omissão.
        @pre V.
        @post *this = n. */
    Racional(int const n = 0);
    /** Constrói racional correspondente a n/d.
        @pre d ≠ 0.
        @post *this = n/d. */
    Racional(int const n, int const d);

   /** Escreve o racional no ecrã no formato de uma fracção.
        @pre *this = r.
        @post *this = r∧(¬cout∨ cout contém n/d (ou simplesmente n se d = 1)
              sendo n/d a fracção canónica correspondente ao racional *this). */
    void escreve(); // Declaração da rotina membro: operação.

    /** Devolve a soma com o racional recebido como argumento.
        @pre *this = r.
        @post *this = r ∧ somaCom = *this + r2. */
    Racional somaCom(Racional const r2);

    /** Lê do teclado um novo valor para o racional, na forma de dois inteiros sucessi-
vos.
```

```
        @pre *this = r.
        @post Se cin e cin tem dois inteiros n' e d' disponíveis para lei-
tura, com d' ≠ 0, então
                *this = n'/d' ∧ cin,
            senão
                *this = r ∧ ¬cin. */
    void lê();

  private:
    int numerador;
    int denominador;
    /** Reduz a fracção que representa o racional.
        @pre denominador ≠ 0 ∧ *this = r.
        @post denominador ≠ 0 ∧ mdc(numerador, denominador) = 1∧
              *this = r. */
    void reduz();

    /** Indica se a condição invariante de classe se verifica.
        @pre *this = r.
        @post *this = r ∧ cumpreInvariante = (0 <
denominador ∧ mdc(numerador, denominador) = 1). */
    bool cumpreInvariante();
};

Racional::Racional(int const n)
    : numerador(n), denominador(1)
{

    assert(cumpreInvariante());
    assert(numerador == n * denominador);
}

Racional::Racional(int const n, int const d)
    : numerador(d < 0 ? -n : n),
      denominador(d < 0 ? -d : d)
{
    assert(d != 0);

    reduz();

    assert(cumpreInvariante());
    assert(numerador * d == n * denominador);
}

void Racional::escreve()
{
```

```
    assert(cumpreInvariante());

    cout << numerador;
    if(denominador != 1)
        cout << '/' << denominador;

    assert(cumpreInvariante());
}

Racional Racional::somaCom(Racional const r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    Racional r;

    r.numerador = numerador * r2.denominador +
        r2.numerador * denominador;
    r.denominador = denominador * r2.denominador;

    r.reduz();

    assert(cumpreInvariante() and r.cumpreInvariante());

    return r;
}

void Racional::lê()
{
    assert(cumpreInvariante());

    int n, d;

    if(cin >> n >> d)
        if(d == 0)
            cin.setstate(ios_base::failbit);
        else {
            numerador = d < 0 ? -n : n;
            denominador = d < 0 ? -d : d;

            reduz();

            assert(cumpreInvariante());
            assert(numerador * d == n * denominador and cin);

            return;
        }
```

```
    assert(cumpreInvariante());
    assert(not cin);
}

void Racional::reduz()
{
    assert(denominador != 0);

    int k = mdc(numerador, denominador);

    numerador /= k;
    denominador /= k;

    assert(denominador != 0 and mdc(numerador, denominador) == 1);
}

bool Racional::cumpreInvariante()
{
    return 0 < denominador and mdc(numerador, denominador) == 1;
}

int main()
{
    // Ler fracções:
    cout << "Introduza duas fracções (numerador denominador): ";
    Racional r1, r2;
    r1.lê();
    r2.lê();

    if(not cin) {
        cerr << "Opps!  A leitura dos racionais falhou!" << endl;
        return 1;
    }

    // Calcular racional soma:
    Racional r = r1.somaCom(r2);

    // Escrever resultado:
    cout << "A soma de ";
    r1.escreve();
    cout << " com ";
    r2.escreve();
    cout << " é ";
    r.escreve();
    cout << '.' << endl;
}
```

Note-se que o compilador se encarrega de garantir que algumas instâncias não mudam de valor durante a execução de um método. É o caso das constantes. É evidente, pois, que se essas constantes cumprem inicialmente a condição invariante de classe, também a cumprirão no final no método, pelo que se pode omitir a verificação explícita através de uma instrução de asserção, tal como se fez para o método `somaCom()`.

A Figura 7.5 mostra a notação usada para representar a condição invariante da classe C++ `Racional`, bem como a pré-condição e a condição objectivo da operação `Racional::somaCom()`.

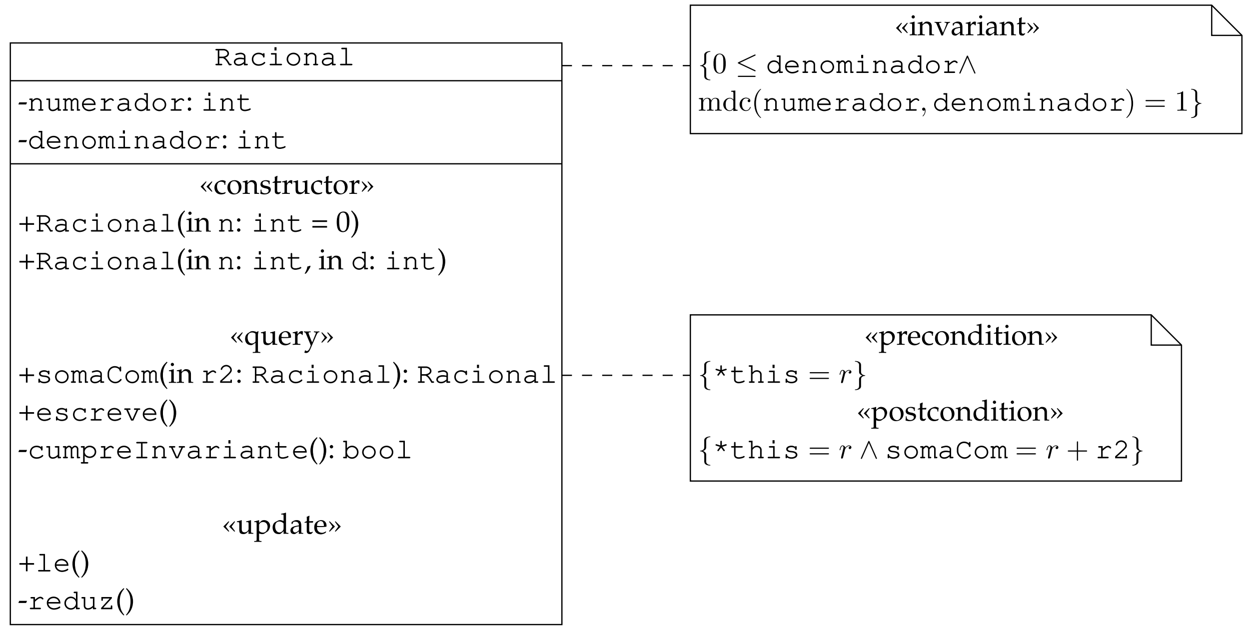


Figura 7.5: A classe C++ `Racional` agora também com condição invariante de instância, pré-condição e condição objectivo indicadas para a operação `Racional::somaCom()`.

7.5 Sobrecarga de operadores

Tal como definida, a classe C++ `Racional` obriga o consumidor a usar uma notação desagradável e pouco intuitiva para fazer operações com racionais. Como se viu, seria desejável que a função `main()`, no programa em desenvolvimento, se pudesse escrever simplesmente como:

```
int main()
{
    // Ler fracções:
    cout << "Introduza duas fracções (numerador denominador): ";
    Racional r1, r2;
    cin >> r1 >> r2;

    if(not cin) {
```

```
        cerr << "Opps!  A leitura dos racionais falhou!" << endl;
        return 1;
    }

    // Calcular racional soma:
    Racional r = r1 + r2;

    // Escrever resultado:
    cout << "A soma de " << r1 << " com " << r2
         << " é " << r << '.' << endl;
}
```

Se se pudesse escrever o programa como acima, claramente a classe `Racional`, uma vez equipada com os restantes operadores dos tipos aritméticos básicos, passaria a funcionar para o consumidor como qualquer outro tipo básico do C++: seria verdadeiramente um *tipo de primeira categoria*.

O C++ possibilita a sobrecarga dos operadores (`+`, `-`, `*`, `/`, `==`, etc.) de modo a poderem ser utilizados com TAD concretizados pelo programador na forma de classes C++. A solução para o problema passa então pela sobrecarga dos operadores do C++ de modo a terem novos significados quando aplicados ao novo tipo `Racional`, da mesma forma que se tinha visto antes relativamente aos tipos enumerados (ver Secção 6.1). Mas, ao contrário do que se fez então, agora as funções de sobrecarga têm de ser membros da classe `Racional`, de modo a poderem aceder aos seus membros privados (alternativamente poder-se-iam usar funções membro amigas da classe, Secção 7.15). Ou seja, a solução é simplesmente alterar o nome da operação `Racional::somaCom()` de `somaCom` para `operator+`:

```
...
/** Representa números racionais.
    @invariant 0 < denominador ∧ mdc(numerador, denominador) = 1. */
class Racional {
  public:

    ...

    /** Devolve a soma com o racional recebido como argumento.
        @pre *this = r.
        @post *this = r ∧ operator+ = *this + r2. */
    Racional operator+(Racional const r2);

    ...

  private:

    ...
```

```
};

...

Racional Racional::operator+(Racional const r2)
{
    ...
}

...
```

Tal como acontecia com a expressão `r1.somaCom(r2)`, a expressão `r1.operator+(r1)` invoca a operação `operator+()` da classe C++ `Racional` usando `r1` como instância (variável) implícita. Só que agora é possível escrever a mesma expressão de uma forma muito mais clara e intuitiva:

```
r1 + r2
```

De facto, sempre que se sobrecarregam operadores usando operações, o primeiro operando (que pode ser o único no caso de operadores unários, i.e., só com um operando) é sempre a instância implícita durante a execução do respectivo método, sendo os restantes operandos passados como argumento à operação.

Se `@` for um operador binário (e.g., `+`, `-`, `*`, etc.), então a sobrecarga do operador `@` pode ser feita:

- Para uma classe C++ *`Classe`*, definindo uma operação *`tipo_de_devolução`* *`Classe`*`::operator@(`*`tipo_do_segundo_operando`*`)`. Numa invocação deste operador, o primeiro operando, obrigatoriamente do tipo *`Classe`*, é usado como instância implícita e o segundo operando é passado como argumento.
- Através de uma rotina não-membro *`tipo_de_devolução`* `operator@(`*`tipo_do_primeiro_operando`*`,` *`tipo_do_segundo_operando`*`)`. Numa invocação deste operador, ambos os operandos são passados como argumentos.

A expressão `a @ b` pode portanto ser interpretada como

```
a.operator@(b)
```

ou

```
operator@(a, b)
```

consoante o operador esteja definido como membro da classe a que `a` pertence ou esteja definido como rotina normal, não-membro.

Se `@` for um operador unário (e.g., `+`, `-`, `++` prefixo, etc.), então a sobrecarga do operador `@` pode ser feita:

- Para uma classe C++ `Classe`, definindo uma operação *`tipo_de_devolução`* *`Classe`*`::operator@()`. Numa invocação deste operador, o seu único operando, obrigatoriamente do tipo *`Classe`*, é usado como instância implícita.

- Através de uma rotina não membro *`tipo_de_devolução`* `operator@(`*`tipo_do_operando`*`)`.

A expressão `@a` (ou `a@` se `@` for sufixo) pode portanto ser interpretada como

```
a.operator@()
```

ou

```
operator@(a)
```

consoante o operador esteja definido como membro da classe a que `a` pertence ou esteja definido como rotina normal, não-membro.

É importante notar que:

1. Quando a sobrecarga de um operador se faz por intermédio de uma operação (rotina membro) de uma classe C++, o primeiro operando (e único no caso de uma operação unária) numa expressão que envolva esse operador *não sofre nunca conversões implícitas de tipo*. Em todos os outros casos as conversões implícitas são possíveis.

2. Nunca se deve alterar a semântica dos operadores. Imagine-se os problemas que traria sobrecarregar o operador + para a classe C++ `Racional` como significando o produto!

3. Nem todos os operadores podem ser sobrecarregados por intermédio rotinas não-membro. Os operadores = (atribuição), `[]` (indexação), `()` (invocação) e -> (selecção), só podem ser sobrecarregados por meio de operações (rotinas membro). Para todas as classes que não os redefinam, os operadores = (atribuição), `&` (unário, endereço de) e **,** (sequenciamento) são definidos implicitamente: por isso é possível atribuir instâncias de classes C++, como a classe `Racional`, sem para isso ter de sobrecarregar o operador de atribuição =).

Falta agora a tarefa algo penosa de sobrecarregar todos os operadores aplicáveis a racionais. Porquê? Porque, apesar de o programa da soma das fracções não necessitar senão dos operadores > > e < <, de extracção e inserção em canais, é instrutivo preparar a classe para utilizações futuras, ainda difíceis de antecipar.

Pretende-se, pois, equipar o TAD `Racional` com todos os operadores usuais para os tipos básicos do C++:

+, -, * e / Operadores aritméticos (binários): adição, subtracção, produto e divisão. Não têm efeitos laterais, i.e., não alteram os operandos.

+ e - Operadores aritméticos (unários): identidade e simétrico. Não têm efeitos laterais.

<, <=, >, >= Operadores relacionais (binários): menor, menor ou igual, maior e maior ou igual. Não têm efeitos laterais.

== e != Operadores de igualdade e diferença (binários). Não têm efeitos laterais.

++ e -- Operadores de incrementação e decrementação prefixo (unários). Têm efeitos laterais, pois alteram o operando. Aliás, são eles a sua principal razão de ser.

++ e -- Operadores de incrementação e decrementação sufixo (unários). Têm efeitos laterais.

+=, -=, *= e /= Operadores especiais de atribuição: adição e atribuição, subtracção e atribuição, produto e atribuição e divisão e atribuição (binários). Têm efeitos laterais, pois alteram o primeiro operando.

> > e < < Operadores de extracção e inserção de um canal (binários). Ambos alteram o operando esquerdo (que é um canal), mas apenas o primeiro altera o operando direito. Têm efeitos laterais.

7.6 Testes de unidade

Na prática, não é fácil a decisão de antecipar ou não utilizações futuras. Durante o desenvolvimento de uma classe deve-se tentar suportar utilizações futuras, difíceis de antecipar, ou deve-se restringir o desenvolvimento àquilo que é necessário em cada momento? Se o objectivo é preparar uma biblioteca de ferramentas utilizáveis por qualquer programador, então claramente devem-se tentar prever as utilizações futuras. Mas, se a classe está a ser desenvolvida para ser utilizada num projecto em particular, a resposta cai algures no meio destas duas opções. É má ideia, de facto, gastar esforço de desenvolvimento[14] a desenvolver ferramentas de utilização futura mais do que dúbia. Mas também é má ideia congelar o desenvolvimento de tal forma que aumentar as funcionalidades de uma classe C++, logo que tal se revele necessário, seja difícil. O ideal, pois, está em não desenvolver prevendo utilizações futuras, mas em deixar a porta aberta para futuros desenvolvimentos.

A recomendação anterior não se afasta muito do preconizado pela metodologia de desenvolvimento *eXtreme Programming* [1]. Uma excelente recomendação dessa metodologia é também o desenvolvimento dos chamados *testes de unidade*. Se se olhar com atenção para a definição da classe C++ `Racional` definida até agora, conclui-se facilmente que a maior parte das condições objectivo das operações não são testadas usando instruções de asserção. O problema é que a condição objectivo das operações está escrita em termos da noção matemática de número racional, e não é fácil fazer a ponte entre uma noção matemática e o código C++... Por exemplo, como explicitar em código a condição objectivo do operador + para racionais? Uma primeira tentativa poderia ser a tradução directa:

```
/** Devolve a soma com o racional recebido como argumento.
    @pre *this = r.
    @post *this = r ∧ operator+ = *this + r2. */
```

[14] A não ser para efeitos de estudo e desenvolvimento pessoal, claro.

```
Racional Racional::operator+(Racional const r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    Racional r;

    r.numerador = numerador * r2.denominador +
        r2.numerador * denominador;
    r.denominador = denominador * r2.denominador;

    r.reduz();

    assert(cumpreInvariante() and r.cumpreInvariante());
    assert(r == *this + r2);

    return r;
}
```

Há dois problemas neste código. O primeiro é que o operador == ainda não está definido. Este problema resolver-se-á facilmente mais à frente neste capítulo. O segundo é muito mais importante: a asserção, tal como está escrita, recorre recursivamente ao próprio operador +! Claramente, o caminho certo não passa por aqui.

Os testes de unidade proporcionam uma alternativa interessante às instruções de asserção para as condições objectivo das operações. A ideia é que se deve escrever um conjunto exaustivo de testes para as várias operações da unidade e mantê-los durante toda a vida do código. Por unidade entende-se aqui uma unidade de modularização, tipicamente uma classe C++ e rotinas associadas que concretizam um TAD ou uma classe propriamente dita. Os testes de unidade só muito parcialmente substituem as instruções de asserção para as condições objectivo das operações da classe:

1. As instruções de asserções estão sempre activas e verificam a validade da condição objectivo sempre que o operação é invocada. Por outro lado, os testes de unidade apenas são executados de tempos e tempos, e de uma forma independente à do programa, ou dos programas, no qual a unidade testada está integrada.

2. As instruções de asserção verificam a validade da condição objectivo para todos os casos para os quais o programa, ou os programas, invocam a respectiva operação da classe C++. No caso dos testes de unidade, no entanto, é impensável testar exaustivamente as operações em causa.

3. As instruções de asserção estão activas durante o desenvolvimento e durante a exploração do programa desenvolvido[15], enquanto os testes de unidade são executados de tempos a tempos, durante o desenvolvimento ou manutenção do programa.

[15]Uma das características das instruções de asserção é que pode ser desactivadas facilmente, bastando para isso definir a macro `NDEBUG`. No entanto, não é muito boa ideia desactivar as instruções de asserção. Ver discussão sobre o assunto no !! citar capítulo sobre tratamento de erros.

Justificados que foram os testes de unidade, pode-se agora criar o teste de unidade para o TAD `Racional`:

```
#ifdef TESTE

#include <fstream>

/** Programa de teste do TAD Racional e da função mdc(). */
int main()
{
    assert(mdc(0, 0) == 1);
    assert(mdc(10, 0) == 10);
    assert(mdc(0, 10) == 10);
    assert(mdc(10, 10) == 10);
    assert(mdc(3, 7) == 1);
    assert(mdc(8, 6) == 2);
    assert(mdc(-8, 6) == 2);
    assert(mdc(8, -6) == 2);
    assert(mdc(-8, -6) == 2);

    Racional r1(2, -6);

    assert(r1.numerador() == -1 and r1.denominador() == 3);

    Racional r2(3);

    assert(r2.numerador() == 3 and r2.denominador() == 1);

    Racional r3;

    assert(r3.numerador() == 0 and r2.denominador() == 1);

    assert(r2 == 3);
    assert(3 == r2);
    assert(r3 == 0);
    assert(0 == r3);

    assert(r1 < r2);
    assert(r2 > r1);
    assert(r1 <= r2);
    assert(r2 >= r1);
    assert(r1 <= r1);
    assert(r2 >= r2);

    assert(r2 == +r2);
    assert(-r1 == Racional(1, 3));
```

```
    assert(++r1 == Racional(2, 3));
    assert(r1 == Racional(2, 3));

    assert(r1++ == Racional(2, 3));
    assert(r1 == Racional(5, 3));
    assert((r1 *= Racional(7, 20)) == Racional(7, 12));
    assert((r1 /= Racional(3, 4)) == Racional(7, 9));
    assert((r1 += Racional(11, 6)) == Racional(47, 18));
    assert((r1 -= Racional(2, 18)) == Racional(5, 2));

    assert(r1 + r2 == Racional(11, 2));
    assert(r1 - Racional(5, 7) == Racional(25, 14));
    assert(r1 * 40 == 100); assert(30 / r1 == 12);

    ofstream saída("teste");
    saída << r1 << ' ' << r2;
    saída.close();

    ifstream entrada("teste");
    Racional r4, r5;
    entrada >> r4 >> r5;

    assert(r1 == r4);
    assert(r2 == r5);
}

#endif // TESTE
```

São de notar os seguintes pontos:

- Alguma da sintaxe utilizada neste teste só será introduzida mais tarde. O leitor deve regressar a este teste quando o TAD `Racional` for totalmente desenvolvido.
- Cada teste consiste essencialmente numa instrução de asserção. Há melhores formas de escrever os testes de unidade, sem recorrer a asserções, nomeadamente recorrendo a bibliotecas de teste. Mas tais bibliotecas estão fora do âmbito deste texto.
- O teste consiste numa função `main()`. De modo a não entrar em conflito com a função `main()` do programa propriamente dito, envolveu-se a função `main()` de teste entre duas directivas de pré-compilação, `#ifdef TESTE` e `#endif // TESTE`. Isso faz com que toda a função só seja levada em conta pelo compilador quando estiver definida a macro `TESTE` (coisa que num compilador em Linux se consegue tipicamente com a opção de compilação `-DTESTE`). Este assunto será visto com rigor no Capítulo 9, onde se verá também como se pode preparar um TAD como o tipo `Racional` para ser utilizado em qualquer programa onde seja necessário trabalhar com racionais.

7.7 Devolução por referência

Começar-se-á o desenvolvimento dos operadores para o TAD `Racional` pelo operador de incrementação prefixo. Uma questão nesse desenvolvimento é saber o que é que devolve esse operador e, de uma forma mais geral, todos os operadores de incrementação e decrementação prefixos e especiais de atribuição.

7.7.1 Mais sobre referências

Na Secção 3.2.11 viu-se que se pode passar um argumento por referência a um procedimento se este tiver definido o parâmetro respectivo como uma referência. Por exemplo,

```
void troca(int& a, int& b)
{
    int auxiliar = a;
    a = b;
    b = auxiliar;
}
```

é um procedimento que troca os valores das duas variáveis passadas como argumento. Este procedimento pode ser usado no seguinte troço de programa

```
int x = 1, y = 2;
troca(x, y);
cout << x << ' ' << y << endl;
```

que mostra

```
2 1
```

no ecrã.

O conceito de referência pode ser usado de formas diferentes. Por exemplo,

```
int i = 1;
int& j = i; // a partir daqui j é sinónimo de i!
j = 3;
cout << i << endl;
```

mostra

```
3
```

no ecrã, pois alterar a variável `j` é o mesmo que alterar a variável `i`, já que `j` é sinónimo de `i`.

As variáveis que são referências, caso de `j` no exemplo anterior e dos parâmetros `a` e `b` do procedimento `troca()`, têm de ser inicializadas com a variável de que virão a ser sinónimos. Essa inicialização é feita explicitamente no caso de `j`, e implicitamente no caso das variáveis `a` e `b`, neste caso através da passagem de `x` e `y` como argumento na chamada de `troca()`.

Necessidade da devolução por referência

Suponha-se o código:

```
int i = 0;
++(++i);
cout << i << endl;
```

(Note-se que este código é muito pouco recomendável! Só que, como a sobrecarga dos operadores deve manter a mesma semântica que esses mesmos operadores possuem para os tipos básicos, é necessário conhecer bem "os cantos à casa", mesmo os mais obscuros, infelizmente.)

Este código resulta na dupla incrementação da variável `i`, como seria de esperar. Mas para isso acontecer, o operador `++`, para além de incrementar a variável `i`, tem de devolver a própria variável `i`, e não uma sua cópia, pois de outra forma a segunda aplicação do operador `++` levaria à incrementação da cópia, e não do original.

Para que este assunto fique mais claro, começar-se-á por escrever um procedimento `incrementa()` com o mesmo objectivo do operador de incrementação. Como este procedimento deve afectar a variável passada como argumento, neste caso `i`, deve receber o argumento por referência:

```
/** Incrementa o inteiro recebido como argumento e devolve-o.
    @pre i = i.
    @post i = i + 1. */
void incrementa(int& v)
{
    v = v + 1;
}
...
int i = 0;
incrementa(incrementa(i));
cout << i << endl;
```

Infelizmente este código não compila, pois a invocação mais exterior do procedimento recebe como argumento o resultado da primeira invocação, que é `void`. Logo, é necessário devolver um inteiro nesta rotina:

```
         /** Incrementa o inteiro recebido como argumento e devolve-o.
             @pre i = i.
             @post incrementa = i ∧ i = i + 1. */
         int incrementa(int& v)
         {
/* 1 */      v = v + 1;
/* 2 */      return v;
         }
         ...
```

```
        int i = 0;
/* 3 */            incrementa(i)
/* 4 */ incrementa(            );
/* 5 */ cout << i << endl;
```

Este código tem três problemas. O primeiro problema é que, dada a definição actual da linguagem, não compila, pois o valor temporário devolvido pela primeira invocação da rotina não pode ser passado por referência (não-constante) para a segunda invocação. A linguagem C++ proíbe a passagem por referência (não-constante) de valores, ou melhor, de variáveis temporárias[16]. O segundo problema é que, ao contrário do que se recomendou no capítulo sobre modularização, esta rotina não é um procedimento, pois devolve alguma coisa, nem uma função, pois afecta um dos seus argumentos. Note-se que continua a ser indesejável escrever este tipo de código. Mas a emulação do funcionamento do operador ++, que é um operador com efeitos laterais, obriga à utilização de uma função com efeitos laterais... O terceiro problema, mais grave, é que, mesmo que fosse possível a passagem de uma variável temporária por referência, o código acima ainda não faria o desejado, pois nesse caso a segunda invocação da rotina `incrementa()` acabaria por alterar apenas essa variável temporária, e não a variável `i`, como se pode ver na Figura 7.6. Para resolver este problema, a rotina deverá devolver não uma cópia de `i`, mas a própria variável `i`, que é como quem diz, um sinónimo da variável `i`. Ou seja, a rotina deverá devolver `i` por referência.

```
        /** Incrementa o inteiro recebido como argumento e devolve-o.
            @pre i = i.
            @post incrementa ≡ i ∧ i = i + 1. */
        int& incrementa(int& v)
        {
/* 1 */     v = v + 1;
/* 2 */     return v; // ou simplesmente return v = v + 1;
        }
        ...
        int i = 0;
/* 3 */            incrementa(i)
/* 4 */ incrementa(            );
/* 5 */ cout << i << endl;
```

Esta versão da rotina `incrementa()` já leva ao resultado pretendido, usando para isso uma *devolução por referência*. Repare-se na condição objectivo desta rotina e compare-se com a usada para a versão anterior da mesma `rotina()`, em que se devolvia por valor: neste caso é necessário dizer que o que se devolve é não apenas igual a `i`, mas também é *idêntico* a `i`, ou seja, é o próprio `i`, como se pode ver na Figura 7.7[17]. Para isso usou-se o símbolo $\equiv$ em vez do usual $=$.

Para se compreender bem a diferença entre a devolução por valor e devolução por referência, comparem-se as duas rotinas abaixo:

[16] Esta restrição é razoavelmente arbitrária, e está em discussão a sua possível eliminação numa próxima versão da linguagem C++.

[17] É necessário clarificar a diferença entre igualdade e identidade. Pode-se dizer que dois gémeos são iguais, mas não que são idênticos, pois são indivíduos diferentes. Por outro lado, pode-se dizer que Fernando Pessoa e Alberto Caeiro não são apenas iguais, mas também idênticos, pois são nomes que se referem à mesma pessoa.

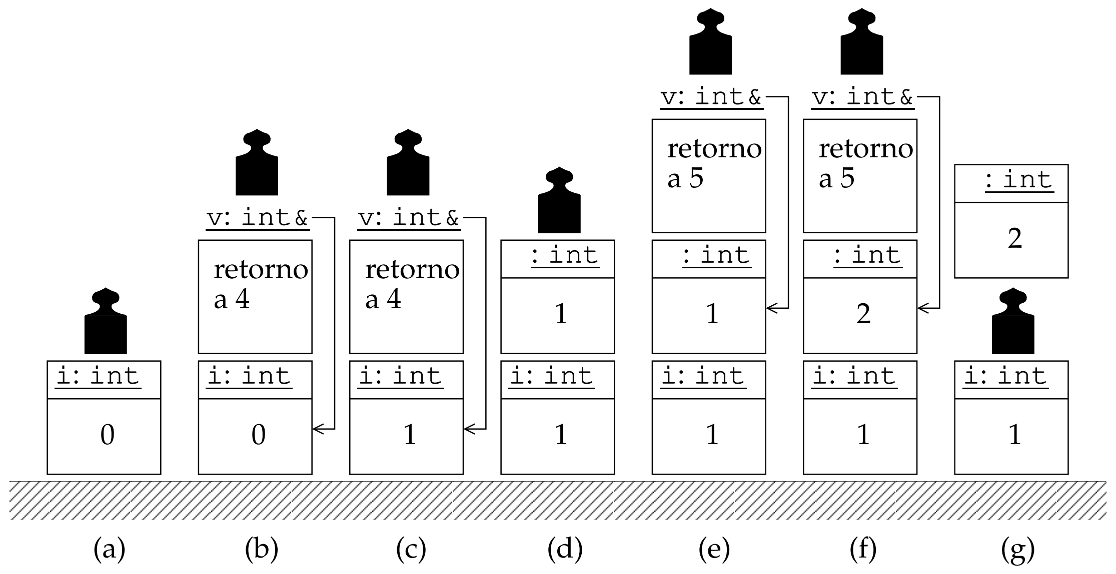


Figura 7.6: Estado da pilha durante a execução. (a) Imediatamente antes das invocações nas linhas 3 e 4; (b) imediatamente antes da instrução 1, depois de invocada a rotina pela primeira vez, já com o parâmetro `v` na pilha; (c) entre as instruções 1 e 2, já depois de incrementado `v` e portanto `i` (pois `v` é sinónimo de `i`); (d) imediatamente após a primeira invocação da rotina e imediatamente antes da sua segunda invocação, já com os parâmetros retirados da pilha, sendo de notar que o valor devolvido está guardado numa variável temporária, sem nome, no topo da pilha (a variável está abaixo do "pisa-papeis" que representa o topo da pilha, e não acima, como é habitual com os valores devolvidos, por ir ser construída uma referência para alea, que obriga a que seja preservada mais tempo); (e) imediatamente antes da instrução 1, depois de invocada a rotina pela segunda vez; (f) entre as instruções 1 e 2, já depois de incrementado `v` e portanto já depois de incrementada a variável temporária, de que `v` é sinónimo; e (g) imediatamente antes da instrução 5, depois de a rotina retornar. Vê-se claramente que a variável incrementada da segunda vez não foi a variável `i`, como se pretendia, mas uma variável temporária, entretanto destruída.

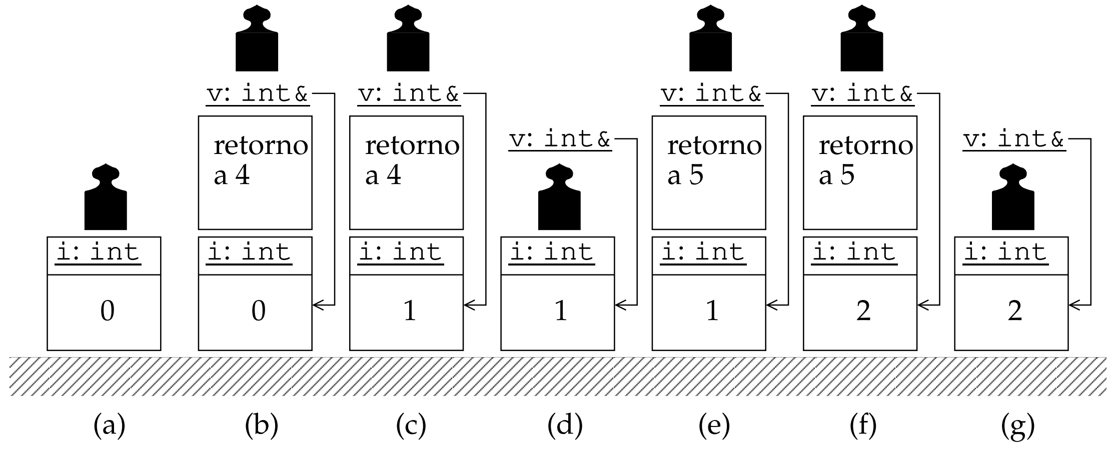


Figura 7.7: Estado da pilha durante a execução. (a) Imediatamente antes das invocações nas linhas 3 e 4; (b) imediatamente antes da instrução 1, depois de invocada a rotina pela primeira vez, já com o parâmetro `v` na pilha; (c) entre as instruções 1 e 2, já depois de incrementado `v` e portanto `i` (pois `v` é sinónimo de `i`); (d) imediatamente após a primeira invocação da rotina e imediatamente antes da sua segunda invocação, já com os parâmetros retirados da pilha, sendo de notar que a referência devolvida se encontra no topo da pilha; (e) imediatamente antes da instrução 1, depois de invocada a rotina pela segunda vez, tendo a referência `v` sido inicializada à custa da referência no topo da pilha, e portanto sendo `v` sinónimo de `i`; (f) entre as instruções 1 e 2, já depois de incrementado `v` e portanto já depois de incrementada a variável `i` pela segunda vez; e (g) imediatamente antes da instrução 5, depois de a rotina retornar. Vê-se claramente que a variável incrementada da segunda vez foi exactamente a variável `i`, como se pretendia.

```
int cópia(int v)
{
    return v;
}

int& mesmo(int& v)
{
    return v;
}
```

A primeira rotina devolve uma cópia do parâmetro `v`, que por sua vez é já uma cópia do argumento passado à rotina. Ou seja, devolve uma cópia do argumento, coisa que aconteceria mesmo que o argumento fosse passado por referência. A segunda rotina, pelo contrário, recebe o seu argumento por referência e devolve o seu parâmetro também por referência. Ou seja, o que é devolvido é um sinónimo do próprio parâmetro `v`, que por sua vez é um sinónimo do argumento passado à rotina. Ou seja, a rotina devolve um sinónimo do argumento. Uma questão filosófica é que esse sinónimo ... não tem nome! Ou melhor, é a própria expressão de invocação da rotina que funciona como sinónimo do argumento. Isso deve ser claro no seguinte código:

```
int main()
{
    int valor = 0;

    cópia(valor) = 10; // erro!
    mesmo(valor) = 10;
}
```

A instrução envolvendo a rotina `cópia()` está errada, pois a rotina devolve um valor temporário, que não pode surgir do lado esquerdo de uma atribuição. Na terminologia da linguagem C++ diz-se que `cópia(valor)` não é um *valor esquerdo* (*lvalue* ou *left value*). Pelo contrário a expressão envolvendo a rotina `mesmo()` está perfeitamente correcta, sendo absolutamente equivalente a escrever:

```
valor = 10;
```

Na realidade, ao se devolver por referência numa rotina, está-se a dar a possibilidade ao consumidor desse procedimento de colocar a sua invocação do lado esquerdo da atribuição. Por exemplo, definido a rotina `incrementa()` como acima, é possível escrever

```
int a = 11;
incrementa(a) = 0;   // possível (mas absurdo), incrementa e depois atri-
bui zero a  a.
incrementa(a) /= 2; // possível (mas má ideia), incrementa e depois di-
vide a por dois.
```

Note-se que a devolução de referências implica alguns cuidados adicionais. Por exemplo, a rotina

```
int& mesmoFalhado(int v)
{
    return v;
}
```

contém um erro grave: devolve uma referência (ou sinónimo) para uma variável local, visto que o parâmetro `v` não é uma referência, mas sim uma variável local cujo valor é uma cópia do argumento respectivo. Como essa variável local é destruída exactamente aquando do retorno da rotina, a referência devolvida fica a referir-se a ... coisa nenhuma!

Uma digressão pelo operador `[]`

Uma vez que o operador de indexação `[]`, usado normalmente para as matrizes e vectores, pode ser sobrecarregado por tipos definidos pelo programador, a devolução de referências permite, por exemplo, definir a classe `VectorDeInt` abaixo, que se comporta aproximadamente como a classe `vector<int>` descrita na Secção 5.2, embora com verificação de erros de indexação:

```
#include <iostream>
#include <vector>
#include <cstdlib>

using namespace std;

class VectorDeInt {
  public:
    VectorDeInt(vector<int>::size_type const dimensão_inicial = 0,
                int const valor_inicial_dos_itens = 0);

    ...

    int& operator[](vector<int>::size_type const índice);

    ...

  private:
    vector<int> itens;
};

VectorDeInt::VectorDeInt(vector<int>::size_type const dimensão_inicial,
                         int const valor_inicial_dos_itens)
    : v(dimensão_inicial, valor_inicial_dos_itens)
```

```
{
}

...

int& VectorDeInt::operator[](vector<int>::size_type const índice)
{
    assert(0 <= índice and índice < itens.size());

    return itens[índice];
}

int main()
{
    VectorDeInt v(10);

    v[0] = 1;
    v[10] = 3; // índice errado! aborta com asserção falhada.
}
```

7.7.2 Operadores ++ e −− prefixo

O operador ++ prefixo necessita de alterar o seu único operando. Assim, é conveniente sobrecarregá-lo na forma de uma operação da classe C++ `Racional`. Uma vez que tem um único operando, este será usado como instância, neste caso variável, implícita durante a execução do respectivo método, pelo que a operação não tem qualquer parâmetro. É importante perceber que a incrementação de um racional pode ser feita de uma forma muito mais simples do que recorrendo à soma de racionais em geral: a adição de um a um racional representado por uma fracção canónica $\frac{n}{d}$ é

$$\frac{n+d}{d},$$

que também é uma fracção no formato canónico[18], pelo que o código é simplesmente

[18]Seja $\frac{n}{d}$ o racional guardado numa instância da classe C++ `Racional`, e que portanto verifica a condição invariante dessa classe, ou seja, $0 < d \wedge \mathrm{mdc}(n,d) = 1$. É óbvio que

$$\frac{n}{d} + 1 = \frac{n+d}{d}.$$

Mas será que a fracção

$$\frac{n+d}{d}$$

verifica a condição invariante de classe? Claramente o denominador d é positivo, pelo que resta verificar se $\mathrm{mdc}(n+d,d) = 1$. Suponha-se que existe um divisor $1 < k$ comum ao numerador e ao denominador. Nesse caso existem n' e d' tais que $kn' = n + d$ e $kd' = d$, de onde se conclui facilmente que $kn' = n + kd'$, ou seja, $n = k(n' - d')$. Só que isso significaria que $1 < k \leq \mathrm{mdc}(n,d)$, o que é contraditório com a hipótese de partida de que $\mathrm{mdc}(n,d) = 1$. Logo, não existe divisor comum ao numerador e denominador superior à unidade, ou seja, $\mathrm{mdc}(n+d,d) = 1$ como se queria demonstrar.

```
/** Representa números racionais.
      @invariant 0 < denominador ∧ mdc(numerador, denominador) = 1. */
class Racional {
  public:

    ...

    /** Incrementa e devolve o racional.
          @pre *this = r.
          @post operador++ ≡ *this ∧ *this = r + 1. */
    Racional& operator++();

    ...

};

...

Racional& Racional::operator++()
{
    assert(cumpreInvariante());

    numerador += denominador;

    assert(cumpreInvariante());

    return ?;
}

...
```

não se necessitando de reduzir a fracção depois desta operação.

Falta resolver um problema, que é o que devolver no final do método. Depois da discussão anterior com a rotina `incrementa()`, deve já ser claro que se deseja devolver o próprio operando do operador, que neste caso corresponde à variável implícita. Como se viu atrás, é possível explicitar a variável implícita usando a construção `*this`, pelo que o código fica simplesmente:

```
Racional& Racional::operator++()
{
    assert(cumpreInvariante());

    numerador += denominador;

    assert(cumpreInvariante());
```

```
        return *this;
    }
```

A necessidade de devolver a própria variável implícita ficará porventura mais clara se se observar um exemplo semelhante ao que se usou mais atrás, mas usando racionais em vez de inteiros:

```
Racional r(1, 2);
++ ++r; // o mesmo que ++(++r);
r.escreve();
cout << endl;
```

Este código é absolutamente equivalente ao seguinte, que usa a notação usual de invocação de operações de uma classe C++:

```
Racional r(1, 2);
(r.operator++()).operator++();
r.escreve();
cout << endl;
```

Aqui torna-se perfeitamente clara a necessidade de devolver a própria variável implícita, para que esta possa ser usada par invocar pela segunda vez o mesmo operador.

Quanto ao operador `--` prefixo, a sua definição é igualmente simples:

```
/** Representa números racionais.
    @invariant 0 < denominador ∧ mdc(numerador, denominador) = 1. */
class Racional {
  public:

    ...

    /** Decrementa e devolve o racional.
        @pre *this = r.
        @post operador-- ≡ *this ∧ *this = r − 1. */
    Racional& operator--();

    ...

};

...

inline Racional& Racional::operator--()
```

```
{
    assert(cumpreInvariante());

    numerador -= denominador;

    assert(cumpreInvariante());

    return *this;
}

...
```

7.7.3 Operadores ++ e −− sufixo

Qual é a diferença entre os operadores de incrementação e decrementação prefixo e sufixo? Como já foi referido no Capítulo 2, a diferença está no que devolvem. As versões prefixo devolvem o próprio operando, já incrementado, e as versões sufixo devolvem uma cópia do valor do operando *antes de incrementado*. Para que tal comportamento fique claro, convém comparar cuidadosamente os seguintes troços de código:

```
int i = 0;
int j = ++i;
```

e

```
int i = 0;
int j = i++;
```

Enquanto o primeiro troço de código inicializa a variável `j` como valor de `i` já incrementado, i.e., com 1, o segundo troço de código inicializa a variável `j` com o valor de `i` *antes de incrementado*, ou seja, com 0. Em ambos os casos a variável `i` é incrementada, ficando com o valor 1.

Clarificada esta diferença, há agora que implementar os operadores de incrementação e decrementação sufixo para a classe C++ `Racional`. A primeira questão, fundamental, é sintáctica: sendo os operadores prefixo e sufixo ambos unários, como distingui-los na definição dos operadores? Se forem definidos como operações da classe C++ `Racional`, então ambos terão o mesmo nome e ambos não terão nenhum parâmetro, distinguindo-se apenas no tipo de devolução, visto que as versões prefixo devolvem por referência e as versões sufixo devolvem por valor. O mesmo se passa se os operadores forem definidos como rotinas normais, não-membro. O problema é que o tipo de devolução não faz parte da assinatura das rotinas, membro ou não, pelo que o compilador se queixará de uma dupla definição do mesmo operador...

Face a esta dificuldade, os autores da linguagem C++ tomaram uma das decisões mais arbitrárias que poderiam ter tomado. Arbitraram que para as assinaturas entre os operadores de incrementação e decrementação prefixo serem diferentes das respectivas versões sufixo, estas últimas teriam como que um operando adicional, inteiro, implícito, e cujo valor deve ser ignorado. É um pouco como se os operadores sufixo fossem binários...

Por razões que ficarão claras mais à frente, definir-se-ão os operadores de incrementação e decrementação sufixo como rotinas normais, não-membro.

Comece-se pelo operador de incrementação sufixo. Sendo sufixo, a sua definição assume que o operador é binário, tendo como primeiro operando o racional a incrementar e como segundo operando um inteiro cujo valor deve ser ignorado. Como o operador será sobrecarregado através de uma rotina normal, ambos os operandos correspondem a parâmetros da rotina, sendo o primeiro, corresponde ao racional a incrementar, passado por referência:

```
/** Incrementa o racional recebido como argumento, devolvendo o seu valor antes de incrementado.
    @pre *this = r.
    @post operator++ = r ∧ *this = r + 1. */
Racional operator++(Racional& r, int valor_a_ignorar)
{
    Racional const cópia = r;

    ++r;

    return cópia;
}
```

Como a parâmetro `valor_a_ignorar` é arbitrário, servindo apenas para compilador perceber que se está a sobrecarregar o operador sufixo, e não o prefixo, não é necessário sequer dar-lhe um nome, pelo que a definição pode ser simplifica para

```
/** Incrementa o racional recebido como argumento, devolvendo o seu valor antes de incrementado.
    @pre *this = r.
    @post operator++ = r ∧ *this = r + 1. */
Racional operator++(Racional& r, int)
{
    Racional const cópia = r;

    ++r;

    return cópia;
}
```

É interessante notar como se recorre ao operador de incrementação prefixo, que já foi definido, na implementação do operador sufixo. Ao contrário do que pode parecer, tal não ocorre simplesmente porque se está a sobrecarregar o operador sufixo como uma rotina não-membro da classe `Racional`. De facto, mesmo que o operador fosse definido como membro da classe

```
/* A sobrecarga também se poderia fazer à custa de uma operação da classe! */
Racional Racional::operator++(int)
{
    Racional const cópia = *this;

    ++*this;

    return cópia;
}
```

continuaria a ser vantajoso fazê-lo: é que o código de incrementação propriamente dito fica concentrado numa única rotina, pelo que, se for necessário mudar a representação dos racionais, apenas será necessário alterar a implementação do operador prefixo.

Repare-se como, em qualquer dos casos, é necessário fazer uma cópia do racional antes de incrementado e devolver essa cópia por valor, o que implica realizar ainda outra cópia. Finalmente compreende-se a insistência, desde o início deste texto, em usar a incrementação prefixo em detrimento da versão sufixo, mesmo onde teoricamente ambas produzem o mesmo resultado, tal como em incrementações ou decrementações isoladas (por exemplo no progresso de um ciclo): é que a incrementação ou decrementação sufixo é quase sempre menos eficiente do que a respectiva versão prefixo.

O operador de decrementação sufixo define-se exactamente de mesma forma:

```
/** Decrementa o racional recebido como argumento, devolvendo o seu valor antes de decrementado.
    @pre *this = r.
    @post operator-- = r ∧ *this = r − 1. */
Racional operator--(Racional& r, int)
{
    Racional const cópia = r;

    --r;

    return cópia;
}
```

Como é óbvio, tendo-se devolvido por valor em vez de por referência, não é possível escrever

```
Racional r;
r++ ++; // erro!
```

que de resto já era uma construção inválida para os tipos básicos do C++.

7.8 Mais operadores para o TAD `Racional`

Falta ainda sobrecarregar muitos operadores para o TAD `Racional`. Um facto curioso, como se verificará em breve, é que os operadores aritméticos sem efeitos laterais se implementam facilmente à custa dos operadores aritméticos com efeitos laterais, e que a versão alternativa, em que se implementam os operadores com efeitos laterais à custa dos que não os têm, conduz normalmente a menores eficiências, pois estes últimos operadores implicam frequentemente a realização de cópias. Assim, tendo-se já sobrecarregado os operadores de incrementação e decrementação, o próximo passo será o de sobrecarregar os operadores de atribuição especiais. Depois definir-se-ão os operadores aritméticos normais. Aliás, no caso do operador + será uma re-implementação.

7.8.1 Operadores de atribuição especiais

Começar-se-á pelo operador `*=`, de implementação muito simples.

Tal como os operadores de incrementação e decrementação, também os operadores de atribuição especiais são mal comportados. São definidos à custas de rotinas que são mistos de função e procedimento, ou funções com efeitos laterais. O operador `*=` não é excepção. Irá ser sobrecarregado à custa de uma operação da classe C++ `Racional`, pois necessita de alterar os atributos da classe. Como o operador `*=` tem dois operandos, o primeiro será usado com instância (aliás, variável) implícita, e o segundo será passado como argumento. A operação terá, pois um único parâmetro. Todos os operadores de atribuição especiais devolvem uma referência para o primeiro operando, tal como os operadores de incrementação e decrementação prefixo. É isso que permite escrever o seguinte pedaço de código, muito pouco recomendável, mas idêntico ao que se poderia também escrever para variáveis dos tipos básicos do C++:

```
Racional a(4), b(1, 2);
(a *= b) *= b;
```

Deve ser claro que este código multiplica `a` por $\frac{1}{2}$ duas vezes, ficando `a` com o valor 1.

A implementação do operador produto e atribuição é simples::

```
...

class Racional {
  public:

    ...

    /** Multiplica por um racional.
        @pre *this = r.
        @post operator*= ≡ *this ∧ *this = r × r2. */
    Racional& operator*=(Racional r2);
```

```
    ...
};

...

Racional& Racional::operator*=(Racional const r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    numerador *= r2.numerador;
    denominador *= r2.denominador;

    reduz();

    assert(cumpreInvariante());

    return *this;
}

...
```

O corpo do método definido limita-se a efectuar o produto da forma usual para as fracções, i.e., o numerador do produto é o produto dos numeradores e o denominador do produto é o produto dos denominadores. Como os denominadores são ambos positivos, o seu produto também o será. Para que o resultado cumpra a condição invariante de classe falta apenas garantir que no final do método $\mathrm{mdc}(n, d) = 1$. Como isso não é garantido (pense-se, por exemplo, o produto de $\frac{1}{2}$ por 2), é necessário reduzir a fracção resultado. Tal como no caso dos operadores de incrementação e decrementação prefixo, também aqui se termina devolvendo a variável implícita, i.e., o primeiro operando.

O operador `/=` sobrecarrega-se da mesma forma, embora tenha de haver o cuidado de garantir que o segundo operando não é zero:

```
...

class Racional {
  public:

    ...

    /** Divide por um racional.
          @pre *this = r ∧ r2 ≠ 0.
          @post operator/= ≡ *this ∧ *this = r/r2. */
    Racional& operator/=(Racional r2);

    ...
```

```
};

...

Racional& Racional::operator/=(Racional const r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    assert(r2 != 0);

    if(r2.numerador < 0) {
        numerador *= -r2.denominador;
        denominador *= -r2.numerador;
    } else {
        numerador *= r2.denominador;
        denominador *= r2.numerador;
    }

    reduz();

    assert(cumpreInvariante());

    return *this;
}

...
```

Há neste código algumas particularidades que é preciso estudar.

A divisão por zero é impossível, pelo que a pré-condição obriga `r2` a ser diferente de zero. A instrução de asserção reflecte isso mesmo, embora contenha um erro: por ora não é possível comparar dois racionais através do operador `!=`, quanto mais um racional e um inteiro (0 é um do tipo `int`). Pede-se ao leitor que seja paciente, pois dentro em breve este problema será resolvido sem ser necessário alterar em nada este método!

O cálculo da divisão é muito simples: o numerador da divisão é o numerador do primeiro operando multiplicado pelo denominador do segundo operando, e vice-versa. Uma versão simplista do cálculo da divisão seria:

```
numerador *= r2.denominador;
denominador *= r2.numerador;
```

Este código, no entanto, não só não garante que o resultado esteja reduzido, e daí a invocação de `reduz()` no código mais acima. (tal como acontecia para o operador `*=`) como também não garante que o denominador resultante seja positivo, visto que o numerador de `r2` pode perfeitamente ser negativo. Prevendo esse caso o código fica

```
if(r2.numerador < 0) {
    numerador *= -r2.denominador;
    denominador *= -r2.numerador;
} else {
    numerador *= r2.denominador;
    denominador *= r2.numerador;
}
```

tal como se pode encontrar no no método acima.

Relativamente ao operador += é possível resolver o problema de duas formas. A mais simples neste momento é implementar o operador += à custa do operador +, pois este já está definido. Nesse caso a solução é:

```
Racional& Racional::operator+=(Racional const r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    *this = *this + r2

    assert(cumpreInvariante());

    return *this;
}
```

Esta solução, no entanto, tem o inconveniente de obrigar à realização de várias cópias entre racionais, além de exigir a construção de um racional temporário para guardar o resultado da adição antes de este ser atribuído à variável implícita. Como se verá, a melhor solução é desenvolver o operador += de raiz e implementar o operador + à sua custa.

Os operadores += e −= sobrecarregam-se de forma muito semelhante:

```
...

class Racional {
  public:

    ...

    /** Adiciona de um racional.
          @pre *this = r.
          @post operator+= ≡ *this ∧ *this = r + r2. */
    Racional& operator+=(Racional r2);

    /** Subtrai de um racional.
          @pre *this = r.
```

```
        @post operator-= ≡ *this ∧ *this = r − r2. */
    Racional& operator-=(Racional r2);

    …
};

…

Racional& Racional::operator+=(Racional const r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    numerador = numerador * r2.denominador +
        r2.numerador * denominador;
    denominador *= r2.denominador;

    reduz();

    assert(cumpreInvariante());

    return *this;
}

Racional& Racional::operator-=(Racional const r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    numerador = numerador * r2.denominador -
        r2.numerador * denominador;
    denominador *= r2.denominador;

    reduz();

    assert(cumpreInvariante());

    return *this;
}

…
```

7.8.2 Operadores aritméticos

Os operadores aritméticos usuais podem ser facilmente implementados à custa dos operadores especiais de atribuição. Implementar-se-ão aqui como rotinas normais, não-membro, por razões que serão clarificadas em breve. Começar-se-á pelo operador `*`. A ideia é criar uma variável local temporária cujo valor inicial seja uma cópia do primeiro operando, e em seguida usar o operador `*=` para proceder à soma:

```
/** Produto de dois racionais.
    @pre V.
    @post operator* = r1 × r2. */
Racional operator*(Racional const r1, Racional const r2)
{
    Racional auxiliar = r1;
    auxiliar *= r2;
    return auxiliar;
}
```

Observando cuidadosamente este código, conclui-se facilmente que o parâmetro `r1`, desde que deixe de ser constante, pode fazer o papel da variável `auxiliar`, visto que a passagem se faz por valor:

```
/** Produto de dois racionais.
    @pre r1 = r₁.
    @post operator* = r₁ × r2. */
Racional operator*(Racional r1, Racional const r2)
{
    r1 *= r2;
    return r1;
}
```

Finalmente, dado que o operador `*=` devolve o primeiro operando, podem-se condensar as duas instruções do método numa única instrução idiomática:

```
/** Produto de dois racionais.
    @pre r1 = r₁.
    @post operator* = r₁ × r2. */
Racional operator*(Racional r1, Racional const r2)
{
    return r1 *= r2;
}
```

A implementação dos restantes operadores aritméticos faz-se exactamente da mesma forma:

```
/** Divisão de dois racionais.
    @pre r1 = r₁ ∧ r2 ≠ 0.
    @post operator/ = r₁/r2. */
Racional operator/(Racional r1, Racional const r2)
{
    assert(r2 != 0);
```

```
    return r1 /= r2;
}
```

/** Adição de dois racionais.
@pre r1 $= r_1$.
@post operator+ $= r_1 +$ r2. */

```
Racional operator+(Racional r1, Racional const r2)
{
    return r1 += r2;
}
```

/** Subtracção de dois racionais.
@pre r1 $= r_1$.
@post operator- $= r_1 -$ r2. */

```
Racional operator-(Racional r1, Racional const r2)
{
    return r1 -= r2;
}
```

Para além da vantagem já discutida de implementar um operador à custa de outro, agora deve já ser clara a vantagem de ter implementado o operador `*` à custa do operador `*=` e não o contrário: a operação `*=` tornou-se muito mais eficiente, pois não obriga a copiar ou construir qualquer racional, enquanto a operação `*` continua a precisar a sua dose de cópias e construções...

Mas, porquê definir estes operadores como rotinas normais, não-membro? Há uma razão de peso, que tem a ver com as conversões implícitas.

7.9 Construtores: conversões implícitas e valores literais

7.9.1 Valores literais

Já se viu que a definição de classes C++ concretizando TAD permite a acrescentar à linguagem C++ novos tipos que funcionam praticamente como os seus tipos básicos. Mas haverá equivalente aos valores literais? Recorde-se que, num programa em C++, `10` e `100.0` são valores literais dos tipos `int` e `double`, respectivamente. Será possível especificar uma forma para, por exemplo, escrever valores literais do novo tipo `Racional`? Infelizmente isso é impossível em C++. Por exemplo, o código

```
Racional r;
r = 1/3;
```

redunda num programa aparentemente funcional mas com um comportamento inesperado. Acontece que a expressão `1/3` é interpretada como a divisão inteira, que neste caso tem resultado zero. Esse valor inteiro é depois convertido implicitamente para o tipo `Racional` e atribuída à variável `r`. Logo, `r`, depois da atribuição, conterá o racional zero!

Existe uma alternativa elegante aos inexistentes valores literais para os racionais. É proporcionada pelos construtores da classe, e funciona quase como se de valores literais se tratasse: os construtores podem ser chamados explicitamente para criar um novo valor dessa classe. Assim, o código anterior deveria ser corrigido para:

```
Racional r;
r = Racional(1, 3);
```

7.9.2 Conversões implícitas

Se uma classe `A` possuir um construtor que possa ser invocado passando um único argumento do tipo `B` como argumento, então está disponível uma conversão implícita do tipo `B` para a classe `A`. Por exemplo, o primeiro construtor da classe `Racional` (ver Secção 7.4.1) pode ser chamado com apenas um argumento do tipo `int`, o que significa que, sempre que o compilador esperar um `Racional` e encontrar um `int`, converte o `int` implicitamente para um valor `Racional`. Por exemplo, estando definido um operator + com operandos do tipo `Racional`, o seguinte pedaço de código

```
Racional r1(1, 3);
Racional r2 = r1 + 1;
```

é perfeitamente legal, tendo o mesmo significado que

```
Racional r1(1, 3);
Racional r2 = r1 + Racional(1);
```

e colocando em `r2` o racional $\frac{4}{3}$.

Em casos em que esta conversão implícita de tipos é indesejável, pode-se preceder o respectivo construtor da palavra-chave `explicit`. Assim, se a classe `Racional` estivesse definida como

```
...

class Racional {
  public:
     /** Constrói racional com valor inteiro. Construtor por omissão.
         @pre V.
         @post *this = n∧
               0 < denominador ∧ mdc(numerador, denominador) = 1. */
    explicit Racional(int const n = 0);
    ...

};

...
```

o compilador assinalaria erro ao encontrar a expressão `r1 + 1`. Neste caso, no entanto, a conversão implícita de `int` para `Racional` é realmente útil, pelo que o qualificador `explicit` é desnecessário.

7.9.3 Sobrecarga de operadores: operações ou rotinas?

Suponha-se por um instante que o operador + para a classe C++ Racional é sobrecarregado através de uma operação. Isto é, regresse-se à versão do operador + apresentada na Secção 7.5. Nesse caso o seguinte código

```
Racional r(1, 3);
Racional s = r + 3;
```

é válido, pois o valor inteiro 3 é convertido implicitamente para `Racional` e seguidamente é invocado o operador + definido.

Ou seja, o código acima é equivalente a

```
Racional r(1, 3);
Racional s = r + Racional(3);
```

Porém, o código

```
Racional r(1, 3);
Racional s = 3 + r;
```

é inválido, pois a linguagem C++ proíbe conversões na instância através da qual se invoca um método. Se o operador tivesse sido sobrecarregado à custa de uma normal rotina não-membro, todos os seus argumentos poderiam sofrer conversões implícitas, o que resolveria o problema. Mas foi exactamente isso que se fez nas secções anteriores! Logo, o código acima é perfeitamente legal e equivalente a

```
Racional r(1, 3);
Racional s = Racional(3) + r;
```

Este facto será utilizado para implementar alguns dos operadores em falta para a classe C++ `Racional`.

7.10 Operadores igualdade, diferença e relacionais

Os operadores de igualdade, diferença e relacionais serão desenvolvidos usando algumas das técnicas já apresentadas. Estes operadores serão sobrecarregados usando rotinas não-membro, de modo a se tirar partido das conversões implícitas de `int` para `Racional`, e tentar-se-á que sejam implementados à custa de outros módulos preexistentes, por forma a minimizar o impacte de possíveis alterações na representação (interna) dos números racionais.

Os primeiros operadores a sobrecarregar serão o operador igualdade, ==, e o operador diferença, !=. Viu-se na Secção 7.4.4 que o facto de as instâncias da classe C++ `Racional` cumprirem a condição invariante de classe, i.e., de $\frac{\texttt{numerador}}{\texttt{denominador}}$ ser uma fracção no formato canónico, permitia simplificar muito a comparação entre racionais. De facto, assim é, pois dois racionais são iguais se e só se tiverem representações em fracções canónicas iguais. Assim, uma primeira tentativa de definir o operador == poderia ser:

```
/** Indica se dois racionais são iguais.
    @pre V.
    @post operator== = (r1 = r2). */
bool operator==(Racional const r1, Racional const r2)
{
    return r1.numerador == r2.numerador and
           r1.denominador == r2.denominador;
}
```

O problema deste código é que, sendo o operador uma rotina não-membro, não tem acesso aos membros privados da classe C++. Por outro lado, se o operador fosse uma operação da classe C++, embora o problema do acesso aos membros se resolvesse, deixariam de ser possíveis conversões implícitas do primeiro operando do operador. Como resolver o problema?

Há duas soluções para este dilema. A primeira passa por tornar a rotina que sobrecarrega o operador == *amigo* da classe C++ `Racional` (ver Secção 7.15). Esta solução é desaconselhável, pois há uma alternativa simples que não passa por amizades (e, por isso, não está sujeita a introduzir quaisquer promiscuidades): deve-se explorar o facto de a rotina precisar de saber os valores do numerador e denominador da fracção canónica correspondente ao racional, *mas não precisar de alterar o seu valor*.

7.10.1 Inspectores e interrogações

Se se pensar cuidadosamente nas possíveis utilizações do TAD `Racional`, conclui-se facilmente que o programador consumidor pode necessitar de conhecer a fracção canónica correspondente ao racional. Se assim for, convém equipar a classe C++ com duas funções membro que se limitam a devolver o valor do numerador e denominador dessa fracção canónica. Como a representação de um `Racional` é justamente feita à custa de uma fracção canónica, conclui-se que as duas funções membro são muito fáceis de implementar[19]:

[19] A condição objectivo da operação `denominador()` é algo complexa, pois evita colocar na interface da classe referências à sua implementação, como seria o caso se se referisse ao atributo `denominador_`. Assim, usa-se a definição de denominador de fracção canónica. O valor devolvido `denominador` tem de ser tal que exista um numerador n tal que

1. a fracção $\frac{n}{\texttt{denominador}}$ é igual à instância implícita e
2. a fracção $\frac{n}{\texttt{denominador}}$ está no formato canónico,

```
...

class Racional {
  public:

    ...

    /** Devolve numerador da fracção canónica correspondente ao racional.
        @pre *this = $r$.
        @post *this = $r \wedge \frac{\texttt{numerador}}{\texttt{denominador()}} = $ *this. */
    int numerador();

    /** Devolve denominador da fracção canónica correspondente ao racional.
        @pre *this = $r$.
        @post *this = $r \wedge$
$\left(\mathbf{E}\, n : \mathcal{V} : \frac{n}{\texttt{denominador}} = \texttt{*this} \wedge 0 < \texttt{denominador} \wedge \mathrm{mdc}(n, \texttt{denominador}) = 1\right)$. */
    int denominador();

    ...

  private:
    int numerador_;
    int denominador_;

    ...
};

...

int Racional::numerador()
{
    assert(cumpreInvariante());

    assert(cumpreInvariante());
    return numerador_;
}

int Racional:: denominador()
{
    assert(cumpreInvariante());
```

ou seja, o valor devolvido é o denominador da fracção canónica correspondente à instância implícita.
A condição objectivo da operação `numerador()` é mais simples, pois recorre à definição da operação `denominador()` para dizer que se o valor devolvido for o numerador de uma fracção cujo denominador é o valor devolvido pela operação `denominador()`, então essa fracção é igual à instância implícita. Como o denominador usado é o denominador da fracção canónica igual à instância implícita, conclui-se que o valor devolvido é de facto o numerador dessa mesma fracção canónica.

```
        assert(cumpreInvariante());

        return denominador_;
    }

    ...
```

Às operações de uma classe C++ que se limitam a devolver propriedades das suas instâncias chama-se *inspectores*. Invocá-las também se diz *interrogar* a instância. Os inspectores permitem obter os valores de propriedades de um instância sem que se exponham as suas partes privadas à manipulação pelo público em geral.

É de notar que a introdução destes novos operadores trouxe um problema prático. As novas operações têm naturalmente o nome que antes tinham os atributos da classe. Repare-se que não se decidiu dar outros nomes às operações para evitar os conflitos: que produz uma classe deve estar preparado para, em nome do fornecimento de uma interface tão clara e intuitiva quanto possível, fazer alguns sacrifícios. Neste caso o sacrifício é o de alterar o nome dos atributos, aos quais é comum acrescentar um sublinhado (_) para os distinguir de operações com o mesmo nome, e, sobretudo, alterar os nomes desses atributos em todas as operações entretanto definidas (e note-se que já são algumas...).

7.10.2 Operadores de igualdade e diferença

Os inspectores definidos na secção anterior são providenciais, pois permitem resolver facilmente o problema do acesso aos atributos. Basta recorrer a eles para comparar os dois racionais:

```
/** Indica se dois racionais são iguais.
    @pre V.
    @post operator== = (r1 = r2). */
bool operator==(Racional const r1, Racional const r2)
{
    return r1.numerador() == r2.numerador() and
           r1.denominador() == r2.denominador();
}
```

O operador != sobrecarrega-se de uma forma ainda mais simples: negando o resultado de uma invocação ao operador == definido acima:

```
/** Indica se dois racionais são diferentes.
    @pre V.
    @post operator!= = (r1 ≠ r2). */
bool operator!=(Racional const r1, Racional const r2)
{
    return not (r1 == r2);
}
```

7.10.3 Operadores relacionais

O operador < pode ser facilmente implementado para a classe C++ `Racional`, bastando recorrer ao mesmo operador para os inteiros. Suponha-se que se pretende saber se

$$\frac{n_1}{d_1} < \frac{n_2}{d_2},$$

em que $\frac{n_1}{d_1}$ e $\frac{n_2}{d_2}$ são fracções no formato canónico. Como $0 < d_1$ e $0 < d_2$, a desigualdade acima é equivalente a

$$n_1 d_2 < n_2 d_1.$$

Logo, a sobrecarga do operador < pode ser feita como se segue:

```
/** Indica se o primeiro racional é menor que o segundo.
    @pre V.
    @post operator< = (r1 < r2). */
bool operator<(Racional const r1, Racional const r2)
{
    return r1.numerador() * r2.denominador() <
           r2.numerador() * r1.denominador();
}
```

Os restantes operadores relacionais podem ser definidos todos à custa do operador <. É instrutivo ver como, sobretudo no caso desconcertantemente simples do operador >:

```
/** Indica se o primeiro racional é maior que o segundo.
    @pre V.
    @post operator> = (r1 > r2). */
bool operator>(Racional const r1, Racional const r2)
{
    return r2 < r1;
}

/** Indica se o primeiro racional é menor ou igual ao segundo.
    @pre V.
    @post operator<= = (r1 ≤ r2). */
bool operator<=(Racional const r1, Racional const r2)
{
    return not (r2 < r1);
}

/** Indica se o primeiro racional é maior ou igual ao segundo.
    @pre V.
    @post operator>= = (r1 ≥ r2). */
```

```
bool operator>=(Racional const r1, Racional const r2)
{
    return not (r1 < r2);
}
```

Curiosamente (ou não), também os operadores == e != se podem implementar à custa apenas do operador <. Fazê-lo fica como exercício para o leitor.

7.11 Constância: verificando erros durante a compilação

Uma boa linguagem de programação permite ao programador escrever programas com um mínimo de erros. Um bom programador que tira partido das ferramentas que a linguagem possui para reduzir ao mínimo os seus próprios erros.

Há três formas importantes de erros:

1. Erros lógicos. São erros devidos a um raciocínio errado do programador: a sua resolução do problema, incluindo algoritmos e estruturas de dados, ainda que correctamente implementados, não leva ao resultado pretendido, ou seja, na realidade não resolve o problema. Este tipo de erro é o mais difícil de corrigir. A facilidade ou dificuldade da sua detecção varia bastante conforme os casos, mas é comum que ocorram erros lógicos de difícil detecção.

2. Erros de implementação. Ao implementar a resolução do problema idealizada, foram cometidos erros não-sintácticos (ver abaixo), i.e., o programa não é uma implementação do algoritmo idealizado. Erros deste tipo são fáceis de corrigir, desde que sejam detectados. A detecção dos erros tanto pode ser fácil como muito difícil, por exemplo, quando os erros ocorrem em casos fronteira que raramente ocorrem na prática, ou quando o programa produz resultados que, sendo errados, parecem plausíveis.

3. Erros sintácticos. São "gralhas". O próprio compilador se encarrega de os detectar. São fáceis de corrigir.

Antes de um programa ser disponibilizado ao utilizador final, é testado. Antes de ser testado, é compilado. Antes de ser compilado, é desenvolvido. Com excepção dos erros durante o desenvolvimento, é claro que quanto mais cedo no processo ocorrerem os erros, mais fáceis serão de detectar e corrigir, e menor o seu impacte. Assim, uma boa linguagem é aquela que permite que os (inevitáveis) erros sejam sobretudo de compilação, detectados facilmente pelo compilador, e não de implementação ou lógicos, detectados com dificuldade pelo programador ou pelos utilizadores do programa.

Para evitar os erros lógicos, uma linguagem deve possuir uma boa biblioteca, que liberte o programador da tarefa ingrata, e sujeita a erros, de desenvolver algoritmos e estruturas de dados bem conhecidos. Mas como evitar os erros de implementação? Há muitos casos em que a linguagem pode ajudar. É o caso da possibilidade de usar constantes em vez de variáveis, que permite ao compilador detectar facilmente tentativas de alterar o seu valor, enquanto que a utilização de uma variável para o mesmo efeito impediria do compilador de detectar o erro, deixando esse trabalho nas mãos do programador. Outro caso é o do encapsulamento. A categorização de membros de uma classe C++ como privados permite ao compilador detectar tentativas erróneas de acesso a esses membros, coisa que seria impossível se os membros fossem públicos, recaindo sobre os ombros do programador consumidor da classe a responsabilidade de não aceder a determinados membros, de acordo com as especificações do programador produtor. Ainda outro caso é a definição das variáveis tão perto quando possível da sua primeira utilização, que permite evitar utilizações erróneas dessa variável antes do local onde é realmente necessária, e onde, se a variável for de um tipo básico, toma um valor arbitrário (lixo).

Assim, é conveniente usar os mecanismos da linguagem de programação que permitem exprimir no próprio código determinadas opções de implementação e condições de utilização, e que permitem que seja o próprio compilador a verificar do seu cumprimento, tirando esse peso dos ombros do programador, que pode por isso dedicar mais atenção a outros assuntos mais importantes. É o caso da classificação de determinadas instâncias como constantes, estudada nesta secção no âmbito da classe `Racional`.

7.11.1 Passagem de argumentos

Até agora viram-se duas formas de passagem de argumentos: por valor e por referência. Com a utilização da palavra-chave `const` as possibilidades passam a quatro, ou melhor, a três e meia...

A forma mais simples de passagem de argumentos é por valor. Neste caso os parâmetros são variáveis locais à rotina, inicializadas à custa dos argumentos respectivos. Ou seja, os parâmetros são *cópias* dos argumentos:

```
// Declaração:
TipoDeDevolução rotina(TipoDoParâmetro parâmetro);

// Definição:
TipoDeDevolução rotina(TipoDoParâmetro parâmetro)
{
    ...
}
```

É também possível que os parâmetros sejam constantes:

```
// Declaração:
TipoDeDevolução rotina(TipoDoParâmetro const parâmetro);

// Definição:
TipoDeDevolução rotina(TipoDoParâmetro const parâmetro)
{
    ...
}
```

No entanto, a diferença entre um parâmetro variável ou constante, no caso da passagem de argumentos por valor, não tem qualquer espécie de impacte sobre o código que invoca a rotina. Ou seja, para o programador consumidor da rotina é irrelevante se os parâmetros são variáveis ou constantes: o que lhe interessa é que serão cópias dos argumentos, que por isso não serão afectados pelas alterações que os parâmetros possam ou não sofrer[20]: a interface da rotina não é afectada, e as declarações

```
TipoDeDevolução rotina(TipoDoParâmetro parâmetro);
```

e

```
TipoDeDevolução rotina(TipoDoParâmetro const parâmetro);
```

são idênticas, pelo que se sói usar apenas a primeira forma, excepto quando for importante deixar clara a constância do parâmetro devido ao facto de ele ocorrer na condição objectivo da rotina, i.e., quando se quiser dizer que o parâmetro usado na condição objectivo tem o valor original, à entrada da rotina.

Já do ponto de vista do programador programador, ou seja, durante a definição da rotina, faz toda a diferença que o parâmetro seja constante: se o for, o compilador detectará tentativas de o alterar no corpo da rotina, protegendo o programador dos seus próprios erros no caso de a alteração do valor do parâmetro ser de facto indesejável.

Finalmente, note-se que a palavra-chave `const`, no caso da passagem de argumentos por valor, é eliminada automaticamente da assinatura da rotina, pelo que é perfeitamente possível que surja apenas na sua definição (implementação), sendo eliminada da declaração (interface):

```
// Declaração:
TipoDeDevolução rotina(TipoDoParâmetro parâmetro);

// Definição:
TipoDeDevolução rotina(TipoDoParâmetro const parâmetro)
{
    ... // Alteração de parâmetro proibida!
}
```

No caso da passagem por referência a palavra-chave `const` faz toda a diferença em qualquer caso, quer do ponto de vista da interface, quer do ponto de vista da implementação. Na passagem de argumentos por referência,

[20]Curiosamente é possível criar classes cujo construtor por cópia (ver Secção 7.4.2) altere o original! É normalmente muito má ideia fazê-lo, pois perverte a semântica usual da cópia, mas em alguns casos poderá ser uma prática justificada. É o caso do tipo genérico `auto_ptr`, da biblioteca padrão do C++. Mas mesmo no caso de uma classe C++ ter um construtor por cópia que altere o original, tal alteração ocorre durante a passagem de um argumento dessa classe C++ por valor, seja ou não o parâmetro respectivo constante, o que só vem reforçar a irrelevância para a interface de uma rotina de se usar a palavras chave `const` para qualificar parâmetros que não sejam referências.

```
// Declaração:
TipoDeDevolução rotina(TipoDoParâmetro& parâmetro);

// Definição:
TipoDeDevolução rotina(TipoDoParâmetro& parâmetro)
{
    ...
}
```

os parâmetros funcionam como *sinónimos* dos argumentos (ou *referências [variáveis]* para os argumentos). Assim, qualquer alteração de um parâmetro repercute-se sobre o argumento respectivo. Como neste tipo de passagem de argumentos não é realizada qualquer cópia, ela tende a ser mais eficiente que a passagem por valor, pelo menos para tipos em que as cópias são onerosas computacionalmente, o que não é o caso dos tipos básicos da linguagem. Por outro lado, este tipo de passagem de argumentos proíbe a passagem como argumento de constantes, como é natural, mas também de variáveis temporárias, tais como resultados de expressões que não sejam *lvalues* (ver Secção 7.7.1). Este facto impede a passagem de argumentos de tipos diferentes dos parâmetros mas para os quais exista uma conversão implícita.

Quando a passagem de argumentos se faz por referência constante,

```
// Declaração:
TipoDeDevolução rotina(TipoDoParâmetro const& parâmetro);

// Definição:
TipoDeDevolução rotina(TipoDoParâmetro const& parâmetro)
{
    ...
}
```

os parâmetros funcionam como *sinónimos constantes* dos argumentos (ou *referências constantes* para os parâmetros). Sendo constantes, as alterações aos parâmetros são proibidas. Por um lado, a passagem de argumentos por referência constante é semelhante à passagem por valor, pois não só impossibilita alterações aos argumentos como permite que sejam passados valores constantes e temporários como argumentos e que estes sofram conversões implícitas: uma referência constante pode ser sinónimo tanto de uma variável como de uma constante, pois uma variável pode sempre ser tratada como uma constante (e não o contrário), e pode mesmo ser sinónimo de uma variável ou constante *temporária*. Por outro lado, este tipo de passagem de argumentos é semelhante à passagem de argumentos por referência simples, pois não obriga à realização de cópias.

Ou seja, a passagem de argumentos por referência constante tem a vantagem das passagens por referência, ou seja, a sua maior eficiência na passagem de tipos não básicos, e a vantagens da passagem por valor, ou seja, a impossibilidade de alteração do argumento através do respectivo parâmetro e a possibilidade de passar instâncias (variáveis ou constantes) temporárias ou não. Assim, como regra geral, é sempre recomendável a passagem de argumentos por referência constante, em detrimento da passagem por valor, quando estiverem em causa tipos não básicos e quando não houver necessidade por alguma razão de alterar o valor do parâmetro durante a execução da rotina em causa.

Esta regra deve ser aplicada de forma sistemática às rotinas membro e não-membro desenvolvidas, no caso deste capítulo às rotinas associadas ao TAD `Racional` em desenvolvimento. A título de exemplo mostra-se a sua utilização na sobrecarga dos operadores `+=`, `/=` e `+` para a classe C++ `Racional`:

```
...

class Racional {
  public:

    ...

    /** Adiciona de um racional.
        @pre *this = r.
        @post operator+= ≡ *this ∧ *this = r + r2. */
    Racional& operator+=(Racional const& r2);

    /** Divide por um racional.
        @pre *this = r ∧ r2 ≠ 0.
        @post operator/= ≡ *this ∧ *this = r/r2. */
    Racional& operator/=(Racional const& r2);

   ...
};

...

Racional& Racional::operator+=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    numerador = numerador * r2.denominador +
        r2.numerador * denominador;
    denominador *= r2.denominador;

    reduz();

    assert(cumpreInvariante());

    return *this;
}

Racional& Racional::operator/=(Racional const& r2)
{
```

```
    assert(cumpreInvariante() and r2.cumpreInvariante());

    assert(r2 != 0);

    int numerador2 = r2.numerador_;
    if(r2.numerador_ < 0) {
        numerador_ *= -r2.denominador_;
        denominador_ *= -numerador2;
    } else {
        numerador_ *= r2.denominador_;
        denominador_ *= numerador2;
    }

    reduz();

    assert(cumpreInvariante());

    return *this;
}

...

/** Adição de dois racionais.
    @pre r1 = r_1.
    @post operator+ = r_1 + r2. */
Racional operator+(Racional r1, Racional const& r2)
{
    return r1 += r2;
}

...
```

Preservou-se a passagem por valor do primeiro argumento do operador + por ser desejável que nesse caso o parâmetro seja uma cópia do argumento, de modo a sobre ele se poder utilizar o operador +=.

É de notar uma alteração importante à definição da sobrecarga do operador `/=`: passou a ser feita um cópia do numerador do segundo operando, representado pelo parâmetro `r2`. É fundamental fazê-lo para que o código tenha o comportamento desejável no caso de se invocar o operador da seguinte forma:

```
r /= r;
```

Fica como exercício para o leitor verificar que o resultado estaria longe do desejado se esta alteração não tivesse sido feita (dica: a variável implícita e a variável da qual `r2` é sinónimo são a mesma variável).

7.11.2 Constantes implícitas: operações constantes

É possível definir constantes de um TAD concretizado à custa de uma classe C++. Por exemplo, para a classe C++ `Racional` é possível escrever o código

```
Racional const um_terço(1, 3);
```

que define uma constante `um_terço`. O problema está em que, tal como a classe C++ `Racional` está definida, esta constante praticamente não se pode usar. Por exemplo, o código

```
cout << "O denominador é " << um_terço.denominador() << endl;
```

resulta num erro de compilação.

A razão para o erro é simples: o compilador assume que as operações da classe C++ `Racional` alteram a instância implícita, ou seja, assume que as operações têm sempre uma variável e não uma constante implícita. Assim, como o compilador assume que há a possibilidade de a constante `um_terço` ser alterada, o que é um contra-senso, simplesmente proíbe a invocação da operação inspectora `Racional::denominador()`.

Note-se que o mesmo problema já existia no código desenvolvido: repare-se na rotina que sobrecarrega o operador ==, por exemplo:

```
/** Indica se dois racionais são iguais.
    @pre V.
    @post operator== = (r1 = r2). */
bool operator==(Racional const& r1, Racional const& r2)
{
    return r1.numerador() == r2.numerador() and
           r1.denominador() == r2.denominador();
}
```

Os parâmetros `r1` e `r2` desta rotina funcionam como sinónimos constantes dos respectivos argumentos (que podem ser constantes ou não). Logo, o compilador assinala um erro no corpo desta rotina ao se tentar invocar os inspectores `Racional::numerador()` e `Racional::denominador()` através das duas constantes: o compilador não adivinha que uma operação não altera a instância implícita. Aliás, nem o poderia fazer, pois muitas vezes no código que invoca a operação o compilador não tem acesso ao respectivo método, como se verá no 9, pelo que não pode verificar se de facto assim é.

Logo, é necessário indicar explicitamente ao compilador quais as operações que não alteram a instância implícita, ou seja, quais as operações que tratam a instância implícita como uma constante implícita. Isso consegue-se acrescentando a palavra-chave `const` ao cabeçalho das operações em causa e respectivos métodos, pois esta palavra-chave passará a fazer parte da respectiva assinatura (o que permite sobrecarregar uma operação com o mesmo nome e lista de parâmetros onde a única coisa que varia é a constância da instância implícita). Por exemplo, o inspector `Racional::numerador()` deve ser qualificado como não alterando a instância implícita:

```
...

class Racional {
  public:

    ...

    /** Devolve numerador da fracção canónica correspondente ao racional.
        @pre V.
        @post numerador / denominador() = *this. */
    int numerador() const;

    ...
};

...

int Racional::numerador() const
{
    assert(cumpreInvariante());

    assert(cumpreInvariante());
    return numerador_;
}

...
```

É importante perceber que o compilador verifica se no método correspondente a uma *operação constante*, que é o nome que se dá a uma operação que garante a constância da instância implícita, se executa alguma instrução que possa alterar a constante implícita. Isso significa que o compilador proíbe a invocação de operações não-constantes através da constante implícita e também que proíbe a alteração dos atributos, pois os atributos de uma constante assumem-se também constantes!

É o facto de a constância da instância implícita ser agora claramente indicada através do qualificador `const` e garantida pelo compilador que permitiu deixar de explicitar essa constância através de um termo extra na pré-condição e na condição objectivo: a constância da instância implícita continua a estar expressa no contrato destas operações, mas agora não na pré-condição e na condição objectivo mas na própria sintaxe do cabeçalho das operações[21].

Todas as operações inspectoras são naturalmente operações constantes. Embora também seja comum dizer-se que as operações constantes são inspectoras, neste texto reserva-se o nome inspector para as operações que devolvam propriedades da instância para a qual são invocados. Pelo contrário, às operações que alteram a instância implícita, ou que a permitem alterar

[21]Exactamente da mesma forma que as pré-condições não se referem normalmente ao tipo dos parâmetros (e.g., "o primeiro parâmetro tem de ser um `int`"), pois esse facto é expresso na própria linguagem C++ e garantido pelo compilador (bem, quase sempre, como se verá quando se distinguir tipo estático de tipo dinâmico...).

indirectamente, chama-se normalmente *operações modificadoras*, embora também seja possível distinguir entre várias categorias de operações não-constantes.

Resta, pois, qualificar como constantes todas as operações e respectivos métodos que garantem a constância da instância implícita:

```
...

class Racional {
  public:

    ...

    /** Devolve numerador da fracção canónica correspondente ao racional.
          @pre V.
          @post numerador / denominador() = *this. */
    int numerador() const;

    /** Devolve denominador da fracção canónica correspondente ao racional.
          @pre V.
          @post (E n : V : n / denominador = *this ∧ 0 < denominador ∧ mdc(n, denominador) = 1). */
    int denominador() const;

   /** Escreve o racional no ecrã no formato de uma fracção.
          @pre V.
          @post ¬cout∨ cout contém n/d (ou simplesmente n se d = 1)
                sendo n/d a fracção canónica correspondente ao racional *this. */
    void escreve() const;

    ...

  private:

    ...

    /** Indica se a condição invariante de classe se verifica.
          @pre V.
          @post cumpreInvariante = (0 <
denominador_ ∧ mdc(numerador_, denominador_) = 1). */
    bool cumpreInvariante() const;

    ...

};

...
```

```
int Racional::numerador() const
{
    assert(cumpreInvariante());

    return numerador_;
}

int Racional:: denominador() const
{
    assert(cumpreInvariante());

    return denominador_;
}

void Racional::escreve() const
{
    assert(cumpreInvariante());

    cout << numerador_;
    if(denominador_ != 1)
        cout << '/' << denominador_;
}

...

bool Racional::cumpreInvariante() const
{
    return 0 < denominador_ and mdc(numerador_, denominador_) == 1;
}

...
```

Note-se que nas operações que garantem a constância da instância implícita, tendo-se verificado a veracidade da condição invariante de classe no seu início, não é necessário voltar a verificá-la no seu final. Note-se também que, pela sua natureza, a operação que indica se a condição invariante de instância se verifica, tipicamente chamada `cumpreInvariante()`, é uma operação constante.

É interessante verificar que uma classe C++ tem duas interface distintas. A primeira, mais pequena, é a interface disponível para utilização com constantes dessa classe, e consiste no conjunto das operações que garantem a constância da instância implícita. A segunda, que engloba a primeira, é a interface disponível para utilização com variáveis da classe.

Finalmente, é muito importante pensar logo nas operações de uma classe como sendo ou não constantes, ou melhor, como garantindo ou não a constância da instância implícita, e não fazê-lo à posteriori, como neste capítulo! O desenvolvimento do TAD `Racional` feito neste capítulo não é feito pela ordem mais apropriada na prática (para isso ver o próximo capítulo), mas sim pela ordem que se julgou mais conveniente pedagogicamente para introduzir os muitos conceitos associados a classes C++ que o leitor tem de dominar para as desenhar com proficiência.

7.11.3 Devolução por valor constante

Outro assunto relacionado com a constância é a devolução de constantes. A ideia de devolver constantes pode parecer estranha à primeira vista, mas repare-se no seguinte código:

```
Racional r1(1, 2), r2(3, 2);
++(r1 + r2);
```

Que faz ele? Define duas variáveis `r1` e `r2`, soma-as, e finalmente *incrementa a variável temporária* devolvida pelo operador `+`. Tal código é mais provavelmente fruto de erro do programador do que algo desejado. Além disso, semelhante código seria proibido se em vez de racionais as variáveis fossem do tipo `int`. Como se pretende que o TAD `Racional` possa ser usado como qualquer tipo básico da linguagem, é desejável encontrar uma forma de proibir a invocação de operações modificadoras através de instâncias temporárias.

À luz da discussão na secção anterior, é fácil perceber que o problema se resolve se as funções que devolvem instâncias temporárias de classes C++, i.e., as funções que devolvem instâncias de classes C++ por valor, forem alteradas de modo a devolverem constantes temporárias, e não variáveis. No caso do TAD em desenvolvimento, são apenas as rotinas que sobrecarregam os operadores aritméticos usuais e os operadores de incrementação e decrementação sufixo que precisam de ser alteradas:

```
/** Adição de dois racionais.
    @pre r1 = r₁∧.
    @post operator+ = r₁ + r2. */
Racional const operator+(Racional r1, Racional const& r2)
{
    return r1 += r2;
}

/** Subtracção de dois racionais.
    @pre r1 = r₁∧.
    @post operator- = r₁ − r2. */
Racional const operator-(Racional r1, Racional const& r2)
{
    return r1 -= r2;
}

/** Produto de dois racionais.
    @pre r1 = r₁.
    @post operator* = r₁ × r2. */
```

```
Racional const operator*(Racional r1, Racional const& r2)
{
    return r1 *= r2;
}
```

/** Divisão de dois racionais.
@pre `r1` $= r_1 \wedge$ `r2` $\neq 0$.
@post `operator/` $= r_1/$`r2`. */

```
Racional const operator/(Racional r1, Racional const& r2)
{
    assert(r2 != 0);

    return r1 /= r2;
}
```

/** Incrementa o racional recebido como argumento, devolvendo o seu valor antes de incrementado.
@pre `*this` $= r$.
@post `operator++` $= r \wedge$ `*this` $= r + 1$. */

```
Racional const operator++(Racional& r, int valor_a_ignorar)
{
    Racional const cópia = r;

    ++r;

    return cópia;
}
```

/** Decrementa o racional recebido como argumento, devolvendo o seu valor antes de decrementado.
@pre `*this` $= r$.
@post `operator-` $= r \wedge$ `*this` $= r - 1$. */

```
Racional const operator--(Racional& r, int)
{
    Racional const cópia = r;

    --r;

    return cópia;
}

...
```

Ficaram a faltar ao TAD `Racional` os operadores + e − unários. Começar-se-á pelo segundo.

O operador − unário pode ser sobrecarregado quer através de uma operação da classe C++ `Racional`

```
...

class Racional {
  public:

    ...

    /** Devolve simétrico do racional.
        @pre V.
        @post operator- = -*this. */
    Racional const operator-() const;

    ...
};

...

Racional const Racional::operator-() const
{
    assert(cumpreInvariante());

    Racional r;
    r.numerador_ = -numerador_;
    r.denominador_ = denominador_;

    assert(r.cumpreInvariante());

    return r;
}

...
```

quer através de uma função normal

```
Racional const operator-(Racional const& r)
{
    return Racional(-r.numerador(), r.denominador());
}
```

Embora a segunda versão seja muito mais simples, ela implica a invocação do construtor mais complicado da classe C++, que verifica o sinal do denominador e reduz a fracção correspondente ao numerador e denominador passados como argumento. Neste caso essas verificações são inúteis, pois o denominador não varia, mantendo-se positivo, e mudar o sinal do numerador mantém numerador e denominador mutuamente primos. Assim, é preferível a primeira versão, onde se constrói um racional usando o construtor por omissão, que é muito eficiente, e em seguida se alteram directamente e sem mais verificações os valores do numerador e denominador. Em qualquer dos casos é devolvido um racional por valor e, por isso, constante.

7.11.4 Devolução por referência constante

Em alguns casos também é possível utilizar devolução por referência constante. Esta têm a vantagem de ser mais eficiente do que a devolução por valor, podendo ser utilizada quando o valor a devolver não for uma variável local à função, nem uma instância temporária construída dentro da função, pois tal redundaria na devolução de um sinónimo constante de uma instância entretanto destruída... É o caso do operador + unário que, por uma questão de simetria, se sobrecarrega por intermédio de uma operação da classe C++ `Racional`:

```
...

class Racional {
  public:

    ...

    /** Devolve versão constante do racional.
        @pre V.
        @post operator+ ≡ *this. */
    Racional const& operator+() const;

    ...
};

...

Racional const& Racional::operator+() const
{
    assert(cumpreInvariante());

    return *this;
}

...
```

Como contra exemplo, suponha-se que a rotina que sobrecarrega o operador ++ sufixo devolvia por referência constante:

```
/** Incrementa o racional recebido como argumento, devolvendo o seu valor an-
tes de incrementado.
    @pre *this = r.
    @post operator++ = r ∧ *this = r + 1. */
Racional const& operator++(Racional& r, int)
{
    Racional const cópia = r;
```

```
        ++r;

        return cópia; // Erro! Devolução de referência para variável local!
    }
```

Seria claramente um erro fazê-lo, pois seria devolvida uma referência para uma instância local, que é destruída logo que a função retorna.

7.12 Reduzindo o número de invocações com `inline`

O mecanismo de invocação de rotinas (membro ou não) implica tarefas de "arrumação da casa" algo morosas, como se viu na Secção 3.4: é necessário colocar na pilha o endereço de retorno e os respectivo argumentos, executar as instruções do corpo da rotina, depois retirar os argumentos da pilha, e retornar, eventualmente devolvendo o resultado no seu topo. Logo, a invocação de rotinas pode ser, em alguns casos, um factor limitador da eficiência dos programas. Suponha-se as instruções:

```
Racional r(1, 3);
Racional s = r + 2;
```

Quantas invocações de rotinas são feitas neste código? A resposta é surpreendente, mesmo ignorando as instruções de asserção (que aliás podem ser facilmente "desligadas"):

1. O construtor para construir `r`, que invoca
2. a operação `Racional::reduz()`, cujo método invoca
3. a função `mdc()`.
4. O construtor para converter implicitamente o valor literal 2 num racional.
5. O construtor por cópia (ver Secção 7.4.2) para copiar o argumento `r` para o parâmetro `r1` durante a invocação d'
6. a função `operator+`, que invoca
7. a operação `Racional::operator+=`, cujo método invoca
8. a operação `Racional::reduz()`, cujo método invoca
9. a função `mdc()`.
10. O construtor por cópia para devolver `r1` por valor na função `operator+`.
11. O construtor por cópia para construir a variável `s` à custa da constante temporária devolvida pela função `operator+`.

Mesmo tendo em conta que o compilador pode eventualmente optimizar algumas destas invocações, 11 invocações para duas inocentes linhas de código parece demais. Não será lento? Como evitá-lo?

A linguagem C++ fornece uma forma simples de reduzir o peso da "arrumação da casa" aquando da invocação de uma rotina: rotinas muito simples, tipicamente não fazendo uso de ciclos e consistindo em apenas duas ou três linhas (excluindo instruções de asserção), podem qualificadas como *em-linha* ou *inline*. A palavra-chave `inline` pode ser usada para este efeito, qualificando-se com ela as definições das rotinas que se deseja que sejam em-linha.

Mas o que significa a definição de uma rotina ser em-linha? Que o compilador, se lhe parecer apropriado (e o compilador pode-se recusar a fazê-lo) em vez de traduzir o código da rotina em linguagem máquina, colocá-lo num único local do programa executável e chamá-lo quando necessário, coloca o código da rotina em linguagem máquina directamente nos locais onde ela deveria ser invocada.

Por exemplo, é natural que o código máquina produzido por

```
inline int soma(int const& a, int const& b)
{
    return a + b;
}

int main()
{
    int x1 = 10;
    int x2 = 20;
    int x3 = 30;
    int r = 0;

    r = soma(x1, x2); r = soma(r, x3);
}
```

seja idêntico ao produzido por

```
int main()
{
    int x1 = 10;
    int x2 = 20;
    int x3 = 30;
    int r = 0;

    r = x1 + x2;
    r = r + x3;
}
```

Para melhor compreender o que foi dito, é boa ideia fazer uma digressão pela linguagem *assembly*, aliás a única nestas folhas. Para isso recorrer-se-á à máquina MAC-1, desenvolvida por Andrew Tanenbaum para fins pedagógicos e apresentada em [13, Secção 4.3] (ver também MAC-1 asm `http://www.daimi.aau.dk/~bentor/html/useful/asm.html`).

A tradução para o *assembly* do MAC-1 do programa original é:

Se não levasse em conta o qualificador `inline`, um compilador de C++ para *assembly* MAC-1 poderia gerar:

```
        jump main

        # Variáveis:
        x1 = 10
        x2 = 20
        x3 = 30
        r = 0
main:   # Programa principal:
        lodd x1   # Carrega variável x1 no acumulador.
        push      # Coloca acumulador no topo da pilha.
        lodd x2   # Carrega variável x2 no acumulador.
        push      # Coloca acumulador no topo da pilha.
        # Aqui a pilha tem os dois argumentos x1 e x2:
        call soma # Invoca a função soma().
        insp 2    # Repõe a pilha.
        # Aqui o acumulador tem o valor devolvido.
        stod r    # Guarda o acumulador na variável r.

        lodd r
        push
        lodd x3
        push
        # Aqui a pilha tem os dois argumentos r e x3:
        call soma
        insp 2
        # Aqui o acumulador tem o valor devolvido.
        stod r

        halt

soma:   # Função soma():
        lodl 1    # Carrega no acumulador o segundo parâmetro.
        addl 2    # Adiciona primeiro parâmetro ao acumulador.
        retn      # Retorna, devolvendo resultado no acumulador.
```

Se levasse em conta o qualificador `inline`, um compilador de C++ para *assembly* MAC-1 provavelmente geraria:

```
        jump main
```

```
        # Variáveis:

        x1 = 10
        x2 = 20
        x3 = 30
        r = 0

main:   # Programa principal:
        lodd x1    # Carrega variável x1 no acumulador.
        addd x2    # Adiciona variável x2 ao acumulador.
        stod r     # Guarda o acumulador na variável r.

        lodd r
        addd x3
        stod r

        halt
```

A diferença entre os dois programas em *assembly* é notável. O segundo é claramente mais rápido, pois evita todo o mecanismo de invocação de funções. Mas também é mais curto, ou seja, ocupa menos espaço na memória do computador! Embora normalmente haja sempre um ganho em termos do número de instruções a efectuar, se o código a colocar em-linha for demasiado extenso, o programa pode-se tornar mais longo, o que pode inclusivamente levar ao esgotamento da memória física, levando à utilização da memória virtual do sistema operativo, que tem a lamentável característica de ser ordens de grandeza mais lenta. Assim, é necessário usar o qualificador `inline` com conta, peso e medida.

Para definir uma operação como em-linha, pode-se fazer uma de duas coisas:

1. Ao definir a classe C++, definir logo o método (em vez de a declarar apenas a respectiva operação).
2. Ao definir o método correspondente à operação declarada na definição da classe, preceder o seu cabeçalho do qualificador `inline`.

Em geral a segunda alternativa é preferível à primeira, pois torna mais evidente a separação entre a interface e a implementação da classe, separando claramente operações de métodos.

A definição de uma rotina, membro ou não-membro, como em-linha não altera a semântica da sua invocação, tendo apenas consequências em termos da tradução do programa para código máquina.

No caso do código em desenvolvimento neste capítulo, relativo ao TAD `Racional`, todas as rotinas são suficientemente simples para que se justifique a utilização do qualificador `inline`, com excepção apenas da função `mdc()`, por envolver um ciclo, e da operação `Racional::lê()`, por ser demasiado extensa. Exemplificando apenas para o primeiro construtor da classe:

```
...

class Racional {
  public:
    /** Constrói racional com valor inteiro. Construtor por omissão.
        @pre V.
        @post *this = n. */
    Racional(int const n = 0);

    ...
};

inline Racional::Racional(int const n)
    : numerador_(n), denominador_(1)
{

    assert(cumpreInvariante());
    assert(numerador_ == n * denominador_);
}

...
```

7.13 Optimização dos cálculos com racionais

Um dos problemas com a representação escolhida para a classe C++ `Racional` é o facto de os atributos `numerador_` e `denominador_` serem do tipo `int`, que tem limitações devidas à sua representação na memória do computador. Essa foi parte da razão pela qual se insistiu em que os racionais fossem sempre representados pelo numerador e denominador de uma fracção no formato canónico, i.e., com denominador positivo e formando uma fracção reduzida. No entanto, esta escolha não é suficiente. Basta olhar para a definição da função que sobrecarrega o operador < para a classe C++ `Racional`

```
/** Indica se o primeiro racional é menor que o segundo.
    @pre V.
    @post operator< = (r1 < r2). */
inline bool operator<(Racional const& r1, Racional const& r2)
{
    return r1.numerador() * r2.denominador() <
           r2.numerador() * r1.denominador();
}
```

para se perceber imediatamente que, mesmo que os racionais sejam representáveis, durante cálculos intermédios que os envolvam podem ocorrer transbordamentos. No entanto, embora seja impossível eliminar totalmente a possibilidade de transbordamentos (excepto eventualmente abandonando o tipo `int` e usando um TAD representando números inteiros de dimensão arbitrária), é possível minorar o seu impacte. Por exemplo, no caso do operador < é possível encontrar divisores comuns aos numeradores e aos denominadores dos racionais a comparar e usá-los para reduzir ao máximo a magnitude dos inteiros a comparar:

```
/** Indica se o primeiro racional é menor que o segundo.
    @pre V.
    @post operator< = (r1 < r2). */
inline bool operator<(Racional const& r1, Racional const& r2)
{
    int dn = mdc(r1.numerador(), r2.numerador());
    int dd = mdc(r1.denominador(), r2.denominador());

    return (r1.numerador() / dn) * (r2.denominador() / dd) <
           (r2.numerador() / dn) * (r1.denominador() / dd);
}
```

As mesmas ideias podem ser aplicadas a outras operações, pelo que se discutem nas secções seguintes. Durante estas secções admite-se que as fracções originais ($\frac{n}{d}$, $\frac{n_1}{d_1}$ e $\frac{n_2}{d_2}$) estão no formato canónico. Recorda-se também que se admite uma extensão da função mdc de tal forma que $\mathrm{mdc}(0,0) = 1$.

7.13.1 Adição e subtracção

O resultado da soma de fracções dado por

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 \times d_2 + n_2 \times d_1}{d_1 \times d_2},$$

embora tenha forçosamente o denominador positivo, pode não estar no formato canónico. Se $k = \mathrm{mdc}(d_1, d_2)$ e $l = \mathrm{mdc}(n_1, n_2)$, então, dividindo ambos os termos da fracção resultado por k e pondo l em evidência,

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{l \times (n'_1 \times d'_2 + n'_2 \times d'_1)}{k \times d'_1 \times d'_2},$$

onde $d'_1 = d_1/k$ e $d'_2 = d_2/k$ são mutuamente primos, i.e., $\mathrm{mdc}(d'_1, d'_2) = 1$, e $n'_1 = n_1/l$ e $n'_2 = n_2/l$ são mutuamente primos, i.e., $\mathrm{mdc}(n'_1, n'_2) = 1$.

Este novo resultado, apesar da divisão por k de ambos os termos da fracção, pode ainda não estar no formato canónico, pois pode haver divisores não-unitários comuns ao numerador e ao denominador. Repare-se no exemplo

$$\frac{1}{10} + \frac{1}{15},$$

em que $k = \mathrm{mdc}(10, 15) = 5$. Aplicando a equação acima obtém-se

$$\frac{1}{10} + \frac{1}{15} = \frac{1 \times 3 + 1 \times 2}{5 \times 2 \times 3} = \frac{5}{30}.$$

Neste caso, para reduzir a fracção aos termos mínimos é necessário dividir ambos os termos da fracção por 5.

Em vez de tentar reduzir a fracção resultado tomando quer o numerador quer o denominador como um todo, é preferível verificar primeiro se é possível haver divisores comuns entre os respectivos factores. Considerar-se-ão dois factores para o numerador (l e $n'_1 \times d'_2 + n'_2 \times d'_1$) e dois factores para o denominador (k e $d'_1 \times d'_2$), num total de quatro combinações onde é possível haver divisores comuns.

Será que podem haver divisores não-unitários comuns a l e a k? Suponha-se que existe um divisor $1 < i$ comum a l e a k. Nesse caso, dado que $d_1 = d'_1 \times k$ e $n_1 = n'_1 \times l$, ter-se-ia de concluir que $i \leq \mathrm{mdc}(n_1, d_1)$, ou seja, $i \leq 1$, o que é uma contradição. Logo, l e a k não têm divisores comuns não-unitários.

Será que pode haver divisores não-unitários comuns a l e a $d'_1 d'_2$? Suponha-se que existe um divisor $1 < i$ comum a l e a $d'_1 d'_2$. Nesse caso, existe forçosamente um divisor $1 < j$ comum a l e a d'_1 ou a d'_2. Se j for divisor comum a l e a d'_1, então j é também divisor comum a n_1 e a d_1, ou seja, $j \leq \mathrm{mdc}(n_1, d_1)$, donde se conclui que $j \leq 1$, o que é uma contradição. O mesmo argumento se aplica se j for divisor comum a l e a d'_2. Logo, l e $d'_1 d'_2$ não têm divisores comuns não-unitários.

Será que podem haver divisores não-unitários comuns a $n'_1 \times d'_2 + n'_2 \times d'_1$ e a $d'_1 \times d'_2$? Suponha-se que existe um divisor $1 < h$ comum $n'_1 \times d'_2 + n'_2 \times d'_1$ e de $d'_1 \times d'_2$. Nesse caso, existe forçosamente um divisor $1 < i$ comum a $n'_1 \times d'_2 + n'_2 \times d'_1$ e a d'_1 ou a d'_2. Seja então $1 < i$ um divisor comum a $n'_1 \times d'_2 + n'_2 \times d'_1$ e a d'_1. Nesse caso tem de existir um divisor $1 < j$ comum a d'_1 e a n'_1 ou a d'_2. Isso implicaria que $j \leq \mathrm{mdc}(n_1, d_1)$ ou que $j \leq \mathrm{mdc}(d'_1, d'_2) \neq 1$. Em qualquer dos casos conclui-se que $j \leq 1$, o que é uma contradição. O mesmo argumento se aplica se $1 < i$ for divisor comum a $n'_1 \times d'_2 + n'_2 \times d'_1$ e a d'_2. Logo, $n'_1 \times d'_2 + n'_2 \times d'_1$ e $d'_1 \times d'_2$ não têm divisores comuns não-unitários.

Assim, a existirem divisores não-unitários comuns ao denominador e numerador da fracção

$$\frac{l \times (n'_1 \times d'_2 + n'_2 \times d'_1)}{k \times d'_1 \times d'_2},$$

eles devem-se à existência de divisores não-unitários comuns a $n'_1 \times d'_2 + n'_2 \times d'_1$ e a k. Assim, sendo

$$m = \mathrm{mdc}(n'_1 \times d'_2 + n'_2 \times d'_1, k),$$

a fracção

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{l \times ((n'_1 \times d'_2 + n'_2 \times d'_1)/m)}{(k/m) \times d'_1 \times d'_2},$$

está no formato canónico.

Qual foi a vantagem de factorizar l e k e proceder aos restantes cálculos face à alternativa, mais simples, de calcular a fracção como

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{(n_1 \times d_2 + n_2 \times d_1)/h}{(d_1 \times d_2)/h},$$

com $h = \mathrm{mdc}(n_1 \times d_2 + n_2 \times d_1, d_1 \times d_2)$?

A vantagem é meramente computacional. Apesar de os cálculos propostos exigirem mais operações, os valores intermédios dos cálculos são em geral mais pequenos, o que minimiza a possibilidade de existirem valores intermédios que não sejam representáveis em valores do tipo `int`, evitando-se assim transbordamentos.

A fracção canónica correspondente à adição pode ser portanto calculada pela equação acima. A fracção canónica correspondente à subtracção pode ser calculada por uma equação semelhante

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{l \times ((n'_1 \times d'_2 - n'_2 \times d'_1)/m)}{(k/m) \times d'_1 \times d'_2}.$$

Pode-se agora actualizar a definição dos métodos `Racional::operator+=` e `Racional::operator-=` para:

```
Racional& Racional::operator+=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    int dn = mdc(numerador_, r2.numerador_);
    int dd = mdc(denominador_, r2.denominador_);

    // Devido a r += r:
    int d2 = r2.denominador_;

    numerador_ /= dn;
    denominador_ /= dd;

    numerador_ = numerador_ * (d2 / dd) +
        r2.numerador_ / dn * denominador_;

    dd = mdc(numerador_, dd);

    numerador_ = dn * (numerador_ / dd);
    denominador_ *= d2 / dd;

    assert(cumpreInvariante());

    return *this;
}
```

```
Racional& Racional::operator-=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    int dn = mdc(numerador_, r2.numerador_);
    int dd = mdc(denominador_, r2.denominador_);

    // Devido a r -= r:
    int d2 = r2.denominador_;

    numerador_ /= dn;
    denominador_ /= dd;

    numerador_ = numerador_ * (d2 / dd) -
        r2.numerador_ / dn * denominador_;

    dd = mdc(numerador_, dd);

    numerador_ = dn * (numerador_ / dd);
    denominador_ *= d2 / dd;

    assert(cumpreInvariante());

    return *this;
}
```

Uma vez que ambos os métodos ficaram bastante extensos, decidiu-se retirar-lhes o qualificador `inline`.

7.13.2 Multiplicação

Relativamente à multiplicação de fracções,

$$\frac{n_1}{d_1} \times \frac{n_2}{d_2} = \frac{n_1 \times n_2}{d_1 \times d_2},$$

apesar de o denominador ser forçosamente positivo, é possível que o resultado não esteja no formato canónico, bastando para isso que existam divisores não-unitários comuns a n_1 e d_2 ou a d_1 e n_2. É fácil verificar que, sendo $k = \mathrm{mdc}(n_1, d_2)$ e $l = \mathrm{mdc}(n_2, d_1)$, a fracção

$$\frac{n_1}{d_1} \times \frac{n_2}{d_2} = \frac{(n_1/k) \times (n_2/l)}{(d_1/l) \times (d_2/k)}$$

está, de facto, no formato canónico.

Pode-se agora actualizar a definição do método `Racional::operator*=` para:

```
inline Racional& Racional::operator*=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    int n1d2 = mdc(numerador_, r2.denominador_);
    int n2d1 = mdc(r2.numerador_, denominador_);

    numerador_ = (numerador_ / n1d2) * (r2.numerador_ / n2d1);
    denominador_ = (denominador_ / n2d1) * (r2.denominador_ / n1d2);

    assert(cumpreInvariante());

    return *this;
}
```

7.13.3 Divisão

O caso da divisão de fracções,

$$\frac{n_1}{d_1}/\frac{n_2}{d_2} = \frac{n_1 \times d_2}{d_1 \times n_2} \text{ se } n_2 \neq 0,$$

é muito semelhante ao da multiplicação, sendo mesmo possível usar os métodos acima para a calcular. Em primeiro lugar é necessário garantir que $n_2 \neq 0$. Se $n_2 = 0$ a divisão não está definida. Admitindo que $n_2 \neq 0$, então a divisão é equivalente a uma multiplicação:

$$\frac{n_1}{d_1}/\frac{n_2}{d_2} = \frac{n_1}{d_1} \times \frac{d_2}{n_2}.$$

No entanto, é necessário verificar se n_2 é positivo, pois de outra forma o resultado da multiplicação não estará no formato canónico, uma vez que terá denominador negativo. Se $0 < n_2$, a divisão é calculada multiplicando as fracções canónicas $\frac{n_1}{d_1}$ e $\frac{d_2}{n_2}$. Se $n_2 < 0$, multiplicam-se as fracções canónicas $\frac{n_1}{d_1}$ e $\frac{-d_2}{-n_2}$. Para garantir que o resultado está no formato canónico usa-se a mesma técnica que para a multiplicação.

Pode-se agora actualizar a definição do método `Racional::operator/=` para:

```
inline Racional& Racional::operator/=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    assert(r2 != 0);

    int dn = mdc(numerador_, r2.numerador_);
    int dd = mdc(denominador_, r2.denominador_);

    if(r2.numerador_ < 0) {
```

```
            numerador_ = (numerador_ / dn) * (-r2.denominador_ / dd);
            denominador_ = (denominador_ / dd) * (-r2.numerador_ / dn);
        } else {
            numerador_ = (numerador_ / dn) * (r2.denominador_ / dd);
            denominador_ = (denominador_ / dd) * (r2.numerador_ / dn);
        }

        assert(cumpreInvariante());

        return *this;
    }
```

7.13.4 Simétrico e identidade

O caso das operações de cálculo do simétrico e da identidade,

$$\begin{aligned} -\frac{n}{d} &= \frac{-n}{d} \text{ e} \\ +\frac{n}{d} &= \frac{n}{d}, \end{aligned}$$

não põe qualquer problema, pois os resultados estão sempre no formato canónico.

7.13.5 Operações de igualdade e relacionais

Sendo dois racionais r_1 e r_2 e as respectivas representações na forma de fracções canónicas $r_1 = \frac{n_1}{d_1}$ e $r_2 = \frac{n_2}{d_2}$, é evidente que $r_1 = r_2$ se e só se $n_1 = n_2 \wedge d_1 = d_2$. Da mesma forma, $r_1 \neq r_2$ se e só se $n_1 \neq n_2 \vee d_1 \neq d_2$.

Relativamente aos mesmos dois racionais, a expressão $r_1 < r_2$ é equivalente a $\frac{n_1}{d_1} < \frac{n_2}{d_2}$ ou ainda a $n_1 d_2 < n_2 d_1$, pois ambos os denominadores são positivos. Assim, é possível comparar dois racionais usando apenas comparações entre inteiros. Os inteiros a comparar podem ser reduzidos calculando $k = \mathrm{mdc}(d_1, d_2)$ e $l = \mathrm{mdc}(n_1, n_2)$ e dividindo os termos apropriados da expressão:

$$(n_1/l)\,(d_2/k) < (n_2/l)\,(d_1/k)\,.$$

Da mesma forma podem-se reduzir todas as comparações entre racionais (com $<$, $>$, $\geq$ ou $\leq$) às correspondentes comparações entre inteiros.

Pode-se agora actualizar a definição da rotina `operator<`:

```
/** Indica se o primeiro racional é menor que o segundo.
    @pre V.
    @post operator< = (r1 < r2). */
inline bool operator<(Racional const& r1, Racional const& r2)
{
    int dn = mdc(r1.numerador(), r2.numerador());
```

```
    int dd = mdc(r1.denominador(), r2.denominador());

    return (r1.numerador() / dn) * (r2.denominador() / dd) <
           (r2.numerador() / dn) * (r1.denominador() / dd);
}
```

7.13.6 Operadores especiais

O TAD `Racional` tal como concretizado até agora, suporta operações simultâneas entre racionais e inteiros, sendo para isso fundamental a conversão implícita entre valores do tipo `int` e o tipo `Racional` fornecida pelo primeiro construtor da respectiva classe C++. No entanto, é instrutivo seguir a ordem dos acontecimentos quando se calcula, por exemplo, a soma de um racional com um inteiro:

```
Racional r(1, 2);

cout << r + 1 << endl;
```

A soma implica as seguintes invocações:

1. Construtor da classe C++ `Racional` para converter o inteiro 1 no correspondente racional.
2. Rotina `operator+()`.
3. Operação `Racional::operator+=()`.

Será possível evitar a conversão do inteiro em racional e, sobretudo, evitar calcular a soma de um racional com um inteiro recorrendo à complicada maquinaria necessária para somar dois racionais? Certamente. Basta fornecer versões especializadas para operandos inteiros das sobrecargas dos operadores em causa:

```
...

class Racional {
  public:

    ...

    /** Adiciona de um inteiro.
        @pre *this = r.
        @post operator+= ≡ *this ∧ *this = r + n. */
    Racional& operator+=(int const n);

    ...
```

```
};

...

Racional& Racional::operator+=(int const i)
{
    assert(cumpreInvariante());

    numerador_ += i * denominador_;

    assert(cumpreInvariante());

    return *this;
}

/** Adição de um racional e um inteiro.
    @pre r = r.
    @post operator+ = r + i. */
inline Racional const operator+(Racional r, int const i)
{
    return r += i;
}

/** Adição de um inteiro e um racional.
    @pre r = r.
    @post operator+ = i + r. */
inline Racional const operator+(int const i, Racional r)
{
    return r += i;
}
```

Fica como exercício para o leitor desenvolver as sobrecargas de operadores especializadas em todos os outros casos em que se possam fazer operações conjuntas entre inteiros e racionais.

7.14 Operadores de inserção e extracção

É possível e desejável sobrecarregar o operador << de inserção num canal e o operador >> de extracção de um canal. Aliás, estas sobrecargas são, digamos, a cereja sobre o bolo. São o toque final que permite escrever o programa da soma de racionais exactamente da mesma forma como se faria se se pretendesse somar inteiros:

```
int main()
{
    // Ler fracções:
```

```
    cout << "Introduza duas fracções (numerador denominador): ";
    Racional r1, r2;
    cin >> r1 >> r2;

    if(not cin) {
        cerr << "Opps!  A leitura dos racionais falhou!" << endl;
        return 1;
    }

    // Calcular racional soma:
    Racional r = r1 + r2;

    // Escrever resultado:
    cout << "A soma de " << r1 << " com " << r2
         << " é " << r << '.' << endl;
}
```

7.14.1 Sobrecarga do operador <<

Começar-se-á pelo operador de inserção, por ser mais simples. O operador << é binário, tendo por isso dois operandos. Por exemplo, se se pretender escrever um valor inteiro no ecrã pode-se usar a instrução

```
cout << 10;
```

onde o primeiro operando, `cout`, é um canal de saída, ligado normalmente ao ecrã, e 10 é um valor literal inteiro. O efeito da operação é fazer surgir o valor 10 no ecrã. O segundo operando é claramente do tipo `int`, mas, qual é o tipo do primeiro operando? Aliás, o que é `cout`? A resposta a estas perguntas é muito importante para a sobrecarga do operador << desejada: o primeiro operando, `cout` é uma variável global do tipo `ostream`, ou seja, *canal de saída*. Ambos, variável `cout` e tipo `ostream`, estão declarados no ficheiro de interface[22] `iostream`. Por outro lado, o operador << é um operador binário como qualquer outro, e por isso tem associatividade à esquerda. Isso quer dizer que a instrução

```
cout << 10 << ' ';
```

é interpretada como

```
(cout << 10) << ' ';
```

A primeira operação com o operador << faz-se, por isso, com o primeiro operando do tipo `ostream` e o segundo do tipo `char`. A segunda operação com o operador << faz-se claramente com o segundo operando do tipo `char`. De que tipo será o primeiro operando nesse

[22]Um ficheiro de interface é usado na modularização física para permitir a um módulo físico (ficheiro) usar ferramentas definidas noutro módulo físico. Estes assuntos são matéria do Capítulo 9.

caso? E que valor possui? As respostas são evidentes se se lembrar que a instrução acima é equivalente a

```
cout << 10;
cout << ' ';
```

É claro que o primeiro operando da segunda operação com o operador << tem de ser não apenas do tipo `ostream`, para que o segundo operando seja inserido num canal, mas deve ser exactamente o canal `cout`, para que a inserção se faça no local correcto.

A sobrecarga do operador << não se pode fazer à custa de uma operação da classe `Racional`, pois o primeiro operando do operador deve ser do tipo `ostream`. Sendo este tipo uma classe, quando muito o operador << poderia ser sobrecarregado por uma operação dessa classe. Mas como a classe está pré-definida na biblioteca padrão do C++, não é possível fazê-lo. Assim, a sobrecarga será feita usando uma rotina normal, e por isso com dois parâmetros. O primeiro parâmetro corresponderá ao canal onde se deve realizar a operação de inserção, e o segundo ao racional a inserir. Como o canal é certamente modificado pela operação de inserção, terá de ser passado por referência:

```
tipo_de_devolução operator<<(ostream& saída, Racional const& r);
```

O tipo de devolução, para que o canal seja passado sucessivamente em instruções de inserção encadeadas tais como

```
Racional r1, r2;

...

cout << r1 << ' ' << r2 << endl;
```

terá naturalmente de ser `ostream&`, ou seja, o canal terá de ser devolvido por referência. Aliás, a justificação para o fazer é idêntica à que se usou para os operadores de incrementação e decrementação prefixo e para os operadores especiais de atribuição. Ou seja, a definição da rotina `operator<<` deverá tomar a forma:

```
/** Insere o racional no canal de saída no formato de uma fracção.
    @pre 𝒱.
    @post ¬saída∨ saída contém n/d (ou simplesmente n se d = 1)
          sendo n/d a fracção canónica correspondente ao racional r. */
ostream& operator<<(ostream& saída, Racional const& r)
{
    ...

    return saída;
}
```

Dada a existência de operações de inspecção que permitem obter o numerador e o denominador da fracção canónica correspondente a um racional, seria perfeitamente possível eliminar a operação `Racional::escreve()` da classe C++ `Racional` e usar o seu corpo, com adaptações, como corpo da rotina `operator<<`. No entanto, adoptar-se-á uma solução diferente. Manter-se-á a operação `Racional::escreve()`, embora com um nome mais apropriado, e implementar-se-á a rotina `operator<<` à sua custa. Para isso é fundamental tornar a operação mais genérica, de modo a inserir o racional num canal arbitrário:

```
...

class Racional {
  public:
    ...

    /** Insere o racional no canal no formato de uma fracção.
        @pre V.
        @post ¬saída∨ saída contém n/d (ou simplesmente n se d = 1)
              sendo n/d a fracção canónica correspondente ao racional *this. */
    void insereEm(ostream& saída) const;

    ...
};

...

...
inline void Racional::insereEm(ostream& saída) const
{
    assert(cumpreInvariante());

    saída << numerador_;
    if(denominador_ != 1)
        saída << '/' << denominador_;
}

...

/** Insere o racional no canal de saída no formato de uma fracção.
    @pre V.
    @post ¬saída∨ saída contém n/d (ou simplesmente n se d = 1)
          sendo n/d a fracção canónica correspondente ao racional r. */
ostream& operator<<(ostream& saída, Racional const& r)
{
    r.insereEm(saída);

    return saída;
```

```
}

...
```

7.14.2 Sobrecarga do operador > >

O caso do operador > > é muito semelhante, embora neste caso o primeiro operando do operador seja do tipo `istream`, ou seja, *canal de entrada*, e se deva passar o racional por referência, para permitir a sua alteração:

```
...

class Racional {
  public:
    ...

    /** Extrai do canal um novo valor para o racional, na forma de dois inteiros sucessivos.
        @pre *this = r.
        @post Se entrada e entrada tem dois inteiros n' e d' disponíveis para leitura, com d' ≠ 0, então
                  *this = n'/d' ∧ entrada,
              senão
                  *this = r ∧ ¬entrada. */
    void extraiDe(istream& entrada);

    ...
};

...

...
void Racional::extraiDe(istream& entrada)
{
    assert(cumpreInvariante());

    int n, d;

    if(entrada >> n >> d)
        if(d == 0)
            entrada.setstate(ios_base::failbit);
        else {
            numerador_ = d < 0 ? -n : n;
            denominador_ = d < 0 ? -d : d;
```

```
            reduz();

            assert(cumpreInvariante());
            assert(numerador_ * d == n * denominador_ and cin);

            return;
        }

    assert(cumpreInvariante());
    assert(not entrada);
}

...

/** Extrai do canal um novo valor para o racional, na forma de dois inteiros sucessivos.
    @pre r = r.
    @post Se entrada e entrada tem dois inteiros n' e d' disponíveis para lei-
tura, com d' ≠ 0, então
            r = n'/d' ∧ entrada,
        senão
            r = r ∧ ¬entrada. */
istream& operator>>(istream& entrada, Racional& r)
{
    r.extraiDe(entrada);

    return entrada;
}

...
```

Neste caso a vantagem de implementar a rotina `operator>>` à custa de uma operação correspondente na classe C++ fica mais clara. Como a rotina `operator>>` não pode ser membro da classe e, no entanto, necessita de alterar os atributos da classe, a solução foi delegar a tarefa da extracção ou leitura para uma operação da classe, que não tem quaisquer restrições de acesso.

7.14.3 Lidando com erros

No início deste capítulo apresentou-se a primeira versão da rotina de leitura de racionais, então vistos simplesmente como fracções, sem mais explicações. Chegou agora a altura de explicar o código dessa rotina, entretanto convertida na operação `Racional::extraiDe()`, apresentada abaixo sem instruções de asserção, i.e., reduzida ao essencial:

```
void Racional::extraiDe(istream& entrada)
{
     int n, d;
```

```
        if(entrada >> n >> d)
            if(d == 0)
                entrada.setstate(ios_base::failbit);
            else {
                numerador_ = d < 0 ? -n : n;
                denominador_ = d < 0 ? -d : d;

                reduz();
            }
    }
```

Em primeiro lugar, note-se que a extracção do numerador e do denominador se faz, não directamente para os respectivos atributos, mas para duas variáveis criadas para o efeito. De outra forma, se a extracção do numerador tivesse sucesso, mas a extracção do denominador falhasse, a extracção do racional como um todo teria falhado e este teria mudado de valor, o que, para além de ser má ideia, violaria o estabelecido no contrato da operação.

Usa-se o idioma do C++

```
    if(entrada >> n >> d)
        ...
```

para fazer simultaneamente a extracção dos dois valores inteiros canal de entrada e verificar se essa leitura teve sucesso. Esta instrução poderia (e talvez devesse...) ser decomposta em duas:

```
    entrada >> n >> d;
    if(entrada)
        ...
```

A primeira destas instruções serve para fazer as duas extracções. Tal como se viu para o operador <<, a primeira instrução é interpretada como

```
    (entrada >> n) >> d;
```

devido à associatividade esquerda do operador >>. Que acontece quando a primeira extracção falha, por exemplo quando no canal não se encontra uma sequência de dígitos interpretável como um número inteiro, mas sim, por exemplo, caracteres alfabéticos? Nesse caso a primeira extracção não tem qualquer efeito e o canal `entrada` fica em estado de erro. Nesse caso, a segunda falhará também, pois todas as operações de inserção e extracção realizadas sobre um canal em estado de erro estão condenadas ao fracasso. Ou seja, se a primeira operação falhar, a segunda não tem qualquer efeito, o que é equivalente a não ser realizada. Como o valor de um canal interpretado como um valor booleano é verdadeiro se o canal não estiver em estado de erro e falso se estiver, as instruções acima são equivalentes a

```
if(not entrada.fail()) {
    entrada >> n;
    if(not entrada.fail()) {
        entrada >> d;
        if(not entrada.fail())
            ...
    }
}
```

onde `istream::fail()` é uma operação da classe `istream` que indica se o canal está em estado de erro.

Uma vez realizadas as duas extracções com sucesso, é necessário verificar ainda assim se os valores lidos são aceitáveis. Neste caso isso corresponde a verificar se o denominador é nulo. Se for, a leitura deve falhar. I.e., o racional deve manter o valor original *e o canal deve ficar em estado de erro*, de modo a que a falha de extracção possa ser detectada mais tarde. Só dessa forma será possível, por exemplo, escrever o seguinte código:

```
Racional r;

r.extraiDe(cin);

if(not cin){
    cerr << "Opps!  A leitura falhou!" << endl;
    ...
}
```

ou mesmo

```
Racional r;

cin >> r;

if(not cin){
    cerr << "Opps!  A leitura falhou!" << endl;
    ...
}
```

utilizando o operador `>>` já sobrecarregado para os racionais. Pretende-se de novo que o comportamento de um programa usando racionais seja tão semelhante quanto possível do ponto de vista semântico com o mesmo programa usando inteiros.

Assim, no caso de se extrair um denominador nulo, deve-se colocar o canal `entrada` em estado de erro. Para o fazer usa-se a instrução[23]

[23]Também se pode limpar o estado de erro de um canal, usando-se para isso a operação `istream::clear()`:

```
entrada.clear();
```

```
entrada.setstate(ios_base::failbit);
```

A parte restante do código é auto-explicativa.

7.14.4 Coerência entre os operadores << e >>

Tal como sobrecarregados, os operadores << e >> para racionais, ou melhor, as operações `Racional::insereEm()` e `Racional::extraiDe()` não são coerentes: a extracção não suporta o formato usado pela inserção, nomeadamente não espera o símbolo '/' entre os termos da fracção, nem admite a possibilidade de o racional não ter denominador explícito, nomeadamente quando é também um valor inteiro. É excelente ideia que o operador de extracção consiga extrair racionais em qualquer dos formatos produzidos pelo operador de inserção. Para o conseguir é necessário complicar um pouco o método `Racional::extraiDe()`, e indicar claramente o novo formato admissível no contrato das rotinas envolvidas:

```
...

class Racional {
  public:
    ...

    /** Extrai do canal um novo valor para o racional, na forma de uma fracção.
        @pre *this = r.
        @post Se entrada e entrada contém n′/d′ (ou apenas n′, assumindo-
se d′ = 1), em
              que n′ e d′ são inteiros com d′ ≠ 0, então
                *this = n′/d′ ∧ entrada,
              senão
                *this = r ∧ ¬entrada. */
    void extraiDe(istream& entrada);

    ...
};

...

...
void Racional::extraiDe(istream& entrada)
{
    assert(cumpreInvariante());

    int n;
    int d = 1;

    if(entrada >> n) {
```

```
        if(entrada.peek() != '/') {
            numerador_ = n;
            denominador_ = 1;
        } else {
            if(entrada.get() and isdigit(entrada.peek()) and
               entrada >> d and d != 0)
            {
                numerador_ = d < 0 ? -n : n;
                denominador_ = d < 0 ? -d : d;

                reduz();
            } else if(entrada)
                entrada.setstate(ios_base::failbit);
        }
    }

    assert(cumpreInvariante());

    assert(numerador_ * d == n * denominador_ or not entrada.good());
}

...

/** Extrai do canal um novo valor para o racional, na forma de uma fracção.
    @pre r = r.
    @post Se entrada e entrada contém n′/d′ (ou apenas n′, assumindo-
se d′ = 1), em
        que n′ e d′ são inteiros com d′ ≠ 0, então
            r = n′/d′ ∧ entrada,
        senão
            r = r ∧ ¬entrada. */
istream& operator>>(istream& entrada, Racional& r)
{
    r.extraiDe(entrada);

    return entrada;
}

...
```

São de notar os seguintes pontos:

- A operação `istream::peek()` devolve o valor do próximo caractere no canal *sem o extrair*!
- A operação `istream::get()` extrai o próximo caractere do canal (e devolve-o).

- A função `isdigit()`[24] indica se o caractere passado como argumento é um dos dígitos decimais.

Fica como exercício para o leitor o estudo pormenorizado do método `Racional::extraiDe()`.

7.14.5 Leitura e escrita de ficheiros

Uma vez sobrecarregados os operadores << e >> para a classe C++ `Racional`, é possível usá-los para fazer extracções e inserções não apenas do teclado e para o ecrã, mas de e para onde quer que um canal esteja estabelecido. É possível estabelecer canais de entrada e saída para ficheiros, por exemplo. Para isso usam-se as classes `ifstream` e `ofstream`, compatíveis com `istream` e `ostream`, respectivamente, e que ficam disponíveis se se incluir a seguinte linha no início do programa:

```
#include <fstream>
```

Escrita em ficheiros

Para se poder escrever num ficheiro é necessário estabelecer um canal de escrita ligado a esse ficheiro e inserir nele os dados a escrever no ficheiro. O estabelecimento de um canal de escrita para um ficheiro é feito de uma forma muito simples. A instrução

```
ofstream saída("nome do ficheiro");
```

constrói um novo canal chamado `saída` que está ligado ao ficheiro da nome "*nome do ficheiro*". Note-se que `saída` é uma variável como outra qualquer, só que representa um canal de escrita para um dado ficheiro. A instrução acima é possível porque a classe `ofstream` possui um construtor que recebe uma cadeia de caracteres clássica do C++. Isso significa que é possível usar cadeias de caracteres para especificar o nome do ficheiro ao qual o canal deve estar ligado, desde que se faça uso da operação `string::c_str()`, que devolve a cadeia de caracteres clássica corresponde a uma dada cadeia de caracteres:

```
cout << "Diga o nome do ficheiro: ";
string nome_do_ficheiro;
cin >> nome_do_ficheiro;

ofstream saída(nome_do_ficheiro.c_str());
```

[24] Acrescentar

```
#include <cctype>
```

no início do programa para usar esta função.

O estabelecimento de um canal de saída para um dado ficheiro é uma operação destrutora: se o ficheiro já existir, é esvaziado antes. Dessa forma a escrita começa sempre do nada. Claro está que o estabelecimento de um canal de escrita pode falhar. Por exemplo, o disco rígido pode estar cheiro, pode não haver permissões para escrever no directório em causa, ou pode existir já um ficheiro com o mesmo nome protegido para escrita. Se o estabelecimento do canal falhar durante a sua construção, este fica em estado de erro, sendo muito fácil verificar essa situação tratando o canal como se de um booleano se tratasse:

```
cout << "Diga o nome do ficheiro: ";
string nome_do_ficheiro;
cin >> nome_do_ficheiro;

ofstream saída(nome_do_ficheiro.c_str());

if(not saída) {
    cerr << "Opps...  Não consegui criar ficheiro \""
         << nome_do_ficheiro << "\"!" << endl;
    ...
}
```

É ainda possível estabelecer e encerrar um canal usando as operações `ofstream::open()`, que recebe o nome do ficheiro como argumento, e `ofstream::close()`, que não tem argumentos. Só se garante que todos os dados inseridos num canal ligado a um ficheiro já nele foram escritos se:

1. o canal tiver sido explicitamente encerrado através da operação `ostream::close()`;
2. o canal tiver sido destruído da forma usual (e.g., no final do bloco onde a variável que o representa está definida);
3. tiver sido invocada a operação `ostream::flush()`;
4. tiver sido inserido no canal o manipulador `flush` (i.e., `saída << flush`); ou
5. tiver sido inserido no canal o manipulador `endl` (i.e., `saída << endl`), que coloca um fim-de-linha no canal e invoca automaticamente a operação `ostream::flush()`.

Ou seja, tudo funciona como se o canal tivesse um certa capacidade para dados, tal como uma mangueira tem a capacidade de armazenar um pouco de água. Tal como numa mangueira só se pode garantir que toda a água que nela entrou saiu pela outra ponta tomando algumas diligências, também num canal de saída só se pode garantir que os dados chegaram ao seu destino, e não estão ainda em circulação no canal se se diligenciar como indicado acima.

A partir do momento em que um canal de saída está estabelecido, pode-se usá-lo da mesma forma que ao canal `cout`. Por exemplo:

```
cout << "Diga o nome do ficheiro: ";
string nome_do_ficheiro;
cin >> nome_do_ficheiro;

ofstream saída(nome_do_ficheiro.c_str());

if(not saída) {
    cerr << "Opps...  Não consegui criar ficheiro \""
         << nome_do_ficheiro << "\"!" << endl;
    …
}

Racional r(1, 3);

saída << r << endl;
```

Leitura de ficheiros

Para se poder ler de um ficheiro é necessário estabelecer um canal de leitura ligado a esse ficheiro e extrair dele os dados a ler do ficheiro. O estabelecimento de um canal de leitura para um ficheiro é feito de uma forma muito simples. A instrução

```
ifstream entrada("nome do ficheiro");
```

constrói um novo canal chamado `entrada` que está ligado ao ficheiro da nome "*`nome do ficheiro`*". Mais uma vez `entrada` é uma variável como outra qualquer, só que representa um canal de leitura para um dado ficheiro. É também possível usar cadeias de caracteres para especificar o nome do ficheiro ao qual o canal deve estar ligado:

```
cout << "Diga o nome do ficheiro: ";
string nome_do_ficheiro;
cin >> nome_do_ficheiro;

ifstream entrada(nome_do_ficheiro.c_str());
```

O estabelecimento de um canal de entrada pode falhar. Por exemplo, se o ficheiro não existir, ou se estiver protegido para leitura. Se o estabelecimento do canal falhar durante a sua construção, este fica em estado de erro, sendo de novo muito fácil verificar essa situação:

```
cout << "Diga o nome do ficheiro: ";
string nome_do_ficheiro;
cin >> nome_do_ficheiro;

ifstream entrada(nome_do_ficheiro.c_str());
```

```
if(not entrada) {
    cerr << "Opps...  Não consegui ligar a ficheiro \""
         << nome_do_ficheiro << "\"!" << endl;
    ...
}
```

Tal como para os canais de saída, é também possível estabelecer e encerrar um canal usando as operações `ofstream::open()`, que recebe o nome do ficheiro como argumento, e `ofstream::close()`, que não tem argumentos.

A partir do momento em que um canal de entrada está estabelecido, pode-se usá-lo da mesma forma que ao canal `cin`. Por exemplo:

```
cout << "Diga o nome do ficheiro: ";
string nome_do_ficheiro;
cin >> nome_do_ficheiro;

ofstream entrada(nome_do_ficheiro.c_str());

if(not entrada) {
    cerr << "Opps...  Não consegui ligar a ficheiro \""
         << nome_do_ficheiro << "\"!" << endl;
    ...
}

Racional r;
entrada >> r;
```

Se num mesmo programa se escreve e lê de um mesmo ficheiro, é possível usar canais que permitem simultaneamente inserções e extracções. Porém, a forma mais simples é alternar escritas e leituras no mesmo ficheiro usando canais diferentes, desde que se garanta que todos os dados inseridos no respectivo canal de saída foram já escritos no respectivo ficheiro antes de os tentar extrair a partir de um canal de entrada ligado ao mesmo ficheiro. Repare-se nas linhas finais do teste de unidade do TAD `Racional`:

```
...

Racional r1(2, -6);

...

Racional r2(3);

...
```

```
ofstream saída("teste");
saída << r1 << ' ' << r2;
saída.close();

// Só o fecho explícito garante que a extracção do canal entrada tem sucesso.

ifstream entrada("teste");
Racional r4, r5;
entrada >> r4 >> r5;

...
```

7.15 Amizades e promiscuidades

7.15.1 Rotinas amigas

Há casos em que pode ser conveniente definir rotinas normais (i.e., não-membro) que tenham acesso aos membros privados de uma classe C++. Por exemplo, suponha-se que se pretendia definir a rotina `operator>>` para a classe C++ Racional, como se fez mais atrás, mas sem delegar o trabalho na operação `Racional::extraiDe()`. Nesse caso podia-se tentar definir a rotina como:

```
void Racional::extraiDe(istream& entrada)
{
}

...

/** Extrai do canal um novo valor para o racional, na forma de uma fracção.
    @pre r = r.
    @post Se entrada e entrada contém n′/d′ (ou apenas n′, assumindo-se d′ =
1), em
        que n′ e d′ são inteiros com d′ ≠ 0, então
            r = n′/d′ ∧ entrada,
         senão
            r = r ∧ ¬entrada. */
istream& operator>>(istream& entrada, Racional& r)
{
    assert(r.cumpreInvariante());

    int n;
    int d = 1;

    if(entrada >> n) {
        if(entrada.peek() != '/') {
```

```
            r.numerador_ = n;
            r.denominador_ = 1;
        } else {
            if(entrada.get() and isdigit(entrada.peek()) and
               entrada >> d and d != 0)
            {
                r.numerador_ = d < 0 ? -n : n;
                r.denominador_ = d < 0 ? -d : d;

                reduz();
            } else if(entrada)
                entrada.setstate(ios_base::failbit);
        }
    }

    assert(r.cumpreInvariante());

    assert(r.numerador_ * d == n * r.denominador_ or not canal);

    return entrada;
}
```

Como a rotina definida não é membro da class C++ `Racional`, não tem acesso aos seus membros privados. Para resolver este problema pode-se usar o conceito de amizade: se a classe C++ `Racional` declarar que é amiga, e por isso confia, nesta rotina, ela passa a ter acesso total às suas "partes íntimas". Para o conseguir basta colocar o cabeçalho da rotina em qualquer ponto da definição da classe C++[25], precedendo-a do qualificador `friend`:

```
...

class Racional {
  public:

    ...

  private:

    ...

    friend istream& operator>>(istream& entrada, Racional& r);
};

...
```

[25]Para o compilador é irrelevante se a declaração de amizade é feita na parte pública ou na parte privada da classe. No entanto, sendo as amizades uma questão de implementação, é desejável colocar a declaração na parte privada da classe, tipicamente no seu final.

Qualquer rotina não-membro que faça uso de uma classe C++ é conceptualmente parte das operações que concretizam o comportamento desejado para o correspondente TAD. No entanto, se uma rotina não-membro não for amiga da classe C++, embora seja parte da implementação do TAD, não é parte da implementação da correspondente classe C++. Isso deve-se ao facto de não ter acesso às suas partes privadas, e por isso não ter nenhuma forma directa de afectar o estado das suas instâncias. Porém, a partir do momento em que uma rotina se torna amiga de uma classe C++, passa a fazer parte da implementação dessa classe. É por isso que a rotina `operator>>` definida acima se preocupa com o cumprimento da condição invariante de classe: se pode afectar directamente o estado das instâncias da classe é bom que assim seja. É que o produtor de uma rotina não-membro amiga de uma classe deve ser visto como produtor da classe C++ respectiva, enquanto o produtor de uma rotina não-membro e não-amiga que faça uso de uma classe não pode ser visto como produtor dessa classe C++, mas sim como seu consumidor.

7.15.2 Classes amigas

Da mesma forma que no caso das rotinas, também se podem declarar classes inteiras como amigas de uma classe C++. Nesse caso todos os métodos da classe declarada como amiga têm acesso às partes privadas da classe C++ que declarou a amizade. Este tipo de amizade pode ser muito útil em algumas circunstâncias, como se verá no Capítulo 10, embora em geral sejam de evitar. A sintaxe da declaração de amizade de classes é semelhante à usada para as rotinas. Por exemplo,

```
class B {
    ...
};

class A {
  public:

    ...

  private:

    ...

    // Declaração da classe C++ B como amiga da classe A.
    friend class B;
};
```

7.15.3 Promiscuidades

Em que casos devem ser usadas rotinas ou classes amigas de uma classe C++? A regra geral é o mínimo possível. Se se puder evitar usar rotinas e classes amigas tanto melhor, pois evita-se introduzir uma excepção à regra de que só os membros de uma classe C++ têm acesso à suas partes privadas e à regra de que a implementação de uma classe corresponde às suas partes privadas e aos respectivos métodos. Todas as excepções são potencialmente geradoras de erros. Como mnemónica do perigo das amizades em C++, fica a frase "amizades normalmente trazem promiscuidades". Assim, e respeitando a regra geral enunciada, manter-se-á a implementação da rotina `operator>>` à custa da operação `Racional::extraiDe()`.

7.16 Código completo do TAD `Racional`

O TAD `Racional` foi concretizado na forma de uma classe C++ com o mesmo nome *e de rotinas (não-membro) associadas*. Conceptualmente o TAD é definido pelas respectivas operações, pelo que a classe C++ não é suficiente para o concretizar: faltam as rotinas associadas, que não são membro da classe. Ou seja, as rotinas que operam com racionais fazem logicamente parte do TAD, mesmo não pertencendo à classe C++ que o concretiza.

Note-se que se concentraram as declarações das rotinas no início do código, pois fazem logicamente parte da interface do TAD, tendo-se colocado a respectiva definição no final do código, junto com a restante implementação do TAD. Esta organização será útil aquando da organização do código em módulos físicos, ver 9, de modo a permitir a fácil utilização do TAD desenvolvido em qualquer programa.

```
#include <iostream>
#include <cctype>
#include <cassert>

using namespace std;

/** Devolve o máximo divisor comum dos inteiros passados como argumento.
    @pre V.
    @post mdc = { mdc(m, n)   m ≠ 0 ∨ n ≠ 0
                { 1           m = 0 ∧ n = 0 . */
int mdc(int const m, int const n);

/** Representa números racionais.
    @invariant 0 < denominador_ ∧ mdc(numerador_, denominador_) = 1. */
class Racional {
  public:
    /** Constrói racional com valor inteiro. Construtor por omissão.
        @pre V.
        @post *this = n. */
    Racional(int const n = 0);

    /** Constrói racional correspondente a n/d.
        @pre d ≠ 0.
        @post *this = n/d. */
    Racional(int const n, int const d);
```

```
/** Devolve numerador da fracção canónica correspondente ao racional.
    @pre V.
    @post numerador/denominador() = *this. */
int numerador() const;

/** Devolve denominador da fracção canónica correspondente ao racional.
    @pre V.
    @post (E n : V : n/denominador = *this ∧ 0 < denominador ∧ mdc(n, denominador) = 1). */
int denominador()  const;

/** Devolve versão constante do racional.
    @pre V.
    @post operator+ ≡ *this. */
Racional const& operator+() const;

/** Devolve simétrico do racional.
    @pre V.
    @post operator- = −*this. */
Racional const operator-() const;

/** Insere o racional no canal no formato de uma fracção.
    @pre V.
    @post ¬saída∨  saída  contém n/d (ou simplesmente n se d = 1)
          sendo n/d a fracção canónica correspondente ao racional  *this. */
void insereEm(ostream& saída) const;

/** Extrai do canal um novo valor para o racional, na forma de uma fracção.
    @pre  *this = r.
    @post Se  entrada  e  entrada  contém n′/d′ (ou apenas n′, assumindo-
se d′ = 1), em
          que n′ e d′ são inteiros com d′ ≠ 0, então
              *this = n′/d′ ∧ entrada,
          senão
              *this = r ∧ ¬entrada. */
void extraiDe(istream& entrada);

/** Adiciona de um racional.
    @pre *this = r.
    @post operator+= ≡ *this ∧ *this = r + r2. */
Racional& operator+=(Racional const& r2);

/** Subtrai de um racional.
    @pre *this = r.
    @post operator-= ≡ *this ∧ *this = r − r2. */
Racional& operator-=(Racional const& r2);
```

```
    /** Multiplica por um racional.
        @pre *this = r.
        @post operator*= ≡ *this ∧ *this = r × r2. */
    Racional& operator*=(Racional const& r2);

    /** Divide por um racional.
        @pre *this = r ∧ r2 ≠ 0.
        @post operator/= ≡ *this ∧ *this = r/r2. */
    Racional& operator/=(Racional const& r2);

  /** Incrementa e devolve o racional.
        @pre *this = r.
        @post operador++ ≡ *this ∧ *this = r + 1. */
    Racional& operator++();

/** Decrementa e devolve o racional.
        @pre *this = r.
        @post operador-- ≡ *this ∧ *this = r − 1. */
    Racional& operator--();

   private:
    int numerador_;
    int denominador_;
    /** Reduz a fracção que representa o racional.
        @pre denominador_ ≠ 0 ∧ *this = r.
        @post denominador_ ≠ 0 ∧ mdc(numerador_, denominador_) = 1∧
             *this = r. */
    void reduz();

    /** Indica se a condição invariante de classe se verifica.
        @pre V.
        @post cumpreInvariante = (0 <
denominador_ ∧ mdc(numerador_, denominador_) = 1). */
    bool cumpreInvariante() const;
};

/** Adição de dois racionais.
    @pre V.
    @post operator+ = r1 + r2. */
Racional const operator+(Racional const r1, Racional const& r2);

/** Subtracção de dois racionais.
    @pre V.
    @post operator- = r1 − r2. */
Racional const operator-(Racional const r1, Racional const& r2);
```

```
/** Produto de dois racionais.
    @pre V.
    @post operator* = r1 × r2. */
Racional const operator*(Racional const r1, Racional const& r2);

/** Divisão de dois racionais.
    @pre r2 ≠ 0.
    @post operator/ = r1/r2. */
Racional const operator/(Racional const r1, Racional const& r2);

/** Incrementa o racional recebido como argumento, devolvendo o seu valor an-
tes de incrementado.
    @pre *this = r.
    @post operator++ = r ∧ *this = r + 1. */
Racional const operator++(Racional& r, int);

/** Decrementa o racional recebido como argumento, devolvendo o seu valor an-
tes de decrementado.
    @pre *this = r.
    @post operator-- = r ∧ *this = r − 1. */
Racional const operator--(Racional& r, int);

/** Indica se dois racionais são iguais.
    @pre V.
    @post operator== = (r1 = r2). */
bool operator==(Racional const& r1, Racional const& r2);

/** Indica se dois racionais são diferentes.
    @pre V.
    @post operator== = (r1 ≠ r2). */
bool operator!=(Racional const& r1, Racional const& r2);

/** Indica se o primeiro racional é menor que o segundo.
    @pre V.
    @post operator< = (r1 < r2). */
bool operator<(Racional const& r1, Racional const& r2);

/** Indica se o primeiro racional é maior que o segundo.
    @pre V.
    @post operator> = (r1 > r2). */
bool operator>(Racional const& r1, Racional const& r2);

/** Indica se o primeiro racional é menor ou igual ao segundo.
    @pre V.
    @post operator<= = (r1 ≤ r2). */
```

```
bool operator<=(Racional const& r1, Racional const& r2);

/** Indica se o primeiro racional é maior ou igual ao segundo.
    @pre 𝒱.
    @post operator>= = (r1 ≥ r2). */
bool operator>=(Racional const& r1, Racional const& r2);

/** Insere o racional no canal de saída no formato de uma fracção.
    @pre 𝒱.
    @post ¬saída∨ saída contém n/d (ou simplesmente n se d = 1)
          sendo n/d a fracção canónica correspondente ao racional r. */
ostream& operator<<(ostream& saída, Racional const& r);

/** Extrai do canal um novo valor para o racional, na forma de uma fracção.
    @pre r = r.
    @post Se entrada e entrada contém n′/d′ (ou apenas n′, assumindo-
se d′ = 1), em
         que n′ e d′ são inteiros com d′ ≠ 0, então
             r = n′/d′ ∧ entrada,
         senão
             r = r ∧ ¬entrada. */
istream& operator>>(istream& entrada, Racional& r);

int mdc(int m, int n)
{
    if(m == 0 and n == 0)
        return 1;

    if(m < 0)
        m = -m;
    if(n < 0)
        n = -n;

    while(true) {
        if(m == 0)
            return n;
        n = n % m;
        if(n == 0)
            return m;
        m = m % n;
    }
}

inline Racional::Racional(int const n)
    : numerador_(n), denominador_(1)
{
```

```
    assert(cumpreInvariante());
    assert(numerador_ == n * denominador_);
}

inline Racional::Racional(int const n, int const d)
    : numerador_(d < 0 ? -n : n),
      denominador_(d < 0 ? -d : d)
{
    assert(d != 0);

    reduz();

    assert(cumpreInvariante());
    assert(numerador_ * d == n * denominador_);
}

inline int Racional::numerador() const
{
    assert(cumpreInvariante());

    return numerador_;
}

inline int Racional::denominador() const
{
    assert(cumpreInvariante());

    return denominador_;
}

inline Racional const& Racional::operator+() const
{
    assert(cumpreInvariante());

    return *this;
}

inline Racional const Racional::operator-() const
{
    assert(cumpreInvariante());

    Racional r;
    r.numerador_ = -numerador_;
    r.denominador_ = denominador_;
```

```
    assert(r.cumpreInvariante());

    return r;
}

inline void Racional::insereEm(ostream& saída) const
{
    assert(cumpreInvariante());

    saída << numerador_;
    if(denominador_ != 1)
        saída << '/' << denominador_;
}

void Racional::extraiDe(istream& entrada)
{
     assert(cumpreInvariante());

    int n;
    int d = 1;

    if(entrada >> n) {
        if(entrada.peek() != '/') {
            numerador_ = n;
            denominador_ = 1;
        } else {
            if(entrada.get() and isdigit(entrada.peek()) and
               entrada >> d and d != 0)
            {
                numerador_ = d < 0 ? -n : n;
                denominador_ = d < 0 ? -d : d;

                reduz();
            } else if(entrada)
                entrada.setstate(ios_base::failbit);
        }
    }

    assert(cumpreInvariante());

    assert(numerador_ * d == n * denominador_ or not entrada);
}

Racional& Racional::operator+=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());
```

```
    int dn = mdc(numerador_, r2.numerador_);
    int dd = mdc(denominador_, r2.denominador_);

    // Devido a r += r:
    int n2 = r2.numerador_;
    int d2 = r2.denominador_;

    numerador_ /= dn;
    denominador_ /= dd;

    numerador_ = numerador_ * (d2 / dd) + n2 / dn * denominador_;

    dd = mdc(numerador_, dd);

    numerador_ = dn * (numerador_ / dd);
    denominador_ *= d2 / dd;

    assert(cumpreInvariante());

    return *this;
}

Racional& Racional::operator-=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    int dn = mdc(numerador_, r2.numerador_);
    int dd = mdc(denominador_, r2.denominador_);

    // Devido a r += r:
    int n2 = r2.numerador_;
    int d2 = r2.denominador_;

    numerador_ /= dn;
    denominador_ /= dd;

    numerador_ = numerador_ * (d2 / dd) - n2 / dn * denominador_;

    dd = mdc(numerador_, dd);

    numerador_ = dn * (numerador_ / dd);
    denominador_ *= d2 / dd;

    assert(cumpreInvariante());
```

```
    return *this;
}

inline Racional& Racional::operator*=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    int n1d2 = mdc(numerador_, r2.denominador_);
    int n2d1 = mdc(r2.numerador_, denominador_);

    numerador_ = (numerador_ / n1d2) * (r2.numerador_ / n2d1);
    denominador_ = (denominador_ / n2d1) * (r2.denominador_ / n1d2);

    assert(cumpreInvariante());

    return *this;
}

inline Racional& Racional::operator/=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    assert(r2 != 0);

    int dn = mdc(numerador_, r2.numerador_);
    int dd = mdc(denominador_, r2.denominador_);

    if(r2.numerador_ < 0) {
        numerador_ = (numerador_ / dn) * (-r2.denominador_ / dd);
        denominador_ = (denominador_ / dd) * (-r2.numerador_ / dn);
    } else {
        numerador_ = (numerador_ / dn) * (r2.denominador_ / dd);
        denominador_ = (denominador_ / dd) * (r2.numerador_ / dn);
    }

    assert(cumpreInvariante());

    return *this;
}

inline Racional& Racional::operator++()
{
    assert(cumpreInvariante());

    numerador_ += denominador_;
```

```
    assert(cumpreInvariante());

    return *this;
}

inline Racional& Racional::operator--()
{
    assert(cumpreInvariante());

    numerador_ -= denominador_;

    assert(cumpreInvariante());

    return *this;
}

inline void Racional::reduz()
{
    assert(denominador_ != 0);

    int k = mdc(numerador_, denominador_);

    numerador_ /= k;
    denominador_ /= k;

    assert(denominador_ != 0 and mdc(numerador_, denominador_) == 1);
}

inline bool Racional::cumpreInvariante() const
{
    return 0 < denominador_ and mdc(numerador_, denominador_) == 1;
}

inline Racional const operator+(Racional r1, Racional const& r2)
{
    return r1 += r2;
}

inline Racional const operator-(Racional r1, Racional const& r2)
{
    return r1 -= r2;
}

inline Racional const operator*(Racional r1, Racional const& r2)
{
    return r1 *= r2;
```

```
}

inline Racional const operator/(Racional r1, Racional const& r2)
{
    assert(r2 != 0);

    return r1 /= r2;
}

inline Racional const operator++(Racional& r, int)
{
    Racional const cópia = r;

    ++r;

    return cópia;
}

inline Racional const operator--(Racional& r, int)
{
    Racional const cópia = r;

    --r;

    return cópia;
}

inline bool operator==(Racional const& r1, Racional const& r2)
{
    return r1.numerador() == r2.numerador() and
           r1.denominador() == r2.denominador();
}

inline bool operator!=(Racional const& r1, Racional const& r2)
{
    return not (r1 == r2);
}

inline bool operator<(Racional const& r1, Racional const& r2)
{
    int dn = mdc(r1.numerador(), r2.numerador());
    int dd = mdc(r1.denominador(), r2.denominador());

    return (r1.numerador() / dn) * (r2.denominador() / dd) <
           (r2.numerador() / dn) * (r1.denominador() / dd);
}
```

```
inline bool operator>(Racional const& r1, Racional const& r2)
{
    return r2 < r1;
}

inline bool operator<=(Racional const& r1, Racional const& r2)
{
    return not (r2 < r1);
}

inline bool operator>=(Racional const& r1, Racional const& r2)
{
    return not (r1 < r2);
}

ostream& operator<<(ostream& saída, Racional const& r)
{
    r.insereEm(saída);

    return saída;
}

istream& operator>>(istream& entrada, Racional& r)
{
    r.extraiDe(entrada);

    return entrada;
}

#ifdef TESTE

#include <fstream>

/** Programa de teste do TAD Racional e da função mdc(). */
int main()
{
    assert(mdc(0, 0) == 1);
    assert(mdc(10, 0) == 10);
    assert(mdc(0, 10) == 10);
    assert(mdc(10, 10) == 10);
    assert(mdc(3, 7) == 1);
    assert(mdc(8, 6) == 2);
    assert(mdc(-8, 6) == 2);
    assert(mdc(8, -6) == 2);
    assert(mdc(-8, -6) == 2);
```

```
Racional r1(2, -6);

assert(r1.numerador() == -1 and r1.denominador() == 3);

Racional r2(3);

assert(r2.numerador() == 3 and r2.denominador() == 1);

Racional r3;

assert(r3.numerador() == 0 and r2.denominador() == 1);

assert(r2 == 3);
assert(3 == r2);
assert(r3 == 0);
assert(0 == r3);

assert(r1 < r2);
assert(r2 > r1);
assert(r1 <= r2);
assert(r2 >= r1);
assert(r1 <= r1);
assert(r2 >= r2);

assert(r2 == +r2);
assert(-r1 == Racional(1, 3));

assert(++r1 == Racional(2, 3));
assert(r1 == Racional(2, 3));

assert(r1++ == Racional(2, 3));
assert(r1 == Racional(5, 3));
assert((r1 *= Racional(7, 20)) == Racional(7, 12));
assert((r1 /= Racional(3, 4)) == Racional(7, 9));
assert((r1 += Racional(11, 6)) == Racional(47, 18));
assert((r1 -= Racional(2, 18)) == Racional(5, 2));

assert(r1 + r2 == Racional(11, 2));
assert(r1 - Racional(5, 7) == Racional(25, 14));
assert(r1 * 40 == 100); assert(30 / r1 == 12);

ofstream saída("teste");
saída << r1 << ' ' << r2;
saída.close();
```

```
    ifstream entrada("teste");
    Racional r4, r5;
    entrada >> r4 >> r5;

    assert(r1 == r4);
    assert(r2 == r5);
}

#else // TESTE

int main()
{
    // Ler fracções:
    cout << "Introduza duas fracções (numerador denominador): ";
    Racional r1, r2;
    cin >> r1 >> r2;

    if(not cin) {
        cerr << "Opps!  A leitura dos racionais falhou!" << endl;
        return 1;
    }

    // Calcular racional soma:
    Racional r = r1 + r2;

    // Escrever resultado:
    cout << "A soma de " << r1 << " com " << r2
         << " é " << r << '.' << endl;
}

#endif // TESTE
```

O código acima representa o resultado final de um longo périplo, começado no início deste capítulo. Será que valeu a pena o esforço posto no desenvolvimento do TAD `Racional`? Depende. Se o objectivo era simplesmente somar duas fracções, certamente que não. Se o objectivo, por outro lado, era permitir a escrita simples de programas usando números racionais, então valeu a pena. Em geral, quanto mais esforço for dispendido pelo programador produtor no desenvolvimento de uma módulo, menos esforço será exigido do programador consumidor. No entanto, não se deve nunca perder o pragmatismo de vista e desenhar um módulo tentando prever todas as suas possíveis utilizações. Em primeiro lugar porque essas previsões provavelmente falharão, conduzindo a código inútil, e em segundo porque todo esse código inútil irá servir apenas como fonte de complexidade e erros desnecessários. Neste caso, no entanto, o negócio foi bom: o desenvolvimento serviu não apenas para fazer surgir naturalmente muitas particularidades associadas ao desenvolvimento de TAD em C++, servindo assim como ferramenta pedagógica, como também redundou numa classe que é realmente útil[26], ou que

[26]É de tal forma assim que se está a estudar a inclusão de uma classe genérica semelhante na biblioteca padrão

pelo menos está no bom caminho para o vir a ser.

7.17 Outros assuntos acerca de classes C++

Ao longo deste capítulo, e a pretexto do desenvolvimento de um TAD `Racional`, introduziram-se muitos conceitos importantes relativos às classes C++. Nenhum exemplo prático esgota todos os assuntos, pelo que se reservou esta secção final para introduzir alguns conceitos adicionais sobre classes C++.

7.17.1 Constantes membro

Suponha que se pretende definir uma classe C++ para representar vectores num espaço de dimensão fixa e finita. De modo a ser fácil alterar a dimensão do espaço onde se encontra o vector sem grande esforço, é tentador definir um atributo constante para representar a dimensão desse espaço. Depois, as coordenadas do vector poderiam ser guardadas numa matriz clássica do C++, membro da classe, com a dimensão dada. Ou seja:

```
class Vector {
  public:
    int const dimensão = 3; // erro!
    ...

  private:
    double coordenadas[dimensão];
    ...
};
```

Infelizmente, não é possível definir e inicializar uma constante dentro de uma classe. A razão para a proibição é simples. Sendo `dimensão` uma constante membro (de instância), cada instância da classe C++ possuirá a sua própria cópia dessa constante. Mas isso significa que, para ser verdadeiramente útil, essa constante deverá poder tomar valores diferentes para cada instância da classe C++, i.e., para cada variável dessa classe C++ que seja construída. Daí que não se possam inicializar atributos constantes (de instância) na sua própria definição. Seria possível, por exemplo, inicializar a constante nos construtores da classe:

```
class Vector {
  public:
    int const dimensão;

    Vector();

    ...
```

do C++ (ver `http://www.boost.org`).

```
  private:
    double coordenadas[dimensão];
    ...
};

Vector::Vector()
    : dimensão(3), ...
{
    ...
}
```

As listas de inicializadores são extremamente úteis, sendo utilizadas para inicializar não só atributos constantes, como também atributos que sejam referências e atributos que sejam de uma classe sem construtor por omissão, i.e., que exijam uma inicialização explícita. Mas neste caso a solução a que chegámos não é a mais adequada. Se todos os vectores devem ter a mesma dimensão, porquê fornecer cada instância da classe C++ com a sua própria constante `dimensão`? O ideal seria partilhar essa constante entre todas as instâncias da classe.

7.17.2 Membros de classe

Até agora viu-se apenas como definir membros de instância, i.e., membros dos quais cada instância da classe C++ possui uma versão própria. Como fazer para definir um atributo do qual exista apenas uma versão, comum a todas as instâncias da classe C++? A solução é simples: basta declarar o atributo com sendo um atributo *de classe* e não *de instância*. Isso consegue-se precedendo a sua declaração do qualificador `static`:

```
class Vector {
  public:
    static int const dimensão;

    Vector();

    ...

  private:
    double coordenadas[dimensão]; // Opps... Não funciona...

    ...
};
```

Um atributo de classe não pode ser inicializado nas listas de inicializadores dos construtores. Nem faria qualquer sentido, pois um atributo de classe têm apenas uma instância, partilhada entre todas as instâncias da classe de que é membro, não fazendo sentido ser inicializada senão uma vez, antes do programa começar. Assim, é necessário definir esse atributo fora da classe, sendo durante essa definição que se procede à respectiva inicialização:

```
int const Vector::dimensão = 3;
```

Infelizmente esta solução não compila correctamente. É que em C++ só se pode especificar a dimensão de uma matriz clássica usando um constante *com valor conhecido pelo compilador*. Como um atributo de classe só é inicializado na sua definição, que está fora da classe, o atributo `dimensão` é inicializado tarde demais: o compilador "engasga-se" na definição da matriz membro, dizendo que não sabe o valor da constante.

Este problema não é, em geral, resolúvel, excepto quando o atributo de classe em definição for de um tipo básico inteiro do C++. Nesse caso é possível fazer a definição (como inicialização) dentro da própria classe:

```
class Vector {
  public:
    static int const dimensão = 3;

    Vector();

    ...

  private:
    double coordenadas[dimensão]; // Ah! Já funciona...

    ...
};
```

onde a definição externa à classe deixa de ser necessária.

Da mesma forma, é possível declarar operações da classe C++ que não precisam de ser invocadas através de nenhuma instância: são as chamadas operações de classe. Suponha-se, por absurdo, que se queria que a classe C++ `Vector` mantivesse uma contagem do número de instâncias de si própria existente em cada instante. Uma possibilidade seria:

```
class Vector {
  public:
    static int const dimensão = 3;

    Vector();

    static int númeroDeInstâncias();

    ...

  private:
    double coordenadas[dimensão]; // Ah! Já funciona...
```

```
    static int número_de_instâncias;

    ...
};

Vector::Vector()
    : ...
{
    ...

    ++número_de_instâncias;
}

inline int Vector::númeroDeInstâncias()
{
    return número_de_instâncias;
}

int Vector::número_de_instâncias = 0;
```

Os membros de classe (e não de instância) são precedidos do qualificador `static` aquando da sua declaração dentro da definição da classe. O atributo de classe `número_de_instâncias` é declarado durante a definição da classe e só é definido (i.e., construída de facto com o valor inicial 0) depois da classe C++, tal como acontece com as rotinas membro.

O contador de instâncias acima tem dois problemas. O primeiro é que a contabilização falha se algum vector for construído por cópia a partir de outro vector:

```
Vector v1; // Ok, incrementa contador.

Vector v2(v1); // Erro! Não incrementa contador.
```

Para já ignorar-se-á este problema, até porque só se falará do fornecimento explícito de construtores por cópia, que substituem o construtor por cópia fornecido implicitamente pela linguagem, num capítulo posterior.

O segundo problema é que a contabilização falha quando se destrói alguma instância da classe.

7.17.3 Destrutores

Da mesma forma que os construtores de uma classe C++ são usados para inicializar as suas instâncias quando estas são construídas, podem-se usar destrutores, i.e., código que deve é executado quando a instância é destruída, para "arrumar a casa" no final do tempo de vida das instâncias de uma classe C++. Os destrutores são extremamente úteis, particularmente quando se utilizam variáveis dinâmicas ou, em geral, quando os construtores da classe C++ reservam algum recurso externo para uso exclusivo da instância construída. Utilizações mais interessantes do conceito ver-se-ão mais tarde, bastando para já apresentar um exemplo ingénuo da sua utilização no âmbito da classe C++ `Vector`. Os destrutores declaram-se e definem-se como os construtores, excepto que se coloca o símbolo ~ antes do seu nome (que é também o nome da classe):

```
class Vector {
  public:
    static int const dimensão = 3;

    Vector();

    ~Vector();

    static int númeroDeInstâncias();

    ...

  private:
    double coordenadas[dimensão]; // Ah! Já funciona...

    static int número_de_instâncias;

    ...
};

Vector::Vector()
    : ...
{
    ...

    ++número_de_instâncias;
}

Vector::~Vector()
{
    --número_de_instâncias;

    ...
}

inline int Vector::númeroDeInstâncias()
{
    return número_de_instâncias;
}
```

```
int Vector::número_de_instâncias = 0;
```

Note-se que a linguagem fornece implicitamente um destrutor para as classes definidas sempre que este não seja definido explicitamente pelo programador fabricante.

7.17.4 De novo os membros de classe

Suponha-se o seguinte código usando a classe desenvolvida:

```
Vector a;

int main()
{
    {
        Vector b;

        for(int i = 0; i != 3; ++i) {
            Vector c;
            cout << "Existem " << Vector::númeroDeInstâncias()
                 << " instâncias." << endl;
            static Vector d;
        }

        cout << "Existem " << Vector::númeroDeInstâncias()
             << " instâncias." << endl;
    }

    cout << "Existem " << C::númeroDeInstâncias()
         << " instâncias." << endl;
}
```

É de notar que a invocação da operação de classe C++ `Vector::númeroDeInstâncias()` faz-se não através do operador de selecção de membro `.`, o que implicaria a invocação da operação através de uma qualquer instância da classe C++, mas através do operador de resolução de âmbito `::`. No entanto, é também possível, se bem que inútil, invocar operações de classe através do operador de selecção de membro. As mesmas observações se podem fazer no que diz respeito aos atributos de classe.

A execução do programa acima teria como resultado:

```
Existem 3 instâncias.
Existem 4 instâncias.
Existem 4 instâncias.
Existem 3 instâncias.
Existem 2 instâncias.
```

De aqui em diante utilizar-se-á a expressão "membro" como significando "membro de instância", i.e., membros dos quais cada instância da classe C++ a que pertencem possui uma cópia própria, usando-se sempre a expressão "membro de classe" para os membros partilhados entre todas as instâncias da classe C++.

Para que o resultado do programa acima seja claro, é necessário recordar que:

- instâncias automáticas (i.e., variáveis e constantes locais sem o qualificador `static`) são construídas quando a instrução da sua definição é executada e destruídas quando o bloco de instruções na qual foram definidas termina;
- instâncias estáticas globais são construídas antes de o programa começar e destruídas depois do seu final (depois de terminada a função `main()`); e
- instâncias estáticas locais são construídas quando a instrução da sua definição é executada pela primeira vez e destruídas depois do final do programa (depois de terminada a função `main()`).

7.17.5 Construtores por omissão

Todas as classes C++ têm construtores. A um construtor que possa ser invocado sem qualquer argumento (porque não tem qualquer parâmetro ou porque todos os parâmetros têm valores por omissão) chama-se construtor *por omissão*. Se o programador não declarar qualquer construtor, é fornecido implicitamente, sempre que possível, um construtor por omissão. Por exemplo, no código

```
class C {
  private:
    Racional r1;
    Racional r2;
    int i;
    int j;
};

C c; // nova instância, construtor por omissão invocado.
```

é fornecido implicitamente pelo compilador um construtor por omissão para a classe C++ `C`. Este construtor invoca os construtores por omissão de todos os atributos, com excepção dos pertencentes a tipos básicos do C++ que, infelizmente, não são inicializados implicitamente. Neste caso, portanto, o construtor por omissão da classe `C` constrói os atributos `r1` r `r2` com o valor racional zero, deixando os tributos `i` e `j` por inicializar.

O construtor por omissão fornecido implicitamente pode ser invocado explicitamente,

```
C c = C(); // ou C c(C());
```

embora neste caso seja também invocado o construtor por cópia, também fornecido implicitamente, que constrói a variável `c` à custa da instância temporária construída pelo construtor por omissão. Para que esse facto fique claro, repare-se no código

```
cout << Racional() << endl;
```

que mostra no ecrã o valor da instância temporária da classe C++ `Racional` construída pelo respectivo construtor por omissão, i.e., o valor zero. Neste caso o construtor por cópia, da classe C++ `Racional`, não é invocado.

Se o programador declarar algum construtor explicitamente, então o construtor por omissão deixa de ser fornecido implicitamente. Por exemplo, se a classe C++ `C` fosse

```
class C {
  public:
    C(int i, int j);

  private:
    Racional r1;
    Racional r2;
    int i;
    int j;
};
```

o código

```
C c; // ops... falta o construtor por omissão!
```

resultaria num erro de compilação.

No exemplo seguinte, o construtor por omissão faz exactamente o mesmo papel que o construtor por omissão fornecido implicitamente para a classe C++ `C` no exemplo original:

```
class C {
  public:
    C();

  private:
    Racional r1;
    Racional r2;
    int i;
    int j;
};

C::C()
    : r1(), r2()
{
}
```

Sempre não sejam colocados na lista de inicialização de um construtor, os atributos que não sejam de tipos básicos do C++ são inicializados implicitamente através do respectivo construtor por omissão (se ele existir, bem entendido). Assim, o construtor no exemplo acima pode-se simplificar para:

```
C::C()
{
}
```

Antes de ser executado o corpo do construtor envolvido numa construção, todos os atributos da classe C++ são construídos *pela ordem da sua definição na classe*, sendo passados aos construtores os argumentos indicados na lista de inicializadores entre parênteses após o nome do atributo, se existirem, ou os construtores por omissão na falta do nome do atributo na lista, com excepção dos atributos de tipos básicos do C++, que têm de ser inicializados explicitamente, caso contrário ficam por inicializar. Por exemplo, no código

```
class C {
  public:
    C(int n, int d, int i);

  private:
    Racional r1;
    Racional r2;
    int i;
    int j;
};

C::C(int const n, int const d, int const i)
    : r1(n, d), i(i), j()
{
    r2 = 3;
}

C c(2, 10, 1);
```

ao ser construída a variável `c` é invocado o seu construtor, o que resultará nas seguintes operações:

1. Construção de `r1` por invocação do construtor da classe C++ `Racional` com argumentos `n` e `d` (que neste caso inicializa `r1` com $\frac{1}{5}$).

2. Construção de `r2` por invocação implícita do construtor da classe C++ `Racional` que pode ser invocado sem argumentos (que inicializa `r2` com o racional zero ou $\frac{0}{1}$).

3. Construção do atributo `i` a partir do parâmetro `i` (que neste caso fica com 1). É usado o construtor por cópia dos `int`.

4. Construção do atributo `j` através do construtor por omissão dos `int`, que é necessário invocar explicitamente (e que inicializa `j` com o valor zero).

5. Execução do corpo do construtor da classe C++ `C`:

   (a) Conversão do valor literal 3 de `int` para `Racional`.

   (b) Atribuição do racional $\frac{3}{1}$ a `r2`.

É de notar que a variável membro `r2` é construída e inicializada com o valor zero e só depois lhe é atribuído o racional $\frac{3}{1}$, o que é uma perda de tempo. Seria preferível incluir `r2` na lista de inicializadores.

7.17.6 Matrizes de classe

É possível definir matrizes clássicas do C++ tendo como elementos valores de TAD, por exemplo da classe C++ `Racional`:

```
Racional m1[10];
Racional m2[10] = {1, 2, Racional(3), Racional(), Racional(1, 3)};
```

Os 10 elementos de `m1` e os últimos cinco elementos de `m2` são construídos usando o construtor por omissão da classe `Racional` (que os inicializa com o racional zero). Os dois primeiros elementos da matriz `m2` são inicializados a partir de inteiros implicitamente convertidos para racionais usando o construtor com um único argumento da classe C++ `Racional` (i.e., o segundo construtor, usando-se um segundo argumento tendo valor por omissão um). Essa conversão é explicitada no caso do terceiro elemento de `m2`. Já para o quarto e o quinto elementos, eles são construídos por cópia a partir de um racional construído usando o construtor por omissão, no primeiro caso, e o construtor completo, com dois argumentos, no segundo caso. Note-se que se a classe C++ em causa não possuir construtores por omissão, é obrigatório inicializar todos os elementos da matriz explicitamente.

7.17.7 Conversões para outros tipos

Viu-se já que ao se definir numa classe C++ um construtor passível de ser invocado com um único argumento, se fornece uma forma de conversão implícita de tipo (ver Secção 7.9.2). Por exemplo, a classe C++ `Racional` fornece um construtor que pode ser invocado com um único argumento inteiro, o que possibilita a conversão implícita de tipo entre valores do tipo `int` e valores da classe `Racional`,

```
Racional r(1, 3);

...

if(r < 1) // 1 convertido implicitamente de int para Racional.
```

sem haver necessidade de explicitar essa conversão:

```
Racional r(1, 3);

...

if(r < Racional(1)) // não é necessário...
```

E se se pretendesse equipar a classe C++ `Racional` com uma conversão implícita desse tipo para `double`, i.e., se se pretendesse tornar o seguinte código válido?

```
Racional r(1, 2);
double x = r;
cout << r << ' ' << x << endl; // mostra: 1/2 0.5
cout << double(r) << endl; // mostra: 0.5
```

A solução poderia passar por alterar a classe C++ `double`, de modo a ter um construtor que aceitasse um racional. O problema é que nem o tipo `double` é uma classe, nem, se por absurdo o fosse, estaria nas mãos do programador alterá-la.

A solução passa por alterar a classe C++ `Racional` indicando que se pode converter implicitamente um racional num `double`. De facto, a linguagem C++ permite fazê-lo. É possível numa classe C++ definir uma conversão implícita de um tipo para essa classe, mas também o contrário: definir uma conversão implícita da classe para um outro tipo. Isso consegue-se sobrecarregando um operador de conversão para o tipo pretendido, neste caso `double`. Esse operador chama-se `operator double`. Este tipo de operadores, de conversão de tipo, têm algumas particularidades:

1. Só podem ser sobrecarregados através de rotinas membro da classe C++ a converter.

2. Não incluem tipo de devolução no seu cabeçalho, pois este é indicado pelo próprio nome do operador.

Ou seja, no caso em apreço

```
...
class Racional {
  public:

    ...

    /** Devolve o racional visto como um double.
        @pre V.
        @post O valor devolvido é uma aproximação tão boa quanto
              possível do racional representado por *this. */
```

```
        operator double() const;

    ...

    };

    ...

    Racional::operator double() const
    {
        return double(numerador()) / double(denominador());
    }
```

A divisão do numerador pelo denominador é feita depois de ambos serem convertidos para `double`. De outra forma seria realizada a divisão inteira, cujo resultado não é bem aquilo que se pretendia...

O problema deste tipo de operadores de conversão de tipo, que devem ser usados com moderação, é que levam frequentemente a ambiguidades. Por exemplo, definido o operador de conversão para `double` de valores da classe C++ `Racional`, como deve ser interpretado o seguinte código?

```
Racional r(1,3);

...

if(r == 1)
    ...
```

O compilador pode interpretar a guarda da instrução de selecção ou condicional como

```
double(r) == double(1)
```

ou como

```
r == Racional(1)
```

mas não sabe qual escolher.

As regras do C++ para resolver este tipo de ambiguidades são algo complicadas (ver [12, Secção 7.4]) e não resolvem todos os casos. No exemplo dado o programa é de facto ambíguo e portanto resulta num erro de compilação. Como é muito mais natural e frequente a conversão implícita de um `int` num `Racional` do que a conversão implícita de um `Racional` num `double`, a melhor solução é simplesmente não sobrecarregar o operador de conversão para `double`.

7.17.8 Uma aplicação mais útil das conversões

Há casos em que os operadores de conversão podem ser muito úteis. Suponha-se que se pretende definir um tipo `Palavra` que se comporta como um `int` em quase tudo, mas fornece algumas possibilidades adicionais, tais como acesso individualizado aos seus *bits* (ver Secção 2.7.4). O novo tipo poderia ser concretizado à custa de uma classe C++:

```
class Palavra {
  public:
    Palavra(int const valor = 0);
    operator int() const;
    bool bit(int const n) const;

  private:
    int valor;
};

inline Palavra::Palavra(int const valor)
    : valor(valor)
{
}

inline Palavra::operator int() const
{
    return valor;
}

inline bool Palavra::bit(int const n) const
{
    return (valor & (1 << n)) != 0;
}
```

Esta definição tornaria possível escrever

```
Palavra p = 99996;

p = p + 4;

std::cout << "Valor é: " << p << std::endl;

std::cout << "Em binário: ";

for(int i = 32; i != 0; --i)
    std::cout << p.bit(i - 1);
std::cout << std::endl;
```

que resultaria em

```
Valor é: 100000
Em binário: 00000000000000011000011010100000
```

Esta classe C++, para ser verdadeiramente útil, deveria proporcionar outras operações, que ficam como exercício para o leitor.

Capítulo 8

Programação baseada em objectos

!!!!!!Atenção! Colocar diagrama UML completo das classes nesse capítulo!

!!Falar de programação centrada nos dados. Preocupa-nos queobjectos existem no domínio do problema, que tipos de objectos são (ou a que classe pertencem), que operações se podem realizar com eles e que relações existem entre eles.

!! Num capítulo à parte: Exemplo de desenho e utilização de TAD. Aula 14. Exemplo da calculadora simples. Depois mudar de double para Racional. Acrescentar parênteses ou, alternativamente, deixá-los como exercício para o leitor (pode-se dar umas dicas).

!!Fazer capítulo sobre desenvolvimento. Aqui é que deve vir para a parte sobre tipos abstractos de dados. Falar acerca do desenvolvimento. Identificação de classes, relações entre classes, desenvolvimento incremental, teste antecipado, guardar testes!

!! Inspectores/interrogações: sempre funções e const. Não há funções sem ser const? Há, mas nesse caso não alteram directamente: devolvem referência através da qual se pode alterar a variável implícita (ver à frente). Modificadores: procedimentos não const. Actuadores: procedimentos const (afectam globais ou parâmetros por referência). Modificadores/actuadores: e.g., lê, afecta implícita e canal. Manípulos: funções não const que não mudam a variável implícita mas devolvem referência ou classe referência que a permite alterar. Const: query ou interrogações.

!!Falar de nomes: classes = substantivos. Objectos reais ou conceptuais. Hmm... distinguir TAD de Classes?

!!Colocar aqui exemplo dos mapas (calculadora mais sofisticada?). Aproveitar para introduzir o conceito de mutable. Distinguir entre estado e estado observável (estado: produtor, estado observável: consumidor).

!!!!!!!

Incluir parênteses e verificação de erros? Só parênteses? Erros transmitidos como? Como excepções?

!!!!!!!!!

Hoje vamos falar um pouco mais sobre desenho de TAD. Normalmente os TAD não surgem do nada. Surgem por serem úteis para resolver um determinado problema. Vamos então resolver um problema. Suponhamos que queremos implementar uma calculadora simples. Esta calculadora deve suportar as operações de adição, subtracção, multiplicação, divisão e potenciação sobre valores de vírgula flutuante (usaremos double). Para simplificar vamos supor que os parênteses são proibidos. As expressões a calcular devem ser lidas do canal de entrada cin, ligado ao teclado, e os resultados devem ser escritos no canal de saída cout, ligado ao ecrã. A calculadora pode calcular várias expressões, pelo que se arbitra que estas são separadas umas das outras pelo símbolo ; (ponto-e-vírgula). A calculadora termina o cálculo de expressões quando encontra o símbolo . (ponto). Por exemplo, se a entrada for: 2 + 2 ; 2 * 2.5 . a saída deve ser 4 5 Para percebermos melhor como resolver o problema, vamos tentar perceber como nós próprios efectuaríamos os cálculos caso tivéssemos acesso a um símbolo de cada vez. Por símbolo quero dizer ou valores numéricos ou os operadores permitidos. Seja então a expressão 1 + 4 * 3 / 2 ^ 3 - 2 . Qualquer um de nós calcula rapidamente o resultado que é... Exacto: 0,5. E se só virmos um símbolo de cada vez. Que fazer? Discutir. Concluir que o melhor é ir guardando os valores e os operadores! Mas onde? O.k., vamos passo a passo. Primeiro recebemos 1 Que fazer? Discutir. Depois recebemos + Que fazer? Discutir. Deixar a dica que talvez seja boa ideia usar um TAD para representar os operadores. Depois recebemos 4 Pode-se já calcular? Discutir. Não! Porquê? Porque a seguir pode vir outro operador com maior precedência! E se agora fosse + ? Discutir. Nesse caso podia-se logo calcular a primeira operação! O mesmo se a expressão até aqui fosse: 1 * 4 + ! Mas não é... Temos de ler mais um símbolo, que é: * Calma lá. Quantos valores já lemos? E quantos operadores? E onde os guardamos? Discutir, mesmo sem concluir. Depois vem 3 Pouco ajuda... E depois vem / E agora? Discutir! Qual é a regra? Podem-se calcular todos os operadores em espera que tenham precedência superior ou igual ao que acabámos de ler! Mas por que ordem? Discutir. Concluir que os cálculos prosseguem pela ordem inversa à da leitura! Ou seja, podemos empilhar as operações! E os valores? Discutir. Uma pilha também serve para os valores! Exemplificar desde o início ao fim da expressão, desenhando a evolução das pilhas. Explicar o que acontece no . (ponto). Então já temos uma ideia da forma de cálculo da nossa calculadora. Falta definirmos os TAD necessários. Em primeiro lugar, vai-nos dar muito jeito um TAD para representar os operadores! Apresentar acetato! Explicar cuidadosamente a classe. Explicar o operator()(). Explicar o operador <: um operador é menor que outro se tiver menos precedência. Explicar membro de classe cuidadosamente! Dar exemplos de invocação! Em segundo lugar vai-nos ser útil uma classe para representar a calculadora! Apresentar acetato. Explicar detalhadamente o seu funcionamento. Deixar claro que não se verificam nenhuns erros do utilizador! A intenção é simplificar a resolução. Fica como exercício... Fazer um traçado admitindo a mesma entrada. Depois de eles se terem convencido da bondade da solução, avançar. Disse-vos que ia falar do desenho de TAD e na volta apresentei as classes C++ correspondentes já prontas e acabadas... Batotice... Bom, não inteiramente. Faltam as pilhas! Explicar que ainda não temos os tipos PilhaDeDouble nem PilhaDeOperador, mas que já temos um programa escrito que faz uso da classe! Esta é uma boa forma de sabermos que operações deve um TAD suportar: começar a resolver o problema assumindo que a classe existe! Vamos começar pela classe PilhaDeDouble. A classe PilhaDeOperador será praticamente igual. Que operações podemos realizar sobre uma pilha? Discutir. Concluir: Criar pilha vazia. Pôr item na pilha. Altera a pilha. Obter o item do topo. Não altera a pilha! Tirar o item do topo. Altera a pilha. Saber altura da pilha. Não altera a pilha! Verificar se está vazia. Não altera a pilha! Antes de avançarmos, no entanto, é conveniente, aliás, é fundamental, fazermos aquilo que se chama um teste de unidade. Um teste de unidade, ou teste de módulo, é um pequeno programa que nos permite testar todas as operações de um TAD, de modo a ficarmos seguros do seu bom funcionamento. ..... #ifdef TESTE int main() { PilhaDeDouble pilha; assert(pilha.estaVazia()); assert(pilha.altura() == 0); for(int i = 0; i != 10; ++i) { pilha.poe(i); assert(pilha.itemNoTopo() == i); } assert(not pilha.estaVazia()); assert(pilha.altura() == 10); for(int i = 10; i != -1; –i) { assert(pilha.itemNoTopo() == i); pilha.tiraItem(); } assert(pilha.estaVazia()); assert(pilha.altura() == 0); } #endif Explicar bem as directivas de pré-processador, embora sem entrar em pormenores. Dizer que a função main() só chega a ser compilada se o programa for compilado com um comando da forma: c++ -DTESTE ... Agora é só ir definindo a interface da classe C++, incluindo documentação, mas deixando a implementação de lado. Discutir efeito de cada operação e circunstâncias em que falha. /** Representa pilhas de double. @invariant Dizer que não se conhece por ser questão de implementação. */ class PilhaDeDouble { public: typedef double Item; /** Constrói pilha vazia. @pre V. @post estaVazia(). */ PilhaDeDouble(); /** Devolve o item que está no topo da pilha. @pre ¬estaVazia(). @post itemNoTopo idêntico ao item no topo de *this. */ Item const& itemNoTopo() const; /** Indica se a pilha está vazia. @pre V. @post estaVazia = *this está vazia. */ bool estaVazia() const; /** Devolve altura da pilha. @pre V. @post altura = altura de *this. */ int altura() const; /** Põe um novo item na pilha (no topo). @pre V. @post *this contém um item adicional no topo igual a novo_item. */ void poe(Item const& novo_item); /** Tira o item que está no topo da pilha. @pre ¬estaVazia(). @post *this contém os itens originais menos o do topo. */ void tiraItem(); private: ... }; ... Discutir cuidadosamente o typedef e sua utilidade! Com Item é fácil alterar o tipo dos itens! O que fazer para definir a classe PilhaDeOperador? Discutir. O que é que falta? A implementação! É o menos importante! Façam-na vocês: fica como exercício. Conclusões: Os TAD surgem para auxiliar na resolução de um problema. A resolução deve ser usada para sugerir as operações necessárias para o novo TAD. Deve-se começar por escrever o respectivo teste de unidade. Só então se desenha a interface da classe C++ que implementa o TAD. Finalmente passa-se à implementação. O teste de unidade e a interface da classe podem ser desenvolvidos em paralelo, incrementalmente. Mas a implementação deve ser deixada para o fim! Falar de modularização: departamento de marketing + designers definem a interface (o que faz, como se opera, aspecto exterior), departamento de engenharia implementa mecanismo. Programação baseada em objectos: primeiro pensa-se que classes existem (substantivos no vocabulário do problema). Depois pensa-se nos algoritmos. E não o contrário: programação procedimental, em que primeiro se pensa nos algoritmos e só depois nos dados. Por vezes misturam-se os dois tipos de abordagem. Se houver tempo discutir um pouco a implementação. Dizer que as pilhas... Já existem... #include <stack>... push(), pop(), size(), top(), empty(). Referir que no segundo semestre vamos aprender a escrever o código da pilha uma vez apenas de modo a que seja válido com qualquer tipo: classes modelo e programação genérica.

8.1 Desenho de classes

Como se desenha uma classe? Quando e porquê o fazer? Não há respostas taxativas para estas questões, mas tentar-se-á apresentar nesta secção um exemplo de resolução de um problema que sirva para as clarificar um pouco.

Suponha-se que se pretende escrever um programa que leia 10 valores inteiros do teclado e os escreva pela ordem inversa no ecrã. Um pouco de reflexão sobre o problema revela que ele pode ser resolvido com matrizes C++. Mas, antes de aceitar a solução como evidente, convém pensar um pouco no problema.

Ponha-se no lugar do computador e admita que lhe são passadas 10 folhas de papel, com número escritos, e que as deve devolver pela ordem inversa à da entrega. Como organizaria as folhas de papel? A resposta usual a esta pergunta é que os papeis deveriam ser organizados numa pilha (com os números voltados para cima) porque, numa pilha de papeis, o último papel a entrar é o primeiro a sair.

Não há equivalente a uma pilha de papeis na linguagem C++ (embora exista na biblioteca padrão...). Surge então a ideia de construir uma classe para acrescentar esse conceito ao C++. Mas resta uma dúvida: sendo o problema tão simples e facilmente resolúvel usando matrizes directamente, porquê definir uma classe nova?

Mais uma vez não existem respostas taxativas a esta questão. A questão deve ser analisada sob diversos pontos de vista e uma decisão tomada com base nos resultados dessa análise. Por exemplo:

1.O desenvolvimento da nova classe exige substancialmente mais esforço que uma solução ad hoc? 2.A nova classe pode vir a ser útil no futuro, no mesmo ou noutros programas? 3.A modularização inerente à construção de uma classe vai trazer vantagens para o desenvolvimento e correcção do programa?

Neste caso, é evidente que uma classe representando o conceito de pilha de inteiros pode vir a ser útil no futuro, sobretudo se for facilmente adaptável para representar pilhas de entidades de outro tipo tais como pilhas de char ou pilhas de Racional (mais tarde se verá como se podem construir classes genéricas, ou classes modelo).

Capítulo 9

Modularização de alto nível

A modularização é um assunto recorrente em programação. De acordo com [4]:

> **módulo**[1]**.** [Do lat. *modulu*.] ***S. m.*** [...] **4.** Unidade (de mobiliário, de material de construção, etc.) planejada segundo determinadas proporções e destinada a reunir-se ou ajustar-se a outras unidades análogas, de várias maneiras, formando um todo homogêneo e funcional. [...]

Em programação os módulos são também unidades que se podem reunir ou ajustar umas às outras de modo a formar um todo homogéneo e funcional. Mas normalmente exige-se que os módulos tenham algumas características adicionais:

1. Devem ter um único objectivo bem definido.
2. Devem ser coesos, i.e., devem existir *muitas ligações dentro dos módulos*.
3. Devem ser fracamente ligados entre si, i.e., devem existir *poucas ligações entre os diferentes módulos*.
4. Devem separar claramente a sua interface da sua implementação, i.e., devem distinguir claramente entre o que deve estar visível para o consumidor do módulo e que deve estar escondido no seu interior. Ou seja, os módulos devem respeitar o princípio do encapsulamento.

No Capítulo 3 introduziu-se pela primeira vez a noção de modularização. As unidades básicas de modularização são as rotinas, apresentadas nesse capítulo:

1. Têm um único objectivo bem definido (que no caso das funções é devolver um qualquer valor e no caso dos procedimentos é fazer qualquer coisa).
2. São coesas, pois tipicamente, como têm um objectivo bem definido, podem ser expressos em poucas linhas de código, todas interdependentes. É comum, e a maior parte das vezes desejável, que um módulo utilize outros módulos disponíveis, ou seja, que uma rotina utilize outras rotinas para realizar parte do seu trabalho.

3. São fracamente ligadas. É comum uma rotina usar outra rotina, o que estabelece uma ligação entre as duas, mas fá-lo normalmente recorrendo apenas à sua interface, invocando-a, pelo que as ligações se reduzem à ligação entre argumentos e parâmetros, que desejavelmente são em pequeno número, e ao valor devolvido, no caso de uma função.

4. Separam claramente interface de implementação. O cabeçalho de uma rotina contém (quase[1]) tudo aquilo que quem o utiliza precisa de saber, indicando o nome da rotina, o número e tipo dos parâmetros e o tipo de devolução, i.e., a sua interface. O corpo de uma rotina corresponde à sua implementação, indicando como funciona.

Existem vários níveis de modularização adicionais, que podem ser organizados hierarquicamente. Depois das rotinas, o nível seguinte de modularização corresponde às classes:

1. Têm um único objectivo bem definido que é o de representar um dado conceito, físico ou abstracto. Por exemplo, uma classe `Racional` serve para representar o conceito de número racional, com as respectivas operações, enquanto uma classe `CircuitoLógico` pode servir para representar o conceito de circuito lógico e respectivas operações.

2. São coesas, pois servem para representar uma única entidade, e portanto os seus métodos estão totalmente interligados uns aos outros através dos atributos da classe e, muitas vezes, através de utilizações mútuas.

3. Separam claramente interface de implementação. A interface corresponde aos membros públicos da classe, que tipicamente não incluem atributos variáveis. Das operações públicas, apenas fazem parte da interface os respectivos cabeçalhos. A implementação das operações, ou seja, os métodos, está inacessível ao consumidor da classe[2]. Fazem parte da implementação de uma classe todos os membros privados (incluindo normalmente todos os atributos variáveis) e todos os métodos da classe.

O nível seguinte de modularização é o nível físico. É porventura o nível com pior suporte pela linguagem e que é mais facilmente mal utilizado pelo programador inexperiente. A modularização física corresponde à divisão dos programas em ficheiros e tem duas utilidades principais: fazer uma divisão lógica das ferramentas de um programa a um nível superior ao das classes e, uma vez que um programa já não precisa de estar concentrado num único ficheiro, mas pode ser dividido por vários ficheiros, facilitar a reutilização de código em programas diferentes.

Finalmente, a linguagem C++ fornece ainda o conceito de espaço nominativo, que se pode fazer corresponder aproximadamente à noção de pacote. Este é o nível mais alto de modularização, e permite fazer uma divisão lógica das ferramentas de um programa a um nível superior ao da modularização física e, simultaneamente, evitar a colisão de nomes definidos

[1] O cabeçalho de uma rotina diz como ela se utiliza, embora não diga o que faz ou calcula. Para isso, e mesmo que o nome da rotina seja auto-explicativo, acrescenta-se ao cabeçalho um comentário de documentação que indica o que a rotina faz ou calcula e que inclui a sua pré-condição e a sua condição objectivo.

[2] A implementação de um módulo em C++ está muitas vezes visível ao programador consumidor. O programador consumidor de uma função pode, muitas vezes, ver o seu corpo ou implementação. Mas não pode, através do seu programa, afectar directamente essa implementação: o programador consumidor usa a implementação de um módulo sempre indirectamente através da sua interface. É um pouco como se os módulos em C++ fossem transparentes: pode-se ver o mecanismo, mas não se tem acesso directo a ele.

em cada parte de um programa complexo. Assim, um espaço nominativo (ou melhor, um pacote) abarca tipicamente vários módulos físicos.

Em resumo, existem os seguintes níveis de modularização:

1. Modularização procedimental: os módulos a este nível chamam-se *rotinas* (*funções* e *procedimentos*).
2. Modularização de dados: os módulos a este nível chamam-se *classes*.
3. Modularização física: os módulos a este nível chamam-se normalmente *módulos físicos* (ou simplesmente *módulos*) e correspondem a ficheiros (normalmente um par ou um trio de ficheiros, como se verá).
4. Modularização em pacotes: os módulos a este nível chamam-se *pacotes* e correspondem, no C++, a espaços nominativos.

Nas próximas secções serão discutidos em algum pormenor estes dois novos níveis de modularização: ficheiros (modularização física) e pacotes (espaços nominativos).

9.1 Modularização física

Nos capítulos anteriores viu-se que uma forma natural de reaproveitar código no mesmo programa é através da escrita de rotinas. Viu-se também como se criavam novos tipos de dados (i.e., TAD concretizados à custa de classes C++), como se equipavam esses novos tipos das respectivas operações, e como se utilizavam esses novos tipos num dado programa. Mas como reutilizar uma ferramenta, e.g., uma rotina ou uma classe C++, em programas diferentes? Claro está que uma solução seria usar as opções Copiar/Colar do editor de texto. Mas essa não é, decididamente, a melhor solução. Há várias razões para isso, mas a principal talvez seja que dessa forma sempre que se altera a ferramenta copiada, presumivelmente para lhe introduzir correcções ou melhorias, é necessário repetir essas alterações em todas as suas cópias, que ao fim de algum tempo se podem encontrar espalhadas por vários programas. Para obviar a este problema, a linguagem C++ fornece um mecanismo de reutilização mais conveniente: a compilação separada de módulos físicos correspondentes a diferentes ficheiros.

Mesmo que o objectivo não seja a reutilização de código, a modularização física através da colocação de diferentes ferramentas (rotinas, classes, etc.) em ficheiros separados é muito útil, pois permite separar claramente ferramentas com aplicações diversas, agrupando simultaneamente ferramentas com relações fortes entre si. Isto tem a vantagem não apenas de aumentar a estruturação do programa, mas também de facilitar a participação de várias equipas no desenvolvimento de um programa: cada equipa desenvolve um conjunto de módulos físicos diferentes, e consequentemente ficheiros diferentes. Além disso, a compilação separada acelera o processo de construção do ficheiro executável pois, como se verá, uma alteração num módulo de um programa não exige normalmente senão a reconstrução de uma pequena parte do programa, por vezes apenas do módulo afectado.

9.1.1 Constituição de um módulo

Um módulo físico de um programa é constituído normalmente por dois *ficheiros fonte*[3]: o ficheiro de interface (*header file*) com extensão `.H` (alternativamente `.hpp`, `.hh` ou mesmo `.h++`) e o ficheiro de implementação com extensão `.C` (alternativamente `.cpp`, `.cc` ou mesmo `.c++`)[4]. Por vezes um dos ficheiros não existe. É tipicamente o caso do módulo que contém a função `main()` do programa, que não é muito útil para reutilizações noutros programas e que só contém o ficheiro de implementação. Também ocorrem casos em que o módulo contém apenas o ficheiro de interface.

Normalmente cada módulo corresponde a uma unidade de tradução (*translation unit*). Uma unidade de tradução corresponde ao ficheiro de implementação de um módulo já depois de pré-processado, i.e., incluindo todos os ficheiros especificados em directivas `#include` (ver mais abaixo).

O termo módulo por si só refere-se normalmente a um módulo físico, i.e., a uma unidade de modularização física constituída normalmente por um ficheiro de interface e pelo respectivo ficheiro de implementação. Assim, de hora em diante a palavra *módulo* será usada neste sentido mais restrito, sendo utilizações mais latas do termo indicadas explicitamente ou, espera-se, claras pelo contexto.

Os nomes escolhidos para cada um dos ficheiros de um módulo reflectem a sua utilização usual. Normalmente o ficheiro de interface serve para indicar as interfaces de todas as ferramentas disponibilizadas pelo módulo e o ficheiro de implementação serve para implementar essas mesmas ferramentas. Vistos a um nível de abstracção superior, o ficheiro de interface corresponde à interface do módulo e o ficheiro de implementação à sua implementação.

Assim, o ficheiro de interface contém normalmente a declaração de rotinas e a definição das classes C++ disponibilizadas pelo módulo. Cada definição de um classe C++ contém a declaração das respectivas operações, a definição dos atributos de instância, a declaração dos atributos de classe, e a declaração das classes C++ e rotinas amigas da classe C++ em causa. Por outro lado, o ficheiro de implementação contém normalmente as definições de todas as ferramentas usadas dentro do módulo mas que não fazem parte da sua interface, ou seja, ferramentas de utilidade interna ao módulo, e as definições de todas as ferramentas que fazem parte da interface mas que foram apenas declaradas no ficheiro de interface. Mais à frente se apresentarão um conjunto de regras mais explícito acerca do conteúdo típico destes dois ficheiros.

9.2 Fases da construção do ficheiro executável

A construção de um ficheiro executável a partir de um conjunto de diferentes módulos físicos tem três fases: o pré-processamento, a compilação propriamente dita e a fusão.

Inicialmente, um programa chamado *pré-processador* age individualmente sobre o ficheiro de implementação (`.C`) de cada módulo, incluindo nele todos os ficheiros de interface (`.H`) es-

[3]Chamam-se ficheiros fonte pois são os ficheiros que contém o código C++ escrito pelo(s) programador(es) e que darão origem ao ficheiro executável do programa através de um processo de construção que produz uma série de ficheiros intermédios.

[4]É boa ideia reservar as extensões `.h` e `.c` para ficheiros de interface escritos na linguagem C.

pecificados por directivas `#include` e processando todas as directivas de pré-processamento (que correspondem a linhas começadas por um `#`). O resultado do pré-processamento é um ficheiro de extensão `.ii` (em Linux) que contém linguagem C++ e a que se chama uma *unidade de tradução*.

Depois, um programa chamado compilador traduz cada unidade de tradução (`.ii`) de C++ para código relocalizável em linguagem máquina, na forma de um ficheiro objecto de extensão `.o` (em Linux). Os ficheiros objecto, para além de conterem o código em linguagem máquina, contêm também uma lista dos símbolos definidos ou usados nesse ou por esse código e que será usada na fase de fusão. A compilação age sobre cada unidade de tradução independentemente de todas as outras, daí que seja usado o termo compilação separada para referir o processo de construção de um executável a partir dos correspondentes ficheiros fonte.

O termo *compilação* pode ser usado no sentido lato de construção de um executável a partir de ficheiros fonte ou no sentido estrito de tradução de uma unidade de tradução para o correspondente ficheiro objecto. Em rigor só o sentido estrito está correcto, mas é comum (e fazemo-lo neste texto) usar também o sentido lato onde isso for claro pelo contexto.

A última grande fase da construção é a fusão(*linking*). Nesta fase, que é a única que actua sobre todos os módulos em simultâneo (com excepção dos que não têm unidade de tradução), os ficheiros objecto do programa são fundidos (*linked*) num executável. Ao programa que funde os ficheiros objecto chama-se fusor (*linker*)[5].

Note-se que em Linux é o mesmo programa (`c++` ou `g++`) que se encarrega das três fases, podendo-as realizar todas em sequência ou apenas uma delas consoante os casos.

As várias fases da construção de um executável são descritas abaixo com um pouco mais de pormenor.

9.2.1 Pré-processamento

O pré-processamento é uma herança da linguagem C. Não fazendo parte propriamente da linguagem C++, é no entanto fundamental para desenvolver programas em módulos físicos separados. É uma forma muito primitiva de garantir a compatibilidade e a correcção do código escrito em cada módulo. A verdade, porém, é que infelizmente o C++ não disponibiliza nenhum outro método para atingir os mesmos objectivos, ao contrário de outras linguagens como o Java, onde a modularização física é suportada directamente pela linguagem.

O *pré-processador* age sobre um ficheiro de implementação (`.C`), gerando uma unidade de tradução (`.ii`). Esta unidade de tradução contém código C++, tal como o ficheiro de implementação original, mas o pré-processador faz-lhe algumas alterações, i.e., processa-o. O comando para pré-processar um ficheiro de implementação em Linux é[6]:

```
c++ -E nome.C -o nome.ii
```

[5]Neste texto optou-se pelos termos "fundir", "fusor" e "fusão". Porventura teria sido preferível a tradução literal de "*to link*" por "ligar", "*linker*" por "ligador" e "*linkage*" por "ligação".

[6]O comando também pode ser `g++`.

onde `nome` é o nome do módulo a pré-processar, `-E` é uma opção que leva o programa `c++` a limitar-se a proceder à pré-compilação do ficheiro e `-o` é uma opção que serve para indicar em que ficheiro deve ser colocado o resultado do pré-processamento.

O pré-processador copia o código C++ do ficheiro de implementação para a unidade de tradução[7], mas sempre que encontra uma linha começada por um cardinal (`#`) interpreta-a. Estas linhas são as chamadas *directivas de pré-processamento*, que, salvo algumas excepções pouco importantes neste contexto, não existem no ficheiro pré-processado, ou seja, na unidade de tradução gerada.

Directivas de pré-processamento

Existem variadíssimas directivas de pré-processamento. No entanto, só algumas são relevantes neste contexto:

- `#include`
- `#define`
- `#ifdef` (`#else`) e `#endif`
- `#ifndef` (`#else`) e `#endif`

A directiva `#include` tem dois formatos:

1. `#include <nome>`
2. `#include "nome"`

Em ambos os casos o resultado da directiva é a substituição dessa directiva, na unidade de tradução (ficheiro pré-processado), por todo o conteúdo pré-processado do ficheiro indicado por `nome` (e que pode incluir um *caminho*[8]). A diferença entre os dois formatos prende-se com o local onde os ficheiros a incluir são procurados. Os ficheiros incluídos por estas directivas são também pré-processados, pelo que podem possuir outras directivas de inclusão e assim sucessivamente.

Quando o primeiro formato é usado, o ficheiro é procurado nos locais "oficiais" do sistema onde se trabalha. Por exemplo, para incluir os ficheiros de interface que declaram as ferramentas de entradas e saídas a partir de canais da biblioteca padrão faz-se

```
#include <iostream>
```

[7]Na realidade o pré-processador faz mais do que isso. Por exemplo, elimina todos os comentários do código substituindo-os por um espaço. A maior parte das operações levadas a cabo pelo pré-processador, no entanto, não são muito relevantes para esta discussão simplificada.

[8]Optou-se por traduzir por *caminho* o inglês *path*.

pois este é um ficheiro de interface oficial. Neste caso é um ficheiro que faz parte da norma do C++. Isto significa que existe algures no sistema um ficheiro chamado `iostream` (procure-o, em Linux, usando o comando `locate iostream` e veja o seu conteúdo). É possível acrescentar ficheiros de interface aos locais oficiais, desde que se tenha permissões para isso, ou então acrescentar directórios à lista dos locais oficiais de modo a "oficializar" um conjunto de ferramentas por nós desenvolvido. Nos sistemas operativos baseados no Unix há vários directórios com ficheiros de interface oficiais, sendo o mais usual o directório `/usr/include`.

Quando o segundo formato da directiva de inclusão é usado, o ficheiro é procurado primeiro a partir do directório onde se encontra o ficheiro fonte contendo a directiva em causa e, caso não seja encontrado, é procurado nos locais "oficiais".

Os ficheiros incluídos são ficheiros de interface com declarações de ferramentas úteis ao módulo em processamento.

A directiva `#define` tem como objectivo a definição de *macros*, que são como que variáveis do pré-processador. As macros podem ser usadas de formas muito sofisticadas e perigosas, sendo um mecanismo externo à linguagem C++ propriamente dita e que com ela interferem muitas vezes de formas insuspeitas. Por isso recomenda-se cautela na sua utilização. Neste texto ver-se-á apenas a utilização mais simples:

```
#define NOME
```

Depois desta directiva, o pré-processador tem definida uma macro de nome *NOME* com conteúdo nulo. Convencionalmente o nome das macros escreve-se usando apenas maiúsculas para as distinguir claramente dos nomes usados no programa C++.

É possível colocar na unidade de tradução código alternativo ou condicional, consoante uma dada macro esteja definida ou não durante o pré-processamento do código fonte. A isso chama-se compilação condicional ou alternativa e consegue-se usando a directiva condicional `#ifdef` (que significa *if defined*)

```
#ifdef NOME
texto...
#endif
```

ou a directiva de selecção correspondente

```
#ifdef NOME
texto1...
#else
texto2...
#endif
```

ou ainda a directiva negada correspondente, começada por `#ifndef` (que significa *if not defined*).

Quando a directiva condicional é interpretada pelo pré-processador, o texto `texto`... só é pré-processado e incluído na unidade de tradução se a macro *`NOME`* estiver definida. Na directiva de selecção, se a macro estiver definida, então apenas o texto `texto1`... é pré-processado e incluído na unidade de tradução, caso contrário apenas o texto `texto2`... é pré-processado e incluído na unidade de tradução.

Depois do pré-processamento

A unidade de tradução que resulta do pré-processamento de um ficheiro de implementação possui, tipicamente, algumas directivas residuais que são colocadas pelo próprio pré-processador. É o que se passa com o pré-processador da GCC (Gnu Compiler Colection), usado em Linux. A discussão abaixo descreve esse ambiente particular de desenvolvimento.

Para que um programa possa ser executado em modo de depuração deve ser compilado com a opção `-g`. Acontece que, para que o depurador saiba a que linha nos ficheiros fonte corresponde cada instrução (para a poder mostrar no editor, por exemplo o `XEmacs`), a informação acerca da linha e do ficheiro fonte a que corresponde cada instrução em código máquina tem de ficar guardada nos ficheiros objecto e, depois da fusão, no ficheiro executável. Suponha-se o seguinte ficheiro de implementação chamado `olá.C`:

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Olá mundo!" << endl;
}
```

Podem-se usar as seguintes instruções para construir o executável:

```
c++ -E olá.C -o olá.ii
c++ -c olá.ii
c++ -o olá olá.o
```

Note-se que se compilou o ficheiro pré-processado, ou seja, a unidade de tradução. Como pode o compilador saber de onde vieram as linhas do ficheiro `olá.ii`, necessárias durante a depuração? Só se o ficheiro pré-processado contiver essa informação! Para isso servem as directivas introduzidas pelo pré-processador (que, como usualmente, começam por `#`).

Não é apenas para a depuração que essa informação é útil. Se essa informação não estivesse no ficheiro pré-processado, os erros de compilação seriam assinalados no ficheiro `olá.ii`, o que não ajudaria muito o programador, que editou o ficheiro `olá.C` e não quer saber do ficheiro `olá.ii` para nada. Logo, a informação acerca da origem das linhas da unidade de tradução também é útil para o compilador produzir mensagens de erro (experimente-se executar os comandos acima, mas acrescentando a opção `-P` ao pré-processar, e veja-se o que o compilador diz quando encontra um erro...).

As únicas directivas presentes nas unidades de tradução (`.ii`) têm o seguinte formato:

`#` *`linha nome`* [*x*...]

onde:

linha é o número da linha no ficheiro fonte de onde veio a próxima linha da unidade de tradução.

nome é o nome do ficheiro fonte de onde veio a próxima linha na unidade de tradução.

x são um conjunto de números com os seguintes possíveis valores:

1. Indica o começo de um novo ficheiro fonte.
2. Indica que se regressou a um ficheiro fonte (depois de terminar a inclusão de outro).
3. Indica que as linhas que se seguem vêm de um ficheiro de interface "oficial" (serve para desactivar alguns avisos do compilador).
4. Indica que as próximas linhas contêm linguagem C.

Exemplo

Suponha-se que se escreveu uma função `máximoDe()` para calcular o máximo dos primeiros valores contidos num vector de `double`:

```
/** Devolve o maior dos valores dos itens do vector v.
    @pre   PC ≡ 0 < v.size().
    @post  CO ≡ (Q j : 0 ≤ j < v.size() : v[j] ≤ máximoDe) ∧
                (E j : 0 ≤ j < v.size() : v[j] = máximoDe). */
double máximoDe(vector<double> const& v)
{
    double máximo = v[0];
    for(vector<double>::size_type i = 1; i != v.size(); ++i)
        if(máximo < v[i])
            máximo = v[i];
    return máximo;
}
```

Durante o desenvolvimento do programa onde a função acima é usada, é conveniente verificar a pré-condição da função. Desse modo, se o programador consumidor da função se enganar, o programa abortará imediatamente com uma mensagem de erro apropriada, o que antecipará a detecção do erro e consequente correcção. Assim, durante o desenvolvimento, a função deveria fazer a verificação explícita da pré-condição[9]:

[9] O procedimento `exit()` termina abruptamente a execução de um programa. Não se recomenda a sua utilização em nenhuma circunstância. Por favor leia um pouco mais e encontrará melhores alternativas...

```
/** Devolve o maior dos valores dos itens do vector v.
    @pre PC ≡ 0 < v.size().
    @post CO ≡ (Q j : 0 ≤ j < v.size() : v[j] ≤ máximoDe) ∧
              (E j : 0 ≤ j < v.size() : v[j] = máximoDe). */
double máximoDe(vector<double> const& v)
{
    if(v.size() == 0) {
        cerr << "Erro em máximo()! Vector com dimensão nula!"
             << endl;
        exit(1); // Atenção! Não se advoga o uso de exit()!
                 // Leia um pouco mais, por favor...
    }

    double máximo = v[0];
    for(vector<double>::size_type i = 1; i != v.size(); ++i)
        if(máximo < v[i])
            máximo = v[i];
    return máximo;
}
```

O problema desta solução é que, quando o programa está já desenvolvido, testado e distribuído, a instrução condicional acrescentada à função deixa de ter utilidade, pois presume-se que o programador consumidor já garantiu que todas as suas chamadas desta função verificam a pré-condição, servindo a sua verificação explícita apenas para tornar o programa mais lento. Daí que fosse conveniente que essa instrução fosse retirada depois de depurado o programa e que se voltasse a colocar apenas quando o programa fosse actualizado. Ou seja, o ideal seria que as instruções de verificação produzissem efeito em modo de depuração ou desenvolvimento e não produzissem qualquer efeito em modo de distribuição[10].

Mas pôr e tirar instruções desta forma é muito má ideia, mesmo se para o efeito se usarem comentários: é que pode haver muitas, mas mesmo muitas instruções deste tipo num programa, o que levará a erros e esquecimentos. O problema resolve-se recorrendo à compilação condicional:

```
/** Devolve o maior dos valores dos itens do vector v.
    @pre PC ≡ 0 < v.size().
    @post CO ≡ (Q j : 0 ≤ j < v.size() : v[j] ≤ máximoDe) ∧
              (E j : 0 ≤ j < v.size() : v[j] = máximoDe). */
double máximoDe(vector<double> const& v)
{
    #ifndef NDEBUG
    if(v.size() == 0) {
        cerr << "Erro em máximo()! Vector com dimensão nula!"
```

[10]O modo de distribuição (*release*) é o modo do programa tal como ele é distribuído aos seus utilizadores finais. O modo de depuração (*debug*) ou desenvolvimento é o modo do programa enquanto está em desenvolvimento e teste.

```
                << endl;
            exit(1);
        }
        #endif

        double máximo = v[0];
        for(vector<double>::size_type i = 1; i != v.size(); ++i)
            if(máximo < v[i])
                máximo = v[i];
        return máximo;
    }
```

Se a macro NDEBUG (que significa *not debug*) não estiver definida, i.e., se se estiver em fase de depuração, a verificação da pré-condição é feita. Caso contrário, i.e., se não se estiver em fase de depuração mas sim em fase de distribuição e portanto a macro NDEBUG estiver definida, então o código de verificação da pré-condição é eliminado da unidade de tradução e nem chega a ser compilado!

As instruções de asserção descritas na Secção 3.2.19 usam a macro NDEBUG exactamente da mesma forma que no código acima. De resto, as instruções de asserção permitem escrever o código de uma forma mais clara (aproveitou-se para colocar uma verificação parcial da condição objectivo)[11]:

```
    /** Devolve o maior dos valores dos itens do vector v.
        @pre  PC ≡ 0 < v.size().
        @post CO ≡ (Q j : 0 ≤ j < v.size() : v[j] ≤ máximoDe) ∧
                   (E j : 0 ≤ j < v.size() : v[j] = máximoDe). */
    double máximo(vector<double> const& v)
    {
        assert(0 < v.size());

        double máximo = v[0];
        for(vector<double>::size_type i = 1; i != v.size(); ++i)
            if(máximo < v[i])
                máximo = v[i];

        // Sem ciclos não se pode fazer muito melhor (amostragem em três locais):
        assert(v[0] <= máximo and v[v.size() / 2] <= máximo
               and v[v.size() - 1] <= máximo);

        return máximo;
    }
```

Todas as asserções de um módulo podem ser desligadas definindo a macro NDEBUG. Para o fazer não é sequer necessário alterar nenhum ficheiro do módulo! Basta dar a opção de

[11]Note-se que assert() é, na realidade, uma macro com argumentos (noção que não se descreve neste texto).

pré-processamento `-DNDEBUG`. A opção `-D`*`MACRO`* define automaticamente a macro *`MACRO`* em todos os ficheiros pré-processados. Logo, para pré-processar um ficheiro em modo de distribuição pode-se usar o comando

```
c++ -DNDEBUG -E nome.C -o nome.ii
```

ou, se se pretender também compilar o ficheiro, para além de o pré-processar:

```
c++ -DNDEBUG -c nome.C
```

9.2.2 Compilação

O *compilador* age sobre uma unidade de tradução (ficheiro pré-processado) e tradu-lo para linguagem máquina. O ficheiro gerado chama-se *ficheiro objecto* e, apesar de conter linguagem máquina, não é um ficheiro executável, pois contém informação adicional acerca desse mesmo código máquina, como se verá. Os ficheiros objecto têm extensão `.o` (em Linux) e contêm código máquina utilizável (sem necessidade de compilação) por outros programas. Em Linux, o comando para compilar a unidade de tradução de um módulo de nome *`nome`* é

```
c++ -Wall -ansi -pedantic -g -c nome.ii
```

ou, se se quiser pré-processar e compilar usando um só comando:

```
c++ -Wall -ansi -pedantic -g -c nome.C
```

em que a opção `-c` indica que se deve proceder apenas à compilação (e, se necessário, também ao pré-processamento), a opção `-Wall` pede ao compilador para avisar de todos (*all*) os potenciais erros (*Warnings*), as opções `-ansi` e `-pedantic` dizem para o compilador seguir tão perto quanto possível a norma da linguagem, e a opção `-g` indica que o ficheiro objecto gerado deve conter informação de depuração.

A compilação de uma unidade de tradução consiste pois na sua tradução para linguagem máquina e é feita em vários passos:

1. Análise lexical. Separa o texto do programa em símbolos, verificando a sua correcção. Equivale à identificação de palavras e sinais de pontuação que os humanos fazem quando lêem um texto, que na realidade consiste numa sequência de caracteres.

2. Análise sintáctica. Verifica erros gramaticais no código, por exemplo conjuntos de símbolos que não fazem sentido como `a / / b`. Equivale à verificação inconsciente da correcção gramatical de um texto que os humanos fazem. Ao contrário dos humanos, que são capazes de lidar com um texto contendo muitos erros gramaticais (e.g., "eu leste texto este e erro nenhum encontrámos"), o compilador não lida nada bem com erros sintácticos. Embora os compiladores tentem recuperar de erros passados e continuar a analisar o código de modo a produzir um conjunto tão relevante quanto possível de erros e avisos, muitas vezes enganam-se redondamente, assinalando erros que não estão presentes no código. Por isso, apenas o primeiro erro assinalado pelos compiladores é verdadeiramente fiável (mesmo que possa ser difícil de perceber).

3. Análise semântica. Verifica se o texto, apesar de sintacticamente correcto, tem significado. Por exemplo, verifica a adequação dos tipos de dados aos operadores, assinalando erros em expressões como `1.1 % 3.4`. Equivale à verificação inconsciente do sentido de frases gramaticalmente correctas. Por exemplo, o leitor humano reconhece como não fazendo sentido os versos[12]

   O ganso, gostou da dupla e fez também qüem, qüem
   Olhou pro cisne e disse assim vem, vem
   Que um quarteto ficará bem, muito bom, muito bem.

4. Optimização. Durante esta fase o compilador elimina código redundante, simplifica expressões, elimina variáveis desnecessárias, etc., de modo a poder gerar código máquina tão eficiente quanto possível.

5. Geração de código máquina (pode ter uma fase intermédia numa linguagem de mais baixo nível).

O resultado da compilação é um ficheiro objecto, como se viu. Este ficheiro não contém apenas código máquina. Simplificando algo grosseiramente a realidade, cada ficheiro objecto tem guardadas duas tabelas: uma é a tabela das disponibilidades e outra é a tabela das necessidades. A primeira tabela lista os nomes (ou melhor, as assinaturas) das ferramentas definidas pelo respectivo módulo, e portanto disponíveis para utilização em outros módulos. A segunda tabela lista os nomes das ferramentas que são usadas pelo respectivo módulo, mas não são definidas por ele, e que portanto devem ser definidas por outro módulo.

Por exemplo, suponham-se seguintes módulos `A` e `B`:

A.H

```
void meu(); // declaração do procedimento meu() que está definido em A.C.
```

A.C

```
#include "A.H"
#include "B.H"

// Definição do procedimento meu():
void meu()
{
    outro();
}
```

B.H

```
void outro(); // declaração do procedimento outro() que
              // está definido em B.C.
```

[12]"O Pato", de João Gilberto. No entanto, estes versos não estão errados, pois usam a figura da personificação. Os compiladores não têm esta nossa capacidade de entender figuras de estilo, bem entendido...

B.C

```
#include "B.H"

// Definição do procedimento outro():
void outro()
{
}
```

A compilação separada das unidades de tradução destes dois módulos resulta em dois ficheiros objecto `A.o` e `B.o` contendo:

A.o

1. Código máquina.
2. Disponibilidades: `meu()`
3. Necessidades: `outro()`

B.o

1. Código máquina.
2. Disponibilidades: `outro()`
3. Necessidades:

Cada ficheiro objecto, por si só, não é executável. Por exemplo, aos ficheiros `A.o` e `B.o` falta a função `main()` e ao ficheiro `A.o` falta o procedimento `outro()` para poderem ser executáveis. No entanto, mesmo que um ficheiro objecto contenha a definição da função `main()` e de todas as ferramentas usadas, não é executável. O executável é sempre gerado pelo fusor, que utiliza a informação acerca do que cada ficheiro objecto disponibiliza e necessita para fazer o seu trabalho.

9.2.3 Fusão

Quer o pré-processamento quer a compilação, já descritas, agem sobre um único módulo. O pré-processador age sobre o ficheiro de implementação do módulo (`.C`), honrando todas as directivas de `#include` que encontrar, pelo que na realidade o pré-processamento pode envolver vários ficheiros: um de implementação e os outros de interface. O resultado do pré-processamento é um único ficheiro, chamado unidade de tradução. O compilador age sobre uma unidade de tradução de cada vez e independentemente de todas as outras. Por isso se chama muitas vezes à modularização física compilação separada. Mas um programa, mesmo que o seu código esteja espalhado por vários módulos, tem normalmente um único ficheiro executável. Logo, tem de ser a última fase da construção de um executável a lidar com todos os módulos em simultâneo: a fusão.

O *fusor* é o programa que funde todos os ficheiros objecto do programa e gera o ficheiro executável. Em Linux, o comando para fundir os ficheiros objecto num executável é:

```
c++ -o programa módulo1.o módulo2.o ...
```

onde `módulo1.o` etc., são os ficheiros objecto a fundir e `programa` é o nome que se pretende dar ao ficheiro executável.

Se se quiser pré-processar, compilar e fundir usando um só comando, o que é pouco recomendável, pode-se usar o comando

```
c++ -Wall -ansi -pedantic -g -o programa módulo1.C módulo2.C ...
```

que começa por pré-processar o ficheiros de implementação, depois compila as respectivas unidades de tradução, e finalmente funde os ficheiros objecto resultantes.

O fusor basicamente concatena o código máquina de cada um dos ficheiros objecto especificados[13]. Mas durante esse processo:

1. Verifica se há repetições. Não pode haver dois ficheiros objecto a definir a mesma ferramenta. Se houver repetições, o fusor escreve uma mensagem de erro (com um aspecto um pouco diferente das mensagens do compilador).

2. Para cada ferramenta listada como necessária na tabela de necessidades de cada ficheiro objecto, o fusor verifica se ela está listada na tabela de disponibilidades de algum ficheiro objecto. Se faltar alguma ferramenta, o fusor assinala o erro.

3. Verifica se existe uma função `main()`. De outra forma não se poderia criar o executável, cuja execução começa sempre nessa função[14].

É de notar que o fusor não coloca no ficheiro executável todo o código máquina de todos os ficheiros objecto. O fusor tenta fazer uma selecção inteligente de quais as ferramentas que são realmente usadas pelo programa. É por isso que os ficheiros executáveis são normalmente bastante mais pequenos que a soma das dimensões das partes fundidas.

Ficheiros fundidos automaticamente

O fusor acrescenta à lista de ficheiros a fundir o ficheiro de arquivo (de extensão `.a`, ver Secção 9.2.4) contendo os ficheiros objecto da biblioteca padrão do C++. É por isso que é possível usar canais de entrada ou saída (`cin` e `cout`), cadeias de caracteres (`string`), etc., sem grandes preocupações: os respectivos ficheiros objecto são fundidos automaticamente ao construir o executável. Por exemplo, o ficheiro executável do famoso primeiro programa em C++:

`olá.C`

[13] Na realidade tem de fazer bastante mais, mas isso está fora do âmbito deste texto.

[14] Na realidade este último passo não existe, pois o fusor funde sempre automaticamente os ficheiros objecto dados pelo utilizador junto com outros ficheiros fornecidos por si e que contêm a função `main()` na tabela das necessidades.

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Olá mundo!" << endl;
}
```

pode ser construído por

```
c++ -Wall -ansi -pedantic -g -c olá.C
c++ -Wall -g -o olá olá.o
```

em que o primeiro comando pré-processa e compila o ficheiro de implementação `olá.C` e o segundo constrói o ficheiro executável `olá`. Estes comandos parecem indicar que o programa depende apenas do ficheiro `olá.C`. Nada mais falso. A inclusão de `iostream` e utilização de `cout`, `endl` e do operador << obrigam à fusão com, entre outros, o ficheiro objecto do módulo `iostream` existente no ficheiro de arquivo da biblioteca padrão da linguagem C++. O arquivo em causa chama-se `libstdc++.a` e encontra-se algures no sistema (procure-o com o comando `locate libstdc++.a` e use o comando `ar t caminho/libstdc++.a` para ver o seu conteúdo, onde encontrará o ficheiro `ios.o`).

Mesmo o programa mais simples obriga à fusão com um conjunto avulso de outros ficheiros objecto e ficheiros objecto incluídos em arquivos. O comando `c++` especifica esses ficheiros automaticamente, simplificando com isso a vida do programador. Experimente-se passar a opção `-nostdlib` ao comando `c++` ao construir o programa `olá` a partir do ficheiro objecto `olá.o`:

```
c++ -nostdlib -o olá olá.o
```

Como esta opção inibe a fusão automática do ficheiro `olá.o` com o conjunto de ficheiros objecto e arquivos de ficheiros objecto necessários à construção do executável, o comando anterior produz erros semelhantes ao seguinte[15]:

```
/usr/bin/ld: warning: cannot find entry symbol _start; defaulting to 08048074
olá.o: In function `main':
.../olá.C:7: undefined reference to `endl(ostream &)'
.../olá.C:7: undefined reference to `cout'
.../olá.C:7: undefined reference to `ostream::operator<<(char const *)'
.../olá.C:7: undefined reference to `ostream::operator<<(ostream &(*)(ostream &))'
collect2: ld returned 1 exit status
```

[15]Estes erros foram obtidos num sistema Linux, distribuição Red Hat 7.0, com o GCC versão 2.96, pelo que em sistemas com outra configuração as mensagens podem ser diferentes.

O primeiro dos erros apresentados é o menos evidente. Ele indica que o programa não tem sítio para começar. É que os programas em C++, como começam todos na função `main()`, têm de ser fundidos com ficheiros objecto que invocam essa função logo no início do programa. Alguns desses ficheiros objecto estão incluídos no ficheiro de arquivo da biblioteca padrão da linguagem C, que se chama `libc.a`. Se se usar o comando:

```
c++ -nostdlib -o olá /usr/lib/crt1.o /usr/lib/crti.o olá.o /usr/lib/libc.a /usr/lib/gcc-lib/i386-redhat-linux/2.96/libgcc.a
```

a primeira mensagem de erro desaparece, restando apenas:

```
olá.o: In function ‘main’:
.../olá.C:7: undefined reference to ‘endl(ostream &)’
.../olá.C:7: undefined reference to ‘cout’
.../olá.C:7: undefined reference to ‘ostream::operator<<(char const *)’
.../olá.C:7: undefined reference to ‘ostream::operator<<(ostream &(*)(ostream &))’
collect2: ld returned 1 exit status
```

Para conseguir construir o executável com sucesso, no entanto, é necessário especificar todos os ficheiros objecto e arquivos de ficheiros objecto necessários (e que incluem, entre outros, o arquivo da biblioteca padrão do C++ `libstdc++.a`):

```
c++ -nostdlib -o olá /usr/lib/crt1.o /usr/lib/crti.o /usr/lib/gcc-lib/i386-redhat-linux/2.96/crtbegin.o olá.o /usr/lib/gcc-lib/i386-redhat-linux/2.96/libstdc++.a /usr/lib/gcc-lib/i386-redhat-linux/2.96/libgcc.a /usr/lib/libc.a /usr/lib/gcc-lib/i386-redhat-linux/2.96/libgcc.a /usr/lib/gcc-lib/i386-redhat-linux/2.96/crtend.o /usr/lib/crtn.o
```

Não se preocupe se não sabe para que serve cada um dos ficheiros! Basta retirar a opção `-nostdlib` e tudo isto é feito automaticamente...

Fusão dinâmica

Quase todos os sistemas operativos suportam fusão dinâmica. A ideia é que a fusão com os ficheiros de arquivo das bibliotecas usadas pelo programa pode ser adiada, sendo realizada dinamicamente apenas quando o programa é executado. Esta solução tem como vantagens levar a ficheiros executáveis bastante mais pequenos e permitir a actualização dos ficheiros de arquivo das bibliotecas sem obrigar a reconstruir o executável (note-se que o executável e as bibliotecas, que neste caso se dizem dinâmicas ou partilhadas, podem ser fornecidos por entidades independentes). Os ficheiros de arquivo de bibliotecas dinâmicas ou partilhadas têm extensão `.a`, em Linux, e `.dll` em Windows. Para mais pormenores ver [9, Capitulo 7].

9.2.4 Arquivos

É possível colocar um conjunto de ficheiros objecto relacionados num ficheiro de arquivo. Estes ficheiros têm prefixo `lib` e extensão `.a` (de *archive*). É típico que os ficheiros objecto de uma biblioteca de ferramentas sejam colocados num único ficheiro de arquivo, para simplificar a sua fusão com os ficheiros objecto de diferentes programas. Por exemplo, os ficheiros objecto da biblioteca padrão da linguagem C++ (de nome `stdc++`) estão arquivados no ficheiro `libstdc++.a` (que se encontra algures num sub-directório do directório `/usr/lib`).

Para arquivar ficheiros objecto não se usa o fusor: usa-se o chamado *arquivador*. O programa arquivador invoca-se normalmente como se segue:

```
ar ru libbiblioteca.a módulo1.o módulo2.o ...
```

Onde `ru` são opções passadas ao arquivador (`r` significa *replace* e `u` significa *update*), *`biblioteca`* é o nome da biblioteca cujo arquivo se pretende criar, e *`módulo1`* etc., são os nomes dos módulos que constituem a biblioteca e cujos ficheiros objecto se pretende arquivar.

Como cada módulo incluído numa biblioteca tem um ficheiro de interface e produz geralmente um ficheiro objecto que é guardado no arquivo dessa biblioteca, pode-se dizer que as bibliotecas são representadas por:

1. Ficheiro de arquivo dos ficheiros objecto (um ficheiro objecto por módulo com ficheiro de implementação) da biblioteca.
2. Ficheiros de interface de cada módulo (que podem incluir outros ficheiros de interface).

É o que se passa com a biblioteca padrão da linguagem C++, representada em Linux pelo ficheiro de arquivo `libstdc++.a` e pelos vários ficheiros de interface que a compõem (`iostream`, `string`, `cmath`, etc.).

Exemplo

Suponha-se uma biblioteca de nome `saudações` constituída (para já) apenas pelo seguinte módulo:

`olá.H`

```
void olá();
```

`olá.C`

```
#include "olá.H"

#include <iostream>

using namespace std;

void olá()
{
    cout << "Olá mundo!" << endl;
}
```

Para construir o ficheiro de arquivo da biblioteca pode-se usar os seguintes comandos:

```
c++ -Wall -ansi -pedantic -g -c olá.C
ar ru libsaudações.a olá.o
```

em que o primeiro comando pré-processa e compila o ficheiro de implementação do módulo `olá` e o segundo arquiva o ficheiro objecto resultante no arquivo da biblioteca `saudações`, de nome `libsaudações.a`.

Seja agora um programa de nome `saúda` que pretende usar o procedimento `olá()`. Para isso deve começar por incluir o ficheiro de interface `olá.H`, que se pretende seja tomado por um ficheiro oficial (inclusão com <> e não `""`):

`saúda.C`

```
#include <olá.H>

int main()
{
    olá();
}
```

O pré-processamento e compilação do ficheiro de implementação `saúda.C` com o comando

```
c++ -Wall -ansi -pedantic -g -c saúda.C
```

resulta na mensagem de erro

```
saúda.C:1: olá.H: No such file or directory
```

O problema é que, para incluir `olá.H` como um ficheiro de interface oficial, uma de três coisas tem de ocorrer:

1. O ficheiro `olá.H` está num local oficial (e.g., `/usr/include` em Linux).

2. Dá-se a opção `-I.` ao comando `c++` para que ele acrescente o directório corrente (`.`) à lista de directórios onde os ficheiros de interface oficiais são procurados:

   ```
   c++ -Wall -ansi -pedantic -g -I. -c olá.C
   ```

3. Coloca-se o ficheiro `olá.H` num directório criado para o efeito (e.g., `$HOME/include`, onde `$HOME` é o directório "casa" do utilizador corrente em Linux) e coloca-se esse directório na variável de ambiente (*environment variable*) `CPLUS_INCLUDE_PATH`, que contém uma lista de directórios que, para além dos usuais, devem ser considerados como contendo ficheiros de interface oficiais:

```
mkdir $HOME/include
setenv CPLUS_INCLUDE_PATH $HOME/include
mv olá.H $HOME/include
c++ -Wall -ansi -pedantic -g -c saúda.C
```

onde o primeiro comando (criar directório) só têm de ser dado uma vez, e o segundo tem de ser dado sempre que se entra no Linux (e só funciona com o interpretador de comandos `/bin/tcsh`), pelo que é um bom candidato a fazer parte do ficheiro `.tcshrc`.

Falta agora fundir o ficheiro `saúda.o` com os ficheiros objecto do ficheiro arquivo `libsaudações.a` (que por acaso é só um: `olá.o`). Para isso podem-se usar um de três procedimentos:

1. Fusão directa com o ficheiro de arquivo:

   ```
   c++ -o saúda saúda.o libsaudações.a
   ```

2. Fusão usando a opção `-l` (*link with*):

   ```
   c++ -o saúda saúda.o -L. -lsaudações
   ```

   É necessária a opção `-L.` para que o comando `c++` acrescente o directório corrente (`.`) à lista de directórios onde os ficheiros de arquivo oficiais são procurados.

3. Fusão usando a opção `-l`, mas convencendo o comando `c++` de que o arquivo em causa é oficial. Para isso:

   (a) coloca-se o ficheiro `libsaudações.a` num local oficial (e.g., `/usr/lib` em Linux); ou

   (b) coloca-se o ficheiro `libsaudações.a` num directório criado para o efeito, e.g., `$HOME/lib`, e coloca-se esse directório na variável de ambiente `LIBRARY_PATH`, que contém uma lista de directórios que, para além dos usuais, devem ser considerados como contendo os arquivos oficiais:

   ```
   mkdir $HOME/lib
   setenv LIBRARY_PATH $HOME/lib
   mv libsaudações.a $HOME/lib
   c++ -o saúda saúda.o -lsaudações
   ```

   onde o primeiro comando (criar directório) só têm de ser dado uma vez, e o segundo tem de ser dado sempre que se entra no Linux (e só funciona com a shell /bin/tcsh), pelo que é um bom candidato a fazer parte do ficheiro .tcshrc.

Arquivos vs. ficheiros objecto

A fusão com um arquivo de ficheiros objecto não é, em rigor, equivalente à fusão directa com os ficheiros objecto arquivados. Enquanto numa fusão são verificadas definições repetidas em todos os ficheiros objecto existentes, essa verificação não é feita no caso dos ficheiro de arquivo.

Estes são percorridos por ordem de especificação na linha de comandos e a pesquisa pára logo que se encontra um ficheiro objecto arquivado que contenha a ferramenta procurada. Assim, o comando

```
c++ -o saúda saúda.o olá.o olá.o
```

produziria uma mensagem de erro indicando a definição múltipla do procedimento `olá()`, enquanto o comando

```
c++ -o saúda saúda.o libsaudações.a libsaudações.a
```

não produziria qualquer mensagem de erro, pois o fusor pararia ao encontrar o procedimento no primeiro arquivo especificado, ignorando o segundo.

9.3 Ligação dos nomes

Os exemplos das secções anteriores tinham uma particularidade interessante: rotinas definidas num módulo podiam ser utilizadas noutros módulos, desde que apropriadamente declaradas. Será que é sempre assim? E será assim também para variáveis e constantes globais? E o caso das classes?

Nesta secção abordar-se-á a questão da *ligação* dos nomes em C++. A ligação de um nome tem a ver com a possibilidade de uma dada entidade (e.g., uma função, uma constante, uma classe, etc.), com um dado nome, poder ser utilizada fora do módulo (ou melhor, fora da unidade de tradução) onde foi definida. Em C++ existem três tipos de ligação possível para os nomes: nomes sem ligação, com ligação interna, e com ligação externa.

Quando um nome não tem ligação, a entidade correspondente não pode ser usada senão no seu âmbito de visibilidade. Uma variável local, por exemplo, não tem ligação, pois pode ser usada apenas no bloco onde foi definida.

Uma entidade cujo nome tenha ligação interna pode ser usada em variados âmbitos dentro da unidade de tradução onde foi definida, desde que o seu nome tenha sido previamente declarado nesses âmbitos, mas não pode ser usada em outras unidades de tradução do mesmo programa. Normalmente há uma unidade de tradução por módulo (que é o seu ficheiro de implementação pré-processado). Assim, pode-se dizer que uma entidade cujo nome tem ligação interna é apenas utilizável dentro do módulo que a define.

Uma entidade cujo nome tenha ligação externa pode ser usada em qualquer âmbito de qualquer unidade de tradução do programa, desde que o seu nome tenha sido previamente declarado nesses âmbitos. Assim, pode-se dizer que uma entidade cujo nome tem ligação externa é utilizável em qualquer módulo do programa.

Em geral as entidades definidas dentro de blocos de instruções, que são entidades locais, não têm ligação. Assim, esta discussão centrar-se-á essencialmente nas entidades com ligação, sendo ela interna ou externa, e que geralmente são as entidades globais, i.e., definidas fora de qualquer função ou procedimento.

A linguagem C++ impõe a chamada *regra da definição única*. Esta regra diz que, no mesmo contexto, não podem ser definidas mais do que uma entidade com o mesmo nome. Assim, resumidamente:

1. Um programa não pode definir mais do que uma rotina ou método que não seja em-linha e com ligação externa se estes tiverem a mesma assinatura (podem ter o mesmo nome, desde que variem na lista de parâmetros).
2. Uma unidade de tradução não pode definir mais do que uma rotina não-membro que não seja em-linha e com ligação interna se estas tiverem a mesma assinatura.
3. Uma variável ou constante global com ligação externa só pode ser definida uma vez no programa.
4. Uma variável ou constante global com ligação interna só pode ser definida uma vez na sua unidade de tradução.
5. Uma classe ou um tipo enumerado não-local só pode ser definido uma vez em cada unidade de tradução e, se for definido em diferentes unidades de tradução, a sua definição tem de ser idêntica.

9.3.1 Vantagens de restringir a ligação dos nomes

É muito importante perceber que há vantagens em restringir a utilização de determinadas ferramentas a um dado módulo: esta restrição corresponde a distinguir entre ferramentas que fazem apenas parte da implementação do módulo e ferramentas que são disponibilizadas na sua interface. É a possibilidade de distinguir entre o que está restrito e o que não está que faz com que os módulos físicos mereçam o nome que têm...

Suponha-se, por exemplo, um módulo `vectores` com funções e procedimentos especializados em operações sobre vectores de inteiros. Esse módulo poderia consistir nos seguintes ficheiros de interface e implementação:

`matrizes.H`

```
/** Devolve o maior dos valores dos itens do vector v.
    @pre PC ≡ 0 < v.size().
    @post CO ≡ (Q j : 0 ≤ j < v.size() : m[j] ≤ máximoDe) ∧
               (E j : 0 ≤ j < v.size() : m[j] = máximoDe). */
double máximoDe(vector<double> const& v);

/** Devolve o índice do primeiro item com o máximo valor do vector v.
    @pre PC ≡ 1 ≤ v.size().
    @post CO ≡ 0 ≤ índiceDoPrimeiroMáximoDe < v.size()∧
 (Q j : 0 ≤ j < v.size() : v[j] ≤ v[índiceDoPrimeiroMáximoDe]) ∧
 (Q j : 0 ≤ j < índiceDoPrimeiroMáximoDe : v[j] < v[índiceDoPrimeiroMáximoDe]). */
int índiceDoPrimeiroMáximoDe(vector<int> const& v);
```

```
/** Ordena os valores do vector v.
    @pre PC ≡ v = v.
    @post CO ≡ perm(v, v) ∧ (Q i, j : 0 ≤ i ≤ j < v.size() : v[i] ≤ v[j]).
void ordenaPorOrdemNãoDecrescente(vector<int>& v);
```

matrizes.C

```
#include <cassert>

using namespace std;

/** Verifica se o valor de máximo é o máximo dos valores num vector v.
    @pre PC ≡ 0 < v.size().
    @post CO ≡ éMAaximoDe = ((Q j : 0 ≤ j < v.size() : m[j] ≤ máximo) ∧
                             (E j : 0 ≤ j < v.size() : m[j] = máximo)). */
bool éMáximoDe(int const máximo, vector<int> const& v)
{
    assert(0 < v.size());

    bool existe = false;
    // CI ≡ existe = (E j : 0 ≤ j < i : m[j] = máximo) ∧
    //       (Q j : 0 ≤ j < i : m[j] ≤ máximo).
    for(vector<int>::size_type i = 0; i != v.size(); ++i) {
        if(máximo < v[i])
            return false;
        existe = existe or máximo == v[i];
    }
    return existe;
}

/** Verifica se os valores do vector v estão por ordem não decrescente.
    @pre PC ≡ V.
    @post CO ≡ éNãoDecrescente =
                (Q j, k : 0 ≤ j ≤ k < v.size() : v[j] ≤ v[k]). */
bool éNãoDecrescente(vector<int> const& v)
{
    if(v.size() <= 1)
        return true;
    // CI ≡ (Q j, k : 0 ≤ j ≤ k < i : v[j] ≤ v[k]).
    for(vector<int>::size_type i = 1; i != v.size(); ++i)
        if(v[i - 1] > v[i])
            return false;
    return true;
}
```

```
double máximoDe(vector<double> const& v);
{
    assert(0 < v.size());

    double máximo = v[0];
    // CI ≡ (Q j : 0 ≤ j < i : v[j] ≤ máximo) ∧
    //      (E j : 0 ≤ j < i : m[j] = máximo).
    for(vector<double>::size_type i = 1; i != v.size(); ++i)
        if(máximo < v[i])
            máximo = v[i];

    assert(éMáximoDe(máximo, v));

    return máximo;
}

int índiceDoPrimeiroMáximoDe(vector<int> const& v)
{
    assert(1 <= v.size());

    int i = 0;
    // CI ≡ 0 ≤ i < k ∧ (Q j : 0 ≤ j < k : v[j] ≤ v[i]) ∧
    //      (Q j : 0 ≤ j < i : v[j] < v[i]) ∧ 0 ≤ k ≤ v.size().
    for(vector<int>::size_type k = 1; k != v.size(); ++k)
        if(v[i] < v[k])
            i = k;

    assert(0 <= i and i < v.size() and éMáximoDe(v[i], v));

    return i;
}

void ordenaPorOrdemNãoDecrescente(vector<int>& v)
{
    ... // um algoritmo de ordenação eficiente...

    assert(éNãoDecrescente(v));
}
```

No ficheiro de implementação deste módulo existem duas funções de utilidade restrita ao módulo. A primeira, `éMáximoDe()`, serve para verificar se um dado valor é o máximo dos valores contidos num vector. A segunda, `éNãoDecrescente()`, serve para verificar se os itens de um vector estão por ordem crescente (ou melhor, se formam uma sucessão monótona não-decrescente). Ambas são usadas em instruções de asserção para verificar se (pelo menos parte) das condições objectivo das outras rotinas do módulo são cumpridas.

Que estas duas funções não têm qualquer utilidade directa para o programador consumidor desde módulo é evidente. Saber se um valor é o maior dos valores dos itens de um vector não é muito útil em geral. O que é útil é saber qual é o maior dos valores dos itens de um vector. De igual forma não é muito útil em geral saber se um vector está ordenado. É muito mais útil ter um procedimento para ordenar esse vector.

Haverá alguma desvantagem em deixar que essas funções sejam utilizadas fora deste módulo? Em geral sim. Se são ferramentas úteis apenas dentro do módulo, então o programador do módulo pode decidir alterar o seu nome ou mesmo eliminá-las sem "dar cavaco a ninguém". Se isso acontecesse, o incauto programador consumidor dessas funções ficaria numa situação incómoda... Além disso, corre-se o risco de o nome dessas funções colidir com o nome de funções existentes noutros módulos (ver Secção 9.6.1 mais abaixo).

Assim, seria desejável que as duas funções de verificação tivessem ligação interna.

9.3.2 Espaços nominativos sem nome

A melhor forma de restringir a utilização de uma entidade ao módulo em que está definida é colocá-la dentro de um espaço nominativo sem nome (ver Secção 9.6.2). Assim, a solução para o problema apresentado na secção anterior passa por colocar as duas funções de verificação num espaço nominativo sem nome:

`matrizes.C`

```
#include <cassert>

using namespace std;

namespace { // Aqui definem-se as ferramentas de utilidade restrita a este módulo:

    /** Verifica se o valor de máximo é o máximo dos valores num vector v.
        @pre  PC ≡ 0 < v.size().
        @post CO ≡ éMAaximoDe = ((Q j : 0 ≤ j < v.size() : m[j] ≤ máximo) ∧
                                 (E j : 0 ≤ j < v.size() : m[j] = máximo)). */
    bool éMáximoDe(int const máximo, vector<int> const& v)
    {
        assert(0 < v.size());

        bool existe = false;
        // CI ≡ existe = (E j : 0 ≤ j < i : m[j] = máximo) ∧
        //      (Q j : 0 ≤ j < i : m[j] ≤ máximo).
        for(vector<int>::size_type i = 0; i != v.size(); ++i) {
            if(máximo < v[i])
                return false;
            existe = existe or máximo == v[i];
        }
        return existe;
    }
```

```
    /** Verifica se os valores do vector v estão por ordem não decrescente.
        @pre  $PC \equiv \mathcal{V}$.
        @post $CO \equiv$ éNãoDecrescente =
                  $(\mathbf{Q}\, j, k : 0 \leq j \leq k <$ v.size() $:$ v$[j] \leq$ v$[k])$. */
    bool éNãoDecrescente(vector<int> const& v)
    {
        if(v.size() <= 1)
            return true;
        // $CI \equiv (\mathbf{Q}\, j, k : 0 \leq j \leq k <$ i $:$ v$[j] \leq$ v$[k])$.
        for(vector<int>::size_type i = 1; i != v.size(); ++i)
            if(v[i - 1] > v[i])
                return false;
        return true;
    }

}

double máximoDe(vector<double> const& v);
{
    assert(0 < v.size());

    double máximo = v[0];
    // $CI \equiv (\mathbf{Q}\, j : 0 \leq j <$ i $:$ v$[j] \leq$ máximo$) \wedge$
    //       $(\mathbf{E}\, j : 0 \leq j <$ i $:$ m$[j] =$ máximo$)$.
    for(vector<double>::size_type i = 1; i != v.size(); ++i)
        if(máximo < v[i])
            máximo = v[i];

    assert(éMáximoDe(máximo, v));

    return máximo;
}

int índiceDoPrimeiroMáximoDe(vector<int> const& v)
{
    assert(1 <= v.size());

    int i = 0;
    // $CI \equiv 0 \leq$ i $<$ k $\wedge (\mathbf{Q}\, j : 0 \leq j <$ k $:$ v$[j] \leq$ v[i]$) \wedge$
    //       $(\mathbf{Q}\, j : 0 \leq j <$ i $:$ v$[j] <$ v[i]$) \wedge 0 \leq$ k $\leq$ v.size().
    for(vector<int>::size_type k = 1; k != v.size(); ++k)
        if(v[i] < v[k])
            i = k;

    assert(0 <= i and i < n and éMáximoDe(v[i], v));
```

```
        return i;
    }

    void ordenaPorOrdemNãoDecrescente(int m[], int n)
    {
        assert(0 <= n);

        ... // um algoritmo de ordenação eficiente...

        assert(éNãoDecrescente(m, n));
    }
```

Em termos técnicos a definição de entidades dentro de um espaço nominativo sem nome não lhes dá obrigatoriamente ligação interna. O que acontece na realidade é que o compilador atribui um nome único ao espaço nominativo. Em termos práticos o efeito é o mesmo que se as entidades tivessem ligação interna.

9.3.3 Ligação de rotinas, variáveis e constantes

É importante voltar a distinguir entre os conceitos de definição e declaração. Uma definição cria uma entidade. Uma declaração indica a existência de uma entidade, indicando o seu nome e cararacterísticas. Uma definição é sempre também uma declaração. Mas uma declaração no seu sentido estrito não é uma definição, pois indicar a existência de uma entidade e o seu nome não é o mesmo que criar essa entidade: essa entidade foi ou será criada num outro local.

Nas secções seguintes apresentam-se exemplos que permitem distinguir entre declarações e definições de vários tipos de entidades e saber qual a ligação dos seus nomes. Nas tabelas, admite-se sempre que existem dois módulos no programa: o módulo do lado esquerdo (`define`) é o módulo que define a maior parte das entidades enquanto o módulo do lado direito (`usa`) é o módulo que as usa.

Rotinas

A declaração (em sentido estrito) de uma rotina faz-se colocando o seu cabeçalho seguido de `;`. Por exemplo:

```
    /** Devolve a potência n de x.
        @pre PC ≡ 0 ≤ n ∨ x ≠ 0.
        @post CO ≡ potência = x^n. */
    double potência(double const x, int const n); // função.

    /** Troca os valores dos dois argumentos.
        @pre PC ≡ a = a ∧ b = b.
        @post CO ≡ a = b ∧ b = a. */
    void troca(int& a, int& b);                    // procedimento.
```

A definição de uma rotina faz-se indicando o seu corpo imediatamente após o cabeçalho. Por exemplo:

```
double potência(double const x, int const n)
{
    assert(0 <= n or x != 0.0);

    if(n < 0)
        return 1.0 / potência(x, -n);
    double r = 1.0;
    // CI ≡ r = x^i ∧ 0 ≤ i ≤ n.
    for(int i = 0; i != n; ++i)
        r *= x;
    return r;
}

void troca(int& a, int& b)
{
    int const auxiliar = a;
    a = b;
    b = auxiliar;
}
```

Os métodos de uma classe têm ligação externa se a respectiva classe tiver ligação externa. Caso contrário não têm ligação, uma vez que as classes também ou não têm ligação ou têm ligação externa (ver mais abaixo).

Em C++ não é possível definir rotinas locais (mas ver C.2). Sobram, pois, as funções ou procedimentos globais (ou definidos num espaço nominativo), que têm sempre ligação. Por omissão as funções e procedimentos globais têm ligação externa. Para que uma função ou procedimento tenha ligação interna, é necessário preceder a sua declaração do especificador de armazenamento `static`. É um erro declarar uma função ou procedimento como tendo ligação externa e, posteriormente, declará-lo com tendo ligação interna.

Sejam os dois módulos representados na Figura 9.1.

Ao fundir os dois módulos da figura (e admitindo que se eliminaram os erros de compilação assinalados) o fusor gerará erros como os seguintes:

```
.../usa.C:35: undefined reference to `f3(void)'
.../usa.C:37: undefined reference to `f5(void)'
```

No primeiro caso, o erro ocorre porque o único procedimento `f3()` que existe tem ligação interna ao módulo `define` (a sua definição é precedida do especificador `static`). No segundo caso, o procedimento `f5()` foi declarado e usado em `usa.C` mas nenhum módulo o define.

define.C	usa.C
void f4(); // externo void f1() // externo {...} void f2() // externo {...} static void f3() // interno {...} void f4() // externo { f3(); }	*void f1(); // externo* *void f3(); // externo* *extern void f2(); // externo* *void f5(); // externo* *void f6(); // externo* /* Erro de compilação! Declarado externo e depois definido interno: */ static void f6() {...} /* Erro de compilação! Não está definido dentro do módulo: */ *static void f7(); // interno.* *static void f8(); // interno.* *extern void f8(); // interno.* void f8() // interno. {...} int main() { f1(); f2(); f3(); // Erro de compilação, falta declaração: f4(); f5(); f8(); }

Figura 9.1: Exemplo de ligação para rotinas. Declarações em itálico e fontes de erros de compilação a vermelho.

O especificador `extern`, embora útil no caso das variáveis e constantes, como se verá mais abaixo, é inútil no caso das rotinas[16]. O que determina a ligação é o especificador `static`. Se uma função ou procedimento for declarado `static`, terá ligação interna. Assim, o seguinte código declara e define um procedimento com ligação interna:

```
static void f7();
extern void f7()
{ ... }
```

A utilização de da palavra-chave `static` está deprecada[17] . É preferível usar espaços nominativos sem nome, ou anónimos, para restringir a ligação de uma rotina ou instância a uma unidade de tradução.

Variáveis

As variáveis definidas dentro de um bloco, i.e., as variáveis locais, não têm ligação. As variáveis membro de instância de uma classe têm um esquema de acesso próprio, descrito no Capítulo 7. As variáveis membro de classe (ou estáticas, ver Secção 7.17.2) têm ligação externa se a respectiva classe tiver ligação externa. Caso contrário não terão ligação, uma vez que as classes ou não têm ligação ou têm ligação externa. Os casos mais interessantes de ligação de variáveis dizem respeito a variáveis globais (ou definidas num espaço nominativo), que têm sempre ligação interna ou externa.

Antes de discutir a ligação de variáveis globais, porém, é conveniente reafirmar o que vem sendo dito: é má ideia usar variáveis globais!

Até este momento neste texto nunca se declararam variáveis globais. Há dois factores que permitem distinguir uma declaração de uma definição no caso das variáveis. Para que uma declaração de uma variável seja uma declaração no sentido estrito, i.e., para que não seja também uma definição, têm de acontecer duas coisas:

1. A declaração tem de ser precedida do especificador `extern`.

2. A declaração não pode inicializar a variável.

Se uma declaração possuir uma inicialização explícita é forçosamente uma definição, mesmo que possua o especificador `extern`. Se uma definição de uma variável global não possuir uma inicialização explícita, será inicializada implicitamente com o valor por omissão do tipo respectivo, mesmo que o seu tipo seja um dos tipos básicos do C++. Ou seja, ao contrário do

[16]Na realidade o especificador `extern` não é totalmente inútil em conjunto com rotinas. Para se poder invocar, a partir de código C++, uma rotina num módulo em linguagem C, é necessário preceder a declaração dessa função ou procedimento de `extern "C"`, como se segue:

```
extern "C" int atoi(char char[]);
```

[17]Usa-se a palavra deprecar no seu sentido em inglês: deprecar algo é mais forte que não recomendar esse algo: é recomendar não utilizar/fazer esse algo.

que acontece com as variáveis locais dos tipos básicos, as variáveis globais dos tipos básicos são inicializadas implicitamente com valor por omissão do respectivo tipo (que é sempre um valor nulo). Se o tipo da variável for uma classe sem construtor por omissão, então uma definição sem inicialização é um erro.

Por exemplo:

```
int i = 10;        // definição e inicialização explícita com 10.
int j;             // definição e inicialização implícita com 0.
extern int k = 0;  // definição e inicialização com 0.
extern int j;      // declaração.

class A {
  public:
    A(int i)
        : i(i) {
    }
  private:
    int i;
};

A a(10); // definição e inicialização com 10.
A b;     // tentativa de definição: falha porque A não tem construtor por omissão.
```

Uma variável global tem, por omissão, ligação externa. Para que uma variável global tenha ligação interna é necessário preceder a sua declaração do especificador `static`. Na realidade, como os especificadores `static` e `extern` não se podem aplicar em simultâneo, a presença do especificador `static` numa declaração obriga-a a ser também uma definição.

Sejam os dois módulos representados na Figura 9.2.

Ao fundir os dois módulos acima (e admitindo que se eliminaram os erros de compilação assinalados) o fusor gerará erros como os seguintes:

```
.../usa.C:35: undefined reference to 'v3'
.../usa.C:37: undefined reference to 'v5'
```

No primeiro caso, o erro ocorre porque a única variável `v3` que existe tem ligação interna ao módulo `define`, uma vez que a sua definição é precedida do especificador `static`. No segundo caso, a variável `v5` foi declarada em `usa.C` mas nenhum módulo a define.

Constantes

As constantes definidas dentro de um bloco, ou seja, constantes locais, não têm ligação. As constantes membro de instância de uma classe têm um esquema de acesso próprio, descrito no Capítulo 7. As constantes membro de classe (ou estáticas, ver Secção 7.17.2) têm ligação

define.C	usa.C
extern int v4; // externa int v1; // externa int v2 = 1; // externa static int v3 = 3; // interna int v4; // externa	*extern int v1;* // externa *extern int v2;* // externa *extern int v3;* // externa *extern int v5;* // externa *extern int v6;* // externa /* Erro (ou aviso) de compilação! Declarada externa e depois definida interna: */ static int v6 = 12; static int v7; // interna *extern int v7;* // interna int main() { int i = v1 + v2 + v3 + // Erro de compilação, falta declaração: v4 + v5 + v7; }

Figura 9.2: Exemplo de ligação para variáveis. Declarações em itálico e fontes de erros de compilação a vermelho.

externa se a respectiva classe tiver ligação externa. Caso contrário não terão ligação, uma vez que as classes também ou não têm ligação ou têm ligação externa. Os casos mais interessantes de ligação de constantes dizem respeito a constantes globais (ou definidas num espaço nominativo), que têm sempre ligação interna ou externa.

Para que uma declaração de uma constante seja uma declaração no sentido estrito, i.e., para que não seja também uma definição, têm de acontecer duas coisas:

1. A declaração tem de ser precedida do especificador `extern`.
2. A declaração não pode inicializar a constante.

Se uma declaração possuir uma inicialização explícita, então é forçosamente uma definição, mesmo que possua o especificador `extern`. Se uma definição de uma constante global não possuir uma inicialização explícita, será inicializada implicitamente com o valor por omissão do tipo respectivo, excepto se o seu tipo seja um dos tipos básicos do C++ ou, em geral, um POD (*plain old datatype*). Se o tipo da constante for uma classe sem construtor por omissão, então uma definição sem inicialização é um erro. As regras que definem o que é exactamente um POD são complexas. No contexto deste texto bastará dizer que um agregado, entendido como uma classe só com variáveis membro públicas de tipos básicos ou de outros agregados ou matrizes de tipos básicos e agregados, é um POD. Por exemplo, a classe

```
struct Ponto {
    double x;
    double y;
};
```

é um agregado, e portanto um POD, logo:

```
Ponto const p;
```

é um erro, pois falta uma inicialização explícita.

Outros exemplos:

```
int const i = 10;        // definição e inicialização explícita com 10.
int const j;             // tentativa de definição: falha por-
que j é de um tipo básico.
extern int const k = 0;  // definição e inicialização com 0.
extern int const i;      // declaração.

class A {
  public:
    A(int i)
        : i(i) {
    }
```

```
  private:
    int i;
};

A const a(10); // definição e inicialização com 10.
A const b;     // tentativa de definição: falha porque A
               // não tem construtor por omissão.
```

Uma constante global tem, por omissão, ligação interna, i.e., o oposto da ligação por omissão das variáveis. Para que uma constante global tenha ligação externa é necessário preceder a sua declaração do especificador `extern`. Assim sendo, a utilização do especificador `static` na declaração de constantes globais é redundante.

Sejam os dois módulos representados na Figura 9.3.

Ao fundir os dois módulos acima (e admitindo que se eliminaram os erros de compilação assinalados a vermelho) o fusor gerará os seguintes erros:

```
.../usa.C:35: undefined reference to `c3'
.../usa.C:37: undefined reference to `c5'
```

No primeiro caso, o erro ocorre porque a única constante `c3` que existe tem ligação interna ao módulo `define` (por omissão as constantes globais têm ligação interna). No segundo caso, a constante `c5` foi declarada em `usa.C` mas nenhum módulo a define.

9.3.4 Ligação de classes C++ e tipos enumerados

Não é possível simplesmente declarar um tipo enumerado: é sempre necessário também defini-lo.

A distinção entre a declaração e a definição de uma classe C++ é simples e semelhante sintacticamente à distinção entre declaração e definição no caso das rotinas. A declaração no sentido estrito de uma classe C++ termina em **;** logo após o seu nome. Se em vez de **;** surgir um bloco com a declaração os membros da classe, a declaração é também uma definição. Por exemplo:

```
class Racional; // declaração.

class Racional { // definição.
  public:
    ...

  private:
    ...
};
```

define.C	usa.C
`extern int const c4;` // externa `extern int const c1 =1;`// externa `extern int const c2 =1;`// externa `int const c3 = 3;` // interna `int const c4 = 33;` // externa	*`extern int const c1;`* // externa *`extern int const c2;`* // externa *`extern int const c3;`* // externa *`extern int const c5;`* // externa *`extern int const c6;`* // externa `/*` Erro (ou aviso) de compilação! Declarada externa e depois definida interna: `*/` `static int const c6 = 12;` `int const c7 = 11;` // interna *`extern int const c7;`* // interna `int main()` `{ int i = c1 + c2 + c3` `+` // Erro de compilação, falta declaração: `c4 + c5 + c7;` `}`

Figura 9.3: Exemplo de ligação para constantes. Declarações em itálico e fontes de erros de compilação a vermelho.

A declaração em sentido estrito de uma classe C++ é útil em poucas situações. Como só a definição da classe C++ inclui a definição dos seus atributos (variáveis e constantes membro) de instância, só com a definição o compilador pode calcular o espaço de memória ocupado pelas instâncias dessa classe C++. Assim, não é possível definir variáveis de uma classe C++ sem que a classe tenha sido definida antes, embora se possam definir referências e ponteiros (a ver no Capítulo 11) para essa classe C++. A declaração em sentido estrito usa-se principalmente:

1. Quando se estão a definir duas classes C++ que se usam mutuamente, mas em que nenhuma pode ser considerada mais importante que a outra (e portanto nenhuma é embutida). Note-se que pelo menos uma das classes C++ não pode ter atributos de instância da outra classe C++. Por exemplo:

```
class B;

class A {
  public:
    A() {
    }
    A(const B&); // Declaração de B essencial!
    ...
};

class B {
  public:
    B() {
    }
    ...
  private:
    A m[10]; // Definição de A essencial!
};
```

2. Quando se está a definir uma classe C++ com outra embutida e se pretendem separar as duas definições por uma questão de clareza. Por exemplo:

```
class ListaDeInt {
  public:
    ...
    class Iterador;
    class IteradorConstante;
    ...
};

class ListaDeInt::Iterador {
    ...
};
```

```
    class ListaDeInt::IteradorConstante {
        …
    };
```

É importante também perceber que a definição de uma classe C++ não inclui a definição das suas operações, i.e., não inclui os respectivos métodos, excepto se esta definição for feita internamente à classe C++. Assim, pode-se distinguir entre a *definição de uma classe* e a *definição completa de uma classe C++*, que inclui a definição de todos os seus membros.

Podem-se definir tipo enumerados ou mesmo classes C++ dentro de um bloco de instruções, embora com algumas restrições. Por exemplo, o seguinte código

```
#include <iostream>

using namespace std;

int main()
{
    class Média {
      public:
        Média()
            : soma_dos_valores(0.0), número_de_valores(0) {
        }
        Média& operator << (double const novo_valor) {
            soma_dos_valores += novo_valor;
            ++número_de_valores;
            return *this;
        }
        operator double () const {
            assert(número_de_valores != 0);
            return soma_dos_valores / número_de_valores;
        }
      private:
          double soma_dos_valores;
          int número_de_valores;
    };

    Média m;
    m << 3 << 5 << 1;
    cout << m << endl;
}
```

define uma classe `Média`[18] localmente à função `main()`[19]. Note-se que não se pretende aqui defender este tipo de prática: são muito raros os casos em que se justifica definir uma classe localmente e este não é certamente um deles!

Uma classe ou tipo enumerado local não tem ligação. A sua utilização restringe-se ao bloco em que foi definido. Assim, a classe `Média` só pode ser usada no programa acima dentro da função `main()`.

Classes definidas dentro de outras classes, as chamadas *classes embutidas*, ou tipos enumerados definidos dentro de classes, têm a mesma ligação que a classe que as envolve.

Finalmente, classes ou tipos enumerados definidos no contexto global têm sempre ligação externa. Note-se que, ao contrário do que sucede com as funções e procedimentos, a mesma classe ou tipo enumerado pode ser definido em múltiplas unidades de tradução de um mesmo programa: o fusor só verifica as faltas ou duplicações de definições de rotinas globais e de variáveis ou constantes globais. Isto acontece porque a informação presente na definição de uma classe é necessária na sua totalidade durante a compilação, com excepção das definições de operações que não em-linha, que o compilador não precisa de conhecer. Aliás, a informação presente na definição de uma classe é crucial para o compilador poder fazer o seu papel, ao contrário do que acontece, por exemplo, com as rotinas, que para poderem ser usadas têm apenas de ser declaradas. É por esta razão que as classes não são apenas declaradas nos ficheiros de interface: são também definidas. Desse modo, através da directiva `#include`, a definição de uma classe será incluída em todas as unidades de tradução de dela necessitam.

A regra de definição única tem, por isso, uma versão especial para as classes e enumerados: uma classe ou tipo enumerado com ligação externa pode ser definido em diferentes unidades de tradução desde que essas definições sejam rigorosamente equivalentes. É justamente para garantir essa equivalência que as definições de classes e enumerados são colocadas em ficheiros de interface.

9.4 Conteúdo dos ficheiros de interface e implementação

Que ferramentas se devem colocar em cada módulo? É algo difícil responder de uma forma taxativa a esta pergunta. Mas pode-se dizer que cada módulo deve corresponder a um conjunto muito coeso de rotinas, classes, etc. É vulgar um módulo conter apenas uma classe e algumas rotinas associadas intimamente a essa classe. Por vezes há mais do que uma classe, quando essas classes estão muito interligadas, quando cada uma delas não faz sentido sem a outra. Ou então várias pequenas classes independentes mas relacionadas conceptualmente. Outras vezes vezes um módulo não definirá classe nenhuma, como poderia suceder, por exemplo, com um módulo de funções matemáticas.

---

[18] A classe `Média` define um operador de conversão implícita para `double`. Na definição destes operadores de conversão não é necessário (nem permitido) colocar o tipo de devolução, pois este é igual ao tipo indicado após a palavra-chave `operator`. É este conversor que permite inserir uma instância da classe `Média` num canal de saída sem qualquer problema: essa instância é convertida para um `double`, que se pode inserir no canal, sendo para isso calculada a média dos valores inseridos através do operador `<<`.

[19] Esta curiosa possibilidade de definir classes locais permite simular a definição de rotinas locais, que o C++ não suporta. Se está preparado para alguns pontapés na gramática, veja a Secção C.2.

Depois de escolhidas e desenvolvidas as ferramentas que constituem cada módulo físico de um programa, há que decidir para cada módulo o que deve ficar no seu ficheiro de interface (`.H`) e o que deve ficar no seu ficheiro de implementação (`.C`). A ideia geral é que o ficheiro de interface deve conter o que for estritamente necessário para que as ferramentas definidas pelo módulo possam ser usadas em qualquer outro módulo por simples inclusão do correspondente ficheiro de interface. Ou seja, tipicamente o ficheiro de interface declara o que se define no ficheiro de implementação.

Algumas das ferramentas definidas num módulo são de utilidade apenas dentro desse módulo. Essas ferramentas devem ficar inacessíveis do exterior, não dando-lhes ligação interna, conforme explicado nas secções anteriores, mas usando espaços nominativos sem nome, ou anónimos, como sugerido na Secção 9.3.2. Todas as outras ferramentas (com excepção de alguns tipos de constantes) deverão ter ligação externa.

As próximas secções apresentam algumas regras gerais acerca do que deve ou não constar em cada um dos ficheiros fonte de um módulo.

9.4.1 Relação entre interface e implementação

Para que o compilador possa verificar se o conteúdo do ficheiro de interface corresponde ao conteúdo do respectivo ficheiro de implementação, o ficheiro de implementação começa sempre por incluir o ficheiro de interface:

***módulo*.C**

```
#include "módulo.H" // ou <módulo.H>
...
```

Esta inclusão deve ser feita antes de qualquer outra inclusão necessária na ficheiro de implementação. Dessa forma ficará claro se faltar alguma inclusão no próprio ficheiro de interface: se isso acontecer, o compilador queixar-se-á quando encontrar no ficheiro de interface referências a ferramentas que desconhece.

9.4.2 Ferramentas de utilidade interna ao módulo

Definem-se no ficheiro de implementação (`.C`) dentro de um espaço nominativo sem nome:

***módulo*.C**

```
namespace {
    // Definição de ferramentas internas ao módulo, invisíveis do exterior:
    ...
}
```

9.4.3 Rotinas não-membro

As rotinas não-membro que não sejam em-linha devem ser declaradas no ficheiro de interface (`.H`) e definidas no ficheiro de implementação (`.C`):

***módulo*.H**

```
// Declaração de rotinas não-membro e que não sejam em-linha:
tipo nome(parâmetros...);
```

***módulo*.C**

```
// Definição de rotinas não-membro e que não sejam em-linha:
tipo nome(parâmetros...)
{
    ... // corpo.
}
```

As rotinas não-membro em-linha devem ser definidas apenas no ficheiro de interface (`.H`):

***módulo*.H**

```
// Definição de rotinas não-membro e em-linha:
inline tipo nome(parâmetros...) {
    ... // corpo.
}
```

Alternativamente, ou melhor, desejavelmente, as rotinas não-membro em-linha podem ser definidas num terceiro ficheiro do módulo, como descrito na Secção 9.4.15.

9.4.4 Variáveis globais

As variáveis globais não devem ser usadas.

9.4.5 Constantes globais

As constantes serão tratadas de forma diferente consoante o seu "tamanho". Se o tipo da constante for um tipo básico, uma matriz ou uma classe muito simples, então deverá ser definida com ligação interna no ficheiro de interface (`.H`). Desse modo, serão definidas tantas constantes todas iguais quantos os módulos em que esse ficheiro de interface for incluído. Compreende-se assim que a constante deva ser "pequena", para evitar programas ocupando demasiada memória, e rápida de construir, para evitar ineficiências.

***módulo*.H**

```
// Definição de constantes "pequenas" e "rápidas":

tipo const nome1; // valor por omissão.

tipo const nome2 = expressão;

tipo const nome3(argumentos);
```

Se o tipo da constante não verificar estas condições, então a constante deverá ser definida com ligação externa no ficheiro de implementação (`.C`) e declarada no ficheiro de interface (`.H`).

***módulo*.H**

```
// Declaração de constantes "grandes" ou "lentas":

extern tipo const nome1;

extern tipo const nome2;

extern tipo const nome3;
```

***módulo*.C**

```
#include "módulo.H"

...

// Definição de constantes "grandes" ou "lentas":

tipo const nome1; // valor por omissão.

tipo const nome2 = expressão;

tipo const nome3(argumentos);
```

Note-se que não é necessário o especificador `extern` no ficheiro de implementação porque este inclui sempre o respectivo ficheiro de interface, que possui a declaração das constantes como externas.

9.4.6 Tipos enumerados não-membro

Os tipos enumerados têm de ser definidos no ficheiro de interface (`.H`).

***módulo*.H**

```
// Definições de enumerados:

enum Nome { ... };
```

9.4.7 Classes C++ não-membro

As classes C++, por razões que já se viram atrás, devem ser definidas apenas no ficheiro de interface (`.H`).

***módulo*.H**

```
// Definição de classes:

class Nome {
    ... // aqui tanto quanto possível só declarações.
};
```

9.4.8 Operações (rotinas membro)

Os métodos são implementação das operações de uma classe C++. Podem ser definidos dentro da definição da classe, o que os torna automaticamente em-linha. Mas não é recomendável fazê-lo, pois isso leva a uma mistura de implementação com interface que não facilita a leitura do código. Assim, as operações, quer de classe quer de instância, são sempre declaradas dentro da classe e portanto no ficheiro de interface (`.H`). Os correspondentes métodos serão definidos fora da classe. No caso dos métodos que não são em-linha, a definição far-se-á no ficheiro de implementação (`.C`). No caso dos métodos em-linha, a definição far-se-á no ficheiro de interface.

***módulo*.H**

```
class Nome {
    ...
    // Declarações de métodos:

    tipo nome1(parâmetros...); // membro de instância.

    tipo nome2(parâmetros...); // membro de instância.

    static tipo nome3(parâmetros...); // membro de classe.

    static tipo nome4(parâmetros...); // membro de classe.
    ...
};

// Definições de métodos em-linha:

inline tipo Nome::nome2(parâmetros...) { // membro de instância.
    ... // corpo.
}
```

```
inline tipo Nome::nome4(parâmetros...) { // membro de classe.
    ... // corpo.
}
```

***módulo*.C**

```
// Definições de métodos que não são em-linha:

tipo Nome::nome1(parâmetros...) // membro de instância.
    ... // corpo.
}

tipo Nome::nome3(parâmetros...); // membro de classe.
    ... // corpo.
}
```

Alternativamente, os métodos em-linha podem ser definidos num terceiro ficheiro do módulo, como descrito na Secção 9.4.15.

9.4.9 Atributos de instância

Os atributos (variáveis e constantes) de instância são definidos dentro da própria classe C++ e inicializados pelos seus construtores, pelo que não requerem qualquer tratamento especial.

9.4.10 Variáveis membro de classe

As variáveis membro de classe declaram-se dentro da classe C++, e portanto no ficheiro de interface (`.H`), e definem-se no ficheiro de implementação (`.C`).

***módulo*.H**

```
class Nome {
    ...
    // Declarações de variáveis membro de classe:

    static tipo nome;
    ...
};
```

***módulo*.C**

```
// Definições de variáveis membro de classe (alternativas):

tipo Nome::nome; // valor por omissão.
```

```
tipo Nome::nome = expressão;

tipo Nome::nome(argumentos);
```

9.4.11 Constantes membro de classe

As constantes membro de classe tratam-se exactamente como as variáveis membro de classe: declaram-se na classe C++, e portanto no ficheiro de interface (`.H`), e definem-se no ficheiro de implementação (`.C`).

***módulo*.H**

```
class Nome {
    ...
    // Declarações de constantes membro de classe:

    static tipo const nome;
    ...
};
```

***módulo*.C**

```
// Definições de constantes membro de classe:

tipo const Nome::nome; // valor por omissão.

tipo const Nome::nome = expressão;

tipo const Nome::nome(argumentos);
```

Uma excepção a esta regra são as constantes membro de classe de tipos aritméticos inteiros. Estas podem ser definidas dentro da classe C++ e usadas, por isso, para indicar a dimensão de matrizes membro. Por exemplo:

módulo.H

```
class Nome {
    ...
    // Declarações de constantes membro de classe de tipos aritméticos inteiros:

    static int const dimensão = 100;
    ...
    int matriz[dimensão];
    ...
};
```

9.4.12 Classes C++ membro (embutidas)

As classes C++ membro de outras classes, ou seja, embutidas, podem ser definidas dentro da classe C++ que as envolve. Mas é muitas vezes preferível, embora nem sempre possível, defini-las à parte. Em qualquer dos casos a definição será feita dentro do mesmo ficheiro de interface (`.H`) da classe C++ envolvente. Aos membros de classes C++ embutidas aplicam-se os mesmos comentários que aos membros da classe C++ envolvente.

Forma recomendada:

módulo.H

```
class Nome {
    ...
    // Declarações de classes embutidas:

    class OutroNome;
    ...
};

...

class Nome::OutroNome {
    ...
};
```

Alternativa:

módulo.H

```
class Nome {
    ...
    // Definições de classes embutidas:

    class OutroNome {
        ...
    };
    ...
};
```

9.4.13 Enumerados membro

Os tipos enumerados membro têm de ser definidos dentro da classe C++ e portanto no ficheiro de interface (`.H`).

módulo.H

```
class Data {
    ...
    // Definições de enumerados embutidos:

    enum DiaDaSemana {
        primeiro,
        segunda-feira = primeiro,
        terça_feira,
        ...
        domingo,
        ultimo = domingo
    };
    ...
};
```

9.4.14 Evitando erros devido a inclusões múltiplas

A implementação da modularização física em C++, sendo bastante primitiva, põe alguns problemas. Suponha-se que se está a desenvolver um programa dividido em três módulos. Dois deles, `A` e `B`, disponibilizam ferramentas. O terceiro, `C`, contém o programa principal e faz uso de ferramentas de `A` e `B`, sendo constituído apenas por um ficheiro de implementação (`C.C`) que inclui os ficheiros de interface dos outros módulos (`A.H` e `B.H`):

`C.C`

```
#include "A.H"
#include "B.H"
...
Restante conteúdo de C.C.
```

Suponha-se ainda que o ficheiro de interface do módulo `B` necessita de algumas declarações do módulo `A`. Nesse caso:

`B.H`

```
#include "A.H"
...
Restante conteúdo de B.H.
```

Suponha-se finalmente que o ficheiro `A.H` define uma classe (o argumento seria o mesmo com um tipo enumerado, uma rotina em-linha ou uma constante):

`A.H`

```
class A {
    ...
};
```

Que acontece quando se pré-processa o ficheiro `C.C`? O resultado é a unidade de tradução `C.ii`:

`C.ii`

```
# 1 "C.C"
# 1 "A.H" 1
class A {
    ...
};
# 2 "C.C" 2
# 2 "B.H" 1
# 1 "A.H" 1
class A {
    ...
};
# 2 "B.H" 2
...
Restante conteúdo de B.H.
# 3 "C.C" 2
...
Restante conteúdo de C.C.
```

Este ficheiro contém duas definições da classe `A`, o que viola a regra da definição única, pelo que não se poderá compilar com sucesso. O problema deve-se à inclusão múltipla do conteúdo do ficheiro `A.H` na unidade de tradução `C.ii`. Como resolver o problema?

O pré-processador fornece uma forma eficaz de evitar este problema recorrendo a compilação condicional. Para isso basta envolver todo o código dentro dos ficheiros de interface da seguinte forma:

***`módulo`*`.H`**

```
#ifndef MÓDULO_H
#define MÓDULO_H

... // Conteúdo do ficheiro de interface.

#endif // MÓDULO_H
```

Desta forma, na segunda inclusão do ficheiro já estará definida a macro *`MÓDULO`*`_H`, e portanto o código não será colocado segunda vez na unidade de tradução. A macro deve ter um nome único entre todos os módulos, de modo a que este mecanismo não entre em conflito com o mesmo mecanismo noutros módulos também usados. Não é fácil garantir que o nome seja único, o que revela quão primitivo é o modelo de modularização física do C++. Uma das formas de tentar garantir unicidade no nome das macros é anexar-lhes os nomes dos pacotes a que o módulo pertence, como se verá mais à frente.

9.4.15 Ficheiro auxiliar de implementação

Os ficheiros de interface, como se viu, contêm mais informação do que aquela que, em rigor, corresponde à interface das ferramentas disponibilizadas pelo módulo correspondente. Em particular as rotinas e métodos em-linha são totalmente definidos no ficheiro de interface. Para o programador consumidor de um módulo, a definição exacta destas rotinas é irrelevante. A sua presença no ficheiro de interface acaba até por ser um factor de distracção. Uma solução comum para este problema passa por utilizar um terceiro tipo de ficheiro em cada módulo: o ficheiro auxiliar de implementação. Sugere-se, pois, que os módulos sejam constituídos por (no máximo) três ficheiros fonte:

1. Ficheiro de interface (`.H`) – Com o conteúdo sugerido nas secções anteriores, mas sem definições de rotinas e métodos em-linha e contendo as declarações de todas as rotinas não-membro, independentemente de serem ou não em-linha. Este ficheiro deve terminar com a inclusão do respectivo ficheiro auxiliar de implementação (terminado em `_impl.H`).

2. Ficheiro auxiliar de implementação (`_impl.H`) – Com a definição todas as rotinas e métodos em-linha.

3. Ficheiro de implementação (`.C`) - com o conteúdo sugerido anteriormente.

Desta forma a complexidade do ficheiro de interface reduz-se consideravelmente, o que facilita a sua leitura.

A mesma solução pode ser usada para resolver o problema da definição de rotinas e métodos modelo quando se usa compiladores que não suportam a palavra-chave `export`. Ver Capítulo 13 para mais pormenores.

9.5 Construção automática do ficheiro executável

A construção do ficheiro executável de um programa constituído por vários módulos exige uma sequência de comandos (pelo menos se se quiser tirar partido da compilação separada, bem entendido). Em ambientes Unix (como o Linux) é possível automatizar a construção usando o *construtor*, que é um programa chamado `make`, e, se necessário, um ficheiro de construção.

O construtor tem um conjunto de regras implícitas que lhe permitem construir automaticamente pequenos programas. Infelizmente, está especializado para a linguagem C, pelo que são necessários alguns passos de configuração para usar a linguagem C++. Em particular, é necessário indicar claramente quais são os programas e respectivas opções que devem ser usados para pré-processar e compilar e para fundir. Para isso é necessário atribuir valores à variáveis de ambiente `CXX`, que guarda o nome do programa usado para pré-processar e compilar, `CC`, que guarda o nome do programa usado para fundir, `CXXFLAGS`, que guarda as opções de compilação desejadas, e `LDLIBS`, que indica as bibliotecas a usar nas fusões. Se se estiver a usar o interpretador de comandos `/bin/tcsh`, tal pode ser conseguido através dos comandos:

```
setenv CC c++
setenv CXX c++
setenv CXXFLAGS "-g -Wall -ansi -pedantic"
setenv LDLIBS ""
```

Estes comandos também podem ser colocados no ficheiro de configuração do interpretador de comandos: `~/.tcshrc`[20].

Depois das configurações anteriores, é fácil construir pequenos programas constituídos apenas por um ficheiro de implementação:

olá.C

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Olá mundo!" << endl;
}
```

Para construir o ficheiro executável basta dar o comando:

```
make olá
```

Em casos mais complicados, quando existem vários módulos num programa, é necessário criar um ficheiro de construção (*makefile*), de nome `Makefile`. Este ficheiro, cujo conteúdo pode ser muito complicado, consiste nas suas versões mais simples num conjunto de regras de dependência. Estas dependências têm o seguinte formato:

```
alvo: dependência dependência...
```

[20]Essas alterações só têm efeito depois de se voltar a entrar no interpretador ou depois de se dar o comando `source ~/.tcshrc` na consola em causa.

Estas regras servem para indicar ao construtor que dependências existem entre os ficheiros do programa, dependências essas que o construtor não sabe adivinhar.

A melhor forma de perceber os ficheiros de construção é estudar um exemplo. Suponha-se que um projecto deve construir dois ficheiros executáveis, um para calcular médias de alunos e outro para saber a nota de um dado aluno, e que esse projecto é constituído por três módulos, respectivamente `aluno`, `média` e `procura`. Para simplificar o exemplo, não se usará o ficheiro auxiliar de implementação sugerido na secção anterior, pelo que cada módulo terá no máximo dois ficheiros fonte.

O primeiro módulo, `aluno`, define ferramentas para lidar com alunos e respectivas notas, e possui dois ficheiros fonte: `aluno.C` e `aluno.H`. Os outros módulos, `média` e `procura`, definem os programas para cálculo de médias e pesquisa de alunos, e portanto consistem apenas num ficheiro de implementação cada um (`média.C` e `procura.C`). Os ficheiros fonte são[21]:

aluno.H

```
#ifndef ALUNO_H
#define ALUNO_H

#include <string>
#include <iostream>

bool éPositiva(int nota);

const int nota_mínima_aprovação = 10;
const int nota_máxima = 20;

class Aluno {
  public:
    Aluno(std::string const& nome,
          int número, int nota = 20);
    std::string const& nome() const;
    int número() const;
    int nota() const;

  private:
    std::string nome_;
    int número_;
    int nota_;
};
```

[21]Optou-se por não documentar o código para não o tornar exageradamente longo. Optou-se também por eliminar a verificação de erros do utilizador, pela mesma razão. Finalmente, distinguiu-se algo artificialmente entre métodos e rotinas em linha e não-em linha. A razão foi garantir que o ficheiro de implementação `aluno.C` não ficasse vazio.

```
inline Aluno::Aluno(std::string const& nome,
                    int const número, int const nota)
    : nome_(nome), número_(número), nota_(nota) {
    assert(0 <= nota and nota <= nota_máxima);
}

std::ostream& operator << (std::ostream& saída,
                           Aluno const& aluno);

#endif // ALUNO_H
```

aluno.C

```
#include "aluno.H"

using namespace std;

bool éPositiva(int const nota)
{
    return nota >= nota_mínima_aprovação;
}

string const& Aluno::nome() const
{
    return nome_;
}

int Aluno::número() const
{
    return número_;
}

int Aluno::nota() const
{
    return nota_;
}

ostream& operator << (ostream& saída, Aluno const& aluno)
{
    return saída << aluno.nome() << ' ' << aluno.número()
                 << ' ' << aluno.nota();
}
```

média.C

```
#include <iostream>
#include <string>
```

```
#include <vector>

using namespace std;

#include "aluno.H"

int main()
{
    cout << "Introduza número de alunos: ";
    int número_de_alunos;
    cin >> número_de_alunos;

    cout << "Introduza os alunos (nome número nota):"
         << endl;
    vector<Aluno> alunos;
    for(int i = 0; i != número_de_alunos; ++i) {
        string nome;
        int número;
        int nota;
        cin >> nome >> número >> nota;
        alunos.push_back(Aluno(nome, número, nota));
    }

    double soma = 0.0;
    for(int i = 0; i != número_de_alunos; ++i)
        soma += alunos[i].nota();

   cout << "A média é: " << soma / número_de_alunos
        << " valores." << endl;
}
```

procura.C

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

#include "aluno.H"

int main()
{
    vector<Aluno> alunos;
    alunos.push_back(Aluno("Zé", 1234, 15));
    alunos.push_back(Aluno("Zacarias", 666, 20));
```

```
        alunos.push_back(Aluno("Marta", 5465, 18));
        alunos.push_back(Aluno("Maria", 1111, 14));

        cout << "Pauta:" << endl;
        for(vector<Aluno>::size_type i = 0;
            i != alunos.size(); ++i)
            cout << alunos[i] << endl;

        cout << "De que aluno deseja saber a nota? ";
        string nome;
        cin >> nome;
        for(vector<Aluno>::size_type i = 0;
            i != alunos.size(); ++i)
            if(alunos[i].nome() == nome) {
                cout << "O aluno " << nome << " teve "
                     << alunos[i].nota() << endl;
                if(not éPositiva(alunos[i].nota()))
                    cout << "Este aluno reprovou." << endl;
            }
    }
```

Observando estes ficheiros fonte, é fácil verificar que:

1. Sempre que um dos ficheiros fonte do módulo `aluno` for alterado, o respectivo ficheiro objecto deve ser produzido de novo, pré-processando e compilando o respectivo ficheiro de implementação. Como o construtor sabe que os ficheiros objecto dependem dos ficheiros de implementação respectivos, só é necessário indicar explicitamente que o ficheiro objecto depende do ficheiro de interface. Ou seja, deve-se colocar no ficheiro de construção a seguinte linha:

   ```
   aluno.o: aluno.H
   ```

2. O ficheiro de implementação do módulo `média` inclui o ficheiro de interface `aluno.H`. Assim, é necessário indicar explicitamente que o ficheiro objecto `média.o` depende do ficheiro de interface `aluno.H`:

   ```
   média.o: aluno.H
   ```

3. O mesmo se passa para o módulo procura:

   ```
   procura.o: aluno.H
   ```

4. Finalmente, é necessário indicar que os ficheiros executáveis são obtidos, e portanto dependem, de determinados ficheiros objecto:

   ```
   média: média.o aluno.o
   procura: procura.o aluno.o
   ```

Assim, o ficheiro de construção ficará:

Makefile

```
média: média.o aluno.o
procura: procura.o aluno.o

aluno.o: aluno.H
média.o: aluno.H
procura.o: aluno.H
```

As três últimas regras podem ser geradas automaticamente pelo compilador se se usar a opção `-MM`. Sugere-se que o ficheiro de construção seja inicializado da seguinte forma:

```
c++ -MM aluno.C média.C procura.C > Makefile
```

O resultado será o ficheiro:

Makefile

```
aluno.o: aluno.C aluno.H
média.o: média.C aluno.H
procura.o: procura.C aluno.H
```

onde faltam as regras de construção dos ficheiros executáveis, que têm de ser acrescentadas à mão.

As regras obtidas automaticamente pelo comando `c++ -MM` indicam explicitamente que os ficheiros objecto dependem dos ficheiros de implementação, o que é redundante, pois o construtor sabe dessa dependência.

Com o ficheiro de construção sugerido, construir o programa `média`, por exemplo, é fácil. Basta dar o comando

```
make média
```

que o construtor `make` se encarregará de pré-processar, compilar e fundir apenas aquilo que for necessário para construir o ficheiro executável `média`. O construtor tenta sempre construir os alvos que lhe forem passados como argumento. Se se invocar o construtor sem quaisquer argumentos, então ele tentará construir o primeiro alvo indicado no ficheiro de construção. Assim, pode-se acrescentar uma regra inicial ao ficheiro de construção de modo a que o construtor tente construir os dois executáveis quando não lhe forem passados argumentos:

Makefile

```
    all: média procura

    média: média.o aluno.o
    procura: procura.o aluno.o

    aluno.o: aluno.H
    média.o: aluno.H
    procura.o: aluno.H
```

Neste caso para construir os dois executáveis basta dar o comando:

```
make
```

9.6 Modularização em pacotes

Já se viu atrás que o nível de modularização acima da modularização física é o nível de pacote. Um pacote é constituído normalmente pelas ferramentas de vários módulos físicos. Em C++ não existe suporte directo para a noção de pacote. A aproximação utilizada consiste na colocação dos vários módulos físicos de um pacote num *espaço nominativo* comum ao pacote. A noção de espaço nominativo é descrita brevemente nas próximas secções.

9.6.1 Colisão de nomes

Um problema comum em grandes projectos é a existência de entidades (classes C++, instâncias, rotinas, etc.) com o mesmo nome, embora desenvolvidos por pessoas, equipas ou empresas diferentes. Quando isto acontece diz-se que ocorreu uma colisão de nomes.

Por exemplo, suponha-se que se está a desenvolver uma aplicação de gestão. Pode-se decidir comprar duas bibliotecas de ferramentas, uma com ferramentas de contabilidade, outra de logística. Esta é uma opção acertada a maior parte das vezes: é provavelmente mais rápido e barato comprar as bibliotecas a terceiros do que desenvolver as respectivas ferramentas de raiz.

Suponha-se que os fornecedores dessas bibliotecas são duas empresas diferentes e que estas foram fornecidas no formato de ficheiros de interface e ficheiros de arquivo contendo os ficheiro objecto de cada biblioteca.

Suponha-se ainda que ambas as bibliotecas definem um procedimento com a mesma assinatura[22]:

```
void consolida();
```

Simplificando, os ficheiros fornecidos por cada empresa são:

[22] Fica a cargo do leitor imaginar o que cada um desses dois procedimentos faz.

Biblioteca de logística

`liblogística.a` Ficheiro de arquivo da biblioteca. Contém (entre outros) o ficheiro objecto `produtos.o`, que define (entre outros) o procedimento `consolida()`.

`produtos.H`

```
#ifndef PRODUTOS_H
#define PRODUTOS_H

...
void consolida();
...

#endif // PRODUTOS_H
```

outros ficheiros de interface

Biblioteca de contabilidade

`libcontabilidade.a` Ficheiro de arquivo da biblioteca. Contém (entre outros) o ficheiro objecto `balanço.o`, que define (entre outros) o procedimento `consolida()`.

`balanço.H`

```
#ifndef BALANÇO_H
#define BALANÇO_H

...
void consolida();
...

#endif // BALANÇO_H
```

outros ficheiros de interface

Suponha-se que a aplicação em desenvolvimento contém um módulo `usa.C` com apenas a função `main()` e contendo uma invocação do procedimento `consolida()` da biblioteca de contabilidade:

`usa.C`

```
#include <balanço.H>

int main()
{
    consolida();
}
```

A construção do programa consiste simplesmente em pré-processar e compilar os ficheiros de implementação (entre os quais `usa.C`) e em seguida fundir os ficheiros objecto resultantes com os ficheiros de arquivo das duas bibliotecas (admite-se que algum outro módulo faz uso de ferramentas da biblioteca de logística):

```
c++ -Wall -ansi -pedantic -g -c *.C
c++ -o usa *.o -llogística -lcontabilidade
```

Que sucede? O inesperado: o executável é gerado sem problemas e quando se executa verifica-se que o procedimento `consolida()` executado é o da biblioteca de logística!

Porquê? Porque o fusor não se importa com duplicações nos ficheiros de arquivo. Os ficheiros de arquivo são pesquisados pela ordem pela qual são indicados no comando de fusão e a pesquisa pára quando a ferramenta procurada é encontrada. Como se colocou `-llogística` antes de `-lcontabilidade`, o procedimento encontrado é o da biblioteca de logística...

Mas há um problema mais grave. E se se quiser invocar ambos os procedimentos na função `main()`? É impossível distingui-los.

Como resolver este problema de colisão de nomes? Há várias possibilidades, todas más:

1. Pedir a um dos fornecedores para alterar o nome do procedimento. É uma má solução. Sobretudo porque provavelmente a empresa fornecedora tem outros clientes e não se pode dar ao luxo de ter uma versão diferente da biblioteca para cada cliente. Provavelmente protegeu-se de semelhantes problemas no contrato de fornecimento e recusará o pedido...

2. Esquecer, para cada nome em colisão, a versão da biblioteca contabilidade, e desenvolver essas ferramentas atribuindo-lhes outros nomes. É reinventar a roda, e isso custa tempo e dinheiro.

9.6.2 Espaços nominativos

A solução é escolher fornecedores que garantam um máximo de protecção a priori contra a possibilidade de colisões de nomes. Cada uma das empresas, se fosse competente, deveria ter colocado o seu código dentro de um espaço nominativo apropriado. Neste caso as escolhas acertadas seriam os espaços nominativos `Logística` e `Contabilidade` (os nomes dos espaços nominativos começam tipicamente por maiúscula, como os das classes). Nesse caso as bibliotecas fornecidas consistiriam em:

Biblioteca de logística

`liblogística.a` Ficheiro de arquivo da biblioteca. Contém (entre outros) o ficheiro objecto `produtos.o`, que define (entre outros) o procedimento `Logística::consolida()`.

`produtos.H`

```
#ifndef PRODUTOS_H
#define PRODUTOS_H

namespace Logística {
    ...
    void consolida();
    ...
}

#endif // PRODUTOS_H
```

outros ficheiros de interface

Biblioteca de contabilidade

`libcontabilidade.a` Ficheiro de arquivo da biblioteca. Contém (entre outros) o ficheiro objecto `balanço.o`, que define (entre outros) o procedimento `Contabilidade::consolida()`.

`balanço.H`

```
#ifndef BALANÇO_H
#define BALANÇO_H

namespace Contabilidade {
    ...
    void consolida();
    ...
}

#endif // BALANÇO_H
```

outros ficheiros de interface

O efeito prático da construção

```
namespace Nome {
    ...
}
```

é o de fazer com que todos os nomes declarados dentro das chavetas passem a ter o prefixo `Nome::`. Assim, com estas novas versões das bibliotecas, passaram a existir dois procedimentos diferentes com diferentes nomes: `Logística::consolida()` e `Contabilidade::consolida()`.

Agora a função `main()` terá de indicar o nome completo do procedimento:

`usa.C`

```
#include <balanço.H>

int main()
{
    Contabilidade::consolida();
}
```

9.6.3 Directivas de utilização

Para evitar ter de preceder o nome das ferramentas da biblioteca de contabilidade de `Contabilidade::`, pode-se usar uma *directiva de utilização*:

usa.C

```
#include <balanço.H>

using namespace Contabilidade;

int main()
{
    consolida();
}
```

Uma directiva de utilização injecta todos os nomes do espaço nominativo indicado no espaço nominativo corrente. Note-se que, quando uma entidade não está explicitamente dentro de algum espaço nominativo, então está dentro do chamado *espaço nominativo global*, sendo o seu prefixo simplesmente `::`, que pode ser geralmente omitido. Assim, no código acima a directiva de utilização injecta o nome `consolida()` no espaço nominativo global, pelo que este pode ser usado directamente.

Pode-se simultaneamente usar o procedimento `consolida()` da biblioteca de logística, bastando para isso indicar o seu nome completo:

usa.C

```
#include <balanço.H>
#include <produtos.H>

using namespace Contabilidade;

int main()
{
    consolida();
    Logística::consolida();
}
```

É mesmo possível injectar todos os nomes de ambos os espaços nominativos no espaço nominativo global. Nesse caso os nomes duplicados funcionam como se estivessem sobrecarregados, se forem nomes de rotinas. I.e., se existirem duas rotinas com o mesmo nome, então terão de ter assinaturas (número e tipo dos parâmetros) diferentes. Se tiverem a mesma assinatura, então uma tentativa de usar o nome sem o prefixo apropriado gera um erro de compilação, uma vez que o compilador não sabe que versão invocar. Por exemplo:

usa.C

```
#include <balanço.H>
#include <produtos.H>

using namespace Contabilidade;
using namespace Logística;

int main()
{
    consolida(); // ambíguo!
    Logística::consolida();
}
```

Um possível solução é

usa.C

```
#include <balanço.H>
#include <produtos.H>

using namespace Contabilidade;
using namespace Logística;

int main()
{
    Contabilidade::consolida();
    Logística::consolida();
}
```

onde se usaram nomes completos para discriminar entre as entidades com nomes e assinaturas iguais em ambos os espaços nominativos.

9.6.4 Declarações de utilização

A utilização de directivas de utilização tem a consequência nefasta de injectar no espaço nominativo corrente todos os nomes declarados no espaço nominativo indicado. Por isso, deve-se usar directivas de utilização com conta peso e medida nos ficheiros de implementação e *nunca nos ficheiros de interface*.

Uma alternativa menos drástica que as directivas são as *declarações de utilização*. Se se pretender injectar no espaço nominativo corrente apenas um nome, pode-se usar uma declaração de utilização:

usa.C

```
#include <balanço.H>
#include <produtos.H>

int main()
{
    using Contabilidade::consolida;

    consolida();
    Logística::consolida();
}
```

Nas declarações de utilização apenas se indica o nome da entidade: no caso de uma rotina, por exemplo, é um erro indicar o cabeçalho completo.

9.6.5 Espaços nominativos e modularização física

Podem-se colocar vários módulos num mesmo espaço nominativo[23]. É típico dividir uma biblioteca, por exemplo, num conjunto de pacotes com funções diversas, atribuindo a cada pacote um espaço nominativo, e consistindo cada pacote num conjunto de módulos físicos.

No exemplo apresentado o problema da colisão de nomes não foi totalmente afastado. Foi simplesmente minimizado. É que as duas bibliotecas poderiam, com azar, usar o mesmo nome para um dos seus espaços nominativos. Para reduzir ainda mais essa possibilidade, é conveniente que todos os pacotes de uma empresa pertençam a um super-pacote com o nome da empresa. Suponha-se que a biblioteca de contabilidade foi fornecido pela empresa Verão *Software*, Lda. e que a biblioteca de logística foi fornecido pela empresa Inverno *Software*, Lda.

Outra fonte de possíveis colisões são os nomes das macros usadas para evitar os erros associados a inclusões múltiplas. Como até agora se convencionou que estas macros seriam baseadas apenas no nome do respectivo módulo, podem surgir problemas graves se existirem dois módulos com o mesmo nome nas duas bibliotecas. O problema pode ser resolvido convencionando que as macros incluem também os nomes dos pacotes a que os módulos pertencem.

Se ambas as empresas levando em conta os problemas acima, os ficheiros fornecidos poderiam ser:

Biblioteca de logística

[23]E vice-versa, mas não é tão útil...

`liblogística.a` Ficheiro de arquivo da biblioteca. Contém (entre outros) o ficheiro objecto `produtos.o`, que define (entre outros) o procedimento `InvernoSoftware::Logística::consolida()`.

`produtos.H`

```
#ifndef INVERNOSOFTWARE_LOGISTICA_PRODUTOS_H
#define INVERNOSOFTWARE_LOGISTICA_PRODUTOS_H

namespace InvernoSoftware {
    namespace Logística {
        ...
        void consolida();
        ...
    }
}

#endif // INVERNOSOFTWARE_LOGISTICA_PRODUTOS_H
```

outros ficheiros de interface

Biblioteca de contabilidade

`libcontabilidade.a` Ficheiro de arquivo da biblioteca. Contém (entre outros) o ficheiro objecto `balanço.o`, que define (entre outros) o procedimento `VerãoSoftware::Contabilidade::consolida()`.

`balanço.H`

```
#ifndef VERAOSOFTWARE_CONTABILIDADE_BALANÇO_H
#define VERAOSOFTWARE_CONTABILIDADE_BALANÇO_H

namespace VerãoSoftware {
    namespace Contabilidade {
        ...
        void consolida();
        ...
    }
}

#endif // VERAOSOFTWARE_CONTABILIDADE_BALANÇO_H
```

outros ficheiros de interface

Os espaços nominativos podem, portanto, ser organizados hierarquicamente, pelo que um pacote pode conter outros pacotes para além das suas ferramentas.

Neste caso a utilização poderia ter o seguinte aspecto:

`usa.C`

```
#include <balanço.H>
```

```
#include <produtos.H>

int main()
{
    using VerãoSoftware::Contabilidade::consolida;

    consolida();
    InvernoSoftware::Logística::consolida();
}
```

Apresentam-se abaixo algumas regras gerais sobre o conteúdo dos ficheiros fonte de cada módulo quando se usam espaços nominativos.

9.6.6 Ficheiros de interface

Os ficheiros de interface devem ter o conteúdo descrito anteriormente (ver Secção 9.4), excepto que todas as declarações e definições serão envolvidas nos espaços nominativos necessários, como se viu nos exemplos acima. Não se devem fazer inclusões dentro das chavetas de um espaço nominativo, nem mesmo do ficheiro auxiliar de implementação! Por outro lado, a macro usada para evitar erros associados a inclusões múltiplas deve reflectir não apenas o nome do módulo, mas também o nome dos pacotes a que o módulo pertença.

9.6.7 Ficheiros de implementação e ficheiros auxiliares de implementação

Os ficheiros de implementação e auxiliar de implementação, que contêm definições de entidades apenas declaradas no ficheiro de interface, não devem envolver todas as definições nos espaços nominativos a que pertencem. Para evitar erros, os nomes usados na definição devem incluir como prefixo os espaços nominativos a que pertencem, para que o compilador possa garantir que esses nomes foram previamente declarados dentro desses espaços nominativos no ficheiro de interface do módulo.

Por exemplo, os ficheiros de implementação dos módulos `produtos` (da empresa Inverno Software) e `balanço` (da empresa Verão Software), poderiam ter o seguinte aspecto:

`produtos.C`

```
#include "produtos.H"

... // outros #include.

...
void InvernoSoftware::Logística::consolida()
{
    ...
}
...
```

balanço.C

```
#include "balanço.H"

... // outros #include.

...
void VerãoSoftware::Contabilidade::consolida()
{
    ...
}
```

Note-se que a solução anterior só funciona se as empresas pertencerem ao mesmo espaço nacional: nesse caso o registo de nomes de empresas garante que não há duplicação de nomes. A nível internacional o problema complica-se. Nesse caso pode ser uma ideia que as empresas acrescentem um espaço nominativo exterior a todos os outros e que identifica o país de origem usando o código ISO. Aplicando ao exemplo já apresentado, o ficheiro de interface `produtos.H` ficaria

```
#ifndef PT_INVERNOSOFTWARE_LOGISTICA_PRODUTOS_H
#define PT_INVERNOSOFTWARE_LOGISTICA_PRODUTOS_H

namespace pt {
    namespace InvernoSoftware {
        namespace Logística {
            ...
            void consolida();
            ...
        }
    }
}

#endif // PT_INVERNOSOFTWARE_LOGISTICA_PRODUTOS_H
```

9.6.8 Pacotes e espaços nominativos

Um pacote é uma unidade de modularização. No entanto, a sua implementação à custa de espaços nominativos tem um problema: não separa claramente interface de implementação. A única separação que existe é a que decorre do facto de os pacotes serem constituídos por módulos físicos e estes por classes, funções e procedimentos, que são níveis de modularização em que esta separação existe. Sendo os pacotes constituídos por módulos físicos, seria útil poder classificar os módulos físicos em módulos físicos públicos (que fazem parte da interface do pacote) e módulos físicos privados (que fazem parte da implementação do pacote). Isso pode-se conseguir de uma forma indirecta colocando os módulos físicos privados num espaço nominativo especial, com um nome pouco evidente, e não disponibilizando para os consumidores do pacote senão os ficheiros de interface dos módulos físicos públicos.

Esta solução é de fácil utilização quando o pacote é fornecido integrado numa biblioteca, visto que as bibliotecas são tipicamente fornecidas como um conjunto de ficheiros de interface (e só se fornecem os que dizem respeito a módulos físicos públicos) e um ficheiro de arquivo (de que fazem parte os ficheiros objecto de todos os módulos, públicos ou privados, mas nos quais as ferramentas dos módulos privados são de mais difícil acesso, pois encontram-se inseridas num espaço nominativo de nome desconhecido para o consumidor).

9.7 Exemplo final

Apresenta-se aqui o exemplo da Secção 9.5, mas melhorado de modo a usar ficheiros auxiliares de implementação e espaços nominativos e a fazer um uso realista de rotinas e métodos em-linha.

O exemplo contém um pacote de ferramentas de gestão de uma escola de nome `Escola`. Este pacote, implementado usando um espaço nominativo, contém dois módulos físicos descritos abaixo: `nota` e `aluno`. Módulos:

`nota` Um módulo físico criado para conter ferramentas relacionadas com notas. Pertence ao pacote `Escola`. Colocaram-se todas as ferramentas dentro de um espaço nominativo `Nota` para evitar nomes demasiado complexos para as ferramentas. Pode-se considerar, portanto, que o pacote `Escola` é constituído por um módulo físico `aluno` e por um pacote `Nota`, que por sua vez é constituído por um único módulo físico `nota`. Os ficheiros fonte do módulo `nota` são (não há ficheiro de implementação, pois todas as rotinas são em-linha):

`nota.H`

```
#ifndef ESCOLA_NOTA_NOTA_H
#define ESCOLA_NOTA_NOTA_H

/// Pacote que contém todas as ferramentas de gestão da escola.
namespace Escola {

    /// Pacote que contém ferramentas relacionadas com notas.
    namespace Nota {

        /** Devolve verdadeiro se a nota for considerada positiva.
            @pre 0 <= nota e nota <= máxima.
            @post éPositiva =
                      (mínima_de_aprovação <= nota). */
        bool éPositiva(int nota);

        /// A nota mínima para obter aprovação.
        int const mínima_de_aprovação = 10;

        /// A nota máxima.
```

```
        int const máxima = 20;
    }
}

#include "nota_impl.H"

#endif // ESCOLA_NOTA_NOTA_H
```

nota_impl.H

```
inline bool Escola::Nota::éPositiva(int const nota) {
    return mínima_de_aprovação <= nota;
}
```

aluno Um módulo físico criado para conter ferramentas relacionadas com alunos. Pertence ao pacote `Escola`. Os ficheiros fonte do módulo `aluno` são (não há ficheiro de implementação, pois todas as funções e procedimentos são em-linha):

aluno.H

```
#ifndef ESCOLA_ALUNO_H
#define ESCOLA_ALUNO_H

#include <string>
#include <iostream>

#include "nota.H"

namespace Escola {

    /// Representa o conceito de aluno de uma escola.
    class Aluno {
      public:
        /// Constrói um aluno dado o nome, o número e, opcionalmente, a nota.
        Aluno(string const& nome,
              int número, int nota = Nota::maxima);

        /// Devolve o nome do aluno.
        string const& nome() const;

        /// Devolve o número do aluno.
        int número() const;

        /// Devolve a nota do aluno.
        int nota() const;

      private:
        string nome_;
```

```
            int número_;
            int nota_;
        };

        /// Operador de inserção de um aluno num canal.
        ostream& operator << (ostream& saída,
                              Aluno const& aluno);
    }

    #include "aluno_impl.H"

    #endif // ESCOLA_ALUNO_H
```

aluno_impl.H

```
    inline Escola::Aluno::Aluno(string const& nome,
                                int const número,
                                int const nota)
        : nome_(nome), número_(número), nota_(nota) {
        assert(0 <= nota and nota <= Nota::máxima);
    }

    inline string const& Escola::Aluno::nome() const {
        return nome_;
    }

    inline int Escola::Aluno::número() const {
        return número_;
    }

    inline int Escola::Aluno::nota() const {
        return nota_;
    }

    inline ostream& Escola::operator << (ostream& saída,
                                         Aluno const& aluno) {
        return saída << aluno.nome() << ' ' << aluno.número()
                     << ' ' << aluno.nota();
    }
```

`média` e `procura` São os módulos físicos com as funções `main()` dos programas a construir. Não têm ficheiro de interface nem de implementação auxiliar:

média.C

```
    #include <iostream>
    #include <string>
    #include <vector>
```

```
using namespace std;

#include "aluno.H"

using namespace Escola;
```

/** Programa que lê informação sobre um conjunto de alunos do teclado e depois mostra a média das suas notas. */

```
int main()
{
    cout << "Introduza número de alunos: ";
    int número_de_alunos;
    cin >> número_de_alunos;

    cout << "Introduza os alunos "
         << "(nome número nota):" << endl;
    vector<Aluno> alunos;
    for(int i = 0; i != número_de_alunos; ++i) {
        string nome;
        int número;
        int nota;
        cin >> nome >> número >> nota;

        alunos.push_back(Aluno(nome, número, nota));
    }

    double soma = 0.0;
    for(int i = 0; i != número_de_alunos; ++i)
        soma += alunos[i].nota();
    cout << "A média é: " << soma / número_de_alunos
         << " valores." << endl;
}
```

procura.C

```
#include <iostream>
#include <string>
#include <vector>

using namespace std;

#include "nota.H"
#include "aluno.H"

using namespace Escola;
```

```
/** Programa que mostra informação acerca de um aluno à escolha do utiliza-
dor. */
int main()
{
    vector<Aluno> alunos;
    alunos.push_back(Aluno("Zé", 1234, 15));
    alunos.push_back(Aluno("Zacarias", 666, 20));
    alunos.push_back(Aluno("Marta", 5465, 18));
    alunos.push_back(Aluno("Maria", 1111, 14));

    cout << "Pauta:" << endl;
    for(vector<Aluno>::size_type i = 0;
        i != alunos.size(); ++i)
        cout << alunos[i] << endl;

    cout << "De que aluno deseja saber a nota? ";
    string nome;
    cin >> nome;
    for(vector<Aluno>::size_type i = 0;
        i != alunos.size(); ++i)
        if(alunos[i].nome() == nome) {
            cout << "O aluno " << nome << " teve "
                 << alunos[i].nota() << endl;
            if(not Nota::éPositiva(alunos[i].nota()))
                cout << "Este aluno reprovou." << endl;
        }
}
```

Makefile

```
all: média procura

procura.o: procura.C nota.H nota_impl.H aluno.H aluno_impl.H
média.o: média.C nota.H nota_impl.H aluno.H aluno_impl.H
```

Capítulo 10

Listas e iteradores

!!Falar de promiscuidade e intimidade!

Antes de se avançar para as noções de ponteiro e variáveis dinâmicas, estudadas no próximo capítulo, far-se-á uma digressão pelos conceitos de lista e iterador. Estes dois conceitos permitem uma considerável diversidade de implementações, todas com a mesma interface mas diferentes eficiências. Neste capítulo ir-se-á tão longe quanto possível no sentido de uma implementação apropriada dos conceitos de lista e iterador. No próximo capítulo continuar-se-á este exercício, mas já usando ponteiros e variáveis dinâmicas. Assim, a implementação eficiente de listas e iteradores surgirá como justificação parcial para essas noções.

O estudo das listas e dos iteradores associados tem algumas vantagens adicionais, tais como abordar ao de leve questões relacionadas com as estruturas de dados e a eficiência de algoritmos, bem como habituar o leitor à compreensão de classes com um grau já apreciável de complexidade.

10.1 Listas

Todos nós conhecemos o significado da palavra "lista", pelo menos intuitivamente. As listas ocorrem na nossa vida prática em muitas circunstâncias: lista de compras, de afazeres, de pessoas... O Novo Aurélio [4] define lista da seguinte forma:

> **lista.** [Do fr. *liste* < it. *lista* < germ. **lista* (al. mod. *Leiste*).] ***S. f.* 1.** Relação de nomes de pessoas ou de coisas: relação, rol, listagem. [...]

Escusado será dizer que as definições de "relação", "rol" e "listagem" do Novo Aurélio remetem de volta à noção de lista[1]! Tentar-se-á aqui dar uma definição do conceito de lista um pouco menos genérica.

Suponha-se uma lista de tarefas a realizar por um determinado aluno mais ou menos aplicado e com alguma ideia das prioridades:

[1] É uma fatalidade que tal aconteça num dicionário. Mas esperamos sempre que os ciclos de definições não sejam tão curtos...

- Comprar CD dos *Daft Punk*.
- Combinar ida ao concerto *Come Together*.
- Fazer trabalho de Sistemas Operativos.
- Estudar Programação Orientada para Objectos.

Uma lista parece portanto corresponder simplesmente a um *conjunto* de itens. No entanto, haverá alguma razão para, tal como acontece nos conjuntos, não existirem repetições de itens? Na realidade não. Note-se na seguinte lista de afazeres, bastante mais razoável que a anterior:

- Estudar Programação Orientada para Objectos.
- Comprar CD dos *Daft Punk*.
- Estudar Programação Orientada para Objectos.
- Combinar ida ao concerto *Come Together*.
- Estudar Programação Orientada para Objectos.
- Fazer trabalho de Sistemas Operativos.
- Estudar Programação Orientada para Objectos.

Assim, melhor seria dizer que uma lista é uma *colecção* de itens. Porém, a noção de colecção não é suficiente, pois ignora um factor fundamental: os itens de uma lista têm uma determinada ordem, que não tem forçosamente a ver com os valores dos itens (como é evidente no último exemplo)[2].

Assim, como definição tentativa, pode-se dizer que uma lista é uma sequência de itens com ordem arbitrária mas relevante.

Naturalmente que uma definição apropriada de lista terá de incluir as operações que se podem realizar com listas. Aliás, a noção de lista (e de iterador) podem ser totalmente definidas à custa das respectivas operações.

10.1.1 Operações com listas

A operação fundamental a realizar com qualquer tipo abstracto de dados é naturalmente a construção de instâncias desse tipo. Depois de construída uma lista, deve ser possível pôr novos itens na lista e tirar itens existentes da lista, aceder aos itens, e obter informação geral acerca do estado da lista. Grosso modo, as operações a realizar são:

- Construir nova lista.

[2] A ordem é normalmente determinada pela forma como os seus itens foram colocados na lista.

- Pôr novo item algures na lista. A lista não pode estar cheia, se tiver limite pré-estabelecido.
- Tirar algum item existente da lista. A lista não pode estar vazia.
- Aceder a um qualquer item da lista. A lista não pode estar vazia.
- Saber comprimento da lista.
- Saber se a lista está vazia.
- Saber se está cheia, se tiver limite pré-estabelecido.

Esta listagem de operações deixa um problema por resolver: pôr itens, tirar itens e aceder a itens implica ser capaz de lhes especificar a posição na lista. Naturalmente que uma possibilidade seria indicar as posições através de um número de ordem, por exemplo numerando os itens a partir de zero desde o início da lista, e usar depois índices para especificar posições, como se faz usualmente nas matrizes e vectores. Esta solução tem, no entanto, um problema. Esse problema só se tornará verdadeiramente claro quando se fizer uma implementação eficiente do conceito de lista em secções posteriores e tem a ver com a eficiência dos seus métodos: quando se melhorar a implementação inicial das listas (ainda por fazer...) de modo a permitir inserções eficientes de novos itens *em qualquer posição*, concluir-se-á que a utilização de índices se tornará extremamente ineficiente...

Porquê esta preocupação com a eficiência quando se está ainda a definir a interface das listas, ou seja, as suas operações e respectivos efeitos? Acontece que se pode e deve considerar que a eficiência faz parte do contrato das rotinas e métodos. Embora a questão da eficiência de algoritmos esteja fora do contexto desta disciplina, é fundamental apresentar aqui pelo menos uma ideia vaga destas ideias.

Suponha-se que se pretende desenvolver um procedimento para ordenar os itens de um vector por ordem crescente. A interface do procedimento inclui, como se viu em capítulos anteriores, o seu cabeçalho, que indica como o procedimento se utiliza, e um comentário de documentação onde se indicam a pré-condição e a condição objectivo do procedimento, ou seja o contrato estabelecido entre o programador que o produziu e os programadores que o consumirão (ver Secção 9.3.1):

```
/** Ordena os valores do vector v.
    @pre  PC ≡ v = v.
    @post CO ≡ perm(v, v) ∧ (Q i, j : 0 ≤ i ≤ j < v.size() : v[i] ≤ v[j]).
void ordenaPorOrdemNãoDecrescente(vector<int>& v);
```

No entanto, esta interface não diz tudo. Se se mandar ordenar um vector com 1000 itens, quanto tempo demora o procedimento? E se forem o dobro, 2000 itens? Como cresce o tempo de execução com o número de itens? E quanta memória adicional usará o procedimento?

Estas questões são muitas vezes relevantes, embora nem sempre. Podem fazer a diferença entre um programa que demora dias ou meses (ou anos, séculos ou milénios...) a executar e outro programa que demora segundos. Ou entre um programa que pode ser executado em computadores com memória limitada e outro com um apetite voraz pela memória que o torna inútil na prática.

As eficiências em termos de tempo de execução e em termos de espaço de memória consumido são muito importantes e podem e devem ser considerados parte do contrato de uma rotina ou método de uma classe. No caso do procedimento de ordenação a interface, em rigor, deveria ser melhor especificada:

```
/** Ordena os valores do vector v. O tempo de execução cresce com número n
    de itens ao mesmo ritmo que a função n ln n. Ou seja, a eficiência temporal é
    O(n ln(n)). A ordenação é feita directamente sobre a matriz, recorrendo-se
    apenas a um pequeno número de variáveis auxiliares.
    @pre   PC ≡ v = v.
    @post  CO ≡ perm(v, v) ∧ (Q i, j : 0 ≤ i ≤ j < v.size() : v[i] ≤ v[j]).
void ordenaPorOrdemNãoDecrescente(vector<int>& v);
```

É importante perceber-se que do contracto não fazem parte tempos de execução ao segundo. Há duas razões para isso. A primeira é prática: os tempos exactos de execução não são fáceis de calcular e, sobretudo, variam de computador para computador e de máquina para máquina. A segunda razão prende-se com o facto de ser muito mais importante saber que o tempo de execução ou a memória requerida por um algoritmo crescem de uma forma "bem comportada" do que saber o tempo exacto de execução[3].

Esta digressão veio como justificação para a utilização de critérios de eficiência na exclusão dos índices para indicar localizações de itens em listas. É que se pretende que as listas tenham métodos de inserção de novos itens em locais arbitrários (e respectiva remoção) tão eficientes quanto a sua inserção em (ou remoção de) locais "canónicos", tais como os seus extremos. Isso, como se verá, levará a uma "arrumação" dos itens tal que a sua indexação será inevitavelmente muito ineficiente.

Claro que uma possibilidade de resolução deste problema seria fornecer a possibilidade de usar índices para referenciar posições de itens na lista mas assinalar claramente como ineficientes as respectivas operações. No entanto, tal não será feito aqui, até porque se irá desenvolver uma forma mais apropriada de referenciar os itens de uma lista.

Há que distinguir entre as operações realizadas em locais "canónicos" das listas, que são os seus extremos, e aquelas que se realizam em locais arbitrários, que terão de fazer uso de uma generalização do conceito de indexação, a introduzir na próxima secção. As operações de inserção, remoção e acesso em locais canónicos são:

- Tirar o item da *frente* da lista. A lista não pode estar vazia.

[3] A título de exemplo, suponham-se dois algoritmos possíveis para a implementação do procedimento de ordenação acima. Suponha-se que o primeiro, mais simples, é caracterizado por o tempo de execução crescer com o quadrado do número de itens a ordenar e que o segundo, um pouco mais complicado, tem tempos de execução que crescem com o produto do número de itens a ordenar pelo seu logaritmo. A notação da algoritmia para a eficiência temporal destas algoritmos é $O(n^2)$ e $O(n \log n)$, respectivamente, em que n é o número de itens a ordenar. Cálculos muito simples permitem verificar que, para um número suficientemente grande de itens, o algoritmo mais complicado acaba sempre por ter uma eficiência superior.

- Tirar o item de *trás* da lista. A lista não pode estar vazia.
- Pôr um novo item na frente da lista. A lista não pode estar cheia, se tiver limite pré-estabelecido.
- Pôr um novo item na traseira da lista. A lista não pode estar cheia, se tiver limite pré-estabelecido.
- Aceder ao item na frente da lista (para modificação ou não). A lista não pode estar vazia.
- Aceder ao item na traseira da lista (para modificação ou não). A lista não pode estar vazia.

É possível imaginar muitas mais operações com listas. Algumas delas são realmente úteis, mas só serão introduzidas mais tarde, de modo a manter uma complexidade limitada na primeira abordagem. Outras serão porventura menos úteis em geral e poderão ser dispensadas. Não é fácil, como é óbvio, distinguir operações essenciais de operações acessórias. Algumas são essenciais para que se possa dar o nome de lista ao tipo em análise. As operações escolhidas neste e no próximo capítulo incluem as fundamentais para que se possa de facto falar em listas (como as operações de inserção e remoção de itens em locais canónicos) e também as mais comummente utilizadas na prática. Por outro lado, correspondem também a algumas das operações existentes na interface do tipo genérico `list` da biblioteca padrão do C++[4]. É o caso, por exemplo da seguinte operação, que fará parte da interface das listas:

- Esvaziar a lista, tirando todos os seus itens.

10.2 Iteradores

Como fazer para indicar um local arbitrário de inserção, remoção ou simples acesso a itens da lista? É conveniente aqui observar como um humano resolve o problema. Como o único humano disponível neste momento é o leitor, peço-lhe que entre no jogo e colabore. Considere a seguinte lista de inteiros

$$(1\ 2\ 3\ 11\ 20\ 0\ 354\ 2\ 3\ 45\ 12\ 34\ 30\ 4\ 4\ 23\ 3\ 77\ 4\ \ -1\ \ -20\ 46).$$

Considere um qualquer item da lista e suponha que pretende indicá-lo a um amigo. Suponha que esse amigo está ao seu lado e pode ver a lista. Indique-lhe o item que escolheu.

⋮

⋮

[4] Ou seja, reinventa-se de novo a roda... Ver Nota 1 na página 301.

Provavelmente o leitor indicou-o da forma mais óbvia: apontando-o com o dedo indicador... Pelo menos é assim que a maioria das pessoas faz... O objectivo agora é, pois, conseguir representar o conceito de "dedo indicador" em C++...

E se o leitor quiser indicar o local para inserir um novo item na lista? Provavelmente fá-lo-á indicando um intervalo entre dois itens (excepto se pretender inserir num dos extremos da lista). Aqui, no entanto, usar-se-á uma abordagem diferente: é sempre possível indicar um local de inserção indicando um item existente e dizendo para inserir o novo item imediatamente antes ou depois do item indicado. Isto, claro está, se a lista não estiver vazia! Mas mesmo nesse caso se encontrará uma solução interessante.

Para além da lista em si, tem-se agora um outro conceito para representar. Bons nomes para o novo conceito poderiam ser "indicador" (sugestivo dada a analogia do dedo) ou "cursor". Porém, o termo aqui usado é o usual em programação: *iterador*.

10.2.1 Operações com iteradores

Que operações se podem realizar com os iteradores? A primeira operação é, naturalmente, construir um iterador. É razoável impor que um iterador tenha de estar sempre *associado* a uma determinada lista. Assim, a construção de um iterador associa-o a uma lista e põe-no a *referenciar* um determinado item de essa lista, por exemplo o da sua frente (se existir...).

Imaginem-se agora as operações que se podem realizar com um dedo sobre uma lista. Identificam-se imediatamente as operações mais elementares: avançar e recuar[5] o iterador e aceder ao item referenciado pelo iterador. Por razões que se verá mais tarde, não será eficiente colocar um iterador numa posição arbitrária de uma lista (se exceptuarmos as posições canónicas) nem avançar ou recuar um iterador mais do que um item de cada vez. Estas restrições têm menos a ver com o conceito de iterador em si do que com o tipo de contentor de itens a que está associado[6]. Ou seja, as operações a realizar com iteradores são:

- Construir iterador associado a lista e referenciando o item na sua frente.
- Avançar o iterador. O iterador não pode referenciar o item de trás da lista.
- Recuar o iterador. O iterador não pode referenciar o item na frente da lista.
- Aceder ao item referenciado pelo iterador (para modificação ou não).

[5]Usa-se o termo "avançar" para deslocamentos da frente para trás e o termo "recuar" para deslocamentos de trás para a frente. Isto pode parecer estranho, mas é prática habitual e fará um pouco mais de sentido quando associado aos termos "início" e "fim". A ideia é que avançar é deslocar do início para o fim, tal como o leitor desloca o olhar, avançando, do início para o fim deste texto...

[6]Um *contentor* é um tipo abstracto de dados capaz de conter itens de um outro tipo. Existem variadíssimos contentores, que incluem listas, filas, pilhas, vectores, matrizes, colecções, conjuntos, mapas, etc. Cada um destes tipos de contentores (com excepção das pilhas e das filas) pode ser associado a um tipo específico de iterador. Os iteradores podem ser categorizados em *conceitos*, de acordo com as características específicas do respectivo contentor !!!!citarAustern. Por exemplo, vectores têm associados iteradores ditos de acesso aleatório, muito menos restrivos que os das listas, ditos bidireccionais.

Quantos iteradores se podem associar a uma lista? A solução mais restritiva é permitir apenas um iterador por lista. Esta solução, que não será seguida aqui, tem a vantagem de não exigir a definição de classes auxiliares e de resolver bem o problema da validade dos iteradores depois de alterações à lista que lhe está associada. A solução aqui seguida será a mais genérica: poderão existir um número arbitrário de iteradores associados à mesma lista.

A existência de vários iteradores associados à mesma lista torna necessários os operadores de igualdade e diferença entre iteradores:

- Verificação de igualdade entre iteradores. Os iteradores têm de estar associados à mesma lista.
- Verificação da diferença entre iteradores. Os iteradores têm de estar associados à mesma lista.

10.2.2 Operações se podem realizar com iteradores e listas

Para além das operações realizadas sobre os iteradores, há algumas operações que são realizadas sobre uma lista mas usando um iterador como indicador de uma posição arbitrária. Estas operações são as de inserção e remoção de itens em e de locais arbitrários mencionadas mais atrás. Por outro lado, era conveniente que a lista possuísse métodos construtores de iteradores para as suas posições canónicas.

Relativamente às operações de inserção e remoção, há que tomar algumas decisões.

Em primeiro lugar tem de se decidir se se insere antes ou depois do item referenciado por um iterador. Isto deve-se a que os iteradores referenciam itens, e não intervalos entre itens. Poder-se-ia escolher fornecer duas operações, uma inserindo antes e outra depois. Na prática considera-se suficente fornecer uma versão que insira antes de um iterador, pois durante inserções ordenadas é típico o iterador parar logo que se encontra o primeiro item à direita do qual o novo item não pode ser colocado (ver mais abaixo).

Em segundo lugar tem de se decidir o que fazer a um iterador depois da remoção do item por ele referenciado. É usual escolher-se avançar o iterador para o item imediatamente a seguir.

Ambas as opções, i.e., inserir antes e avançar ao remover, são típicas, embora razoavelmente arbitrárias.

Relativamente às posições canónicas, chamar-se-á *primeiro* a um iterador referenciando o item na frente da lista e *último* a um iterador referenciando o item na traseira da lista. Assim, as operações são:

- Inserir novo item antes do item referenciado por um iterador.
- Remover item referenciado por um iterador (e avançá-lo).
- Construir primeiro iterador, i.e., um iterador referenciando o item na frente da lista.
- Construir último iterador, i.e., um iterador referenciando o item na traseira da lista.

10.2.3 Itens fictícios

Para terminar a discussão acerca das operações de listas e iteradores há que resolver ainda alguns problemas. Nas operações apresentadas até agora teve-se sempre o cuidado de apresentar as respectivas pré-condições. Mas houve alguns "esquecimentos":

- Qual é o primeiro iterador de uma lista vazia? E o último?
- Que item referencia um iterador depois de, através dele, se ter removido o item da traseira da lista?

Outras perguntas igualmente interessantes podem ser feitas:

- Que item referencia um iterador associado a uma lista vazia?
- Como se insere na traseira da lista, através de um iterador, se existe apenas uma operação para inserir *antes* da posição referenciada por um iterador?
- Como se percorre uma lista do princípio ao fim com um iterador?

Para situar um pouco melhor estas perguntas e justificar uma possível resposta, suponha-se o problema de inserir ordenadamente um novo item numa lista de inteiros

$$\texttt{lista} = (1\ 2\ 6\ 9).$$

Suponha-se que o item a inserir é 5. Que fazer? Que algoritmo se pode usar que resolva o problema para qualquer lista ordenada e qualquer novo item?

Uma possibilidade é fazer uma pesquisa sequencial desde o item na frente da lista até encontrar o primeiro item maior ou igual ao novo item e inseri-lo imediatamente antes do item encontrado. Ou seja,

{ Algoritmo para inserir ordenadamente o novo item `novo_item` na lista `lista`. }

{ Construir um iterador referenciando o primeiro item da lista: }
`i` ← primeiro de `lista`
enquanto item referenciado por `i < novo_item` **faça-se:**
    avançar `i`
inserir `novo_item` na lista `lista` imediatamente antes do item referenciado por `i`

E se o novo item for 10? É claro que o algoritmo anterior não está correcto. A sua guarda tem de permitir a paragem mesmo que todos os itens da lista sejam inferiores ao novo item a inserir:

```
{ Algoritmo para inserir ordenadamente o novo item novo_item na lista lista. }

{ Construir um iterador referenciando o primeiro item da lista: }
i ← primeiro de lista
enquanto i não atingiu o fim da lista lista ∧
          item referenciado por i < novo_item faça-se:
     avançar i
inserir novo_item na lista lista imediatamente antes do item referenciado por i
```

O problema é o significado da frase "`i` não atingiu o fim da lista `l`". Que significa o fim da lista? Não é decerto o iterador referenciando o item na traseira. Porquê? Porque nesse caso o algoritmo que segue, que era suposto mostrar todos os itens da lista, pura e simplesmente não funciona: deixa o item de trás da lista de fora...

```
{ Algoritmo para mostrar todos os itens da lista lista. }

{ Construir um iterador referenciando o primeiro item da lista: }
i ← primeiro de lista
enquanto i não atingiu o fim da lista lista faça-se:
     mostrar item referenciado por i
     avançar i
```

Conclui-se facilmente que, a haver um iterador referenciando o fim da lista, ele está para além do último! Mas nesse caso, que item referencia? A esse item, que na realidade não existe na lista, dar-se-á o nome de item *fictício*. Assim, pode-se considerar que as listas, para além dos seus itens, possuem dois itens fictícios nos seus extremos. Estes itens são particulares por várias razões:

1. Existem sempre, mesmo com a lista vazia.
2. Não correspondem a itens reais, pelo que não têm valor.
3. São úteis apenas enquanto âncoras para os iteradores, que se lhes podem referir.

Chamar-se-á aos iteradores antes do primeiro e depois do último respectivamente *início* e *fim*. Veja-se a situação retratada na Figura 10.1.

Na realidade a noção de item fictício foi usada já extensamente quando se desenvolveram ciclos para percorrer matrizes. Repare-se no seguinte troço de programa:

```
double md[4] = {1.1, 2.2, 3.3, 4.4};

for(int i = 0; i != 4; ++i)
    cout << md[i] << endl;
```

Neste ciclo o índice toma todos os valores entre 0 e 4, sendo 4 a dimensão da matriz e, portanto, o índice de um elemento fictício final da matriz. Da mesma forma se pode considerar que uma matriz tem um item fictício inicial:

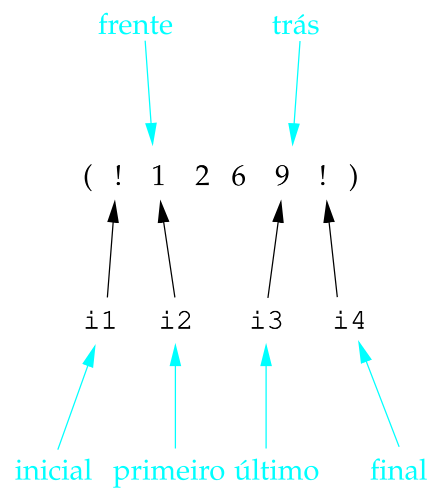


Figura 10.1: Definição de iteradores e itens canónicos das listas. Representam-se por um ponto de exclamação os itens fictícios da lista. Os iteradores primeiro (`i2`) e último (`i3`) referenciam respectivamente os itens na frente e na traseira da lista. Os iteradores início (`i1`) e fim (`i4`) são definidos como sendo respectivamente anterior ao primeiro e posterior ao último.

```
double md[4] = {1.1, 2.2, 3.3, 4.4};

for(int i = 3; i != -1; --i)
    cout << md[i] << endl;
```

Voltando às listas, e dados os dois novos iteradores início e fim, são úteis duas novas operações de construção para os obter:

- Construir iterador início, i.e., o iterador antes do primeiro.
- Construir iterador fim, i.e., o iterador depois do último.

Com estes novos iteradores, torna-se claro que os algoritmos propostos estão correctos. No caso da inserção ordenada, mesmo que o novo item seja superior a todos os itens da lista, o ciclo termina com o iterador final, pelo que inserir o novo item antes da posição referenciada pelo iterador leva-o a ficar na traseira da lista, como apropriado! É também interessante compreender que o posicionamento dos iteradores no caso de uma lista vazia, tal como indicado na Figura 10.2, apesar de contra-intuitivo, leva também ao correcto funcionamento dos algoritmos nesse caso extremo.

10.2.4 Operações que invalidam os iteradores

Que acontece a um iterador referenciando um item não-fictício de uma dada lista se essa lista for esvaziada? É evidente que esse iterador não pode continuar válido. Assim, é importante reconhecer que os iteradores podem estar em estados inválidos, nos quais a sua utilização é um erro. A questão mais importante associada à validade dos iteradores é a de saber que

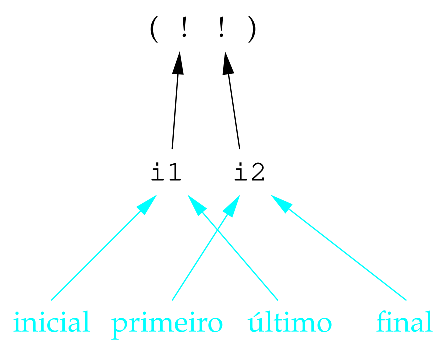


Figura 10.2: Definição de iteradores e itens canónicos de uma lista vazia. Note-se que os iteradores primeiro e último trocam as suas posições relativas usuais. Decorre daqui que são estes iteradores que se definem à custa dos iteradores início e fim, e não o contrário, como se sugeriu atrás.

operações invalidam que iteradores associados a uma lista. Esta questão e as pré-condições das operações discutidas apresentam-se na secção seguinte.

10.2.5 Conclusão

Operações apenas das listas:

- Construtoras:
  - Construir nova lista vazia.
- Inspectoras:
  - Saber comprimento da lista, i.e., quantos itens contém.
  - Saber se a lista está vazia.
  - Saber se está cheia, se tiver limite pré-estabelecido.
  - Saber valor do item na frente da lista. $PC \equiv$ a lista não pode estar vazia.
  - Saber valor do item na traseira da lista. $PC \equiv$ lista não pode estar vazia.
- Modificadoras:
  - Aceder ao item na frente da lista (para possível modificação). $PC \equiv$ a lista não pode estar vazia. Não invalida iteradores.
  - Aceder ao item na traseira da lista (para possível modificação). $PC \equiv$ a lista não pode estar vazia. Não invalida iteradores.
  - Pôr um novo item na frente da lista. $PC \equiv$ a lista não pode estar cheia, se tiver limite pré-estabelecido. Não invalida iteradores.
  - Pôr um novo item na traseira da lista. $PC \equiv$ a lista não pode estar cheia, se tiver limite pré-estabelecido. Não invalida iteradores.

- Tirar o item da frente da lista. $PC \equiv$ a lista não pode estar vazia. Invalida todos os iteradores que referenciem o item da frente da lista.
- Tirar o item de trás da lista. $PC \equiv$ a lista não pode estar vazia.Invalida todos os iteradores que referenciem o item de trás da lista.
- Esvaziar a lista, tirando todos os seus itens. Invalida todos os iteradores associado à lista, com excepção dos que referenciam os seus itens fictícios.

Em todas as operações relacionadas com iteradores (exceptuando as construtoras de iteradores) considera-se como pré-condição que os iteradores têm de ser válidos. Esta condição não se apresenta explicitamente abaixo.

Operações dos iteradores:

- Construir iterador associado a lista (iterador é o primeiro, i.e., se a lista estiver vazia é o seu fim).
- Avançar o iterador. $PC \equiv$ o iterador não pode ser o fim da lista.
- Recuar o iterador. $PC \equiv$ o iterador não pode ser o início da lista.
- Aceder ao item referenciado pelo iterador (para possível modificação). $PC \equiv$ o iterador não pode ser nem o início nem o fim da lista.
- Verificação de igualdade entre iteradores. $PC \equiv$ os iteradores têm de estar associados à mesma lista.
- Verificação da diferença entre iteradores. $PC \equiv$ os iteradores têm de estar associados à mesma lista.

Operações das listas relacionadas com iteradores:

- Inserir novo item antes do item referenciado por um iterador. $PC \equiv$ o iterador não pode ser o início da lista. Nâo invalida iteradores.
- Remover item referenciado por um iterador (e avançá-lo).$PC \equiv$ o iterador não pode ser nem o início nem o fim da lista. Invalida todos os iteradores que referenciem o item removido (com excepção do iterador que foi usado para a remoção, que avança).
- Construir primeiro iterador, i.e., o iterador depois do início da lista.
- Construir último iterador, i.e., o iterador antes do fim da lista.
- Construir iterador início.
- Construir iterador fim.

10.3 Interface

Os conceitos de lista e iterador, para serem verdadeiramente úteis, têm de ser concretizados na forma de classes. As operações estudadas nas secções anteriores correspondem à sua interface. Para simplificar o desenvolvimento supor-se-á que as listas guardam inteiros. Assim, chamar-se-á `ListaDeInt` à classe que concretiza o conceito de lista de inteiros e `Iterador` à classe que concretiza o conceito de iterador de uma lista de inteiros.

Nas secções abaixo vai-se definindo a interface da classe passo por passo. Os contractos das operações declaradas são paresentados informalmente. No final é apresentada uma veroa completa, com os contratos das operações na forma de comentários de documentação.

10.3.1 Interface de `ListaDeInt`

Esta classe é a concretização do conceito de lista de inteiros:

```
class ListaDeInt {
  public:
```

Tipos

Há dois tipos definidos pela classe. O primeiro, `Item`, não é verdadeiramente um novo tipo, pois `Item` é sinónimo de `int`:

```
typedef int Item;
```

Este sinónimo tem a vantagem de simplificar consideravelmente a tarefa de criar listas com itens de outros tipos.

!!!!Dar maior ênfase a isto!

O segundo tipo é a classe `Iterador`:

```
class Iterador;
```

!!!!Reescrever

De modo a que o identificador `Iterador` não fique gasto, uma vez que se podem definir iteradores para muitos outros tipos de contentores para além das listas, é conveniente que a classe `Iterador` seja embutida dentro da classe `ListaDeInt`.

Métodos construtores

Declara-se apenas um método construtor que constrói uma lista vazia:

```
ListaDeInt();
```

Métodos inspectores

Declaram-se três métodos inspectores. Este inspectores podem ser usados em qualquer circunstância para indagar do estado da lista.

O primeiro devolve o número de itens da lista, ou seja, o seu comprimento:

```
int comprimento() const;
```

O segundo serve para verificar se a lista está vazia:

```
bool estáVazia() const;
```

O terceiro serve para verificar se a lista está cheia:

```
bool estáCheia() const;
```

Declaram-se adicionalmente dois métodos inspectores que apenas podem ser invocados se a lista não estiver vazia e que servem para aceder (sem poder modificar) aos itens nas posições canónicas da lista. Ambos têm como pré-condição:

$$PC \equiv \neg\texttt{estáVazia}().$$

O primeiro devolve uma referência constante para o item na frente da lista:

```
Item const& frente() const;
```

O segundo devolve uma referência constante para o item na traseira da lista:

```
Item const& trás() const;
```

Métodos modificadores

Os dois primeiros métodos modificadores declarados são semelhantes aos inspectores dos itens nas posições canónicas da lista. Tal como eles, apenas podem ser invocados se a lista não estiver vazia

$$PC \equiv \neg\texttt{estáVazia}().$$

Ao contrário deles, no entanto, devolvem referências para os itens, o que permite que estes sejam modificados. O primeiro devolve um referência para o item na frente da lista:

```
Item& frente();
```

O segundo devolve uma referência para o item na traseira da lista:

```
Item& trás();
```

Os dois métodos modificadores seguintes permitem pôr um novo item nas posições canónicas da lista. Ambos requerem que a lista não esteja cheia, i.e.,!!!

$$PC \equiv \neg\texttt{estáCheia}().$$

!!!!!Vou aqui

O primeiro dos métodos põe o novo item na frente da lista:

```
void põeNaFrente(Item const& novo_item);
```

O segundo põe o novo item na traseira da lista:

```
void põeAtrás(Item const& novo_item);
```

Declaram-se também dois métodos modificadores que permitem tirar um item das posições canónicas da lista. Ambos requerem que a lista não esteja vazia, i.e.,

$$PC \equiv \neg\texttt{estáVazia}().$$

Ambos invalidam qualquer iterador que esteja associado à lista.

O primeiro dos métodos tira o item da frente da lista:

```
void tiraDaFrente();
```

O segundo tira o item da traseira da lista:

```
void tiraDeTrás();
```

Finalmente, declara-se um método modificador de que não se falou ainda. Este método serve para esvaziar a lista, i.e., para descartar todos os seus itens:

```
void esvazia();
```

Este método invalida qualquer iterador que esteja associado à lista.

Métodos modificadores envolvendo iteradores

Declaram-se dois métodos modificadores que envolvem a especificação de posições através de iteradores.

O primeiro insere um novo item imediatamente antes da posição indicada por um iterador `iterador`, exigindo-se para isso que a lista não esteja cheia, que o iterador seja válido e esteja associado à lista em causa e que o iterador não referencie o iterador início[7]

$PC \equiv$ `iterador` é válido $\wedge$ `iterador` associado à lista $\wedge\neg$`estáCheia`() $\wedge$ `iterador` $\neq$ `início`().

Este método garante que o iterador continua a referenciar o mesmo item que antes da inserção[8]:

```
void insereAntes(Iterador& iterador, Item const& novo_item);
```

Qualquer outro iterador associado à lista é invalidado por este método.

O segundo método remove o item indicado pelo iterador iterador, exigindo-se que este seja válido e esteja associado à lista em causa e que referencie um dos itens válido da lista, pois não faz sentido remover os itens fictícios[9]

$PC \equiv$ `iterador` é válido $\wedge$ `iterador` associado à lista $\wedge$ `iterador` $\neq$ `início`() $\wedge$ `iterador` $\neq$ `fim`().

Este método garante que o iterador fica a referenciar o item logo após o item removido. Por isso o iterador tem de ser passado por referência não-constante.

```
void remove(Iterador& iterador);
```

Qualquer outro iterador associado à lista é invalidado por este método.

[7]Ver declaração do método `início()` mais à frente. Repare-se que não se incluiu na pré-condição nenhum termo que garante que o iterador está de facto associado à lista em causa. Isso será feito mais tarde, quando se falar de ponteiros.

[8]Há aqui uma incoerência que o leitor mais atento detectará. Se o iterador se mantém referenciando o mesmo item que antes da inserção, não deveria ser passado por valor ou referência constante? É verdade que sim. Acontece que a primeira implementação simplista destas classes, que será feita na Secção 10.4, implicará uma alteração física ao iterador para ele conceptualmente se possa manter constante... Este tipo de problemas resolve-se normalmente à custa do qualificador `mutable`, que aplicado a uma variável membro de uma classe, permite que ela seja alterada mesmo que a instância que a contém seja constante. No entanto essa solução levaria a uma maior complexidade da classe e, além disso, o problema resolver-se-á naturalmente quando se melhorar a implementação da classe na Secção 10.5, pelo que mais tarde a declaração do método será mudada paulatinamente de modo ao iterador ser passado por referência constante...

[9]Note-se que não é necessário incluir na pré-condição a garantia de que a lista não está vazia, visto que se o iterador não se refere a nenhum dos itens fictícios, então certamente que a lista não está vazia.

Métodos construtores de iteradores

Estes métodos consideram-se modificadores porque a lista pode ser modificada através dos iteradores devolvidos. Existem quatro métodos construtores de iteradores, correspondentes aos quatro iteradores canónicos definidos na Figura 10.1. Os primeiros métodos devolvem iteradores referenciando os itens fictícios da lista:

```
Iterador início();
Iterador fim();
```

Os dois seguintes devolvem iteradores imediamente depois do início e antes do fim, i.e., nas posições conhecidas por "primeiro" e "último":

```
Iterador primeiro();
Iterador último();
```

Com é evidente os iteradores construídos por qualquer destes métodos são válidos.

Estes dois métodos completam a interface da classe `ListaDeInt`. Mais tarde, quando se passar à implementação, discutir-se-ão os membros privados da classe:

```
  private:
    ... // a completar mais tarde, pois é parte da implementação da classe.
};
```

Interface completa em C++

!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!Que fazer ao contrato? Convinha apresentá-lo!

!!!!Ir buscar tudo de novo aos guiões. Não esquecer de acrescentar invalidação de iteradores nos contratos das operações!

```
class ListaDeInt {
  public:
    typedef int Item;

    class Iterador;

    ListaDeInt();

    int comprimento() const;

    bool estáVazia() const;
    bool estáCheia() const;
```

```
        Item const& frente() const;
        Item const& trás() const;

        Item& frente();
        Item& trás();

        void põeNaFrente(Item const& novo_item);
        void põeAtrás(Item const& novo_item);

        void tiraDaFrente();
        void tiraDeTrás();

        void esvazia();

        void insereAntes(Iterador& iterador, Item const& novo_item);
        void remove(Iterador& iterador);

        Iterador início();
        Iterador fim();

        Iterador primeiro();
        Iterador último();

      private:
        ... // a completar mais tarde, pois é parte da implementação da classe.
    };
```

10.3.2 Interface de `ListaDeInt::Iterador`

Esta classe, `ListaDeInt::Iterador`, é a concretização do conceito de iterador de uma lista de inteiros:

```
    class ListaDeInt::Iterador {
      public:
```

Métodos construtores

Esta classe tem apenas um construtor. Como um iterador tem de estar sempre associado a uma qualquer lista, é natural que o construtor receba a lista como argumento. O iterador construído fica válido e associado à lista passada como argumento e referenciando o item na sua frente. I.e., o iterador construído é o primeiro da lista. Como através do iterador se podem alterar os itens da lista, a lista é passada por referência não-constante:

```
        explicit Iterador(ListaDeInt& lista_a_associar);
```

Note-se a utilização da palavra-chave `explicit`. A sua intenção é impedir que o construtor, podendo ser invocado com um único argumento, introduza uma conversão implícita de lista para iterador. Uma instrução como

```
ListaDeInt lista;
...
ListaDeInt i(lista);
...
if(i == lista) // comparação errónea se não houver conversão implícita.
    ...
```

resulta portanto num erro de compilação, ao contrário do que sucederia se a conversão implícita existisse[10].

Métodos inspectores

Declara-se um único método inspector, que serve para aceder ao item referenciado pelo iterador. Este método devolve uma referência para o item referenciado pelo iterador, de modo a que seja possível alterá-lo. É importante perceber-se que este método é constante, apesar de permitir alterações ao item referenciado: a constância de um iterador refere-se ao iterador em si, e não à lista associada ou aos itens referenciados, que não pertencem ao iterador, mas sim à lista associada[11]. Um iterador constante (`const`) não pode ser alterado (e.g., avançar ou recuar), mas permite alterar o item por ele referenciado na lista associada.

O método requer que o iterador seja válido e não seja nem o início nem o fim da lista, pois não faz sentido aceder aos itens fictícios

$$PC \equiv \text{é válido} \wedge \text{não é iterador inicial da lista} \wedge \text{não é iterador final da lista}.$$

A declaração do método é:

```
Item& item() const;
```

Operadores inspectores

Declaram-se dois operadores inspectores, i.e., dois operadores que não modificam a instância implícita, que é sempre o seu primeiro operando. São os operadores de igualdade (`==`) e diferença (`!=`). Estes operadores são membros da classe porque exigem acesso aos atributos privados da classe[12].

[10] Na realidade essa conversão poderia contribuir para aproximar o modelo das listas e iteradores do modelo de matrizes e ponteiros do C++. Ver Capítulo 11.

[11] Este facto, de resto, irá obrigar mais tarde à definição de uma classe adicional de iterador, menos permissiva, que garanta a constância da lista associada e dos itens referenciados.

[12] Uma alternativa possível, embora indesejável, seria definir os operadores como rotinas normais (não-membro) e torná-los amigos da classe `ListaInt::Iterador`. Isso só seria justificável se fosse importante que ocorressem conversões implícitas em qualquer dos seus operandos, como sucedia no exemplo dos números racionais (ver Capítulo 13). Neste caso essas conversões não são necessárias. Além disso, como se verá mais tarde (ver !!), a tranformação destas classes em classes modelo (genéricas) exige que os operadores de uma classe embutida sejam definidos como membros.

O primeiro operador verifica se a instância implícita (primeiro operando) é igual ao iterador passado como argumento (segundo operando). Dois iteradores são iguais se se referirem ao mesmo item. O segundo verifica se são diferentes:

```
bool operator == (Iterador const& outro_iterador) const;
bool operator != (Iterador const& outro_iterador) const;
```

Só faz sentido invocar estes operadores para iteradores válidos e associados à mesma lista

$$PC \equiv \text{é válido} \wedge \texttt{outro\_ iterador}\ \text{é válido} \wedge \text{associados à mesma lista}.$$

Métodos modificadores

Os únicos métodos modificadores de iteradores são os que permitem avançar e recuar um iterador de um (e um só) item na lista[13]. Em vez de se declararem métodos modificadores com nomes como `avança()` e `recua()`, optou-se por declarar os operadores de incrementação (++) e decrementação (−−), quer prefixos quer sufixos, de modo a que a utilização de iteradores fosse o mais parecida possível quer com a utilização de índices inteiros, quer com a utilização de ponteiros, a estudar no próximo capítulo.

Os operadores de incrementação prefixa e sufixa

```
Iterador& operator ++ ();
Iterador operator ++ (int);
```

exigem ambos que o iterador seja válido e não seja o fim da lista

$$PC \equiv \text{é válido} \wedge \text{não é iterador final da lista}.$$

Os operadores de decrementação prefixa e sufixa

```
Iterador& operator -- ();
Iterador operator -- (int);
```

exigem ambos que o iterador não seja o início da lista

$$PC \equiv \text{é válido} \wedge \text{não é iterador inicial da lista}.$$

Estes dois métodos completam a interface da classe `ListaDeInt::Iterador`. Mais tarde, quando se passar à implementação, discutir-se-ão os membros privados da classe:

```
  private:
    ... // a completar mais tarde, pois é parte da implementação da classe.
};
```

[13] Esta limitação deve-se às mesmas razões pelas quais não se usam índices para localizar itens em listas: implementações adequadas das listas tornariam essas operações onerosas.

Interface completa em C++

!!!O mesmo que para as listas!

```
class ListaDeInt::Iterador {
  public:
    explicit Iterador(ListaDeInt& lista_a_associar);

    Item& item() const;

    bool operator == (Iterador const& outro_iterador) const;
    bool operator != (Iterador const& outro_iterador) const;

    Iterador& operator ++ ();
    Iterador operator ++ (int);

    Iterador& operator -- ();
    Iterador operator -- (int);

  private:
    ... // a completar mais tarde, pois é parte da implementação da classe.
};
```

10.3.3 Usando a interface das novas classes

Uma vez definidos os conceitos de classe e iterador e definidas as respectivas interfaces em C++, pode-se passar directamente ao desenvolvimento de programas que os usem. É claro que não se pode ainda gerar programas executáveis, pois falta a implementação das classes. Como todas as interfaces são contratos, não apenas no caso das rotinas mas também no das classes, o que se possui para já relativamente às classes `ListaDeInt` e `Iterador` é um contrato de promessa. O programador produtor compromete-se a, num determinado prazo, fornecer ao programador consumidor uma implementação para as classes sem qualquer alteração à interface acordada. I.e., compromete-se a, a partir de determinada data, garantir o correcto funcionamento das classes com a interface acordada desde que o programador consumidor garanta, por seu lado, que invoca todos os métodos e rotinas com argumentos que verificam as respectivas pré-condições.

Este contrato permite ao programador consumidor ir desenvolvendo programas usando as classes *ainda antes de estas estarem completas*. É extremamente importante conseguir fazê-lo não apenas porque permite alguns ganhos em produtividade no seio das equipas, mas sobretudo, numa fase de formação, por fazê-lo estimula a capacidade de abstração: se a implementação das classes ainda não existe não é possível que nos distraia ao construirmos código que as usa!

Começa-se por um exemplo simples. Suponha-se uma dada lista `lista`, por exemplo

$$\texttt{lista} = (!\ 1\ 2\ 3\ 11\ 20\ 0\ 354\ 2\ 3\ 45\ 12\ 34\ 30\ 4\ 4\ 23\ 3\ 77\ 4\ \ -1\ \ -20\ 46).$$

Como mostrar os itens desta lista no ecrã? Claramente é necessário percorrê-la, o que só pode ser realizado usando iteradores. O algoritmo é o seguinte:

{ Algoritmo para mostrar todos os itens da lista `lista`. }

{ Construir um iterador referenciando o primeiro item da lista: }
`i` ← primeiro de `lista`
enquanto `i` não atingiu o fim da lista `lista` **faça-se:**
    mostrar item referenciado por `i`
    avançar `i`

A tradução deste algoritmo para C++ pode ser já feita recorrendo às classes cuja interface se definiu nas secções anteriores:

```
ListaDeInt lista;

... // Aqui preenche-se a lista...

ListaDeInt::Iterador i = lista.primeiro();
while(i != lista.fim()) {
    cout << i.item() << endl;
    ++i;
}
```

ou, usando um ciclo `for`,

```
ListaDeInt lista;

... // Aqui preenche-se a lista...

for(ListaDeInt::Iterador i = lista.primeiro(); i != lista.fim(); ++i)
    cout << i.item() << endl;
```

Um problema mais complexo é o de inserir um item `novo_item` por ordem numa lista `lista`. O algoritmo já foi visto atrás:

{ Algoritmo para inserir ordenadamente o novo item `novo_item` na lista `lista`. }

{ Construir um iterador referenciando o primeiro item da lista: }
`i` ← primeiro de `lista`
enquanto `i` não atingiu o fim da lista `lista` ∧
        item referenciado por `i` **<** `novo_item` **faça-se:**
    avançar `i`
inserir `novo_item` na lista `lista` imediatamente antes do item referenciado por `i`

A sua tradução para C++ é mais uma vez imediata:

```
ListaDeInt::Iterador i = lista.primeiro();
while(i != lista.fim() and i.item() < novo_item)
    ++i;

lista.insereAntes(i, novo_item);
```

Assim, é possível escrever código usando classes das quais se conhece apenas a interface. Aliás, esta é a tarefa do programador consumidor, que mesmo que conheça uma implementação, deve-se abstrair dela. É necessário agora implementar estas classes. A responsabilidade de o fazer é do programador produtor. Mas como pode o programador produtor garantir que cumpre a sua parte do contrato? Como pode ter alguma segurança acerca da qualidade dos seus produtos? Para além do cuidado posto no desenvolvimento disciplinado do código, o programador produtor tem de testar o código produzido.

10.3.4 Teste dos módulos

Como pode o programador produtor testar os módulos desenvolvidos, neste caso as classes `ListaDeInt` e `ListaDeInt::Iterador`? Escrevendo código com o objectivo simples de usar os módulos num conjunto de casos considerados interessantes, quer pela sua frequência na prática, quer pela sua natureza extrema. A ideia é levar os módulos desenvolvidos ao limite e verificar se resultam naquilo que se espera.

Tal como o programador consumidor pode começar a escrever código quando ainda só tem disponível a interface de um módulo, também o programador produtor o pode fazer: os testes de um módulo podem ser escritos antes mesmo da sua implementação. Aliás, a palavra "podem" é demasiado fraca: os testes *devem* ser escritos antes da sua implementação. São o ponto de partida para a implementação.

As razões são várias:

- Ao escrever os testes detectam-se erros, inconsistências, faltas e excessos na interface do módulo respectivo.
- Uma vez prontos, os testes podem ser usados, mesmo com implementações incompletas (nesse caso o programador produtor espera e antecipa a ocorrência de erros, naturalmente). Note-se que a execução dos testes com uma implementação incompleta implica que todas as rotinas e métodos do módulo estejam definidos em esqueleto pelo menos[14] (i.e., tem de ser possível construir um executável...).
- Se os testes completos existirem desde o início todas as alterações à implementação são feitas com maior segurança, uma vez que o teste pode ser realizado sem qualquer trabalho e, dessa forma, pode-se confirmar se as alterações tiveram sucesso.

[14] O nome usual para estes esqueletos que compilam embora não façam (ainda) o que é suposto fazerem é *stubs*.

As características desejáveis de um bom teste são pelo menos as seguintes:

- O nível mais baixo ao qual devem existir testes é o de módulo físico. Como é usual que cada módulo físico defina uma única classe (ou pelo menos várias classes totalmente interdependentes), os testes dos módulos físicos normalmente confundem-se com os testes das classes. Este requisito tem apenas a ver com a facilidade com que se podem executar automaticamente todos os testes de um projecto. Por vezes é desejável fugir a esta regra e fazer testes individualizados para algum ou todos os módulos definidos por um módulo físico, mas nesse caso o teste (global) do módulo físico deve-os invocar um por um.
- Os testes devem estar tanto quanto possível embebidos no próprio módulo físico. É usual que em C++ tenham a forma de uma função `main()` colocada dentro do ficheiro de implementação (`.C`) e protegida por uma directiva de compilação condicional à definição de uma macro de nome `TESTE`

  ```
  #ifdef TESTE

  ... // Preâmbulo do teste (#include, etc.)

  int main()
  {
      ... // Conjunto de testes...
  }

  #endif
  ```

  Isto permite facilmente executar os testes sempre que se faz alguma alteração ao módulo.
- Os testes não devem gerar senão mensagens muito simples, informando que se iniciou o teste de um dado módulo físico e de cada um dos seus módulos. No final devem terminar com uma mensagem assinalando o fim dos testes do módulo físico.
- Só em caso de erro devem ser escritas mais mensagens, que deverão se claras e explicativas. As mensagens de erro devem incluir o nome do ficheiro e o número da linha onde o erro foi detectado.
- Caso ocorra algum erro a função `main()` deve devolver 1. Dessa forma podem ser facilmente desenvolvidos programas que geram e executam os testes de todos os módulos físicos de um projecto e geram um relatório global.

!!!!!Valerá a pena manter o apêndice?

Usando as ideias acima desenvolveu-se o programa de teste que se pode encontrar na Secção H.1.3. As listagens completas do módulo físico `lista_int`, que define as classes `ListaDeInt` e `ListaDeInt::Iterador`, encontram-se na Secção H.1.

10.4 Implementação simplista

!!!!!Dizer algures que não se cumpre parte dos contratos! Nomeadamente no que diz respeito à invalidação dos iteradores.

Finalmente chega a altura de se pensar numa implementação para as classes `ListaDeInt` e `ListaDeInt::Iterador`. Mais vale tarde que nunca, dir-se-á. Mas é sempre conveniente adiar a implementação de um módulo até depois de definida a sua interface e de implementada a respectiva função de teste, pelo que esta é a altura certa para atacar o problema.

Nesta secção optar-se-á por uma implementação simplista, e consequentemente ineficiente, para as listas e respectivos iteradores. Discutir-se-ão apenas os aspectos fundamentais dessa implementação, pelo que se recomenda que o leitor complemente esta leitura com o estudo da Secção H.1, onde se encontra listada a implementação completa do módulo.

10.4.1 Implementação de `ListaDeInt`

É natural começar a implementação pelas listas, uma vez que a implementação dos iteradores dependerá da forma como os itens das listas forem organizados.

A questão mais importante é onde guardar os itens. Esta questão pode ser respondida de uma forma simples se se determinar que as listas são limitadas e se impuser que o número máximo de itens é comum a qualquer lista e é exactamente igual a, por exemplo, 100 itens. Com esta restrição, é evidente que os itens podem ser guardados numa matriz clássica[15]. Para isso útil começar por definir uma constante com o número máximo de itens das listas:

```
class ListaDeInt {
  public:
    ...
  private:
    static int const número_máximo_de_itens = 100;
```

Esta constante é partilhada por todas as instâncias da classe lista. Por isso se utilizou a palavra-chave `static`, que aplicada a um atributo torna-o um atributo de classe e não um atributo de instância (ver Secção 7.17.2). O facto de ser um atributo constante de classe e de um tipo aritmético inteiro básico autoriza a sua definição dentro da própria classe. Como o compilador conhece o valor da constante, esta pode ser usada para definir a matriz que guardará os itens:

```
    Item itens[número_máximo_de_itens];
```

Como organizar os itens na matriz? A solução simplista aqui adoptada passa por guardá-los nos primeiros elementos da matriz pela mesma ordem que têm na lista. Dessa forma os

[15]Poderiam ser guardados alternativamente num vector, mas as matrizes têm duas vantagens. A primeira é que demonstram que é possível implementar as classes em causa sem recorrer a outras classes do C++, e portanto de raiz. A segunda é que permitirá uma evolução clara e elegante para as versões mais eficientes da implementação e, finalmente, para o uso de ponteiros e variáveis dinâmicas.

elementos da matriz estarão divididos em duas zonas: os de menor índice estão ocupados com itens da lista, enquanto os de maior índice estão disponíveis para futura utilização. Esta organização torna necessária uma variável que guarde o número de itens da lista em cada instante, i.e., que permita saber a dimensão relativa dessas duas zonas da matriz:

```
        int número_de_itens;
```

A Figura 10.3 ilustra o estado interno de uma lista contendo

`lista` = (! 10 5 3 20 !).

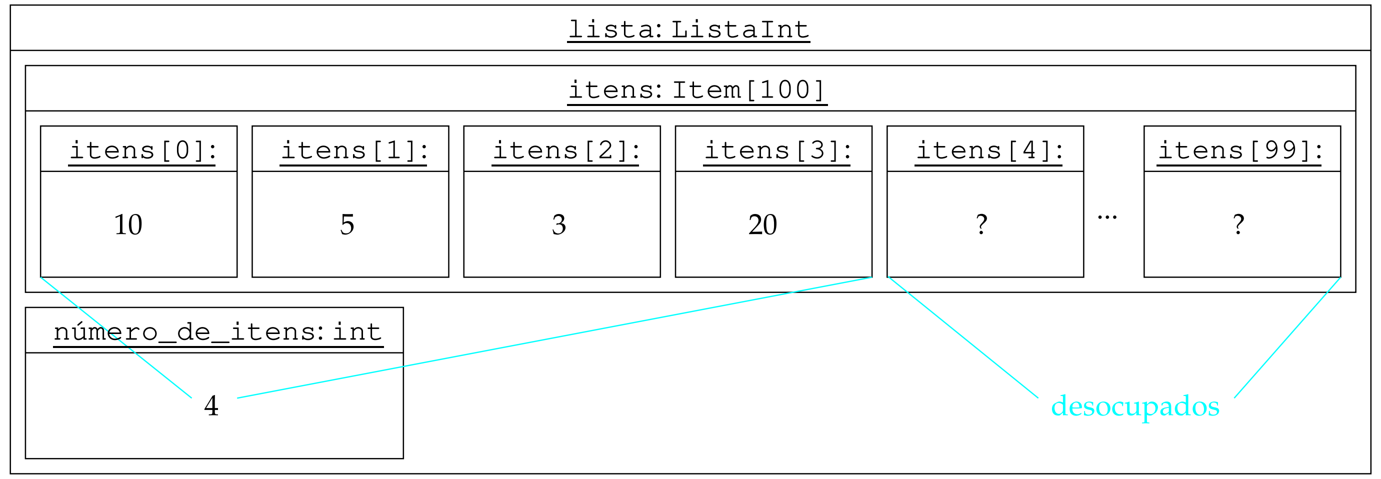


Figura 10.3: Estado interno da lista `lista` = (! 10 5 3 20 !).

Com a implementação escolhida os itens fictícios não têm existência física. Pode-se arbitrar, no entanto, que o item fictício antes da frente é o elemento (inexistente) da matriz com índice -1 e que o item fictício depois da traseira é o elemento (inexistente) com índice `número_de_itens`. Escolheu-se como item fictício final o elemento com índice `número_de_itens` e não `número_máximo_de_itens` de modo a que a passagem entre itens se possa fazer sempre por simples incrementação ou decrementação de índices.

A classe `ListaDeInt::Iterador`, que se implentará na próxima secção, necessita de acesso aos atributos privados da classe `ListaDeInt`. De outra forma não se poderia escrever o método `ListaDeInt::Iterador::item()`, como se verá. Assim, é necessário permitir acesso irrestrito dos iteradores às listas, o que se consegue introduzindo uma amizade:

```
        friend class Iterador;
    };
```

A utilização de amizades é, em geral, má ideia. Neste caso, porém, existe um casamento perfeito[16] entre as duas classes: o conceito de lista só se completa com o conceito de iterador e o conceito de iterador só se pode concretizar para um tipo específico de contentor, neste caso as listas. É natural que, havendo um casamento perfeito entre duas classes, estas tenham acesso aos membros privados (às partes íntimas, digamos) mutuamente. Sendo o casamento legítimo, esta amizade não é promíscua...

Condição invariante de instância

Qualquer classe que mereça esse nome tem uma condição invariante de instância (CIC). Esta condição indica as relações entre os atributos de instância da classe que se devem verificar em cada instante. No caso presente, a condição invariante de instância é simplesmente

$$CIC \equiv 0 \leq \texttt{número\_ de\_ itens} \leq \texttt{número\_ máximo\_ de\_ itens}.$$

Isto é, para que uma instância da classe ListaDeInt represente de facto uma lista, é necessário que a variável `número_de_itens` contenha um valor não-negativo e inferior ou igual ao máximo estipulado. Não é necessário impor qualquer condição adicional.

A condição invariante de instância tem de se verificar para todas as instâncias em jogo quer no início dos métodos públicos da classe quer no seu final[17]. Para segurança do programador produtor, é conveniente colocar asserções no início e no fim de todos esses métodos que verifiquem explicitamente esta condição. Para simplificar a escrita dessas asserções, pode-se acrescentar à classe um método privado `cumpreInvariante()` para verificar se o invariante é ou não cumprido:

```
class ListaDeInt {
  public:
    ...
  private:
    static int const número_máximo_de_itens = 100;

    Item itens[número_máximo_de_itens];
    int número_de_itens;

    bool cumpreInvariante() const;
};
```

Este método é privado porque o programador consumidor não precisa de verificar nunca se uma lista cumpre o invariante. Limita-se a assumir que sim. De resto, o invariante de uma classe é irrelevante para o programador consumidor, pois reflecte uma implementação particular e nada diz acerca da interface. A definição do método é simplesmente

[16] I.e., por amor.

[17] O caso dos construtores e dos destrutores é especial. A condição invariante de instância deve ser válida apenas no final do construtor e no início do destrutor.

```
inline bool ListaDeInt::cumpreInvariante() const {
    return 0 <= número_de_itens and
                número_de_itens <= número_máximo_de_itens;
}
```

10.4.2 Implementação de `ListaDeInt::Iterador`

Dada a implementação da classe `ListaDeInt`, a implementação da classe `ListaDeInt::Iterador` é quase imediata. Em primeiro lugar, cada iterador está associado a uma determinada lista. Assim, é necessário guardar uma referência para a lista associada dentro de cada iterador, pois podem existir variadas lista num programa, cada uma com vários iteradores associados. Em segundo lugar um iterador precisa de saber onde se encontra o item referenciado. Como os itens são guardados na matriz `itens` da classe `ListaDeInt`, basta guardar o índice nessa matriz do item referenciado:

```
class ListaDeInt::Iterador {
  public:
    ...
  private:
    ListaDeInt& lista_associada;
    int índice_do_item_referenciado;
};
```

A Figura 10.5 contém a representação interna da situação retratada na Figura 10.4, em que um iterador `i` referencia o terceiro item de uma lista

$$\texttt{lista} = (10\ 5\ 3\ 20).$$

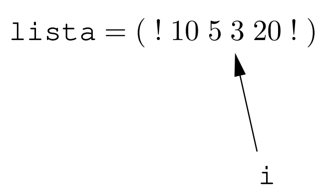


Figura 10.4: Representação simplificada de uma lista `lista` $= (\,!\ 10\ 5\ 3\ 20\ !\,)$ e de um iterador `i` referenciando o seu terceiro item.

Condição invariante de classe

Também os iteradores têm uma condição invariante de classe. Para esta implementação essa condição é muito simples:

$$CIC \equiv -1 \ \leq\ \texttt{índice\_do\_item\_referenciado} \leq \texttt{lista\_associada.número\_de\_itens}.$$

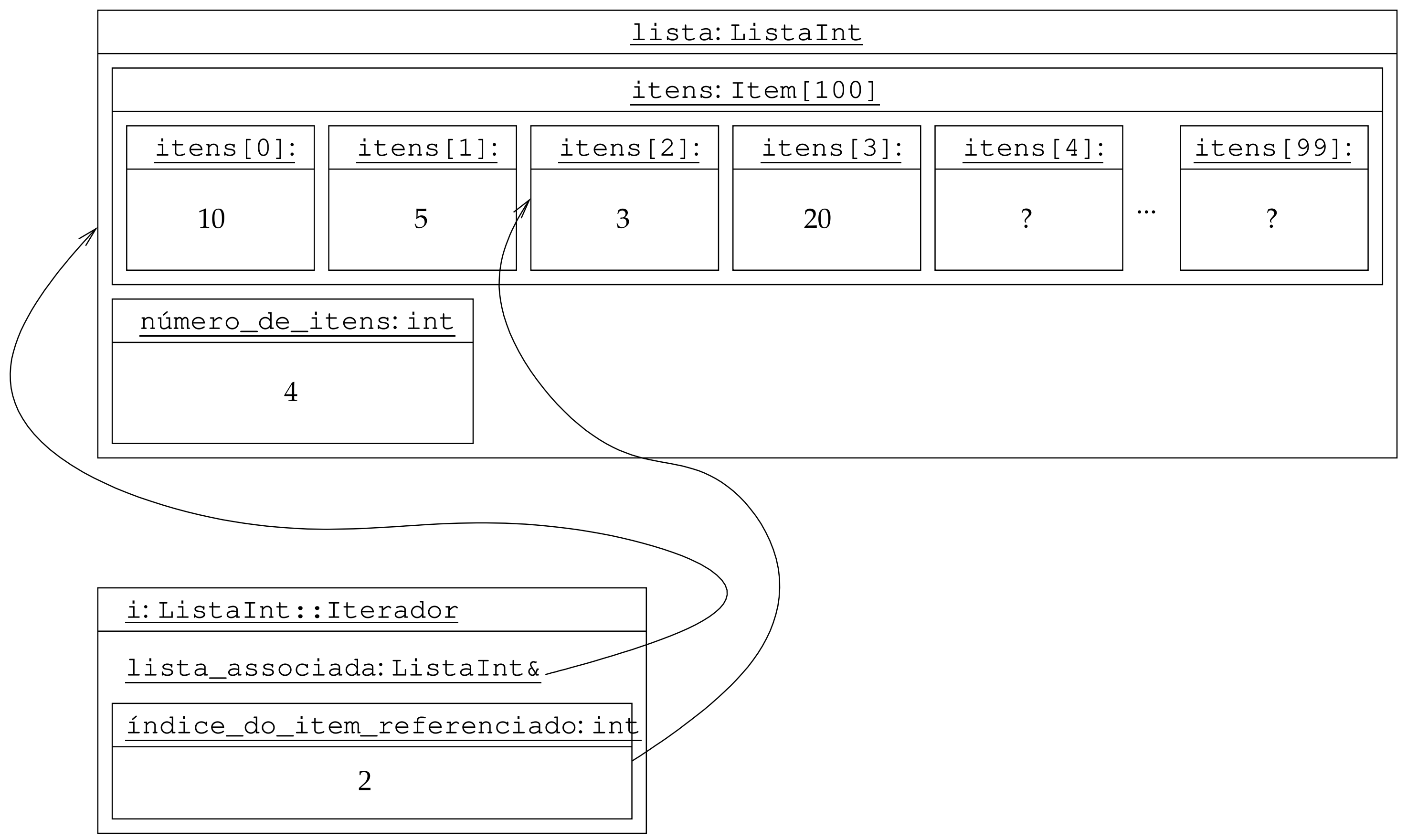


Figura 10.5: Estado interno da lista `lista` = (! 10 5 3 20 !) e de um iterador `i` referenciando o seu terceiro item.

e indica simplesmente que o índice do item referenciado tem de estar entre -1 (se o iterador for o início da lista) e `lista_associada.número_de_itens` (se o iterador for o início da lista).

O problema é que os iteradores, tal como pensados até aqui, têm uma característica infeliz: podem estar em estados inválidos. Por exemplo, se uma lista for esvaziada, todos os iteradores a ela associada ficam inválidos. Essa condição de iterador inválido deveria ser prevista pelo próprio código de tal forma que a condição invariante de instância fosse verdadeira mesmo para iteradores inválidos:

$$\begin{aligned} CIC \equiv \neg\, \text{é válido} \vee -1 \;\leq\;& \texttt{índice\_ do\_ item\_ referenciado} \leq \\ & \texttt{lista\_ associada.número\_ de\_ itens}. \end{aligned}$$

Na prática poder-se-ia guardar informação acerca do estado de validade de um iterador num novo atributo, booleano. Mas este pequeno acrescento exigiria muitas outras alterações, nomeadamente na implementação das listas, o que levaria um aumento considerável da complexidade deste par de classes. Em particular seria necessário prever o estado de iterador inválido em todas as operações que os envolvessem e, na implementação das listas, garantir que os iteradores invalidados fossem assinalados como tal. Fazê-lo é um exercício interessante e útil, mas fora do âmbito deste capítulo e do próximo[18]. Assim, com as implementações que serão feitas neste texto, deverá ser o programador consumidor a garantir que não faz uso de um iterador depois de este ter sido invalidado por alguma operação realizada sobre a lista associada: o código, através do método privado `cumpreInvariante()` que também será definido nesta classe, não o verificará:

```
class ListaDeInt::Iterador {
  public:
    ...
  private:
    ListaDeInt& lista_associada;
    int índice_do_item_referenciado;

    bool cumpreInvariante() const;
};

inline bool ListaDeInt::Iterador::cumpreInvariante() const {
    return -1 <= índice_do_item_referenciado and
        índice_do_item_referenciado <= lista_associada.número_de_itens;
}
```

10.4.3 Implementação dos métodos públicos de **`ListaDeInt`**

Neste secção definir-se-ão alguns dos métodos da classe `ListaDeInt`. Os restantes ficam como exercício para o leitor, que poderá depois conferir com a implementação total do módulo `lista_int` que se encontra na Secção H.1.

[18] O Apêndice I conterá no futuro uma versão mais segura e comentada das classes `ListaInt` e `Iterador` que verifica e mantém informação acerca da validade dos iteradores. !!!referir STLport

O primeiro método a definir é o construtor da lista, que se pode limitar a inicializar o número de itens com o valor zero, para que a lista fique vazia:

```
inline ListaDeInt::ListaDeInt()
    : número_de_itens(0) {
```

termina-se verificando se a nova lista cumpre o seu invariante:

```
    assert(cumpreInvariante());
}
```

O método `ListaDeInt::põeAtrás()` é também muito simples. Basta colocar o novo item após todos os itens existentes (ver Figura 10.3) e incrementar o número de itens:

```
inline void ListaDeInt::põeAtrás(Item const& novo_item) {
```

Começa-se por verificar se a lista cumpre a sua condição invariante de instância,

```
    assert(cumpreInvariante());
```

depois verifica-se a pré-condição do método,

```
    assert(not estáCheia());
```

faz-se a inserção do novo item,

```
    itens[número_de_itens++] = novo_item;
```

e termina-se verificando se no final do método a lista continua a cumprir a sua condição invariante de instância.

```
    assert(cumpreInvariante());
}
```

O método `ListaDeInt::põeNaFrente()` é um pouco mais complicado. O primeiro item está na posição 0 da matriz, pelo que é necessário deslocar todos os itens existentes de modo a deixar espaço para o novo item. Esta é a fonte principal de ineficiência desta implementação e a justificação para a necessidade de se mudar a implementação das listas, como se fará mais tarde:

```
void ListaDeInt::põeNaFrente(Item const& novo_item)
{
```

Começa-se por verificar a condição invariante de instância e a pré-condição do método,

```
        assert(cumpreInvariante());
        assert(not estáCheia());
```

Rearranjam-se os elementos da matriz de modo a deixar um espaço vago na posição 0,

```
        for(int i = número_de_itens; i != 0; --i)
            itens[i] = itens[i - 1];
```

depois insere-se o novo item na posição livre,

```
        itens[0] = novo_item;
```

incrementa-se o contador de itens,

```
        ++número_de_itens;
```

e termina-se verificando se no final do método a lista continua a cumprir a sua condição invariante de instância.

```
        assert(cumpreInvariante());
    }
```

O método `ListaDeInt::insereAntes()` é muito semelhante, embora o elemento da matriz a vagar seja o que contém o item referenciado pelo iterador, pelo que já não é necessário, em geral, rearranjar todos os elementos da matriz:

```
    void ListaDeInt::insereAntes(Iterador& iterador,
                                 Item const& novo_item)
    {
        assert(cumpreInvariante());
        assert(not estáCheia());
        assert(iterador é válido);
        assert(iterador != lista_associada.início());

        for(int i = número_de_itens;
            i != iterador.índice_do_item_referenciado;
            --i)
            itens[i] = itens[i - 1];
```

Tem de se incrementar o índice do item referenciado pelo iterador, para que ele passe a referenciar o item imediatamente depois do que se pretende remover.

```
        itens[iterador.índice_do_item_referenciado++] = novo_item;

        assert(iterador.cumpreInvariante());

        ++número_de_itens;

        assert(cumpreInvariante());
    }
```

Os métodos `ListaDeInt::remove()` e `ListaDeInt::tiraDaFrente()` têm de fazer a operação inversa. Define-se aqui o mais complicado dos dois:

```
    void ListaDeInt::remove(Iterador& iterador) {
        assert(cumpreInvariante());
        assert(iterador é válido);
        assert(iterador != início() and iterador != fim());

        --número_de_itens;
```

A decrementação do número de itens é fundamental para o bom funcionamento do ciclo!

```
        for(int i = iterador.índice_do_item_referenciado;
            i != número_de_itens;
            ++i)
            itens[i] = itens[i + 1];
```

O iterador fica automaticamente no local apropriado.

```
        assert(cumpreInvariante());
    }
```

Para terminar definem-se dois dos métodos construtores de iteradores:

```
    inline ListaDeInt::Iterador ListaDeInt::primeiro() {
        assert(cumpreInvariante());
```

Constrói-se um novo iterador para esta lista, que referencia inicialmente o item na frente da lista (ver construtor da classe `ListaDeInt::Iterador`), e devolve-se imediatamente o iterador criado.

```
        return Iterador(*this);
    }

    inline ListaDeInt::Iterador ListaDeInt::último() {
        assert(cumpreInvariante());

        Iterador iterador(*this);
```

Aqui, depois de construído um novo iterador, altera-se o seu índice, de modo a que referencie o item desejado (o da traseira).

```
        iterador.índice_do_item_referenciado = número_de_itens - 1;
```

Não é boa ideia que seja a classe `ListaDeInt` a invocar directamente o método de verificação do invariante da classe `ListaDeInt::Iterador`: Neste caso está-se mesmo a introduzir um excesso de intimidade... Mais tarde se verá uma melhor solução para este problema.

```
        assert(iterador.cumpreInvariante());

        return iterador;
    }
```

10.4.4 Implementação dos métodos públicos de `ListaDeInt::Iterador`

Neste secção definir-se-ão alguns dos métodos da classe `ListaDeInt::Iterador`. Os restantes ficam como exercício para o leitor, que poderá depois conferir com a implementação total do módulo `lista_int` que se encontra na Secção H.1.

O primeiro método a definir é o construtor da classe. Este método inicializa um novo ponteiro e modo a que referencie o primeiro item de uma lista dada:

```
    inline ListaDeInt::Iterador::Iterador(ListaDeInt& lista_a_associar)
```

O atributo `lista_associada` é uma referência, pelo que neste caso não é apenas conveniente ou recomendável usar uma lista de inicializadores: é obrigatório.

```
        : lista_associada(lista_a_associar),
          índice_do_item_referenciado(0) {
```

Mais uma vez se termina verificando a condição invariante de instância.

```
        assert(cumpreInvariante());
    }
```

O operador de decrementação serve para recuar o item referenciado na lista:

```
    inline ListaDeInt::Iterador& ListaDeInt::Iterador::operator -- () {
        assert(cumpreInvariante());
        // assert(é válido);
        assert(*this != lista_associada.início());
```

Basta decrementar o índice do item referenciado.

```
        --índice_do_item_referenciado;

        assert(cumpreInvariante());
        return *this;
    }
```

O operador de igualdade entre iteradores é muito simples. Dois uteradores válidos e associados à mesma lista são iguais se referenciarem o mesmo item, ou seja, se tiverem o mesmo valor no atributo `índice_do_item_referenciado`:

```
    inline bool ListaDeInt::Iterador::
    operator == (Iterador const& outro_iterador) const {
        assert(cumpreInvariante() and
               outro_iterador.cumpreInvariante());
        // assert(é válido and outro_iterador é válido);
        // assert(iteradores associados à mesma lista...);
```

É importante garantir que os iteradores estão associados à mesma lista. O problema resolver-se-á no próximo capítulo, quando se introduzirem os conceitos e ponteiro e endereço.

```
        return índice_do_item_referenciado ==
               outro_iterador.índice_do_item_referenciado;
    }
```

Finalmente, define-se o método inspector que devolve uma referência para o item referenciado pelo iterador:

```
    inline ListaDeInt::Item& ListaDeInt::Iterador::item() const {
        assert(cumpreInvariante());
        // assert(é válido);
        assert(*this != lista_associada.início() and
               *this != lista_associada.fim());
```

Esta linha de código, em que a classe `ListaDeInt::Iterador` acede a um atributo da classe `ListaDeInt` é uma das razões pelas quais se declarou a primeira classe amiga da segunda.

```
        return lista_associada.itens[índice_do_item_referenciado];
    }
```

10.5 Uma implementação mais eficiente

!!!!!!!!!!Imprimir apêndice com listagens e reproduzir em código! Depois ir buscar versão em código antiga da implementação com cadeias e actualizar de modo a ser o mais parecida possível com a que obtive.

!!!!Aqui o fundamental são os bonecos. Fazer figura sideways!

Começar por escrever as classes ListaDeInt e ListaDeInt::Iterador no quadro (versão simples com matrizes). Pelo menos as partes principais. Explicá-las brevemente.

Vamos ver uma pequena utilização de listas. Suponham que se pretendia ler informação (nome e número) acerca de um conjunto de 100 alunos e depois escrevê-la por ordem alfabética do nome. Podia-se começar por fazer uma classe para representar um aluno:

class Aluno { public: Aluno(string const& nome = "", int número = 0); string const& nome() const { return nome_; } int número() const { return número_; } private: string nome_; int número_; };

Agora, é só alterar a classe ListaDeInt para guardar alunos. O que é preciso fazer?

Discutir alterações. Concluir que é suficiente alterar o typedef e o nome da lista...

Falta agora o programa para ler os 100 alunos e escrevê-los por ordem alfabética.

int main() { ListaAluno lista;

for(int i = 0; i != 100; ++i) { string nome; // uma só palavra! int número; cin >> nome >> número; Aluno aluno(nome, número); // Inserir por ordem... } for(ListaAluno::Iterador i = lista.primeiro(); i != lista.fim(); ++i) cout << i.item().nome() << ' ' << i.item().numero() << endl; }

Como inserir por ordem?

Discutir brevemente o algoritmo. Concluir que tem de se procurar o primeiro nome maior e inserir antes. Se não se encontrar, insere-se antes do fim.

int main() { ListaAluno lista;

for(int i = 0; i != 100; ++i) { string nome; // uma só palavra! int número; cin >> nome >> número; Aluno aluno(nome, número); Iterador i = lista.primeiro(); while(i != lista.fim && i.item().nome() <= nome) ++i; lista.insereAntes(i, aluno); } for(ListaAluno::Iterador i = lista.primeiro(); i != lista.fim(); ++i) cout << i.item().nome() << ' ' << i.item().numero() << endl; }

Perfeito! Vimos que as listas, como estão, são muito simpáticas. Mas... Quando se insere sistematicamente a meio da lista (que é o caso do programa que fizemos) a implementação que usámos para as lista é muito ineficiente. Já viram o que acontece se um 100º aluno for uma Ana? Fica a primeira da lista... e isso implica empurrar todos os alunos que constam na lista...

E reparem que um problema semelhante ocorre quanto se removem itens que não estejam no fim da lista: é preciso puxar os que já lá estão para trás...

Que se pode fazer para melhorar a situação? Suponham que havia apenas três alunos na lista:

Ana Carlos Duarte

e que se pretende inserir o Berto antes do Carlos. Porque é que temos de empurrar o Carlos e o Duarte? Alguém sabe?

Discutir. Solução: porque assumimos que a ordem dos itens na lista era a mesma que a ordem dos itens na matriz...

E se a ordem dos elementos na matriz pudesse ser diferente da ordem dos itens na lista? Aí podía-se pôr o Berto no fim, sem deslocar ninguém...

Colocar.

Mas como saber então que o Berto está a seguir da Ana e antes do Carlos?

Discutir. Concluir que cada elemento da matriz precisa de saber onde está item seguinte da lista.

10.5.1 Cadeias ligadas

Como representar essa informação?

Discutir possibilidades. Concluir que se pode usar um índice em cada elemento da matriz!

Mas então, cada elemento da matriz guarda... Um item e um índice... Logo, já não basta uma matriz de índices...

Concluir que é necessária uma estrutura. Chamar-lhe Elo (da cadeia). Introduzir a noção de CADEIA LIGADA. Introduzir a noção de CADEIA SIMPLESMENTE LIGADA.

Construir a classe e usá-la na lista:

class ListaDeInt { public: ... private: struct Elo { Item item; int seguinte; }; Elo elos[número_máximo_de_itens]; ... };

Representar as ligações por setas no quadro!

E agora? Como implementar o método insereAntes()?

inline void ListaDeInt::insereAntes(Iterador& iterador, Item const& novo_item) { assert(not estáCheia()); ++número_de_itens;

??????? }

Discutir algoritmo. Admitir que há um método reserva() que procura um elo livre na e devolve o seu índice. Concluir onde se guarda o item e que fazer ao índice do seguinte.

inline void ListaDeInt::insereAntes(Iterador& iterador, Item const& novo_item) { assert(not estáCheia());

++número_de_itens;

int const i = reserva(); elos[i].item = item; elos[i].seguinte = iterador.índice; ?????? }

Tudo bem? Não! Porquê? O elo anterior não está ligado ao que acabámos de inserir! Temos de pôr o seu seguinte com o valor i! Mas onde está o elo anterior?

Discutir. Concluir que não é forçosamente o elemento da matriz com o índice anterior.

Que fazer então? Sugestões?

Discutir possibilidades. Concluir que a mais simples é fazer a cadeia DUPLAMENTE LIGADA.

Então é preciso alterar a classe Elo. Alterar.

struct Elo { Item item; int anterior; int seguinte; };

E agora o método insereAntes() fica... Discutir calmamente...

inline void ListaDeInt::insereAntes(Iterador& iterador, Item const& novo_item) { assert(not estáCheia());

++número_de_itens;

int const i = reserva(); elos[i].item = item; elos[i].seguinte = iterador.índice; int const anterior = elos[iterador.índice].anterior; elos[anterior].seguinte = i; // ou simplesmente: // elos[elos[iterador.índice].anterior].seguinte = i; ????? }

Falta alguma coisa? Representar as ligações para trás no quadro... Concluir que falta actualizar o índice do anterior na posição do iterador e do novo elo!

inline void ListaDeInt::insereAntes(Iterador& iterador, Item const& novo_item) { assert(not estáCheia());

++número_de_itens;

int const i = reserva(); elos[i].item = item; elos[i].anterior = elos[iterador.índice].anterior; elos[i].seguinte = iterador.índice; elos[elos[iterador.índice].anterior].seguinte = i; elos[iterador.índice].anterior = i; }

Perfeito! Perfeito? Bolas... E se o iterador se referir ao primeiro item da lista? Que instruções é que não se podem fazer? Pior, como é que se sabe onde a cadeia de itens começa, i.e., onde está o primeiro item da lista? Não é forçosamente o primeiro elemento da matriz!

Discutir assunto. Dizer para se lembrarem dos itens fictícios da lista.

Os itens fictícios da lista têm um característica importante: "existem" (entre aspas) sempre, mesmo com a lista vazia. Logo o problema resolve-se...

Discutir. Concluir que dá jeito dar existência real aos itens fictícios! Sugerir colocá-los no final da matriz, que passa a 102 elementos.

Mas então o que deve fazer o construtor da lista?

Discutir. Concluir que uma possibilidade é:

class ListaDeInt { ... private: ... static int const inicial = numero_máximo_de_itens; static int const final = numero_máximo_de_itens + 1; };

ListaDouble::ListaDouble() : número_de_itens(0) {

elos[inicial].seguinte = final; elos[final].anterior = inicial;

?????? // falta aqui qualquer coisa... }

O anterior do inicial e o seguinte do final podem ficar com lixo... Nunca são precisos!

E agora, o procedimento insere como funciona?

Concluir que funciona sempre sem problemas desde que... o iterador não se refira ao item (fictício) inicial...

inline void ListaDeInt::insereAntes(Iterador& iterador, Item const& novo_item) { assert(not estáCheia()); assert(iterador.índice != inicial);

++número_de_itens;

int const i = reserva(); elos[i].item = item; elos[i].anterior = elos[iterador.índice].anterior; elos[i].seguinte = iterador.índice; elos[elos[iterador.índice].anterior].seguinte = i; elos[iterador.índice].anterior = i; }

E o método remove()?

Discutir.

Concluir:

inline void ListaDouble::remove(Iterador& iterador) { assert(iterador.índice != inicial and iterador.índice != final);

–número_de_itens; int const i = iterador.índice; elos[elos[i].anterior].seguinte = elos[i].seguinte; elos[elos[i].seguinte].anterior = elos[i].anterior;

// Avançar itereador! iterador.índice = elos[i].seguinte;

liberta(i); }

Discutir método liberta().

Adiámos os problemas de reservar uma posição livre da matriz, quando se está a inserir um item, ou de libertar uma posição ocupada, quando se está a remover um item.

Discutir forma de saber que elementos da matriz estão livres. Concluir que uma boa forma é usar um índice para primeiro livre e em cada elemento livre usar o índice para seguinte e fazer cadeia simplesmente ligada sem guardas de elos livres. Notar que o construtor tem de ser alterado de modo a colocar todos os elementos da matriz (excepto as guardas) como elos livres. Deixar para as aulas práticas a discussão pormenorizada.

Discutir interface vs. implementação. Concluir que se melhorou a classe sem alterar a sua interface. Os programas que a usavam não precisam de ser alterados.

Deixar a implementação da parte restante da classe para as aulas práticas.

!!!!!!!!!!!!Ver código das teóricas, versão 2 das listas.

Listagens completas?

Capítulo 11

Ponteiros e variáveis dinâmicas

!!!Não esquecer de criar método listaAssociada() nos iteradores para poder melhorar a PC de insereAntes e remove!

!!!!!!!!!!!!Colocar resumos das aulas 4 e 5.

1. Ponteiros

resumo ou guião da aula 4

2. Listas e iteradores usando ponteiros

mudança simples das listas e iteradores do capítulo anterior

3. Variáveis dinâmicas

resumo ou guião da aula 5

4. Listas e ponteiros usando variáveis dinâmicas

5. Classes com variáveis dinâmicas

espalhar resumo ou guião da aula 6

Construtores e destrutores

Cópias: introdução às noções de igualdade e identidade.

6. Introdução às excepções

bad_alloc. Capturando. Problemas.

!! Falar de construtor por cópia: voltar ao contador de instâncias do capítulo sobre TAD.

Capítulo 12

Herança e polimorfismo

!!!!!O nome do capítulo deve reflectir a utilidade das coisas

Capítulo 13

Programação genérica

!!Expandir resumo da aula 11.

Capítulo 14

Excepções e tratamento de erros

!!Colocar aqui resumo de aula do ano passado. Adaptar de modo a deixar claro que excepções são geralmente mais apropriadas que asserções para lidar com erros do programador (a ideia é que o programa aborte na mesma, mas pode-se capturar a excepção e tentar fazer o programa morrer honradamente, salvando o que puder ser salvo).

Guião da 12ª aula teórica

Sumário

1.Tratamento básico de erros ocorridos durante a execução de programas não críticos: a.Origens dos erros: lógicos, utilizador, recursos externos. b.Tratamento de erros: instruções usuais (condicionais, de selecção, de iteração), asserções e excepções. c.Origens humanas imediatas (programador e utilizador) versus recursos externos (e.g., formato de ficheiros, existência de ficheiros, limitações de memória, etc.). d.Papeis do humano: programador fabricante, programador utilizador e utilizador final. e.Protecção contra erros lógicos (do programador): Utilização de asserções. Os erros devem ser detectados. São corrigidos pelo programador e não pelo próprio programa. Vantagens da utilização de assert. Pré-condições: o programador fabricante simplifica a vida do programador utilizador. Condições objectivo e invariantes (de ciclos ou de classe): o programador fabricante simplifica a sua própria vida. Utilização de assert durante o teste do programa. Desligando o efeito de assert sem retirar as instruções do código. Efeitos de uma asserção falhada: ficheiro core e sua utilidade. f.Protecção contra erros do utilizador: Princípio: o utilizador enganar-se-á muitas vezes e quando os efeitos forem mais nefastos. O utilizador é humano: repetição como solução típica. Os erros são detectados e corrigidos localmente. g.Protecção contra erros com origem em recursos externos: Filosofia: os recursos estão disponíveis e sem problemas excepto em casos excepcionais. Utilização de excepções. Os erros são detectados e são lançadas excepções. Só captura a excepção quem sabe lidar com o erro. A captura das excepções pode ser feita "muito longe" do código que as origina. Efeito de uma excepção não capturada: de novo o ficheiro core. Classes para excepções. Hierarquias de erros e polimorfismo.

Guião

!!!!!!!!!!!!!Isto tranformou-se num capítulo... Tornar capítulo mesmo e refazer guião...

Nesta aula vamos falar de tratamento básico de erros. Note-se que este é um assunto complicado em geral. Vamos assumir que estamos a desenvolver uma aplicação "normal". I.e., vamos admitir que uma falha do programa é muito inconveniente mas não é dramática (nem fatal). É o caso, por exemplo, de um processador de texto. Se fosse um programa de controlo de tráfego aéreo, de cálculo de trajectórias de mísseis, de controlo do arrefecimento de uma central nuclear, teríamos de usar estratégias muito diferentes e nada triviais.

Comecemos por identificar as possíveis fontes de erros num programa:

Discutir com eles! Concluir em:

1.Utilizador humano do programa. 2.Erros do programador (quer como programador utilizador quer como programador fabricante). 3.Recursos externos inadequados (errados, indisponíveis ou insuficientes).

Há um princípio básico na programação: o utilizador errará. Há outro princípio básico na programação: o utilizador falhará na pior ocasião (a única não prevista)

Como deve o programa lidar com erros do utilizador?

Discutir

É garantido que os utilizadores humanos do programa se vão enganar. É inevitável. Com estes erros devemos lidar explicitamente no programa verificando e lidando com os erros onde quer que eles possam ocorrer.

Há outro princípio na programação: o programador errará. Há ainda outro princípio na programação: alguns os erros do programador resistirão a todos os teste e revelar-se-ão apenas depois de o programa ter sido distribuido ao seu utilizador final...

Como se deve lidar com estes erros?

Discutir. Não é preciso concluir nada. Deixar para depois.

E os recursos externos? Estes são recursos que estão fora do controlo de programador e do programa propriamente ditos. Exemplos:

Falta de memória Falta de espaço em disco Falta de ficheiro Formato de ficheiro errado

Por outro lado, não é razoável pedir ao computador para corrigir estes erros, tal como se faria com um humano!

Como se deve lidar com estes erros?

Discutir. Não é preciso concluir nada. Deixar para depois.

Note-se que os erros têm muitas vezes origem humana. O utilizador do programa é tipicamente um humano. O programador, quer como programador utilizador quer como programador fabricante, é também um humano. Muitas vezes é o mesmo humano que desempenha alternada ou simultaneamente papéis diferentes! Um programador fabrica código usando código já escrito, compila e testa o programa, assumindo neste processo os três papéis.

O que temos de descobrir é como se deve lidar em geral com os vários tipos de erros. Como ferramentas temos as instruções de selecção, condicionais ou de iteração da própria linguagem, as asserções e as excepções. Quando é adequado usar cada uma destas ferramentas?

Vamos ver alguns programas de exemplo e estudar os possíveis erros que podem ocorrer no seu contexto.

Por exemplo, suponham um programa simples para cálculo duma raiz quadrada pelo método de Newton:

Explicar vagamente o método e dizer para eles deduzirem a expressão de progressão enquanto passo o programa no quadro. Deixar claro que não é a forma mais eficiente de calcular a raiz quadrada...

#include <iostream> #include <limits>

using namespace std;

// PC: v >= 0 // CO: v × (1 - e) < raiz × raiz < v × (1 + e), onde e (epsilon) é a diferença entre o // menor double maior que 1 e 1, representando portanto de alguma forma o limite mínimo // de erro alcançável. double raiz(double v) { if(v == 0.0) return 0.0; double r = v; double const e = numeric_limits<double>::epsilon(); while(true) { double nr = 0.5 * (r + v / r); if(nr * nr - v < e * v) return nr; r = nr; } }

int main() { cout << "Introduza um valor: "; double valor; cin >> valor; cout << "raiz = " << raiz(valor) << endl; }

Que acontece se o valor dado pelo utilizador for negativo? E se o utilizador pressionar 'a' em vez de um número?

Se o valor for negativo a função entra em ciclo infinito!

De quem é o problema? Do programador, obviamente! Ele deve garantir que as pré-condições da função se verificam. Mas qual programador? Quem programou a função? Ou quem programou o main? Bom, os dois, mas de formas totalmente diferentes.

Quem programa a função (o programador fabricante) não pode assumir muito acerca de futuras utilizações. Responsabiliza-se pelo valor devolvido pela função apenas se as pré-condições se verificarem. É esse o contrato que estabelece com o programador utilizador da função. Mas pode ser simpático e garantir que, enquanto o programa estiver em teste, a passagem de argumentos que não cumpram a PC levem à terminação do programa com uma mensagem apropriada. Para isso usam-se asserções.

Note-se que o programador da função está a prepará-la para detectar erros de outros programadores, os programadores utilizadores (que podem ser a mesma pessoa)! A função não sabe nada acerca do humano que usa o programa! Assim:

#include <iostream> #include <limits> #include <cassert>

using namespace std;

// PC: v >= 0 // CO: v × (1 - e) < raiz × raiz < v × (1 + e), onde e (epsilon) é a diferença entre o // menor double maior que 1 e 1, representando portanto de alguma forma o limite mínimo // de erro alcançável. double raiz(double v) { assert(v >= 0.0); if(v == 0.0) return 0.0; double r = v; double const e = numeric_limits<double>::epsilon(); while(true) { double nr = 0.5 * (r + v / r); if(nr * nr - v < e * v) return nr; r = nr; } }

int main() { cout << "Introduza um valor: "; double valor; cin >> valor; cout << "raiz = " << raiz(valor) << endl; }

Desse modo, o código fornecedor da função contém uma protecção contra erros no código cliente.

É importante perceber que a colocação de uma asserção para verificação das pré-condições da função serve para proteger o programador utilizador dessa função dos seus próprios erros. Por outro lado, é possível também o programador fabricante proteger-se dos seus próprios erros colocando uma asserção para verificação das condições objectivo da função.

Neste caso a CO é razoavelmente complicada, pelo que não se entra em pormenores:

#include <iostream> #include <limits> #include <cassert>

using namespace std;

// PC: v >= 0 // CO: v × (1 - e) < raiz × raiz < v × (1 + e), onde e (epsilon) é a diferença entre o // menor double maior que 1 e 1, representando portanto de alguma forma o limite mínimo // de erro alcançável. double raiz(double v) { assert(v >= 0.0); if(v == 0.0) return 0.0; double r = v; double const e = numeric_limits<double>::epsilon(); while(true) { double nr = 0.5 * (r + v / r); if(nr * nr - v < e * v) { assert(v - nr * nr < e * v && nr * nr - v < e * v); return nr; } r = nr; } }

int main() { cout << "Introduza um valor: "; double valor; cin >> valor; cout << "raiz = " << raiz(valor) << endl; }

Desta forma se o programador fabricante se enganar ao escrever a função, a asserção falhará e assinalará que a aproximação não é suficientemente boa.

As asserções têm uma vantagem adicional: se falharem o programa termina não apenas com uma mensagem de erros mais ou menos inteligível mas também gerando um ficheiro de "core". Este ficheiro contém uma imagem completa do processo no momento em que a asserção falhou. Esta imagem pode ser usada para determinar ou pelo menos para facilitar a determinação das causas para a falha. Para isso basta executar o depurador e dizer-lhe para carregar a imagem do processo. Se o programa se chamasse "raiz", por exemplo, bastava fazer:

gdb raiz core core

A partir deste instante é possível verificar o valor de todas as variáveis no momento da falha. Para isso pode ser útil usar o comando up, que sobe um nível na hierarquia de chamadas de funções e procedimentos, até que a função ou procedimento mostrado seja um dos que estão em desenvolvimento.

Agora, se o utilizador do programa introduzir um valor negativo o programa aborta com uma mensagem antipática. Isso é útil durante o teste do programa. Mas muito indesejável para o seu utilizador! De quem é a culpa? Do programador fabricante de main() e utilizador de raiz()! Ele deveria verificar se o valor dado pelo utilizador é aceitável! A asserção tem como grande utilidade mostrar claramente ao programador de main() que há algo de errado com o seu código. Ele vai verificar este erro durante os testes do programa. Presumivelmente antes de entregar o programa ao utilizador final.

Note-se que há aqui três papeis diferentes: o programador fabricante de raiz(), o programador utilizador de raiz() e o utilizador do programa! Durante o desenvolvimento do programa estes podem ser papeis tomados alternadamente pela mesma pessoa.

A solução passa pois por reconhecer que o utilizador final se pode enganar e escrever o programa à prova desses erros!

#include <iostream> #include <limits> #include <cassert>

using namespace std;

// PC: v >= 0 // CO: v $\times$ (1 - e) < raiz $\times$ raiz < v $\times$ (1 + e), onde e (epsilon) é a diferença entre o // menor double maior que 1 e 1, representando portanto de alguma forma o limite mínimo // de erro alcançável. double raiz(double v) { assert(v >= 0.0); if(v == 0.0) return 0.0; double r = v; double const e = numeric_limits<double>::epsilon(); while(true) { double nr = 0.5 * (r + v / r); if(nr * nr - v < e * v) { assert(v - nr * nr < e * v && nr * nr - v < e * v); return nr; } r = nr; } }

double lePositivo() { while(true) { double valor; cout << "Introduza um valor: "; cin >> valor; if(valor >= 0.0) return valor; cout << "O valor tem de ser >= 0!" << endl; } }

int main() { double valor = lePositivo(); cout << "raiz = " << raiz(valor) << endl; }

Fazer uma nota acerca dos estranhos ciclos! Explicar que estam entre um while e um do while. Não é uma heresia...

Feito isto e testado o programa, ficou garantido que a função raiz será chamada com um valor não negativo. Nesse caso a verificação das asserções já não são necessárias! Logo, pode-se retirá-la do código. Hmmm..... Será boa ideia? Não! O programa ainda pode precisar de acrescentos, melhorias, correcções, e portanto de ser testado de novo! Se isso acontecer dá muito jeito que a asserção lá continue! Como resolver o problema então? É ineficiente estar lá quando o programa está testado mas é conveniente que lá continue porque podemos precisar de testar o programa de novo!

Solução: desligar as asserções. Como? Compilando com a opção -DNDEBUG (no debug).

Para acelerar o teste de uma aplicação é comum distribuir versões não finais, as chamadas versões alfa ou beta, a utilizadores seleccionados. Estas versões devem ser distribuidas sem desligar as asserções. É que se pode pedir aos utilizadores para, em caso de erro, enviarem uma mensagem de correio electrónico com: 1. Mensagem de erro gerada (identifica a asserção falhada). 2. Ficheiro imagem: permite inspecção do estado do programa. 3. Descrição breve dos passos que geraram o erro: facilita perceber como se atingiu o estado de erro.

Problema resolvido... De certeza? E se o utilizador escrever uma letra? Aí a leitura falha! E pior, o canal fica num estado de erro em que todas as tentativas de leituras posteriores falham também... Fica em ciclo infinito!

Explicar.

Solução:

#include <iostream> #include <limits> #include <cassert>

using namespace std;

// PC: v >= 0 // CO: v $\times$ (1 - epsilon) < raiz * raiz < v $\times$ (1 + epsilon), onde epsilon é a // diferença entre o menor double maior que 1 e 1, representando // portanto de alguma forma o limite mínimo de erro alcançável. double raiz(double v) { assert(v >= 0.0); if(v == 0.0) return 0.0; double r = v; double const e = numeric_limits<double>::epsilon(); while(true) { double nr = 0.5 * (r + v / r); if(nr * nr - v < e * v) { assert(v - nr * nr < e * v && nr * nr - v < e * v); return nr; } r = nr; } }

// Ver Slang++ para uma solução mais genérica e elegante... void ignoraLinha() { cin.clear(); char c; while(cin.get(c) && c != '\n') ; }

double lePositivo() { while(true) { double valor; cout << "Introduza um valor: "; if(!(cin >> valor)) { ignoraLinha(); cout << "Tem de ser um número real!" << endl; } else if(valor >= 0.0) return valor; else cout << "O valor tem de ser >= 0!" << endl; } }

int main() { double valor = lePositivo(); cout << "raiz = " << raiz(valor) << endl; }

Mas, e a terceira fonte de erros? Erros devido a recursos externos ao programa? Exemplos? Falta de memória, falta de espaço em disco, ficheiro inexistente, ficheiro com formato errado... Há aqui algo que me cheira a trabalho final...

!!!!!!!!!Este exemplo baseia-se no trabalho final de 1999/2000. Pode precisar de ser refeito em anos posteriores...

!!!!!!!!!!!Ao passar este exemplo para folhas teóricas é conveniente basear-me na hierarquia de formas desenvolvida nos capítulos anteriores. Talvez essa hierarquia de formas deva passar a usar o Slang++....

Suponhamos que queríamos escrever uma classe Polígono usando o Slang++ e que essa classe deveria estar integrada numa hierarquia de conceitos encimada pela classe abstracta Forma:

!!!!!!!!!Nas folhas teóricas pôr Slang::.

class Forma /* simplificada... */ { public: // Com origem em origem: Forma(Posicao const& origem);

// Tem de haver sempre no topo das hierarquias polimórficas: virtual ~Forma() {}

// Todos os tipos concretos de formas têm um nome: virtual string const nomeDoTipo() const = 0;

// Todas as formas podem ser desenhadas no ecrã: virtual desenha(bool seleccionada = false) = 0;

// Devolve origem da forma: Posicao const& origem() const;

// Move a forma estabelecendo uma nova origem: void movePara(Posicao const& nova_origem);

// Move a forma deslocando a sua origem: void moveDe(Posicao const& deslocamento);

private: Posicao origem_; };

class Polígono : public Forma { public: // Sem vértices com origem em origem: Poligono(Posicao const& origem);

// Todos os tipos concretos de formas têm um nome: virtual string const nomeDoTipo() const;

// O vertice v tem coordenadas absolutas... void novoVértice(Posicao const& v);

virtual void desenha(bool seleccionado = false); ...

private: list<Posicao> vertices; };

A ideia é que um polígono é representado pela sequência dos seus vértices, admitindo-se que cada vértice se liga por uma aresta aos vértices adjacentes na sequência, sendo que o primeiro e o último vértice se consideram adjacentes. Note-se que a posição dos vértices é relativa à origem do polígono, que é uma posição convencional.

Mas pretendíamos mais. Deveria ser possível carregar e guardar polígonos de e em ficheiros... Para isso acrescentaríamos:

1.Um construtor para construir um polígono a partir de informação lida de um canal. 2.Um procedimento carrega() para carregar informação de um canal, descartando a informação anteriormente guardada no polígono. 3.Um procedimento guarda() para guardar a informação escrevendo num canal. 4.E faríamos o mesmo a todas as classes da hierarquia...

Ou seja:

class Forma { public: // Com origem em origem: Forma(Posicao const& origem); // Lida de canal: Forma(istream& entrada);

// Tem de haver sempre no topo das hierarquias polimórficas: virtual ~Forma() {}

// Todos os tipos concretos de formas têm um nome: virtual string const nomeDoTipo() const = 0;

// Procedimentos para guardar e carregar: virtual void carrega(istream& entrada); virtual void guarda(ostream& saida) const;

// Todas as formas podem ser desenhadas no ecrã: virtual desenha(bool seleccionada = false) = 0;

// Devolve origem da forma: Posicao const& origem() const;

// Move a forma estabelecendo uma nova origem: void movePara(Posicao const& nova_origem);

// Move a forma deslocando a sua origem: void moveDe(Posicao const& deslocamento);

private: Posicao origem_; };

class Polígono : public Forma { public: // Sem vértices com origem em origem: Poligono(Posicao const& origem);

// Lido de canal: Poligono(istream& entrada);

// Todos os tipos concretos de formas têm um nome: virtual string const nomeDoTipo() const;

// Procedimentos para guardar e carregar: virtual void carrega(istream& entrada); virtual void guarda(ostream& saida) const;

virtual void desenha(bool seleccionado = false) const;

// O vertice v tem coordenadas absolutas... void novoVértice(Posicao const& v);

private: list<Posicao> vertices; };

Discutir implementação.

inline Forma::Forma(Posicao const& origem) : origem_(origem) { }

inline Forma::Forma(istream& entrada) : origem_(entrada) { // Que acontece se o construtor de Posicao falhar? }

inline void Forma::carrega(istream& entrada) { origem_.carrega(entrada); // Que acontece se o procedimento Posicao::carrega() falhar? }

inline void Forma::guarda(ostream& saida) { origem_.guarda(saida); // Que acontece se o procedimento Posicao::guarda() falhar? }

inline Posicao const& Forma::origem() const { return origem_; }

inline void Forma::movePara(Posicao const& nova_origem) { origem_ = nova_origem; }

inline void Forma::moveDe(Posicao const& deslocamento) { origem += deslocamento; }

inline Poligono::Poligono(Posicao const& origem) : Forma(origem) { }

Poligono::Poligono(istream& entrada) : Forma(entrada) // Que acontece se o construtor de Forma falhar? { int numero_de_vertices; if(!(entrada >> numero_de_vertices)) /* Que fazer se a extracção do número de vértices falhar? */; while(numero_de_vertices– != 0) vertices.push_back(Posicao(entrada)); // Que acontece se o construtor de Posicao falhar? }

inline string const Poligono::nomeDoTipo() const { return "Poligono"; }

void Poligono::carrega(istream& entrada) { Forma::carrega(entrada); // Que acontece se o procedimento Forma::carrega() falhar?

vertices.clear();

int numero_de_vertices; if(!(entrada >> numero_de_vertices)) /* Que fazer se a extracção do número de vértices falhar? */; while(numero_de_vertices– != 0) vertices.push_back(Posicao(entrada)); // Que acontece se o construtor de Posicao falhar? }

void Poligono::guarda(ostream& saida) const { Forma::guarda(saida); // Que acontece se o procedimento Forma::guarda() falhar?

if(!(saida << vertices.size())) /* Que fazer se a inserção do número de vértices falhar? */; for(list<Posicao>::const_iterator i = vertices.begin(); i != vertices.end(); ++i) i->guarda(saida)); // Que acontece se o procedimento Posicao::guarda() falhar? }

inline void Poligono::novoVertices(Posicao const& v) { // O vertice v tem coordenadas absolutas, pelo que há que lhe retirar a origem do polígono: vertices.push_back(v - origem()); }

void Poligono::desenha(bool seleccionado) const { ... }

Para evitar repetições de código pode-se factorizar o código repetido num procedimento privado auxiliar da classe Poligono:

class Forma { public: // Com origem em origem: Forma(Posicao const& origem); // Lida de canal: Forma(istream& entrada);

// Tem de haver sempre no topo das hierarquias polimórficas: virtual ~Forma() {}

// Todos os tipos concretos de formas têm um nome: virtual string const nomeDoTipo() const = 0;

// Procedimentos para guardar e carregar: virtual void carrega(istream& entrada); virtual void guarda(ostream& saida) const;

// Todas as formas podem ser desenhadas no ecrã: virtual desenha(bool seleccionada = false) = 0;

// Devolve origem da forma: Posicao const& origem() const;

// Move a forma estabelecendo uma nova origem: void movePara(Posicao const& nova_origem);

// Move a forma deslocando a sua origem: void moveDe(Posicao const& deslocamento);

private: Posicao origem_; };

class Polígono : public Forma { public: // Sem vértices com origem em origem: Poligono(Posicao const& origem);

// Lido de canal: Poligono(istream& entrada);

// Todos os tipos concretos de formas têm um nome: virtual string const nomeDoTipo() const;

// Procedimentos para guardar e carregar: virtual void carrega(istream& entrada); virtual void guarda(ostream& saida) const;

virtual void desenha(bool seleccionado = false) const;

// O vertice v tem coordenadas absolutas... void novoVértice(Posicao const& v);

private: list<Posicao> vertices;

void carregaEspecifico(istream& entrada); };

inline Forma::Forma(Posicao const& origem) : origem_(origem) { }

inline Forma::Forma(istream& entrada) : origem_(entrada) { // Que acontece se o construtor de Posicao falhar? }

inline void Forma::carrega(istream& entrada) { origem_.carrega(entrada); // Que acontece se o procedimento Posicao::carrega() falhar? }

inline void Forma::guarda(ostream& guarda) { origem_.guarda(saida); // Que acontece se o procedimento Posicao::guarda() falhar? }

inline Posicao const& Forma::origem() const { return origem_; }

inline void Forma::movePara(Posicao const& nova_origem) { origem_ = nova_origem; }

inline void Forma::moveDe(Posicao const& deslocamento) { origem += deslocamento; }

inline Poligono::Poligono(Posicao const& origem) : Forma(origem) { }

inline Poligono::Poligono(istream& entrada) : Forma(entrada) { // Que acontece se o construtor de Forma falhar? carregaEspecifico(entrada); // Que acontece se o procedimento carregaEspecifico() falhar? }

inline string const Poligono::nomeDoTipo() const { return "Poligono"; }

void Poligono::carrega(istream& entrada) { Forma::carrega(entrada); // Que acontece se o procedimento Forma::carrega() falhar?

vertices.clear();

carregaEspecifico(entrada); }

void Poligono::guarda(ostream& saida) const { Forma::guarda(saida); // Que acontece se o procedimento Forma::guarda() falhar?

if(!(saida << vertices.size())) /* Que fazer se a inserção do número de vértices falhar? */; for(list<Posicao>::const_iterator i = vertices.begin(); i != vertices.end(); ++i) i->guarda(saida)); // Que acontece se o procedimento Posicao::guarda() falhar? }

void Poligono::carregaEspecifico(istream& entrada) { int numero_de_vertices; if(!(entrada >> numero_de_vertices)) /* Que fazer se a extracção do número de vértices falhar? */; while(numero_de_vertices– != 0) vertices.push_back(Posicao(entrada)); // Que acontece se o construtor de Posicao falhar? }

inline void Poligono::novoVertices(Posicao const& v) { // O vertice v tem coordenadas absolutas, pelo que há que lhe retirar a origem do polígono: vertices.push_back(v - origem()); }

void Poligono::desenha(bool seleccionado) const { ... }

Problemas: que fazer quando a leitura ou a escrita falham?

Em primeiro lugar há que reconhecer que as causas para uma falha são normalmente externas: os recursos usados para leitura e escrita são tipicamente ficheiros. Estes recursos externos não estão totalmente sob o controlo do programador...

Por um lado é claro que se uma leitura de um canal falhar não se pode pedir de novo os dados, pelo simples facto de não se poder assumir que existe um ente inteligente do outro lado do canal: a maior parte das vezes o canal estará ligado a um ficheiro que, se não tiver o formato correcto, dificilmente se auto-corrigirá...

Por outro lado também não é aceitável que o programa simplesmente aborte quando alguma leitura falhar! Se você fosse utilizador de um programa tão temperamental certamente rogaria pragas ao seu programador.

Logo, é necessário que as funções e os procedimentos assinalem os possíveis erros para que o código onde são invocados possa recuperar desses erros. Uma solução clássica mas problemática é fazer os procedimentos devolver um valor booleano verdadeiro se tudo tiver corrido bem e falso no caso contrário... Por exemplo:

inline bool Forma::carrega(istream& entrada) { return origem_.carrega(entrada); // Que acontece se o procedimento Posicao::carrega() falhar? Devolve-se false. }

bool Poligono::carrega(istream& entrada) { if(!Forma::carrega(entrada)) return false; // Que acontece se o procedimento Forma::carrega() falhar? Devolve-se false.

vertices.clear();

return carregaEspecifico(entrada); }

bool Poligono::carregaEspecifico(istream& entrada) { int numero_de_vertices; if(!(entrada >> numero_de_vertices)) return false; while(numero_de_vertices– != 0) vertices.push_back(Posicao(entrada)); // Opps... Construtores não podem devolver valores... // Que acontece se o construtor de Posicao falhar? return true; }

Esta solução tem dois graves problemas:

1.Não é aplicável no caso dos construtores, pois não devolvem qualquer valor. (Também se pode argumentar que devolvem o objecto construído, mas nesse caso é claro que não podem devolver um booleano...) 2.Obriga o programador utilizador do código a verificar sistematicamente o valor devolvido pelos procedimentos. Como isso é muito aborrecido, o programador tende a "distrair-se", e os erros ficam por tratar... 3.Todas as funções e procedimentos tem de lidar com os erros explicitamente: ou recuperando desses erros ou limitando-se a assinar a sua ocorrência devolvendo um valor falso. Toda a sequência de funções e procedimentos invocados desde o local onde ele pode ser corrigido (recuperação) até à sua origem têm de lidar desta forma com os erros. Note-se que o local onde se pode corrigir um erro (recuperar) e o local onde se detecta o erro podem estar muito distantes... Isto é mais uma vez muito maçador e portanto leva a esquecimentos fatais pelos programadores.

Discutir soluções!

A solução ideal é a utilização de excepções. As vantagens são:

1.Podem-se lançar excepções inclusive nos construtores. 2.O programador utilizador não é obrigado a lidar com as excepções. Se ninguém capturar uma excepção lançada o programa aborta gerando um ficheiro de imagem core, que pode ser analisado como indicado acima para verificar as razões que levaram ao lançamento da excepção. 3.Só lida com as excepções quem sabe o que fazer com elas! Assim, código que não sabe que fazer caso seja lançada uma excepção limita-se a não se preocupar: ou algum código mais exterior se encarregará de recuperar dessa excepção ou o programa abortará. Só é obrigado a lidar com uma excepção o código que pretende recuperar do erro que a originou.

O Slang++ já utiliza excepções. Os procedimentos para guardar informação em canais lançam a excepção ErroAoGuardar em caso de erro e os procedimentos para carregar lançam a excepção ErroAoCarregar. Essas classes fazem parte de uma pequena hierarquia de excepções com o seguinte aspecto:

// Esta classe serve de base a uma pequena hierarquia de classes representando excepções. class Erro { public: // Construtor da classe. mensagem é uma mensagem explicando a origem da excepção. Erro(std::string const& mensagem) : mensagem(mensagem) { }

// Destrutor virtual para poder sofrer derivações... virtual ~Erro() { }

/* Inspector da mensagem explicando a origem da excepção na forma de uma conversão implícita para std::string. */ virtual operator std::string () const { return mensagem; }

private: // A mensagem explicando a origem da excepção. std::string mensagem; };

// Esta classe serve para representar excepções de carregamento de objectos a partir de canais. class ErroAoCarregar : public Erro { public: // Construtor da classe. classe é o nome da classe que originou a excepção. ErroAoCarregar(std::string const& classe) : Erro(std::string("Erro ao carregar '") + classe + "'") { } };

// Esta classe serve para representar excepções ao guardar objectos usando canais. class ErroAoGuardar : public Erro { public: // Construtor da classe. classe é o nome da classe que originou a excepção. ErroAoGuardar(std::string const& classe) : Erro(std::string("Erro ao guardar '") + classe + "'") { } };

Assim, deve-se acrescentar código para lançar excepções apropriadas em caso de erro:

class Forma { public: // Com origem em origem: Forma(Posicao const& origem); // Lida de canal: Forma(istream& entrada);

// Tem de haver sempre no topo das hierarquias polimórficas: virtual ~Forma() {}

// Todos os tipos concretos de formas têm um nome: virtual string const nomeDoTipo() const = 0;

// Procedimentos para guardar e carregar: virtual void carrega(istream& entrada); virtual void guarda(ostream& saida) const;

// Todas as formas podem ser desenhadas no ecrã: virtual desenha(bool seleccionada = false) = 0;

// Devolve origem da forma: Posicao const& origem() const;

// Move a forma estabelecendo uma nova origem: void movePara(Posicao const& nova_origem);

// Move a forma deslocando a sua origem: void moveDe(Posicao const& deslocamento);

private: Posicao origem_; };

class Polígono : public Forma { public: // Sem vértices com origem em origem: Poligono(Posicao const& origem);

// Lido de canal: Poligono(istream& entrada);

// Todos os tipos concretos de formas têm um nome: virtual string const nomeDoTipo() const;

// Procedimentos para guardar e carregar: virtual void carrega(istream& entrada); virtual void guarda(ostream& saida) const;

virtual void desenha(bool seleccionado = false) const;

// O vertice v tem coordenadas absolutas... void novoVértice(Posicao const& v);

private: list<Posicao> vertices;

void carregaEspecifico(istream& entrada); };

Forma::Forma(Posicao const& origem) : origem_(origem) { }

Forma::Forma(istream& entrada) : origem_(entrada) { // E se for lançada uma excepção no construtor de Posicao? Deixa-se passar... }

void Forma::carrega(istream& entrada) { origem_.carrega(entrada); // E se for lançada uma excepção no procedimento Posicao::carrega()? Deixa-se passar... }

void Forma::guarda(ostream& saida) { origem_.guarda(saida); // E se for lançada uma excepção no procedimento Posicao::guarda()? Deixa-se passar... }

Posicao const& Forma::origem() const { return origem_; }

void Forma::movePara(Posicao const& nova_origem) { origem_ = nova_origem; }

void Forma::moveDe(Posicao const& deslocamento) { origem += deslocamento; }

inline Poligono::Poligono(Posicao const& origem) : Forma(origem) { }

Poligono::Poligono(istream& entrada) : Forma(entrada) // E se for lançada uma excepção no construtor de Forma? Deixa-se passar... { carregaEspecifico(entrada); // E se for lançada uma excepção no procedimento carregaEspecifico()? Deixa-se passar... }

inline string const Poligono::nomeDoTipo() const { return "Poligono"; }

void Poligono::carrega(istream& entrada) { Forma::carrega(entrada); // E se for lançada uma excepção no procedimento Forma::carrega()? Deixa-se passar...

vertices.clear();

carregaEspecifico(entrada); // E se for lançada uma excepção no procedimento carregaEspecifico()? Deixa-se passar... }

void Poligono::guarda(ostream& saida) const { Forma::guarda(saida); // E se for lançada uma excepção no procedimento Forma::guarda()? Deixa-se passar...

// Em caso de erro na inserção lança-se explicitamente uma excepção! Note-se que as // inserções (operador <<) normalmente não lançam excepções, assinalando erro ao colocar o // canal em estado de erro, que tem de ser verificado explicitamente. if(!(saida << vertices.size())) throw ErroAoGuardar("Poligono");

for(list<Posicao>::const_iterator i = vertices.begin(); i != vertices.end(); ++i) i->guarda(saida)); // E se for lançada uma excepção no procedimento Posicao::guarda()? Deixa-se passar... }

void Poligono::carregaEspecifico(istream& entrada) { // Em caso de erro na extracção lança-se explicitamente uma excepção! Note-se que as // extracções (operador >>) normalmente não lançam excepções, assinalando erro ao colocar o // canal em estado de erro, que tem de ser verificado explicitamente. int numero_de_vertices; if(!(entrada >> numero_de_vertices)) throw ErroAoCarregar("Poligono");

while(numero_de_vertices– != 0) vertices.push_back(Posicao(entrada)); // E se for lançada uma excepção no construtor de Posicao? Deixa-se passar... }

inline void Poligono::novoVertices(Posicao const& v) { // O vertice v tem coordenadas absolutas, pelo que há que lhe retirar a origem do polígono: vertices.push_back(v - origem()); }

void Poligono::desenha(bool seleccionado) const { ... }

Note-se:

1.Se o construtor da Forma falhar será lançada uma excepção! O construtor de Forma limita-se a construir uma posição. Essa construção é que lança a excepção. O programador da classe Forma não tem de se preocupar com o assunto! 2.Idem se houverem erros nos procedimentos Forma::carrega() e Forma::guarda(). 3.Os procedimentos lançam excepções explicitamente caso ocorra algum erro que não resulte no lançamento de uma uma excepção. É o caso dos procedimentos Poligono::guarda() e Poligono::carregaEspecifico(). 4.Os procedimentos não recuperam de erros. Se detectarem erros, lançam excepções. Se os erros forem em funções ou procedimentos por eles invocados, limitam-se a ignorar as excepções por eles lançados.

Quem é responsável por recuperar dos erros? Apenas que entender dever fazê-lo. Suponha-se uma classe Figura que representa o conceito de Figura como uma agregação de Formas:

class Figura { public: Figura(istream& entrada);

void carrega(istream& entrada); void guarda(ostream& saida);

...

void desenha() const;

private: list<Forma*> formas; };

inline Figura::Figura(istream& entrada) { carrega(entrada); }

void Figura::carrega(istream& entrada) { // Em caso de erro na extracção lança-se explicitamente uma excepção! Note-se que as // extracções (operador >>) normalmente não lançam excepções, assinalando erro ao colocar o // canal em estado de erro, que tem de ser verificado explicitamente. int numero_de_formas; if(!(entrada >> numero_de_formas)) throw ErroAoCarregar("Figura");

while(numero_de_formas– != 0) { string tipo_de_forma; if(!(entrada >> tipo_de_forma)) throw ErroAoCarregar("Figura"); if(tipo_de_forma == "Poligono") formas.push_back(new Poligono(entrada)); else if(tipo_de_forma == ...) ... else // Tipo de figura desconhecido. Talvez merecesse uma excepção mais específica... throw ErroAoCarregar("Figura"); } // E se for lançada uma excepção no construtor de Poligono? Deixa-se passar... }

void Figura::guarda(ostream& saida) const { // Em caso de erro na inserção lança-se explicitamente uma excepção! Note-se que as // inserções (operador <<) normalmente não lançam excepções, assinalando erro ao colocar o // canal em estado de erro, que tem de ser verificado explicitamente. if(!(saida << formas.size())) throw ErroAoGuardar("Figura");

for(list<Forma*>::const_iterator i = formas.begin(); i != formas.end(); ++i) { if(!(saida << (*i)->nomeDoTipo())) throw ErroAoGuardar("Figura"); (*i)->guarda(saida)); // E se for lançada uma excepção no procedimento Forma::guarda()? Deixa-se passar... } }

Repare-se como se resolveu o problema de carregar e guardar os objectos usando o nome do seu tipo (é o problema da persistência dos objectos).

Suponha-se ainda uma classe Editor, com a responsabilidade de criar e editar figuras:

class Editor { public: ...

void carrega(); void guarda() const;

... private: bool figura_alterada; Figura* figura; };

void Editor::carrega() { string nome = CaixaDeTexto("Ficheiro a carregar:").executa(); if(ifstream entrada(nome.c_str())) { try { Figura* nova_figura = new Figura(entrada); if(figura_alterada && MenuSimNao("Quer guardar a figura corrente?").executa()) guarda(); delete figura; figura = nova_figura; } catch(Erro& e) { Aviso(e).executa(); } catch(bad_alloc) { Aviso("Memória insuficiente!").executa(); } catch(...) { Aviso("Erro desconhecido ao carregar figura!").executa(); } } else Aviso("Não consegui abrir!").executa(); }

É claro que a classe Figura não quer lidar com possíveis erros de carregamento, por exemplo. Que poderia ela fazer? Por outro lado a classe Editor tem obrigação de fazer alguma coisa. Essa classe é responsável, presume-se, pela interação com o utilizador do programa e por isso tem a responsabilidade de capturar excepções durante o carregamento de figuras, se elas ocorrerem, e delicadamente avisar o utilizador do programa e dar-lhe a chance de prosseguir o seu trabalho.

Note-se a estratégia seguida pelo editor: a figura anterior só é descartada quando a nova foi carregada com sucesso. Assim, em caso de falha, o utilizador é avisado e continua a editar a figura anterior.

É importante perceber-se que as excepções podem, e muitas vezes devem, ser capturadas bastante longe do local onde são lançadas. Repare-se no que sucede se ocorrer um erro na leitura da origem de um polígono da nova figura. A excepção é lançada pelo construtor da classe

Posicao (do Slang++), invocado no construtor de Forma, invocado no construtor de Poligono, invocado no construtor de Figura, invocado no procedimento carrega() da classe Editor, que finalmente a captura!

Note-se também que a herança permite a definição de hierarquias de excepções que podem ser usadas depois para capturas mais específicas ou mais genéricas. Neste caso fez-se a captura de excepções da classe genérica Erro, que se presume representarem qualquer possível tipo de erro do Slang++ (note-se a declaração da excepção capturada como referência de modo a poder fazer uso do polimorfismo). Como podem ser lançadas excepções de outros tipos, nomeadamente bad_alloc se não houver memória ao criar alguma variável dinâmica, fazem-se capturas em sequência, chegando-se ao extremo de capturar qualquer excepção que não seja capturada pelos catch anteriores.

!!!!!!!!!!!!!!!Daqui para baixo são conceitos avançados!

O que se disse até agora é só parte da verdade... É que se uma excepção for lançada a meio de uma operação complexa (e.g., um carregamento) pode haver objectos que fiquem num estado inválido! Isso é muito perigoso! Esta questão tem de ser resolvida pelos programadores fabricantes. Estes não se podem limitar a lançar excepções em caso de erro e ignorar as excepções com que não queiram lidar... Podem fazê-lo mas apenas se garantirem que todos os objectos ficam num estado aceitável. A isto chama-se "segurança face a excepções" [stroustrup]. Segundo Stroustrup [??] há três níveis de segurança face a excepções que podem ser garantidos pelos programadores fabricantes de uma dada função ou procedimento membro ou não membro:

1.Nível básico: todos os objectos ficam num estado válido (ou seja, verificando a CII da sua classe). 2.Nível forte: ou há total sucesso ou é lançada uma excepção e a função ou procedimento não tem quaisquer efeitos, deixando os objectos exactamente no mesmo estado que tinham antes do seu início. 3.Sem lançamento: a função ou procedimento não lançará nenhuma excepção (directa ou indirectamente), tendo sempre sucesso.

Em nenhum dos casos são aceitáveis fugas de memória, como é óbvio.

É claro que se deve lutar sempre por dar as garantias mais fortes! Mas isso não deve ser feito à custa de uma complexidade exagerada dos programas ou de grandes perdas de eficiência.

Para ajudar neste processo, a linguagem C++ faz algumas garantias. A mais importante delas diz que ao se abandonar uma função ou procedimento devido a uma excepção todos os objectos locais construído são automaticamente destruídos. Claro está que esta garantia significa que se for lançada uma excepção num construtor, o objecto em construção não será deixado semi-construído: todos os atributos (variáveis e constantes membros) entretanto construídos, assim como as classes base, serão automaticamente destruídos. Estas garantias são simpáticas, mas não resolvem todos os problemas. Variáveis dinâmicas não são destruídas automaticamente, por exemplo. Em geral, qualquer recurso externo reservado numa função ou procedimento no qual seja lançada uma excepção (explicita ou implicitamente) devem ser libertados sob risco de haver fugas...

A conclusão a que se chega, portanto, é que a interface de uma função ou procedimento não consiste apenas no seu cabeçalho (que indicam como se usa) e das pré-condição e condição objecto (que indicam o que se compromete a fazer e em que circunstâncias): inclui também informação acerca das garantias de segurança face a excepções (que indicam o que sucede aos objectos envolvidos em casos excepcionais).

Analisemos as garantias que são dadas no código apresentado:

Classe Forma

Se, num construtor da classe Forma, for lançada uma excepção na construção do atributo origem_, a construção não tem sucesso. O bom comportamento da classe Forma face a erros durante a construção do atributo origem_ depende por isso do bom comportamento dos construtors da classe Posicao.

E durante o procedimento carrega() da classe Forma? Como a classe Forma se limita a invocar o procedimento carrega() da classe Posicao, tudo depende do bom comportamento desse procedimento.

Acontece que quer os construtores da classe Posicao quer o seu procedimento carrega() oferecem o nível mais forte de segurança face ao lançamento de excepções (esta garantia faz parte da interface da classe). Assim, também os construtores da classe Forma e o seu procedimento carrega() oferecem o o nível mais forte de segurança face a excepções.

E o procedimento guarda()? Aqui o problema é diferente. É que guardar não altera senão o canal onde se escreve: o objecto a guardar nunca é alterado. Como não é possível voltar atrás no que se escreveu num canal de saída genérico, a ideia é que se um procedimento guarda() falhar, deve-se considerar como lixo tudo o que foi escrito no canal... No fundo isto significa que, do ponto de vista do objecto forma, o procedimento oference o nível forte de segurança, mas do ponto de vista do canal não, limitando-se a oferecer o nível básico de segurança.

Calma... E o canal de entrada no caso do construtor a partir de um canal e do procedimento carrega(), ambos da classe Forma? Não se pode simplesmente "des-ler" aquilo que já foi lido (por acaso até pode, mas com limitações). Por isso, também essas operações oferecem apenas o nível básico de segurança relativamente ao canal de entrada...

Cientes das nuances face aos canais de entrada e saída apontadas, daqui em diante, para simplificar a discussão, o estado destes canais deixará de ser tido em conta.

Classe Poligono

Este caso é mais interessante. O construtor de um polígono vazio ao qual se passa apenas a origem oferece naturalmente a segurança de não lançar qualquer excepção.

Por outro lado o construtor a partir de um canal começa por invocar o construtor da classe Forma. Como este oferece o nível forte de segurança, não há qualquer problema. Em seguida é invocado implicitamente o construtor por omissão da lista de vértices. Também este construtor oferece o nível forte de segurança face a excepções. Finalmente, já no corpo do construtor, é invocado o procedimento carregaEspecífico(). Se durante a sua execução for lançada uma excepção toda a construção do Poligono falha, sendo invocados os destrutores da lista de vértices e da classe base. Perfeito.

O procedimento carrega() da classe Poligono é mais complicado. Começa por invocar o procedimento carrega() da classe Forma, que oferece o nível máximo de segurança face a excepções. O problema é que depois invoca o procedimento carregaEspecifico(), que pode falhar... Vamos admitir que o procedimento carregaEspecífico() se comporta "decentemente", i.e., ou consegue carregar aquilo que é específico de um polígono (a lista dos vértices), ou mantém a lista de vértices como estava. Neste caso temos um problema complicado. É que se o carregaEspecifico() falhar, só parte do polígono foi carregado, nomeadamente a sua origem, atra'ves do procedimento Forma::carrega(), ficando a lista de vértices como estava... Para que seja oferecido o nível forte de segurança há que garantir que se consegue voltar atrás no carregamento. A solução típica é:

void Poligono::carrega(istream& entrada) { Poligono novo(entrada); *this = novo; }

Ou seja, a solução passou por recorrer no carrega() ao construtor da classe. Como o construtor oferece o nível forte de segurança, uma de duas coisas podem ocorrer:

1.A construção tem sucesso. Nesse caso o novo polígono é copiado para a instância implícita e depois destruído (implitamente). 2.A construção falha. Nesse caso não só o novo polígono não chega a ser construído como a execução não chega à atribuição devido ao lançamento de uma excepção, ficando o polígono intacto.

Claro está que esta solução tem uma desvantagem: é ineficiente. O que acontece de facto ao longo da sua execução?

1.É construído um novo polígono lido de um canal. 2.O novo polígono é atribuído à instância implícita. Ou seja: a.É esvaziada a lista de vértices da instância implícita. b.São copiados os vértices da nova lista para a lista da instância implícita. 3.Finalmente é invocado o destrutor do novo polígono, levando à destruição da respectiva lista de vértices.

O problema está na feitura de uma cópia. Como evitá-la?

Outro problema da solução acima é que a atribuição entre polígonos pode lançar excepções... Porquê? Porque implica fazer uma cópia dos vértices, e pode não haver memória para isso!

Os dois problemas podem-se resolver de uma forma simples. As listas da biblioteca padrão do C++ disponibilizam uma forma rápida de troca de itens: o procedimento membro swap. Pode-se então reescrever:

void Poligono::carrega(istream& entrada) { Poligono novo(entrada); // Copia o que é genérico (nunca lança excepção): Forma::operator = (novo); // Troca o que é específico (nunca lança excepção): vertices.swap(novo.vertices); }

Como a atribuição entre duas Forma não lança excepções e o mesmo se passa com o procedimento swap(), esse problema está resolvido.

Também fica resolvido o problema da cópia. Agora as listas são trocadas (sem cópia), o que faz com que a lista dos vértices antigos vá parar ao novo polígono, sendo destruída quando esta variável for destruída, i.e., ao se atingir a chaveta final do procedimento.

Classe Figura

A discussão para as figuras é semelhante à discussão para os polígonos, pelo que não se trata aqui.

Assuntos por resolver

Uma questão que não se abordou aqui, mas deveria ter sido abordada, é a dos construtor e operador de atribuição por cópia das classes que recorrem a variáveis dinâmicas: Editor e Figura.

O caso da figura é particularmente interessante. O construtor por cópia limita-se a criar uma nova figura contendo ponteiros para as mesmas formas da figura original. Isso é ou não desejável? Duas figuras são iguais se possuirem as mesmas formas ou se possuirem formas iguais? O que nos interessa é semântica de valor ou de cópia?

Em qualquer dos casos põem-se problemas muito interessantes, que serão abordados mais tarde. ????

!!!!!!!!!Semântica de referência: figuras não podem destruir as suas formas! Que o faz, então? Solução: ponteiros espertos com contadores de referências? Ou contadores de referências dentro da classe?

!!!!!!!!!!!!Semântica de valor: como construir cópias de objectos heterogéneos? Quando se tem um ponteiro polimórfico e se constrói uma nova variável dinâmica igual ao objecto apontado por esse ponteiro diz-se que se fez uma clonagem. A solução passa por usar uma função membro virtual chamada clona(), que devolve um clone, e que é implementada apropriadamente por todas as classes da hierarquia:

class Base { // Abstracta public: Base(Base const&); // construtor por cópia explícito ou implícito. Base* clona() const = 0; // a clonagem é puramente virtual nas classes abstractas e virtual nas concretas. };

classe Derivada : public Base { // Concreta public: Derivada(Derivada const&); // idem. virtual Derivada* clona() const { return new Derivada(*this); } };

A estas funções de clonagem também se chama construtores virtuais, por razões óbvias.

Curiosamente no padrão Compósito (Composto?), o construtor por cópia terá de invocar os clona das formas!

!!!!!!!!!Atenção ao assunto da persistência! Convinha tratá-lo algures!

!!!!!!!!!!!!!!!!!!Especificações de excepções: incluir?

Apêndice A

Notação e símbolos

A.1 Notação e símbolos

1. Todos os pedaços de código (i.e., pedaços de texto numa linguagem de programação) e construções específicas do C++ aparecem em tipo (de letra) de largura constante (como Courier). Por exemplo: `main()` e `i++;`. Em todo o restante texto usa-se um tipo proporcional. Nos comentários em C++ usa-se também um tipo proporcional.[1]

2. As variáveis matemáticas aparecem sempre em itálico. Por exemplo: $n = qm + r$.

3. $\mathbb{N}$ é o conjunto dos números naturais.

4. $\mathbb{Z}$ é o conjunto dos números inteiros.

5. $\mathbb{Q}$ é o conjunto dos números racionais.

6. $\mathbb{R}$ é o conjunto dos números reais.

7. Os conjuntos podem ser indicados em extensão colocando os seus elementos entre $\{\}$.

8. $\{\}$ e $\emptyset$ são representações alternativas para o conjunto vazio.

9. $\sharp$ é o operador cardinal, que faz corresponder a cada conjunto o respectivo número de elementos.

10. Os produtos matemáticos nem sempre são escritos explicitamente: pq significa o mesmo que $p \times q$, ou seja, o produto de p por q.

11. O valor absoluto de um número x representa-se por $|x|$.

12. Os símbolos usados para as igualdades e desigualdades quando inseridos em expressões matemáticas (e não expressões C++, onde a notação do C++ é usada) são os seguintes:

[1] Um tipo diz-se proporcional se a largura dos caracteres varia: um 'i' e muito mais estreito que um 'm'. Pelo contrario, num tipo de largura fixa todos os caracteres tem a mesma largura.

$=$ significa "é igual a" ou "é equivalente a".
$\equiv$ significa "é idêntico a" ou "é o mesmo que"[2].
$\neq$ significa "diferente de".
$>$ significa "maior que".
$\geq$ significa "maior ou igual a".
$<$ significa "menor que".
$\leq$ significa "menor ou igual a".

13. A conjunção e a disjunção representam-se pelos símbolos $\wedge$ e $\vee$ ($\oplus$ significa "ou exclusivo").

14. A pertença a um conjunto indica-se pelo símbolo $\in$.

15. A implicação e a equivalência representam-se pelos símbolos $\Rightarrow$ e $\Leftrightarrow$.

16. Os valores lógicos representam-se por:

    $\mathcal{V}$ significa verdadeiro.
    $\mathcal{F}$ significa falso.

17. A operação de negação representa-se pelo símbolo $\neg$.

18. A operação de obtenção do resto da divisão inteira representa-se pelo símbolo $\div$.

19. O símbolo usado para "aproximadamente igual" é $\approx$.

20. Predicado é uma expressão matemática envolvendo variáveis que, de acordo com o valor dessas variáveis, pode ter valor lógico verdadeiro ou falso. Por exemplo, $x > y$ é um predicado. Se $x = 1$ e $y = 0$ o predicado tem valor lógico $\mathcal{V}$ (verdadeiro).

21. Os quantificadores representam-se como se segue (em todos os casos à variável i chama-se variável [muda] de quantificação):[3]

    Soma: $(\mathbf{S}\, i : m \leq i < n : f(i))$ ou $\sum_{m \leq i < n} f(i)$ é a soma (o somatório) dos valores que a função $f()$ toma para todos os valores do inteiro i verificando $m \leq i < n$. I.e., é o mesmo que $f(m) + f(m+1) + \cdots + f(n-1)$.

    Produto: $(\mathbf{P}\, i : m \leq i < n : f(i))$ ou $\prod_{m \leq i < n} f(i)$ é o produto dos valores que a função $f()$ toma para todos os valores do inteiro i verificando $m \leq i < n$. I.e., é o mesmo que $f(m)f(m+1)\cdots f(n-1)$.

    Qualquer que seja: $(\mathbf{Q}\, i : m \leq i < n : f(i))$ ou $\bigwedge_{m \leq i < n} f(i)$ ou ainda $\forall m \leq i < n : f(i)$ é a conjunção dos valores (lógicos) que o predicado $f()$ toma para todos os valores do inteiro i verificando $m \leq i < n$. I.e., é o mesmo que $f(m) \wedge f(m+1) \wedge \cdots \wedge f(n-1)$, ou seja, é o mesmo que afirmar que "qualquer que seja i com $m \leq i < n$, $f(i)$ é verdadeira", daí que seja conhecido como o **quantificador universal**.

[2] É necessário clarificar a diferença entre igualdade e identidade. Pode-se dizer que dois gémeos são iguais, mas não que são idênticos, pois são indivíduos diferentes. Por outro lado, pode-se dizer que Fernando Pessoa e Alberto Caeiro não são apenas iguais, mas também idênticos, pois são nomes que se referem à mesma pessoa.

[3] Notação retirada de [8].

Existe um: $(\mathbf{E}\, i : m \leq i < n : f(i))$ ou $\bigvee_{m \leq i < n} f(i)$ ou ainda $\exists m \leq i < n : f(i)$ é a disjunção dos valores (lógicos) que o predicado $f()$ toma para todos os valores do inteiro i verificando $m \leq i < n$. I.e., é o mesmo que $f(m) \vee f(m+1) \vee \cdots \vee f(n-1)$, ou seja, é o mesmo que afirmar que "existe pelo menos um i com $m \leq i < n$ tal que $f(i)$ é verdadeira", daí que seja conhecido como o **quantificador existencial**.

Contagem: $(\mathbf{N}\, i : m \leq i < n : f(i))$ é o número dos valores (lógicos) verdadeiros que o predicado $f()$ toma para todos os valores do inteiro i verificando $m \leq i < n$. I.e., é o número de valores lógicos verdadeiros na sequência $f(m), f(m+1), \cdots f(n-1)$.

22. Note-se que a notação $x < y < z$ (onde em vez de $<$ pode surgir $\leq$) é uma forma abreviada de escrever $x < y \wedge y < z$.

23. $\dim(x)$ é a dimensão de x, que tanto pode ser uma matriz como um vector (ver Capítulo 5).

24. Sejam v_1 e v_2 duas sequências de dimensões $\dim(v_1)$ e $\dim(v_2)$ tomando valores num conjunto $\mathbb{A}$. Diz-se que v_1 é uma permutação de v_2 e vice-versa se $\dim(v_1) = \dim(v_2) \wedge (\mathbf{Q}\, x :: (\mathbf{N}\, j : 0 \leq j < \dim(v_1) : v_1[j] = x) = (\mathbf{N}\, j : 0 \leq j < \dim(v_2) : v_2[j] = x))$, ou seja, se as sequências tiverem a mesma dimensão e contiverem exactamente os mesmo valores com o mesmo número de repetições. Define-se um predicado perm para verificar se duas sequências são permutações uma da outra da seguinte forma:

$$\operatorname{perm}(v_1, v_2) \equiv \dim(v_1) = \dim(v_2) \wedge (\mathbf{Q}\, x :: (\mathbf{N}\, j : 0 \leq j < \dim(v_1) : v_1[j] = x) = (\mathbf{N}\, j : 0 \leq j < \dim(v_2) : v_2[j] = x))$$

A notação para os quantificadores divide-se em três partes: $(a : b : c)$. A primeira parte, a, indica o tipo de quantificador e as respectivas variáveis mudas bem como o respectivo conjunto base. Por exemplo, $(\mathbf{P}\, i, j \in \mathbb{N} : \cdots : \cdots)$ indica um produto para todos os i e j inteiros. A segunda parte, b, consiste num predicado que restringe os valores possíveis das variáveis mudas. Esse predicado pode ser omitido se for sempre $\mathcal{V}$. Na realidade, portanto, as variáveis mudas tomam apenas valores que tornam o predicado verdadeiro. Por exemplo, $(\mathbf{P}\, i \in \mathbb{N} : i \div 2 \neq 0 : \cdots)$ indica um produto para todos os inteiros ímpares. O mesmo efeito poderia ser obtido escrevendo $(\mathbf{P}\, i \in \{j \in \mathbb{N} : i \div 2 \neq 0\} : \mathcal{V} : \cdots)$, onde o predicado é sempre verdadeiro e se restringe à partida o conjunto base do quantificador. A parte c indica os termos do quantificador.

O conjunto de valores que a variável muda de um quantificador pode tomar pode ser indicado implicitamente ou explicitamente. Em qualquer dos casos a indicação é feita através de um predicado que será verdadeiro para todos os valores que se pretende que a variável muda tome e falso no caso contrário. O conjunto base da variável muda pode ser indicado na primeira parte do quantificador quando a indicação for implícita. Quando o conjunto base não for indicado assume-se que é $\mathbb{Z}$ (conjunto dos números inteiros). Exemplos:

1. $(\mathbf{Q}\, i : i \in \mathbb{N} : \cdots)$, $(\mathbf{Q}\, i \in \mathbb{N} : \mathcal{V} : \cdots)$, $(\mathbf{Q}\, i \in \mathbb{Z} : i > 0 : \cdots)$ ou $(\mathbf{Q}\, i : i > 0 : \cdots)$: para todos os naturais.

2. $(\mathbf{Q}\, i \in \mathbb{N} : i \div 2 = 0 : \cdots)$ ou, menos formalmente, $(\mathbf{Q}\, i \in \mathbb{N} : i \text{ par} : \cdots)$: para todos os naturais pares.

3. $(\mathbf{Q}\, i\, :\, m \leq i < n\, :\, \cdots)$: para todos os inteiros entre m (*inclusive*) e n (*exclusive*).
4. $(\mathbf{Q}\, i \in \{1,\, 4,\, 5\}\, :\, :\, \cdots)$: para todos os elementos do conjunto $\{1,\, 4,\, 5\}$.

Os quantificadores têm, por definição, os seguintes valores quando o conjunto de valores possíveis para a variável de quantificação é vazio:

- $(\mathbf{S}\, i\, :\, \mathcal{F}\, :\, \cdots) = 0$, a soma de zero termos é nula.
- $(\mathbf{P}\, i\, :\, \mathcal{F}\, :\, \cdots) = 1$, o produto de zero termos é 1.
- $(\mathbf{Q}\, i\, :\, \mathcal{F}\, :\, \cdots) = \mathcal{V}$, a conjunção de zero predicados é verdadeira.
- $(\mathbf{E}\, i\, :\, \mathcal{F}\, :\, \cdots) = \mathcal{F}$, a disjunção de zero predicados é falsa.
- $(\mathbf{N}\, i\, :\, \mathcal{F}\, :\, \cdots) = 0$, a contagem de zero predicados é nula.

Em geral , quando o conjunto de variação da variável muda é vazio, o valor destes quantificadores é o elemento neutro da operação utilizada no quantificador. Assim, para a soma é 0, para o produto é 1, para a conjunção é $\mathcal{V}$ e para a disjunção é $\mathcal{F}$.

Existem (pelo menos) as seguintes equivalências entre quantificadores:

- $(\mathbf{Q}\, i \in A\, :\, p(i)\, :\, f(i)) = \neg\,(\mathbf{E}\, i \in A\, :\, p(i)\, :\, \neg f(i))$.
- $(\mathbf{E}\, i \in A\, :\, p(i)\, :\, f(i)) = \neg\,(\mathbf{Q}\, i \in A\, :\, p(i)\, :\, \neg f(i))$.
- $(\mathbf{Q}\, i \in A\, :\, p(i)\, :\, \neg f(i)) = \neg\,(\mathbf{E}\, i \in A\, :\, p(i)\, :\, f(i))$ é o mesmo que $(\mathbf{N}\, i \in A\, :\, p(i)\, :\, f(i)) = 0$.
- $(\mathbf{Q}\, i \in A\, :\, p(i)\, :\, f(i)) = \neg\,(\mathbf{E}\, i \in A\, :\, p(i)\, :\, \neg f(i))$ é o mesmo que $(\mathbf{N}\, i \in A\, :\, p(i)\, :\, f(i)) = \sharp\{i \in A\, :\, p(i)\}$.

A.2 Abreviaturas e acrónimos

e.g. (do latim *exempli gratia*) por exemplo (também se pode usar v.g., de *verbi gratia*).

i.e. (do latim *id est*) isto é.

vs. (do latim *versus*) versus, contra.

cf. (do latim *confer*) conferir, confrontar, comparar.

viz. (do latim *videlicet*) nomeadamente.

Apêndice B

Um pouco de lógica

As seguintes implicações são úteis durantes as demonstrações:

!!!!!!!!!!!!!Colocar aqui tabelas de verdade. Explicação da implicação. Regras de dedução. São todas Tautologias!. A => A ou C, A e B => A, A e B = B e A. (A=A)=V, Distributividade. ((A ou B)=>C) =(A=>B e B=>C). ((A e B)=>C)=(A=>B ou B=>C), (A=>(B e C)) = (A=>B e A=>C),(A=>(B ou C)) = (A=>B ou A=>C), (A=>B e C=>D)=>((A e C)=>(B e D). Explicar por escrito: = lê-se "equivale" ou "é o mesmo" ou "ou seja". => lê-se "o que que implica", "logo", "por consequência". Ver outras regras no Gries. Quando estiver pronto, fazer referência ao apêndice na introdução das demonstrações com asserções.

Apêndice C

Curiosidades e fenómenos estranhos

Apresentam-se aqui algumas curiosidades acerca da linguagem C++ e da programação em geral. Note-se que este apêndice contém código que vai do completamente inútil até ao interessante ou mesmo, imagine-se, potencialmente útil. Não o leia se não se sentir perfeitamente à vontade com a matéria que consta nos capítulos que a este se referem. A ordem das curiosidades é razoavelmente arbitrária.

C.1 Inicialização

As duas formas de inicialização do C++ diferem em alguns aspectos. Compare-se o programa

```
#include <iostream>

using namespace std;

int main()
{
    int i = 1;
    {
        int i = i;
        cout << i << endl;
    }
}
```

com este outro

```
#include <iostream>

using namespace std;

int main()
```

```
{
    int i = 1;
    {
        int i(i);
        cout << i << endl;
    }
}
```

O segundo escreve 1 no ecrã. O primeiro escreve lixo (só por azar, ou sorte, será 1). Porquê?

Porque no primeiro a variável mais interior já está definida (embora contendo lixo) quando se escreve `= i`, pelo que a variável é inicializada com o seu próprio valor (tinha lixo... com lixo fica). No segundo caso a variável mais interior não está ainda definida quando se escreve `(i)`, pois estes parênteses fazem parte da definição. Assim, a variável interior é inicializada com o valor da variável mais exterior...

C.1.1 Inicialização de membros

Uma consequência interessante do que se disse atrás é que também se aplica nas listas de construtores das classes. Por exemplo:

```
class A {
  public:
    A(int i, int j, int k)
        : i(k), // i membro inicializada com k parâmetro!
          j(j), // j membro inicializada com j parâmetro!
          k(i)  // k membro inicializada com i parâmetro!
    {
    }
  private:
    int i;
    int j;
    int k;
};
```

C.2 Rotinas locais

A linguagem C++ proíbe a definição de rotinas locais: todas as funções e procedimentos são globais. Esta afirmação é verdadeira, mas apenas para funções e procedimentos não membro. Uma vez que existe o conceito de classe local, podem-se usar métodos de classe em classes locais para simular rotinas locais. O método é simples:

```
// Define-se esta macro para tornar a sintaxe de definição mais simpática:
#define local_definitions struct _
```

```
// Define-se esta macro para tornar a sintaxe de invocação mais simpática:
#define local _::

int main()
{
    // As rotinas locais definem-se sempre dentro deste falso ambiente:
    local_definitions {
        // As rotinas locais são sempre métodos de classe (static):
        static int soma(int a, int b)
        {
            return a + b;
        }
    };

    // A invocação de rotinas locais faz-se através da falsa palavra-chave local:
    int n = local soma(10, 45);
}
```

C.3 Membros acessíveis ”só para leitura”

A linguagem C++ proporciona três diferentes categorias de acesso para os membros de uma classe: há membros públicos, protegidos e privados. O valor de variável membro pública (a evitar porque viola o princípio do encapsulamento) pode ser usado ou alterado directamente sem quaisquer restrições. Uma variável membro privada não é acessível de todo do exterior. Não era simpático que existissem variáveis membro cujo valor estivesse disponível para ser usado directamente mas não pudesse ser alterado excepto pela própria classe? Note-se que uma constante membro pública não resolve o problema, pois nem a classe a pode alterar!

A solução passa por usar uma referência constante pública para aceder a uma variável membro privada! Por exemplo, se a variável for do tipo `int` e se chamar `valor`:

```
class A {
  public:
    // O construtor faz valor referenciar valor_:
    A(int v)
        : valor(valor_), valor_(v) {
    }

    // Quando se usa este truque é fundamental fornecer um construtor por cópia
    // apropriado. Porquê? Porque senão a referência da cópia refere-se à variável
    // do original!
    A(A const& a)
        : valor(valor_), valor_(a.valor_) {
    }
```

```
    // Da mesma forma tem de se fornecer o operador de atribuição por cópia. Caso
    // contrário não se poderiam fazer atribuições entre instâncias de classes com o
    // truque. É que o C++ não pode fornecer o operador de atribuição por cópia
    // automaticamente a classes com membros que sejam referências!
    A& operator = (A const& a) {
        valor_ = a.valor_;
        return *this;
    }

    // A referência valor para a variável valor_ é constante para evitar alterações
    // por entidades externas à classe:
    int const& valor;

    // Um procedimento para atribuir um novo valor à variável. Só para efeitos de teste:
    void poeValor(int v) {
        valor_ = v;
    }

  private:
    // A variável propriamente dita é privada...
    int valor_;
};
```

Com esta classe é possível escrever o seguinte código:

```
#include <iostream>

using namespace std;

int main()
{
    A a(10);
    cout << a.valor << endl; // mostra 10.
    a.valor = 20;            // dá erro!
    a.poeValor(30);
    cout << a.valor << endl; // mostra 30.
    A b = a;
    cout << b.valor << endl; // mostra 30.
    A c(40);
    c = a;
    cout << c.valor << endl; // mostra 30.
}
```

Quem disse que não havia a categoria “só para leitura” em C++?

C.4 Variáveis virtuais

O princípio do encapsulamento obriga a esconder as variáveis membro de uma classe (fazê-las privadas) e fornecer funções de inspecção para o seu valor se necessário. Isto é fundamental por um conjunto de razões:

1. A implementação da classe pode precisar de mudar. As variáveis membro fazem naturalmente parte da implementação de uma classe. Uma variável membro pública faz também parte da interface da classe, pelo que uma alteração na implementação pode ter impactos sérios em código que use a classe.

2. Se o valor contido pela variável tiver de ser calculado de forma diferente em classes derivadas, só com funções de inspecção virtuais se pode especializar essa forma de cálculo.

Suponha-se, por exemplo, uma hierarquia de classes representando contas. A classe base da hierarquia representa uma abstracção de conta. Para simplificar considera-se que uma conta permite apenas saber o seu saldo (e possui um destrutor virtual, como todas as classes polimórficas):

```
class Conta {
  public:
    virtual ~Conta() {}
    virtual double saldo() const = 0;
};
```

Definem-se agora duas concretizações do conceito. A primeira é uma conta simples. Uma conta simples tem um saldo como atributo. Possui um construtor que inicializa o saldo e sobrepõe uma função de inspecção especializada que se limita a devolver esse saldo (ou seja, fornece um método para a operação abstracta `saldo()`). A segunda é um *portfolio* de contas, i.e., uma "conta de contas". O portfolio de contas tem um procedimento `acrescentaConta()` que permite acrescentar uma conta ao portfolio, um destrutor para destruir as suas contas e fornece também um método para a operação virtual `saldo()`:

```
#include <list>

class ContaSimples : public Conta {
  public:
    ContaSimples(double saldo)
        : saldo_(saldo) {
    }
    virtual double saldo() const {
        return saldo_;
    }
  private:
    double saldo_;
```

```
};

class Portfolio : public Conta {
  public:
    ~Portfolio() {
        // Destrói contas do portfolio:
        for(std::list<Conta*>::iterator i = contas.begin();
            i != contas.end(); ++i)
            delete *i;
    }
    void acrescentaConta(Conta* nova_conta) {
        contas.push_back(nova_conta);
    }
    virtual double saldo() const {
        // É a soma dos saldos das contas no portfolio:
        double saldo = 0.0;
        for(std::list<Conta*>::const_iterator i = contas.begin();
            i != contas.end(); ++i)
            saldo += (*i)->saldo();
        return saldo;
    }
  private:
    std::list<Conta*> contas;
};
```

Estas classes podem ser usadas como se segue:

```
#include <iostream>

using namespace std;

int main()
{
    // Constrói duas contas simples (ponteiros para Conta!):
    Conta* c1 = new ContaSimples(10);
    Conta* c2 = new ContaSimples(30);

    // Mostra saldos das contas simples:
    cout << c1->saldo() << endl; // mostra 10.

    cout << c2->saldo() << endl; // mostra 30.


    // Constrói uma conta portfolio e acrescenta-lhe as duas contas:
    Portfolio* p = new Portfolio;
    p->acrescentaConta(c1);
```

```
        p->acrescentaConta(c2);

        // Guarda endereço da conta portfolio no ponteiro c para Conta:
        Conta* c = p;

        // Mostra saldo da conta portfolio:
        cout << c->saldo() << endl; // mostra 40.
    }
```

Note-se que o polimorfismo é fundamental para que a invocação da operação `saldo()` leve à execução do método `saldo()` apropriado à classe do objecto apontado e não à classe do ponteiro! Isto seria muito difícil de reproduzir se se tivesse colocado uma variável membro para guardar o saldo na classe base, mesmo que fosse privada. Porquê? Porque nesse caso a operação `saldo()` não seria abstracta, estando definida na classe base como devolvendo a valor dessa variável membro. Por isso o valor dessa variável teria de ser mantido coerente com o saldo da conta. Isso é fácil de garantir para uma conta simples mas muito mais difícil num portfolio, pois sempre que uma classe num portfolio tiver uma mudança de saldo terá de avisar a conta portfolio desse facto para que esta tenha a oportunidade de actualizar o valor da variável.

Se essa variável fosse pública ter-se-ia a desvantagem adicional de futuras alterações na implementação terem impacto na interface da classe, o que é indesejável.

Logo, não se pode usar uma variável membro pública neste caso para guardar o saldo.

Porquê este discurso todo no apêndice de curiosidades e fenómenos estranhos do C++ e numa secção chamada **Variáveis virtuais**? Porque... é possível ter variáveis membro públicas, sem nenhum dos problemas apontados, e ainda apenas com permissão para leitura (restrição fácil de eliminar e que fica como exercício para o leitor).

O truque passa por definir essa variável membro como sendo de uma classe extra `DoubleVirtual` (amiga da classe `Conta`) e que possui:

1. Um construtor que recebe a instância da conta.
2. Um operador de conversão implícita para `double` que invoca o operador virtual de cálculo do saldo.

Adicionalmente, pode-se colocar o operador de cálculo do saldo na parte privada das classes, uma vez que deixa de ser útil directamente para o consumidor das classes. Finalmente, para evitar conflitos de nomes, este operador passa a ter um nome diferente:

```
class Conta {
  public:
    class DoubleVirtual {
    public:
        DoubleVirtual(Conta& base)
        : base(base) {
```

```
            }
            operator double () const {
                return base.calculaSaldo();
            }
        private:
            // Guarda-se uma referência para a conta a que o saldo diz respeito:
            Conta& base;
        };
        friend DoubleVirtual;
        Conta()
            : saldo(*this) {
        }
        Conta(Const const&)
            : saldo(*this) {
        }
        Conta& operator = (Conta const&) {
            return *this;
        }
        virtual ~Conta() {}
        DoubleVirtual saldo;
      private:
        virtual double calculaSaldo() const = 0;
};

#include <list>

class ContaSimples : public Conta {
  public:
    ContaSimples(double saldo)
        : saldo_(saldo) {
    }
private:
    double saldo_;
    virtual double calculaSaldo() const {
        return saldo_;
    }
};

class Portfolio : public Conta {
  public:
    ~Portfolio() {
        for(std::list<Conta*>::iterator i = contas.begin();
            i != contas.end(); ++i)
            delete *i;
    }
    void acrescentaConta(Conta* nova_conta) {
```

```
            contas.push_back(nova_conta);
        }
      private:
        std::list<Conta*> contas;
        virtual double calculaSaldo() const {
            double saldo = 0.0;
            for(std::list<Conta*>::const_iterator i = contas.begin();
                i != contas.end(); ++i)
                saldo += (*i)->saldo;
            return saldo;
        }
    };
```

O programa de teste é agora mais simples, pois aparenta aceder directamente a um atributo para obter o saldo das contas:

```
#include <iostream>

using namespace std;

int main()
{
    // Constrói duas contas simples (ponteiros para Conta!):
    Conta* c1 = new ContaSimples(10);
    Conta* c2 = new ContaSimples(30);

    // Mostra saldos das contas simples:
    cout << c1->saldo << endl;
    cout << c2->saldo << endl;

    // Constrói uma conta portfolio e acrescenta-lhe as duas contas:
    Portfolio* p = new Portfolio;
    p->acrescentaConta(c1);
    p->acrescentaConta(c2);

    // Guarda endereço da conta portfolio no ponteiro c para Conta:
    Conta* c = p;

    // Mostra saldo da conta portfolio:
    cout << c->saldo << endl;
}
```

Este truque pode ser refinado definindo uma classe C++ genérica `TipoVirtual` que simplifique a sua utilização mais genérica:

```
template <typename B, typename T>
```

```
class TipoVirtual {
  public:
    TipoVirtual(B& base, T (B::*calcula)() const)
        : base(base), calcula(calcula) {
    }
    operator T () const {
        return (base.*calcula)();
    }
  private:
    // Guarda-se uma referência para a classe a que variável diz respeito:
    B& base;
    // Guarda-se um ponteiro para a função membro que devolve o valor da variável:
     T (B::*calcula)() const;
};

class Conta {
  public:
    Conta()
        : saldo(*this, &calculaSaldo) {
    }
    Conta(Const const&)
        : saldo(*this, &calculaSaldo) {
    }
    Conta& operator = (Conta const&) {
        return *this;
    }
    virtual ~Conta() {}
    friend TipoVirtual<Conta, double>;
    TipoVirtual<Conta, double> saldo;
  private:
    virtual double calculaSaldo() const = 0;
};

// O resto tudo igual.
```

Com esta classe C++ genérica definida, a utilização de "variáveis virtuais" exige apenas pequenas alterações na hierarquia de classes. Note-se que:

1. Nada se perdeu em encapsulamento. A variável `saldo` é parte da interface mas não é parte da implementação!
2. Nada se perdeu em polimorfismo. Continua a existir uma função para devolução do saldo. Esta função pode (e deve) ser especializada em classes derivadas.
3. Mas que sucede quando se copiam instâncias de classes com variáveis virtuais? Classes com variáveis virtuais têm de definir o construtor por cópia! Têm também de definir o operator de atribuição por cópia!

4. É possível construir outra classe C++ genérica para permitir acesso completo, incluindo escritas. Essa classe C++ genérica fica como exercício!

C.5 Persistência simplificada

!!Texto a completar! Verificar se ainda existe versão com registo automático.

```
/** Este módulo contém uma única classe C++ genérica muito simples, Serializador,
    que serve para simplificar a serialização polimórfica de hierarquias de classes.

    Esta técnica foi desenvolvida por mim independentemente, embora tenha
    benificiado com algumas ideias de:

    Jim Hyslop e Herb Sutter, "Conversations: Abstract Factory, Template Style",
    CUJ Experts, Junho de 2001.

    Jim Hyslop and Herb Sutter, "Conversations: How to Persist an Object",
    CUJ Experts, Julho de 2001.

    @see Serializador.

    Copyright © Manuel Menezes de Sequeira, ISCTE, 2001. */

#ifndef SERIALIZADOR_H
#define SERIALIZADOR_H

#include <map>
#include <string>
#include <typeinfo>

/** Uma classe C++ genérica que serializa polimorficamente objectos referenciados por
    ponteiros. Os únicos requisitos feitos à hierarquia de classes são que dois
    métodos têm de estar presentes:

    Construtor por desserialização:
    Classe(istream& entrada);

    Operação polimórfica de serialização:
    virtual void serializa(istream& saida) const;

    Conceitos (novo): Serializável? */
template <class Base>
class Serializador {
  public:
```

```
    /** Usado para serializar polimorficamente uma instância da hierarquia
        encimada pela classe Base. Um identificador da classe é colocado no
        canal antes dos dados da classe propriamente ditos. */
    static void serializa(std::ostream& saida, Base const* a);

    /** Usado para construir uma nova instância da hierarquia de classes através
        da sua desserialização a partir de um canal de entrada. Assume-se que o
        canal contém um identificador de classe, tal como inserido pelo método
        serializa() acima. */
    static Base* constroi(std::istream& entrada);

    /** Uma classe usada para simplificar o registo de classes na hierarquia. */
    template <class Derivada>
    class Registador {
      public:
        Registador() {
            Serializador<Base>::
                regista_(typeid(Derivada).name(),
                         &Serializador<Base>::template
                             constroi_<Derivada>);
        }
    };

  private:

    typedef Base* Criador(std::istream&);

    static Serializador& instancia();

    Serializador() {}

    Serializador(Serializador const&);

    Serializador& operator = (Serializador const&);

    std::map<std::string, Criador*> registo;

    template <class Derivada>
    static Base* constroi_(std::istream& entrada);

  public:
    // Devia ser privado e Registador devia ser amiga de Serializador,
    // mas o GCC estoirou...
    static void regista_(std::string const& nome_da_classe,
                         Criador* criador);
```

```
};

template <class Base>
template <class Derivada>
inline Base* Serializador<Base>::constroi_(std::istream& entrada)
{
    return new Derivada(entrada);
}

template <class Base>
inline void Serializador<Base>::
    regista_(std::string const& nome_da_classe, Criador* criador)
{
    // Uma asserção é apropriada aqui, uma vez que o registo de classes deve
    // ocorrer antes de a função main() começar...
    assert(instancia().registo.count(nome_da_classe) == 0);

    instancia().registo[nome_da_classe] = criador;
}

template <class Base>
Base* Serializador<Base>::constroi(std::istream& entrada)
{
    // Isto deveria ser uma asserção lançadora de excepções (pré-condição):
    if(not entrada)
        throw string("canal de entrada errado em "
                     "Serializador<Base>::constroi");

    std::string nome_da_classe;
    getline(entrada, nome_da_classe);

    // Isto deveria ser uma excepção para erros extraíndo de um canal de entrada:
    if(not entrada)
        throw string("erro extraíndo identificador de classe");

    // Isto deveria ser uma excepção para classe por registar:
    if(instancia().registo.count(nome_da_classe) == 0)
        throw std::string("Classe '") + nome_da_classe +
              "' por registar!";

    return instancia().registo[nome_da_classe](entrada);
}

template <class Base>
inline void Serializador<Base>::serializa(std::ostream& saida,
                                          Base const* a)
```

```
{
    // Isto deveria ser uma asserção lançadora de excepções (pré-condição):
    if(not saida)
        throw string("canal de saída errado em "
                     "Serializador<Base>::serializa");

    // Desnecessário em rigor, mas útil (traz alguma simetria)...
    // Isto deveria ser uma excepção para classe por registar:
    if(instancia().registo.count(typeid(*a).name()) == 0)
        throw std::string("Classe '") + typeid(*a).name() +
              "' por registar!";

    saida << typeid(*a).name() << std::endl;

    // Isto deveria ser uma excepção para erros inserindo num canal de saída:
    if(not saida)
        throw string("erro inserindo identificador de classe");

    a->serializa(saida);
}

template <class Base>
Serializador<Base>& Serializador<Base>::instancia()
{
    static Serializador instancia;

    return instancia;
}

#endif // SERIALIZADOR_H
```

!!Seguem-se os ficheiros de teste: a.H, a.C, b.H, b.C, c.H, c.C, d.H, d.C, teste.C

a.H

```
#ifndef A_H
#define A_H
#include <iostream>
#include <string>

class A {
  public:
    A();
    A(std::istream& entrada);

    virtual void serializa(std::ostream& saida) const;
```

```
};

inline A::A()
{
    std::clog << "A criado" << std::endl;
}

inline A::A(std::istream& entrada)
{
    if(not entrada)
        throw string("canal de entrada errado em A::A(std::istream&)");

    std::string texto;
    std::getline(entrada, texto);

    if(not entrada or texto != "A")
        throw string("erro desserializando A");

    std::clog << "A criado de canal" << std::endl;
}

inline void A::serializa(std::ostream& saida) const
{
    if(not saida)
        throw string("canal de saída errado em A::serializa");

    saida << "A" << std::endl;

    if(not saida)
        throw string("erro serializando A");

    std::clog << "A serializado" << std::endl;
}

#endif // A_H
```

a.C

```
#include "a.H"

#include "serializador.H"

namespace {
    Serializador<A>::Registador<A> faz_registo;
}
```

b.H

```
#ifndef B_H
#define B_H

#include <iostream>
#include <string>

#include "a.H"

class B : public A {
  public:
    B();
    B(std::istream& entrada);

    virtual void serializa(std::ostream& saida) const;
};

inline B::B()
{
    std::clog << "B criado" << endl;
}

inline B::B(std::istream& entrada)
    : A(entrada)
{
    std::string texto;
    std::getline(entrada, texto);

    if(not entrada or texto != "B")
        throw string("erro desserializando B");

    std::clog << "B criado de canal" << std::endl;
}

inline void B::serializa(ostream& saida) const
{
    if(not saida)
        throw string("canal de saída errado em B::serializa");

    A::serializa(saida);

    saida << "B" << std::endl;

    if(not saida)
        throw string("erro serializando B");

    std::clog << "B serializado" << std::endl;
```

```
}

#endif // B_H
```

b.C

```
#include "b.H"

#include "serializador.H"

namespace {
    Serializador<A>::Registador<B> faz_registo;
}
```

c.H

```
#ifndef C_H
#define C_H

#include <iostream>
#include <string>

#include "a.H"
class C : public A {
  public:
    C();
    C(std::istream& entrada);

    virtual void serializa(std::ostream& saida) const;
};

inline C::C()
{
    std::clog << "C criado" << endl;
}

inline C::C(std::istream& entrada)
    : A(entrada)
{
    std::string texto;
    std::getline(entrada, texto);

    if(not entrada or texto != "C")
        throw string("erro desserializando C");

    std::clog << "C criado de canal" << std::endl;
}
```

```
inline void C::serializa(ostream& saida) const
{
    if(not saida)
        throw string("canal de saída errado em C::serializa");

    A::serializa(saida);

    saida << "C" << std::endl;

    if(not saida)
        throw string("erro serializando C");

    std::clog << "C serializado" << std::endl;
}

#endif // C_H
```

c.C

```
#include "c.H"

#include "serializador.H"

namespace {
    Serializador<A>::Registador<C> faz_registo1;
    Serializador<C>::Registador<C> faz_registo2;
}
```

d.H

```
#ifndef D_H
#define D_H

#include <iostream>
#include <string>

#include "c.H"

class D : public C {
  public:
    D();
    D(std::istream& entrada);

    virtual void serializa(std::ostream& saida) const;
};
```

```
inline D::D()
{
    std::clog << "D criado" << endl;
}

inline D::D(std::istream& entrada)
    : C(entrada)
{
    std::string texto;
    std::getline(entrada, texto);

    if(not entrada or texto != "D")
        throw string("erro desserializando D");

    std::clog << "D criado de canal" << std::endl;
}

inline void D::serializa(ostream& saida) const
{
    if(not saida)
        throw string("canal de saída errado em D::serializa");

    C::serializa(saida);

    saida << "D" << std::endl;

    if(not saida)
        throw string("erro serializando D");

    std::clog << "D serializado" << std::endl;
}

#endif // D_H
```

d.C

```
#include "d.H"

#include "serializador.H"

namespace {
    Serializador<A>::Registador<D> faz_registo1;
    Serializador<C>::Registador<D> faz_registo2;

    /* Mensagens de erro não são difíceis de entender:
```

```
        class X {
          public:
            X(std::istream&) {}
        };
        Serializador<C>::Registador<X> faz_registo3;
        Serializador<D>::Registador<C> faz_registo4;
    */
}
```

teste.C

```
#include <iostream>
#include <fstream>

using namespace std;

#include "a.H"
#include "b.H"
#include "c.H"
#include "d.H"

#include "serializador.H"

class E : public D {
  public:
    E() {}
    E(istream&) {}
};

int main()
{
    A* a = new A;
    A* b = new B;
    A* c = new C;
    A* d = new D;

    try {
        ofstream saida("teste1.txt");

        Serializador<A>::serializa(saida, a);
        Serializador<A>::serializa(saida, b);
        Serializador<A>::serializa(saida, c);
        Serializador<A>::serializa(saida, d);

        saida.flush();
```

```
        ifstream entrada("teste1.txt");

        A* a = Serializador<A>::constroi(entrada);
        A* b = Serializador<A>::constroi(entrada);
        A* c = Serializador<A>::constroi(entrada);
        A* d = Serializador<A>::constroi(entrada);
    } catch(string excepcao) {
        cerr << "Algo de errado... " << excepcao << endl;
    }

    C* x = new C;
    C* y = new D;

    try {
        ofstream saida("teste2.txt");

        Serializador<C>::serializa(saida, x);
        Serializador<C>::serializa(saida, y);

        saida.flush();

        ifstream entrada("teste2.txt");

        C* x = Serializador<C>::constroi(entrada);
        C* y = Serializador<C>::constroi(entrada);
    } catch(string excepcao) {
        cerr << "Algo de errado... " << excepcao << endl;
    }

    // Falha...
    try {
        A* outro = new E;
        Serializador<A>::serializa(cout, outro);
    } catch(string excepcao) {
        cerr << "Algo de errado... " << excepcao << endl;
    }
}
```

Apêndice D

Nomes e seu formato: recomendações

!!!!!!!!!!!! Completar isto!

Entidade	Nome	Formato	Exemplo
função	valor calculado	minúsculas, maiúsculas para separação	`int máximoDivisorComum(int n, int m)`
função devolvendo `bool`	predicado	minúsculas, maiúsculas para separação	`bool éNúmeroPrimo(int n)`
procedimento	acção [+ complementos]	minúsculas, maiúsculas para separação	`void troca(int n, int d)`
funções membro	valor calculado	minúsculas, maiúsculas para separação	`int Racional::denominador() const`

Recomendações sobre nomes de entidades nos programas em C++. Parte das entidades só serão abordadas em capítulos posteriores. Volte a consultar esta tabela Entre [] partes opcionais.

Apêndice E

Palavras-chave do C++

!Colocar explicações com referências para o resto do texto.!!!!!!!!!!!!! !!!!!!!!!!!!!! !!!!!!!!!!!! !!!! !!!!!!!!!!! !!!!!!!!!! !!!!!!!!!!!!! !!!!!!!!!!!!!

Palavra-chave	**Utilização**
`and`	Operador conjunção, o mesmo que `&&`. Ver Secção 2.7.3.
`and_eq`	Operador conjunção *bit*-a-*bit* e atribuição, o mesmo que `&=`. Ver Secção 2.7.4.
`asm`	Serve para introduzir código *assembly*. Só para programação de muito baixo nível!
`auto`	Indica explicitamente que uma variável é automática. A sua utilização é redundante. Ver Secção 3.2.12.
`bitand`	Operador conjunção *bit*-a-*bit*, o mesmo que `&`. Ver Secção 2.7.4.
`bitor`	Operador disjunção *bit*-a-*bit*, o mesmo que `|`. Ver Secção 2.7.4.
`bool`	Um dos tipos básicos do C++, usado para guardar valores booleanos (i.e., $\mathcal{V}$ ou $\mathcal{F}$). Ver Secção 2.3.
`break`	Permite terminar prematuramente as instruções `switch` (ver Secção 4.4.2) e `while`, `for` e `do while` (ver Secção 4.5.4).
`case`	Assinala um ponto de entrada numa instrução `switch`. Ver Secção 4.4.2.
`catch`	
`char`	Tipo básico usado para representar caracteres. Ver Secção 2.3.
`class`	
`compl`	Operador negação *bit*-a-*bit* e ou complemento para um, o mesmo que `~`. Ver Secção 2.7.4.
`const`	Qualificador que indica que uma variável não pode ser alterada depois de inicializada, i.e., que é uma constante, ou que uma função membro de instância não altera a instância implícita.. Ver Ver !!.
`const_cast`	
`continue`	
`default`	

<table>
<tr><td>delete</td><td></td></tr>
<tr><td>do</td><td></td></tr>
<tr><td>double</td><td></td></tr>
<tr><td>dynamic_cast</td><td></td></tr>
<tr><td>else</td><td></td></tr>
<tr><td>enum</td><td></td></tr>
<tr><td>explicit</td><td></td></tr>
<tr><td>export</td><td></td></tr>
<tr><td>extern</td><td></td></tr>
<tr><td>false</td><td></td></tr>
<tr><td>float</td><td></td></tr>
<tr><td>for</td><td></td></tr>
<tr><td>friend</td><td></td></tr>
<tr><td>goto</td><td></td></tr>
<tr><td>if</td><td></td></tr>
<tr><td>inline</td><td></td></tr>
<tr><td>int</td><td></td></tr>
<tr><td>long</td><td></td></tr>
<tr><td>mutable</td><td></td></tr>
<tr><td>namespace</td><td></td></tr>
<tr><td>new</td><td></td></tr>
<tr><td>not</td><td>Operador negação, o mesmo que !. Ver Secção 2.7.3.</td></tr>
<tr><td>operator</td><td></td></tr>
<tr><td>or</td><td>Operador disjunção, o mesmo que ||. Ver Secção 2.7.3.</td></tr>
<tr><td>or_eq</td><td>Operador disjunção bit-a-bit e atribuição, o mesmo que |=. Ver Secção 2.7.4.</td></tr>
<tr><td>private</td><td></td></tr>
<tr><td>protected</td><td></td></tr>
<tr><td>public</td><td></td></tr>
<tr><td>register</td><td></td></tr>
<tr><td>reinterpret_cast</td><td></td></tr>
<tr><td>return</td><td></td></tr>
<tr><td>short</td><td></td></tr>
<tr><td>signed</td><td></td></tr>
<tr><td>sizeof</td><td></td></tr>
<tr><td>static</td><td></td></tr>
<tr><td>static_cast</td><td></td></tr>
<tr><td>struct</td><td></td></tr>
<tr><td>switch</td><td></td></tr>
<tr><td>template</td><td></td></tr>
<tr><td>this</td><td></td></tr>
<tr><td>throw</td><td></td></tr>
<tr><td>true</td><td></td></tr>
</table>

`try`	
`typedef`	
`typeid`	
`typename`	
`union`	
`unsigned`	
`using`	
`virtual`	
`void`	
`volatile`	
`wchat_t`	
`while`	
`xor`	Operador disjunção exclusiva *bit*-a-*bit*, o mesmo que `^`. Ver Secção 2.7.4.
`xor_eq`	Operador disjunção exclusiva *bit*-a-*bit* e atribuição, o mesmo que `^=`. Ver Secção 2.7.4.

Apêndice F

Precedência e associatividade no C++

A tabela seguinte sumariza as regras de precedência e associatividade dos operadores do C++. Operadores colocados na mesma secção têm a mesma precedência. As secções são separadas por § e apresentadas por ordem decrescente de precedência. Quanto à associatividade, apenas os operadores unários (com um único operando) e os operadores de atribuição se associam à direita: todos os outros associam-se à esquerda, como é habitual.

Descrição	**Sintaxe** (itálico: partes variáveis da sintaxe)
resolução de âmbito	*nome_de_classe* `::` *membro*
resolução de âmbito	*nome_de_espaço_nominativo* `::` *membro*
global	`::` *nome*
global	`::` *nome_qualificado*
	§
selecção de membro	*objecto* `.` *membro*
selecção de membro	*ponteiro* `->` *membro*
indexação	*ponteiro* `[` *expressão_inteira* `]`
invocação de função	*expressão* `(` *lista_expressões* `)`
construção de valor	*tipo* `(` *lista_expressões* `)`
incrementação sufixa	*lvalor* `++`
decrementação sufixa	*lvalor* `--`
identificação de tipo	`typeid (` *tipo* `)`
identificação de tipo durante a execução	`typeid (` *expressão* `)`
conversão verificada durante a execução	`dynamic_cast <` *tipo* `> (` *expressão* `)`
conversão verificada durante a compilação	`static_cast <` *tipo* `> (` *expressão* `)`
conversão não verificada (evitar)	`reinterpret_cast <` *tipo* `> (` *expressão* `)`
conversão constante (evitar)	`const_cast <` *tipo* `> (` *expressão* `)`
	§
tamanho de objecto	`sizeof` *expressão*
tamanho de tipo	`sizeof (` *tipo* `)`

incrementação prefixa	`++` *`lvalue`*
decrementação prefixa	`--` *`lvalue`*
negação *bit*-a-*bit* ou complemento para um	`compl` *`expressão`* (ou `~`)
negação	`not` *`expressão`*
simétrico	`-` *`expressão`*
identidade	`+` *`expressão`*
endereço de	`&` *`lvalue`*
conteúdo de	`*` *`expressão`*
construçãode variável dinâmica	`new` *`tipo`*
construçãode variável dinâmica com inicializadores	`new` *`tipo`* `(` *`lista_expressões`* `)`
construção localizada de variável dinâmica	`new (` *`lista_expressões`* `)` *`tipo`*
construção localizada de variável dinâmica com inicializadores	`new (` *`lista_expressões`* `)` *`tipo`* `(` *`lista_expressões`* `)`
destruição de variável dinâmica	`delete` *`ponteiro`*
destruição de matriz dinâmica	`delete[]` *`ponteiro`*
conversão de tipo de baixo nível (crime!)	`(` *`tipo`* `)` *`expressão`*
§	
selecção de membro	*`objecto`* `.*` *`ponteiro_para_membro`*
selecção de membro	*`ponteiro`* `->*` *`ponteiro_para_membro`*
§	
multiplicação	*`expressão`* `*` *`expressão`*
divisão	*`expressão`* `/` *`expressão`*
resto da divisão inteira	*`expressão`* `%` *`expressão`*
§	
adição	*`expressão`* `+` *`expressão`*
subtracção	*`expressão`* `-` *`expressão`*
§	
deslocamento para a esquerda	*`expressão`* `<<` *`expressão`*
deslocamento para a direita	*`expressão`* `>>` *`expressão`*
§	
menor	*`expressão`* `<` *`expressão`*
menor ou igual	*`expressão`* `<=` *`expressão`*
maior	*`expressão`* `>` *`expressão`*
maior ou igual	*`expressão`* `>=` *`expressão`*
§	
igual	*`expressão`* `==` *`expressão`*
diferente	*`expressão`* `!=` *`expressão`* (ou `not_eq`)
§	
conjunção *bit*-a-*bit*	*`expressão`* `bitand` *`expressão`* (ou `&`)
§	
disjunção exclusiva *bit*-a-*bit*	*`expressão`* `xor` *`expressão`* (ou `^`)
§	
disjunção *bit*-a-*bit*	*`expressão`* `bitor` *`expressão`* (ou `|`)

<table>
<tr><td colspan="2">§</td></tr>
<tr><td>conjunção</td><td>expressão and expressão (ou &&)</td></tr>
<tr><td colspan="2">§</td></tr>
<tr><td>disjunção</td><td>expressão or expressão (ou ||)</td></tr>
<tr><td colspan="2">§</td></tr>
<tr><td>operador condicional</td><td>expressão ? expressão : expressão</td></tr>
<tr><td colspan="2">§</td></tr>
<tr><td>atribuição simples</td><td>lvalue = expressão</td></tr>
<tr><td>multiplicação e atribuição</td><td>lvalue *= expressão</td></tr>
<tr><td>divisão e atribuição</td><td>lvalue /= expressão</td></tr>
<tr><td>resto e atribuição</td><td>lvalue %= expressão</td></tr>
<tr><td>adição e atribuição</td><td>lvalue += expressão</td></tr>
<tr><td>subtracção e atribuição</td><td>lvalue -= expressão</td></tr>
<tr><td>deslocamento para a esquerda e atribuição</td><td>lvalue <<= expressão</td></tr>
<tr><td>deslocamento para a direita e atribuição</td><td>lvalue >>= expressão</td></tr>
<tr><td>conjunção bit-a-bit e atribuição</td><td>lvalue &= expressão (ou and_eq)</td></tr>
<tr><td>disjunção bit-a-bit e atribuição</td><td>lvalue |= expressão (ou or_eq)</td></tr>
<tr><td>disjunção exclusiva bit-a-bit e atribuição</td><td>lvalue ^= expressão (ou xor_eq)</td></tr>
<tr><td colspan="2">§</td></tr>
<tr><td>lançamento de excepção</td><td>throw expressão</td></tr>
<tr><td>sequenciamento</td><td>expressão , expressão</td></tr>
</table>

Apêndice G

Tabelas de codificação ISO-8859-1 (Latin-1) e ISO-8859-15 (Latin-9)

Apresenta-se exaustivamente a tabela de codificação ISO-8859-1, também conhecida por Latin-1, incluindo os nomes dos símbolos. Esta tabela foi usada durante bastante tempo nos países da Europa ocidental (e uns quantos outros) e é uma extensão à tabela ASCII. No entanto, esta tabela não contém o símbolo para o euro, além de lhe faltarem alguns símbolos para as línguas francesa e finlandesa. Por isso, o ISO (International Standards Organization) decidiu criar uma nova tabela que deverá passar a ser usada na Europa ocidental: o ISO-8859-15 ou Latin-9. Na tabela são indicados os oito caracteres que variam da tabela Latin-1 para a tabela Latin-9.

Descrição:	**Código:**	**Símbolo:**
Caracteres de controlo:		
nulo	0	
começo de cabeçalho	1	
começo de texto	2	
fim de texto	3	
fim de transmissão	4	
inquirição	5	
confirmação de recepção	6	
campaínha	7	
espaço para trás	8	
tabulador horizontal	9	
nova linha	10	
tabulador vertical	11	
nova página	12	
retorno do carreto	13	
shift out	14	
shift in	15	
escape de ligação de dados	16	

controlo de dispositivo 1	17	
controlo de dispositivo 2	18	
controlo de dispositivo 3	19	
controlo de dispositivo 4	20	
confirmação de recepção negativa	21	
inacção síncrona	22	
fim de transmissão de bloco	23	
cancelamento	24	
fim de meio	25	
substituto	26	
escape	27	
separador de ficheiro	28	
separador de grupo	29	
separador de registo	30	
separador de unidade	31	

Caracteres especiais e dígitos:		
espaço	32	
ponto de exclamação	33	!
aspa	34	"
cardinal	35	#
dólar	36	$
percentagem	37	%
e comercial	38	&
apóstrofe	39	'
parênteses esquerdo	40	(
parênteses direito	41	)
asterisco	42	*
sinal de adição	43	+
vírgula	44	,
hífen	45	-
ponto	46	.
barra	47	/
dígito 0	48	0
dígito 1	49	1
dígito 2	50	2
dígito 3	51	3
dígito 4	52	4
dígito 5	53	5
dígito 6	54	6
dígito 7	55	7
dígito 8	56	8
dígito 9	57	9
dois pontos	58	:

ponto e vírgula	59	;
menor	60	<
igual	61	=
maior	62	>
ponto de interrogação	63	?
Letras maiúsculas:		
arroba	64	@
A maiúsculo	65	A
B maiúsculo	66	B
C maiúsculo	67	C
D maiúsculo	68	D
E maiúsculo	69	E
F maiúsculo	70	F
G maiúsculo	71	G
H maiúsculo	72	H
I maiúsculo	73	I
J maiúsculo	74	J
K maiúsculo	75	K
L maiúsculo	76	L
M maiúsculo	77	M
N maiúsculo	78	N
O maiúsculo	79	O
P maiúsculo	80	P
Q maiúsculo	81	Q
R maiúsculo	82	R
S maiúsculo	83	S
T maiúsculo	84	T
U maiúsculo	85	U
V maiúsculo	86	V
W maiúsculo	87	W
X maiúsculo	88	X
Y maiúsculo	89	Y
Z maiúsculo	90	Z
parênteses recto esquerdo	91	[
barra reversa	92	\
parênteses recto direito	93	]
acento circunflexo	94	^
sublinhado	95	_
Letras minúsculas:		
acento grave	96	`
a minúsculo	97	a
b minúsculo	98	b
c minúsculo	99	c

d minúsculo	100	d
e minúsculo	101	e
f minúsculo	102	f
g minúsculo	103	g
h minúsculo	104	h
i minúsculo	105	i
j minúsculo	106	j
k minúsculo	107	k
l minúsculo	108	l
m minúsculo	109	m
n minúsculo	110	n
o minúsculo	111	o
p minúsculo	112	p
q minúsculo	113	q
r minúsculo	114	r
s minúsculo	115	s
t minúsculo	116	t
u minúsculo	117	u
v minúsculo	118	v
w minúsculo	119	w
x minúsculo	120	x
y minúsculo	121	y
z minúsculo	122	z
chaveta esquerda	123	{
barra vertical	124	|
chaveta direita	125	}
til	126	˜
apagar	127	
Caracteres de controlo estendidos:		
	128 a 159	
Caracteres especiais:		
espaço inquebrável	160	
ponto de esclamação invertido	161	¡
cêntimo	162	¢
libra esterlina	163	£
moeda geral (Latin-9: euro)	164	¤ (\euro)
iene	165	¥
barra vertical interrompida (Latin-9: S maiúsculo caron)	166	¦ (Š)
secção	167	§
trema (Latin-9: s minúsculo caron)	168	¨ (š)
direitos reservados	169	©
ordinal feminino	170	ª
aspa angular esquerda	171	«

negação	172	¬
hífen curto	173	
marca registada	174	®
vogal longa	175	¯
grau	176	°
mais ou menos	177	±
2 elevado	178	²
3 elevado	179	³
acento agudo (Latin-9: Z maiúsculo caron)	180	´ (Ž)
micro	181	μ
parágrafo	182	¶
ponto intermédio	183	·
cedilha (Latin-9: z minúsculo caron)	184	¸ (ž)
1 elevado	185	¹
ordinal masculino	186	º
aspa angular direita	187	»
fracção 1/4 (Latin-9: OE ligado maiúsculo)	188	$\frac{1}{4}$ (Œ)
fracção 1/2 (Latin-9: oe ligado minúsculo)	189	$\frac{1}{2}$ (œ)
fracção 3/4 (Latin-9: Y maiúsculo trema)	190	$\frac{3}{4}$ (Ÿ)
ponto de interrogação invertido	191	¿
Letras maiúsculas Latin-1:		
A maiúsculo grave	192	À
A maiúsculo agudo	193	Á
A maiúsculo circunflexo	194	Â
A maiúsculo til	195	Ã
A maiúsculo trema	196	Ä
A maiúsculo círculo	197	Å
AE ligado maiúsculo	198	Æ
C maiúsculo cedilha	199	Ç
E maiúsculo grave	200	È
E maiúsculo agudo	201	É
E maiúsculo circunflexo	202	Ê
E maiúsculo trema	203	Ë
I maiúsculo grave	204	Ì
I maiúsculo agudo	205	Í
I maiúsculo circunflexo	206	Î
I maiúsculo trema	207	Ï
Eth maiúsculo	208	Ð
N maiúsculo til	209	Ñ
O maiúsculo grave	210	Ò
O maiúsculo agudo	211	Ó
O maiúsculo circunflexo	212	Ô
O maiúsculo til	213	Õ

O maiúsculo trema	214	Ö
sinal de multiplicação	215	×
O maiúsculo cortado	216	Ø
U maiúsculo grave	217	Ù
U maiúsculo agudo	218	Ú
U maiúsculo circunflexo	219	Û
U maiúsculo trema	220	Ü
Y maiúsculo agudo	221	Ý
Thorn maiúsculo	222	Þ
SS ligado minúsculo	223	ß
Letras minúsculas Latin-1:		
a minúsculo grave	224	à
a minúsculo agudo	225	á
a minúsculo circunflexo	226	â
a minúsculo til	227	ã
a minúsculo trema	228	ä
a minúsculo círculo	229	å
ae ligado minúsculo	230	æ
c minúsculo cedilha	231	ç
e minúsculo grave	232	è
e minúsculo agudo	233	é
e minúsculo circunflexo	234	ê
e minúsculo trema	235	ë
i minúsculo grave	236	ì
i minúsculo agudo	237	í
i minúsculo circunflexo	238	î
i minúsculo trema	239	ï
eth minúsculo	240	ð
n minúsculo til	241	ñ
o minúsculo grave	242	ò
o minúsculo agudo	243	ó
o minúsculo circunflexo	244	ô
o minúsculo til	245	õ
o minúsculo trema	246	ö
sinal de multiplicação	247	÷
o minúsculo cortado	248	ø
u minúsculo grave	249	ù
u minúsculo agudo	250	ú
u minúsculo circunflexo	251	û
u minúsculo trema	252	ü
y minúsculo agudo	253	ý
thorn minúsculo	254	þ
y minúsculo trema	255	ÿ

Apêndice H

Listas e iteradores: listagens

Este apêndice contém as listagens completas das várias versões do módulo físico `lista_int` desenvolvidas ao longo do Capítulo 10 e do Capítulo 11.

H.1 Versão simplista

Corresponde à versão desenvolvida no início do Capítulo 10, que não usa ponteiros nem variáveis dinâmicas, e na qual os itens são guardados numa matriz pela mesma ordem pela qual ocorrem na lista.

H.1.1 Ficheiro de interface: `lista_int.H`

Este ficheiro contém a interface do módulo `lista_int`. Contém também a declaração dos membros privados das classes, o que corresponde a uma parte da implementação:

```
#ifndef LISTA_INT_H
#define LISTA_INT_H

#include <iostream>

/** Representa listas de itens do tipo Item. Item é actualmente é um sinónimo
      de int, mas que pode ser alterado facilmente para o tipo que se entender.
      Por convenção, chama-se "frente" e "trás" ao primeiro e último item na lista.
      Os nomes "primeiro", "último", "início" e "fim" são reservados para iteradores. */
class ListaDeInt {
  public:

    /** Sinónimo de int. Usa-se para simplificar a tarefa de criar listas com
          itens de outros tipos. */
    typedef int Item;
```

```
/* Declaração de uma classe embutida que serve para percorrer e manipular
   listas: */
class Iterador;


// Construtores:

/// Construtor da classe, cria uma lista vazia.
ListaDeInt();


// Inspectores:

/// Devolve o comprimento da lista, ou seja, o seu número de itens.
int comprimento() const;

/// Indica se a lista está vazia.
bool estáVazia() const;

/// Indica se a lista está cheia.
bool estáCheia() const;

/** Devolve referência constante para o item na frente da lista.
    @pre PC ≡ ¬estáVazia(). */
Item const& frente() const;

/** Devolve referência constante para o item na traseira da lista.
    @pre PC ≡ ¬estáVazia(). */
Item const& trás() const;


// Modificadores:

/** Devolve referência para o item na frente da lista. Não modifica
    directamente a lista, mas permite modificações através da referência
    devolvida.
    @pre PC ≡ ¬estáVazia(). */
Item& frente();

/** Devolve referência para o item na traseira da lista.
    @pre PC ≡ ¬estáVazia(). */
Item& trás();

/** Põe novo item na frente da lista. Invalida qualquer iterador associado
    à lista.
```

```
      @pre    PC ≡ ¬estáCheia(). */
void pôeNaFrente(Item const& novo_item);

/** Põe novo item na traseira da lista. Invalida qualquer iterador associado
      à lista.
      @pre    PC ≡ ¬estáCheia(). */
void põeAtrás(Item const& novo_item);

/** Tira o item da frente da lista. Invalida qualquer iterador associado
      à lista.
      @pre    PC ≡ ¬estáVazia(). */
void tiraDaFrente();

/** Tira o item da traseira da lista. Invalida qualquer iterador associado
      à lista.
      @pre    PC ≡ ¬estáVazia(). */
void tiraDeTrás();

/** Esvazia a lista. Invalida qualquer iterador associado
      à lista. */
void esvazia();

/** Insere novo item imediatamente antes da posição indicada pelo iterador
      i. Faz com que o iterador continue a referenciar o mesmo item que
      antes da inserção. Invalida qualquer outro iterador associado
      à lista.
      @pre   PC ≡ ¬estáCheia ∧ iterador ≠ fim() ∧ iterador é válido. */
void insereAntes(Iterador& iterador, Item const& novo_item);
/* Atenção! Em todo o rigor o iterador deveria ser constante, pois o mantém-se
    referenciando o mesmo item! */

/** Remove o item referenciado pelo iterador i. O iterador fica a referenciar
      o item logo após o item removido. Invalida qualquer outro iterador
      associado à lista.
      @pre   PC ≡ iterador ≠ início() ∧ iterador ≠ fim()∧
                   iterador é válido. */
void remove(Iterador& i);


/* Funções construtoras de iteradores. Consideram-se modificadoras porque a lista
    pode ser modificada através dos iteradores. */

/** Devolve um novo iterador inicial, i.e., um iterador referenciando o item fictício
      imediatamente antes do item na frente da lista. */
Iterador início();
```

```
    /** Devolve um novo iterador final, i.e., um iterador referenciando o item fictício
        imediatamente após o item na traseira da lista. */
    Iterador fim();

    /** Devolve um novo primeiro iterador, i.e., um iterador referenciando o item na
        frente da lista. Note-se que se a lista estiver vazia o primeiro iterador é igual
        ao iterador final. */
    Iterador primeiro();

    /** Devolve um novo último iterador, i.e., um iterador referenciando o item na
        traseira da lista. Note-se que se a lista estiver vazia o último iterador é igual
        ao iterador inicial. */
    Iterador último();

  private:

    // O número máximo de itens na lista:
    static int const número_máximo_de_itens = 100;

    // Matriz que guarda os itens da lista:
    Item itens[número_máximo_de_itens];

    // Contador do número de itens na lista:
    int número_de_itens;

    /* Função auxiliar que indica se a condição invariante de instância da classe
       se verifica: */
    bool cumpreInvariante() const;

    // A classe de iteração tem acesso irrestrito às listas:
    friend class Iterador;

};

/** Representa iteradores para itens de listas do tipo ListaDeInt.

    Os iteradores têm uma característica infeliz: podem estar em estados inválidos.
    Por exemplo, se uma lista for esvaziada, todos os iteradores a ela associada
    ficam inválidos. É possível resolver este problema, mas à custa de um aumento
    considerável da complexidade deste par de classes. */
class ListaDeInt::Iterador {
  public:

    // Construtores:

    /** Construtor da classe. Associa o iterador com a lista passada como
```

```
        argumento e põe-no a referenciar o item na sua frente. */
explicit Iterador(ListaDeInt& lista_a_associar);


// Inspectores:

/** Devolve uma referência para o item referenciado pelo iterador.
    Note-se que a referência devolvida não é constante. É que um
    iterador const não pode ser alterado (avançar ou recuar), mas
    permite alterar o item por ele referenciado na lista associada.
    @pre PC ≡ O item referenciado não pode ser nenhum dos itens fictícios
              da lista (i.e., nem o item antes da frente da lista, nem o item
              após a sua traseira) e tem de ser válido. */
Item& item() const;

/** Indica se dois iteradores são iguais. Ou melhor, se a instância implícita
    é igual ao iterador passado como argumento. Dois iteradores são iguais
    se se referirem ao mesmo item da mesma lista (mesmo que sejam itens
    fictícios).
    @pre PC ≡ os iteradores têm de estar associados à mesma lista e ser
              válidos. */
bool operator == (Iterador const& outro_iterador) const;

/** Operador de diferença entre iteradores.
    @pre PC ≡ os iteradores têm de estar associados à mesma lista e ser
              válidos. */
bool operator != (Iterador const& outro_iterador) const;


// Modificadores:

/** Avança iterador para o próximo item da lista. Devolve o próprio iterador.
    @pre PC ≡ O iterador não pode ser o fim da lista associada e tem de ser
              válido. */
Iterador& operator ++ ();

/** Avança iterador para o próximo item da lista. Devolve um novo
    iterador com o valor do próprio iterador antes de avançado.
    @pre PC ≡ O iterador não pode ser o fim da lista associada e tem de ser
              válido. */
Iterador operator ++ (int);

/** Recua iterador para o item anterior da lista. Devolve o próprio iterador.
    @pre PC ≡ O iterador não pode ser o início da lista associada e tem de ser
              válido. */
Iterador& operator -- ();
```

```
        /** Recua iterador para o item anterior da lista. Devolve um novo iterador
                com o valor do próprio iterador antes de recuado.
                @pre  PC ≡ O iterador não pode ser o início da lista associada e tem de ser
                           válido.  */
        Iterador operator -- (int);

      private:

        // Referência para a lista a que o iterador está associado:
        ListaDeInt& lista_associada;

        // Índice do item da lista referenciado pelo iterador:
        int índice_do_item_referenciado;

        // Função auxiliar que indica se a condição invariante de instância
              da classe se verifica:
        bool cumpreInvariante() const;

        /* A classe  ListaDeInt  tem acesso irrestrito a todos os membros da classe
               Iterador. É importante perceber que as duas classes,  ListaDeInt  e
              ListaDeInt::Iterador  estão completamente interligadas. Não há
              qualquer promiscuidade nesta relação. São partes do mesmo todo.  */
        friend ListaDeInt;
    };

    #include "lista_int_impl.H"

    #endif // LISTA_INT_H
```

H.1.2 Ficheiro de implementação auxiliar: `lista_int_impl.H`

Este ficheiro contem a definição de todas as rotinas e métodos em-linha do módulo `lista_int`:

```
    #include <cassert>

    inline ListaDeInt::ListaDeInt()
        : número_de_itens(0) {
        assert(cumpreInvariante());
    }

    inline int ListaDeInt::comprimento() const {
        assert(cumpreInvariante());

        return número_de_itens;
```

```
}

inline bool ListaDeInt::estáVazia() const {
    assert(cumpreInvariante());

    return comprimento() == 0;
}

inline bool ListaDeInt::estáCheia() const {
    assert(cumpreInvariante());

    return comprimento() == número_máximo_de_itens;
}

inline ListaDeInt::Item const& ListaDeInt::frente() const {
    assert(cumpreInvariante());
    assert(not estáVazia());

    return itens[0];
}

inline ListaDeInt::Item const& ListaDeInt::trás() const {
    assert(cumpreInvariante());
    assert(not estáVazia());

    return itens[número_de_itens - 1];
}

inline ListaDeInt::Item& ListaDeInt::frente() {
    assert(cumpreInvariante());
    assert(not estáVazia());

    return itens[0];
}

inline ListaDeInt::Item& ListaDeInt::trás() {
    assert(cumpreInvariante());
    assert(not estáVazia());

    return itens[número_de_itens - 1];
}

inline void ListaDeInt::põeAtrás(Item const& novo_item) {
    assert(cumpreInvariante());
    assert(not estáCheia());
```

```
    itens[número_de_itens++] = novo_item;

    assert(cumpreInvariante());
}

inline void ListaDeInt::tiraDeTrás() {
    assert(cumpreInvariante());
    assert(not estáVazia());

    --número_de_itens;

    assert(cumpreInvariante());
}

inline void ListaDeInt::esvazia() {
    assert(cumpreInvariante());

    número_de_itens = 0;

    assert(cumpreInvariante());
}

inline ListaDeInt::Iterador ListaDeInt::início() {
    assert(cumpreInvariante());

    // Cria-se um iterador para esta lista:
    Iterador iterador(*this);

    iterador.índice_do_item_referenciado = -1;

    /* Em bom rigor não é boa ideia que seja um método da lista a verificar se a codição
       invariante de instância do iterador é verdadeira... Mais tarde se verá melhor
       forma de resolver o problema. */
    assert(iterador.cumpreInvariante());

    return iterador;
}

inline ListaDeInt::Iterador ListaDeInt::fim() {
    assert(cumpreInvariante());

    Iterador iterador(*this);

    iterador.índice_do_item_referenciado = número_de_itens;

    assert(iterador.cumpreInvariante());
```

```
        return iterador;
    }

    inline ListaDeInt::Iterador ListaDeInt::primeiro() {
        assert(cumpreInvariante());

        /* Cria-se um iterador para esta lista, que referencia inicialmente o item na frente
           da lista (ver construtor de ListaDeInt::Iterador), e devolve-se
           imediatamente o iterador criado: */
        return Iterador(*this);
    }

    inline ListaDeInt::Iterador ListaDeInt::último() {
        assert(cumpreInvariante());

        Iterador iterador(*this);

        iterador.índice_do_item_referenciado = número_de_itens - 1;

        assert(iterador.cumpreInvariante());

        return iterador;
    }

    inline bool ListaDeInt::cumpreInvariante() const {
        return 0 <= número_de_itens and
               número_de_itens <= número_máximo_de_itens;
    }

    inline ListaDeInt::Iterador::Iterador(ListaDeInt& lista_a_associar)
        : lista_associada(lista_a_associar),
          índice_do_item_referenciado(0) {
        assert(cumpreInvariante());
    }

    inline ListaDeInt::Item& ListaDeInt::Iterador::item() const {
        assert(cumpreInvariante());
        // assert(é válido);
        assert(*this != lista_associada.início() and
               *this != lista_associada.fim());

        return lista_associada.itens[índice_do_item_referenciado];
    }

    inline bool ListaDeInt::Iterador::
```

```
operator == (Iterador const& outro_iterador) const {
    assert(cumpreInvariante() and
           outro_iterador.cumpreInvariante());
    // assert(é válido and outro_iterador é válido);
    // assert(iteradores associados à mesma lista...);

    return índice_do_item_referenciado ==
           outro_iterador.índice_do_item_referenciado;
}

inline bool ListaDeInt::Iterador::
operator != (Iterador const& outro_iterador) const {
    assert(cumpreInvariante() and
           outro_iterador.cumpreInvariante());
    // assert(é válido and outro_iterador é válido);
    // assert(iteradores associados à mesma lista...);

    return not (*this == outro_iterador);
}

inline ListaDeInt::Iterador& ListaDeInt::Iterador::operator ++ () {
    assert(cumpreInvariante());
    // assert(é válido);
    assert(*this != lista_associada.fim());

    ++índice_do_item_referenciado;

    assert(cumpreInvariante());
    return *this;
}

inline ListaDeInt::Iterador ListaDeInt::Iterador::operator ++ (int) {
    assert(cumpreInvariante());
    // assert(é válido);
    assert(*this != lista_associada.fim());

    ListaDeInt::Iterador resultado = *this;
    operator ++ ();
    return resultado;
}

inline ListaDeInt::Iterador& ListaDeInt::Iterador::operator -- () {
    assert(cumpreInvariante());
    // assert(é válido);
    assert(*this != lista_associada.início());
```

```
        --índice_do_item_referenciado;

        assert(cumpreInvariante());
        return *this;
    }

    inline ListaDeInt::Iterador ListaDeInt::Iterador::operator -- (int) {
        assert(cumpreInvariante());
        // assert(é válido);
        assert(*this != lista_associada.início());

        ListaDeInt::Iterador resultado = *this;
        operator -- ();

        return resultado;
    }

    inline bool ListaDeInt::Iterador::cumpreInvariante() const {
        return -1 <= índice_do_item_referenciado and
            índice_do_item_referenciado <= lista_associada.número_de_itens;
    }
```

H.1.3 Ficheiro de implementação: `lista_int.C`

Este ficheiro contém a função `main()` de teste do módulo `lista_int`. Contem também a definição de todas as rotinas e métodos que não são em-linha do módulo.

```
    #include "lista_int.H"

    void ListaDeInt::põeNaFrente(Item const& novo_item)
    {
        assert(cumpreInvariante());
        assert(not estáCheia());

        for(int i = número_de_itens; i != 0; --i)
            itens[i] = itens[i - 1];

        itens[0] = novo_item;

        ++número_de_itens;

        assert(cumpreInvariante());
    }

    void ListaDeInt::insereAntes(Iterador& iterador,
```

```
                              Item const& novo_item)
{
    assert(cumpreInvariante());
    assert(not estáCheia());
    assert(iterador é válido);
    assert(iterador != início());

    // Há que rearranjar todos os itens a partir do referenciado pelo iterador:
    for(int i = número_de_itens;
        i != iterador.índice_do_item_referenciado;
        --i)
        itens[i] = itens[i - 1];

    // Agora já há espaço (não esquecer de revalidar o iterador!):
    itens[iterador.índice_do_item_referenciado++] = novo_item;

    assert(iterador.cumpreInvariante());

    // Mais um...
    ++número_de_itens;

    assert(cumpreInvariante());
}

void ListaDeInt::tiraDaFrente() {
    assert(cumpreInvariante());
    assert(not estáVazia());

    --número_de_itens;
    for(int i = 0; i != número_de_itens; ++i)
        itens[i] = itens[i + 1];

    assert(cumpreInvariante());
}

void ListaDeInt::remove(Iterador& iterador) {
    assert(cumpreInvariante());
    assert(iterador é válido);
    assert(iterador != início() and iterador != fim());

    --número_de_itens;
    for(int i = iterador.índice_do_item_referenciado;
        i != número_de_itens;
        ++i)
        itens[i] = itens[i + 1];
```

```
    assert(cumpreInvariante());
}

#ifdef TESTE

// Macro definida para encurtar a escrita dos testes:
#define erro(mensagem)                                   \
{                                                        \
    cout << __FILE__ << ":" << __LINE__ << ": " \
         << (mensagem) << endl;                  \
    ocorreram_erros = true;                      \
}

int main()
{
    bool ocorreram_erros = false;

    cout << "Testando módulo físico lista_int..." << endl;

    cout << "Testando classes ListaDeInt e ListaDeInt::Iterador..."
         << endl;
    // Definem-se itens canónicos para usar nos testes para que seja
    // fácil adaptar para tipos de itens diferentes:
    ListaDeInt::Item zero = 0;
    ListaDeInt::Item um = 1;
    ListaDeInt::Item dois = 2;
    ListaDeInt::Item tres = 3;
    ListaDeInt::Item quatro = 4;
    ListaDeInt::Item cinco = 5;
    ListaDeInt::Item seis = 6;
    ListaDeInt::Item sete = 7;
    ListaDeInt::Item oito = 8;
    ListaDeInt::Item nove = 9;
    ListaDeInt::Item dez = 10;

    int const número_de_vários = 11;
    ListaDeInt::Item vários[número_de_vários] = {
        zero, um, dois, tres, quatro, cinco,
        seis, sete, oito, nove, dez
    };

    ListaDeInt l;

    if(l.comprimento() != 0)
        erro("l.comprimento() dvia ser 0.");
```

```
l.põeAtrás(tres);
l.põeNaFrente(dois);
l.põeAtrás(quatro);
l.põeNaFrente(um);

if(l.comprimento() != 4)
    erro("l.comprimento() devia ser 4.");

if(l.frente() != um)
    erro("l.frente() devia ser um.");

if(l.trás() != quatro)
    erro("l.trás() devia ser um.");

ListaDeInt::Iterador i = l.início();

++i;

if(i != l.primeiro())
    erro("i devia ser l.primeiro().");

if(i.item() != um)
    erro("i.item() devia ser um.");

++i;

if(i.item() != dois)
    erro("i.item() devia ser dois.");

++i;

if(i.item() != tres)
    erro("i.item() devia ser tres.");

++i;

if(i.item() != quatro)
    erro("i.item() devia ser quatro.");

++i;

if(i != l.fim())
    erro("i devia ser l.fim().");

--i;
```

```
if(i != l.último())
    erro("i devia ser l.último().");

if(i.item() != quatro)
    erro("i.item() devia ser quatro.");

--i;

if(i.item() != tres)
    erro("i.item() devia ser tres.");

--i;

if(i.item() != dois)
    erro("i.item() devia ser dois.");

--i;

if(i.item() != um)
    erro("i.item() devia ser um.");

if(i != l.primeiro())
    erro("i devia ser l.primeiro().");

--i;

if(i++ != l.início())
    erro("i devia ser l.início().");

++i;

l.insereAntes(i, cinco);

if(i.item() != dois)
    erro("i.item() devia ser dois.");

--i;

if(i.item() != cinco)
    erro("i.item() devia ser cinco.");

i--;

l.remove(i);

if(i.item() != cinco)
```

```
        erro("i.item() devia ser cinco.");

    if(--i != l.início())
        erro("i devia ser l.início().");

    ++i;
    i++;
    ++i;
    i++;

    l.remove(i);

    if(i != l.fim())
        erro("i devia ser l.fim().");

    if(l.frente() != cinco)
        erro("l.frente() devia ser cinco.");

    if(l.trás() != tres)
        erro("l.trás() devia ser tres.");

    l.tiraDaFrente();

    if(l.frente() != dois)
        erro("l.frente() devia ser dois.");

    l.tiraDeTrás();

    if(l.trás() != dois)
        erro("l.frente() devia ser dois.");

    l.tiraDeTrás();

    if(l.comprimento() != 0)
        erro("l.comprimento() devia ser 0.");

    int número_de_colocados = 0;
    while(número_de_colocados != número_de_vários and
          not l.estáCheia())
        l.põeAtrás(vários[número_de_colocados++]);

    if(l.comprimento() != número_de_colocados)
        erro("l.comprimento() devia ser número_de_colocados.");

    ListaDeInt::Iterador j = l.último();
    while(not l.estáVazia()) {
```

```
        --número_de_colocados;

        if(l.trás() != vários[número_de_colocados])
            erro("l.trás() devia ser "
                 "vários[número_de_colocados].");

        if(j.item() != vários[número_de_colocados])
            erro("i.item() devia ser "
                 "vários[número_de_colocados].");

        --j;
        l.tiraDeTrás();
    }

    if(número_de_colocados != 0)
        erro("número_de_colocados devia ser 0.");

    if(j != l.início())
        erro("i devia ser l.início().");

    ++j;

    l.insereAntes(j, seis);

    --j;

    l.insereAntes(j, sete);

    l.insereAntes(j, oito);

    --j;

    l.insereAntes(j, nove);

    l.insereAntes(j, dez);

    l.remove(j);

    l.remove(j);

    --j;
    --j;
    --j;

    l.remove(j);
    l.remove(j);
```

```
    if(j.item() != dez)
        erro("j.item() devia ser dez.");

    if(l.trás() != dez)
        erro("l.trás() devia ser dez.");

    if(l.frente() != dez)
        erro("l.frente() devia ser dez.");

    if(l.comprimento() != 1)
        erro("l.comprimento() devia ser 1.");

    l.insereAntes(j, um);
    l.insereAntes(j, dois);
    l.insereAntes(j, tres);
    l.insereAntes(j, quatro);
    l.insereAntes(j, cinco);

    ListaDeInt const lc = l;

    l.esvazia();

    if(not l.estáVazia())
        erro("l.estáVazia() devia ser true.");

    if(lc.frente() != um)
        erro("lc.frente() devia ser um.");

    if(lc.trás() != dez)
        erro("lc.trás() devia ser dez.");

    cout << "Testes terminados." << endl;

    return ocorreram_erros? 1 : 0;
}

#endif // TESTE
```

Apêndice I

Listas e iteradores seguros

!!!!!Repensar listas e iteradores de modo a se saber sempre a validade dos iteradores.

Bibliografia

[1] Kent Beck. *Extreme Programming Explained: Embrace Change*. Adison-Wesley, Boston, 1999. 347

[2] Doug Bell, Ian Morrey, e John Pugh. *Software Engineering: A Programming Approach*. Prentice Hall, Nova Iorque, 2ª edição, 1992. 61

[3] Samuel D. Conte e Carl de Boor. *Elementary Numerical Analysis*. McGraw-Hill International, Auckland, 1983. 39

[4] Aurélio Buarque de Holanda Ferreira. *Novo Aurélio Século XXI: O Dicionário da Lingua Portuguesa*. Editora Nova Fronteira, Rio de Janeiro, 3ª edição, 1999. 261, 453, 523

[5] Edsger Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976. 204, 206

[6] Irv Englander. *The Architecture of Computer Hardware and Systems Software: An Information Technology Approach*. John Wiley & Sons, Inc., Nova Iorque, 1996. 32, 37, 38

[7] Ronald L. Graham, Donald E. Knuth, e Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 2ª edição, 1994. 312

[8] David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, Nova Iorque, 1981 [1989]. 147, 189, 201, 204, 285, 588

[9] Michael K Johnson e Erik W. Troan. *Linux Application Development*. Addison-Wesley, Reading, Massachusetts, 1998. 469

[10] Donald E. Knuth. *Fundamental algorithms*, volume 1 de *The Art of Computer Programming*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2ª edição, 1973. 3, 4, 189

[11] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, Upper Saddle River, New Jersey, 2ª edição, 1997. 110, 285

[12] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 3ª edição, 1998. 20, 176, 291, 297, 445

[13] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice-Hall, 3ª edição, 1989. 393

Índice

algoritmia, 4
algoritmo, 3
    análise, 4
    eficiência, 3
assembladores, 2
assemblers, 2

compiladores, 2
computador, 1

Definitude, 3
discos, 2

Eficácia, 3
Entrada, 3
existe um, *ver* quantificador existencial

Finitude, 3

linguagem
    alto nível, 2
    assembly, 2
    declarativa, 2
    imperativa, 2
    máquina, 2
    natural, 2
    programação, 2

método computational, 4
memória, 2
mnemónica, 2

pré-condição, 10
processador, 1
programa, 3

qualquer que seja, *ver* quantificador universal
quantificador, 588
    contagem, 589
    existencial, 589
    produto, 588
    soma, 588
    universal, 588

registos, 1

Saída, 3
somatório, *ver* quantificador soma