Programação Orientada para Objectos

Manuel Menezes de Sequeira

27 de Fevereiro de 2001

Conteúdo

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 problemas, mesmo em áreas totalmente desconexas, criatividade, intuição, engenho, persis-

[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.

tência, 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 [8] 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.

[2] Definitude: qualidade daquilo que é definido. É um neologismo introduzido pelo autor como tradução do inglês *definite*.

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 [8]. 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 [8] 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 todo o trabalho subsequente noutras 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,

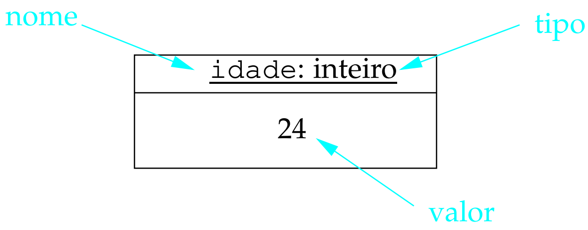


Figura 1.1: Notação para uma variável. A azul explicações sobre a notação gráfica usada.

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 varáveis 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., $PC \equiv \mathtt{n} \geq 0$.

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 $CO \equiv \texttt{soma} = \sum_{j=0}^{\texttt{n}-1} j$, ou seja, a variável `soma` deve conter a soma dos inteiros entre 0 e $\texttt{n}-1$ 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 considerar-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. 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`, 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 aumentar o valor de `i` de uma unidade. Quando o valor de `i` atingir `n` o algoritmo termina. Um pouco mais formalmente[4]:

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

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 '?'.

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

[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 $\frac{\texttt{n}(\texttt{n}-1)}{2}$...

`n`: inteiro
4

`soma`: inteiro
?

`i`: inteiro
?

Figura 1.2: Estado inicial do algoritmo.

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

`n`: inteiro
4

`soma`: inteiro
6

`i`: inteiro
4

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:

$\{PC \equiv \texttt{n} \geq 0.\}$

1

$\texttt{i} \leftarrow 0$

2

$\texttt{soma} \leftarrow 0$

3

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

4

  soma ← soma + i

5

  i ← i + 1

6

fim do enquanto .

7

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

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.

Tabela 1.1: Traçado do algoritmo.

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

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

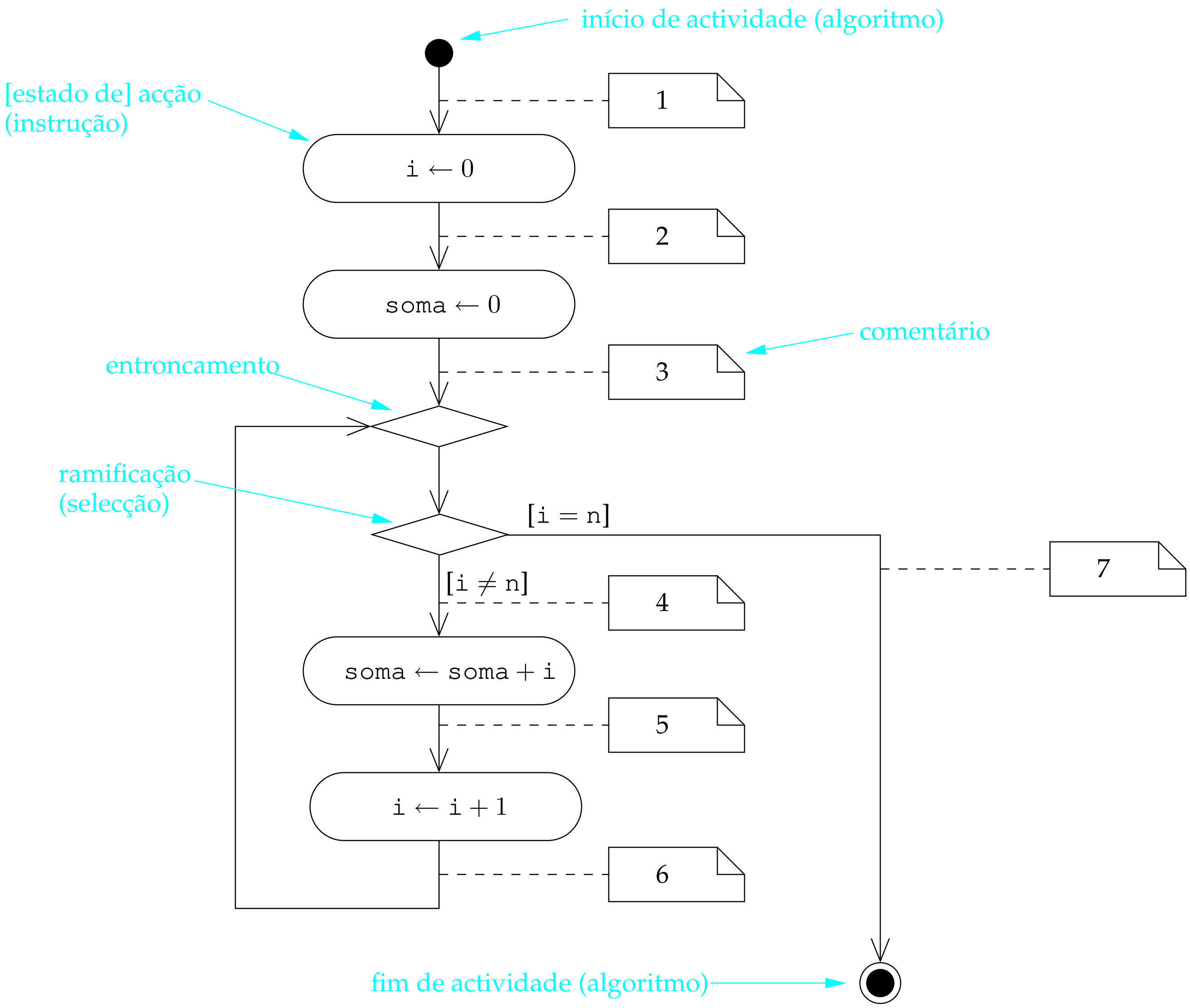
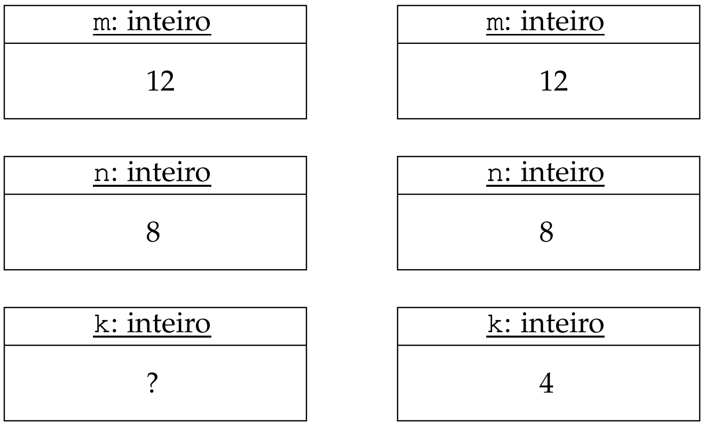


Figura 1.4: Diagrama de actividade do algoritmo. As setas indicam a sequência de execução das acções (instruções). Os losangos têm significados especiais: servem para unir fluxos de execução alternativos (entroncamentos) ou para os começar (ramificações). Em cada um dos ramos saídos de uma ramificação indicam-se as condições que têm de se verificar para que seja escolhido esse ramo. A azul explicações sobre a notação gráfica usada.

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 tem de começar por instruções dizendo para os dois valores serem lidos (ou ouvidos) e guardados em duas variáveis, a que se darão os nomes `m` e `n`, e terminar por um instrução dizendo para o máximo divisor comum ser escrito (ou dito). Para isso é necessária uma variável adicional, a que se dará o nome de `k`. Os passos intermédios do algoritmo indicam a forma de cálculo do máximo divisor comum, e são os que nos interessam aqui. Ou seja, o nosso problema é, dados dois valores quaisquer 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) 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.

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

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

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

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

onde se assume que `m` e `n` não mudam de valor ao longo do algoritmo. O objectivo do problema é pois encontrar um `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 problema

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

em causa não há quaisquer dúvidas: todos os pares de inteiros positivos têm um 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 PC e a CO, 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, $\mathrm{mdc}(m, n) \geq 1$.

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 $m \div k$ uma notação que significa "o resto da divisão (inteira) de m por k". Duas outras propriedades que se verificará serem importantes são:

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

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

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 $\mathrm{mdc}(m, n) > k$, isto é, há divisores comuns maiores que k. Mas então $\mathrm{mdc}(m, n) > k \geq mdc(m, n)$, que é uma falsidade! Logo, $k = \mathrm{mdc}(m, n)$.

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

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

Ou seja, se k é superior ou igual ao máximo divisor comum de m e n e se k 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. Então, se se quiser começar pelo topo, deve-se atribuir a `k` (ou seja, registar na "caixa" `k`) o menor dos valores `m` e `n`:

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

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

Em seguida, deve-se verificar se `k` divide `m` e `n` e, no caso contrário, passar ao próximo candidato:

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 temos

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

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, `k` é o mdc. Como demonstrá-lo? Repare-se que, dadas as propriedades c) e d), quer a inicialização de `k` quer a guarda e o progresso garantem que há uma condição que se mantém sempre válida, desde o início ao fim do ciclo: a chamada condição invariante (CI) do ciclo. Neste caso

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

Dada a inicialização

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

e a PC, tem-se que $\mathtt{k} = \min(\mathtt{m}, \mathtt{n})$, o que, conjugado com a propriedade b), garante que $\mathtt{k} \geq \mathrm{mdc}(\mathtt{m}, \mathtt{n})$.

Durante o ciclo, que acontece se a G for verdadeira? Diminui-se k de uma unidade, isto é progride-se[7]. Mas, admitindo que a CI é verdadeira e sendo G também verdadeira, conclui-se pela propriedade d) que

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

Mas, como o valor guardado em k é inteiro, isso é o mesmo que dizer que $\mathtt{k} - 1 \geq \mathrm{mdc}(\mathtt{m}, \mathtt{n})$. Donde, depois do progresso $\mathtt{k} \leftarrow \mathtt{k} - 1$, a veracidade da CI é recuperada, i.e., $\mathtt{k} \geq \mathrm{mdc}(\mathtt{m}, \mathtt{n})$.

No fundo, demonstrou-se por indução que, como CI se verifica do início do ciclo, também se verifica durante todo o ciclo, *inclusive quando este termina*. Ou seja, demonstrou-se que CI é de facto invariante.

Que acontece se, durante o ciclo, G for falsa? Nesse caso o ciclo termina e conclui-se que há duas condições verdadeiras: CI e também $\neg G$ (onde $\neg$ representa a negação). Mas a conjunção destas condições, pela propriedade c), implica que a CO é verdadeira, i.e., que k termina de facto com o máximo divisor comum de m e n.

A demonstração da correcção do algoritmo passa pois por demonstrar que a CI é verdadeira do início ao fim do ciclo (é invariante) e que quando o ciclo termina (i.e., quando a G é falsa) se tem forçosamente que a CO é verdadeira, i.e.,

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

Assim, o algoritmo completo é:

$\{PC \equiv \mathtt{m} > 0 \wedge \mathtt{n} > 0.\}$
se $\mathtt{m} < \mathtt{n}$ **então:**
    $\mathtt{k} \leftarrow \mathtt{m}$
senão:
    $\mathtt{k} \leftarrow \mathtt{n}$
{*Aqui sabe-se que* $k = \min(m, n) \geq \mathrm{mdc}(m, n)$*, 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..*}

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

onde o texto entre chavetas corresponde a comentários, não fazendo parte do algoritmo.

Uma questão importante que ficou por demonstrar é se é garantido que o ciclo termina sempre. Se não se garantir não se pode chamar a esta sequência de instruções um algoritmo.

A demonstração é simples. Em primeiro lugar, a inicialização garante que o valor inicial de `k` é ≥ 1 (pois `m` e `n` são inteiros positivos). O progresso, por outro lado, faz o valor de `k` diminuir a cada repetição. Como 1 é divisor comum de qualquer par de inteiros positivos, o ciclo, na pior das hipóteses, termina com $\texttt{k} = 1$ ao fim de no máximo $\min(\texttt{m}, \texttt{n}) - 1$ repetições. Logo, é de facto um algoritmo, pois verifica a condição de finitude.

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

1. é finito, terminando sempre ao fim de no máximo $\min(\texttt{m}, \texttt{n}) - 1$ repetições (ou 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 $\texttt{k} = \mathrm{mdc}(\texttt{m}, \texttt{n})$; 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 na prática, muito embora existam algoritmos muito mais eficientes, que serão abordados em aulas posteriores. A análise da eficiência deste algoritmo fica como exercício de Computação e Algoritmia...

O algoritmo apresentado assume que as variáveis `m` e `n` contêm inicialmente os valores das entradas do algoritmo. Como eles lá vão parar não é especificado. De igual forma não se especifica como o valor da variável `k` vai parar à saída no final do algoritmo.

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.

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., verificando a PC 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

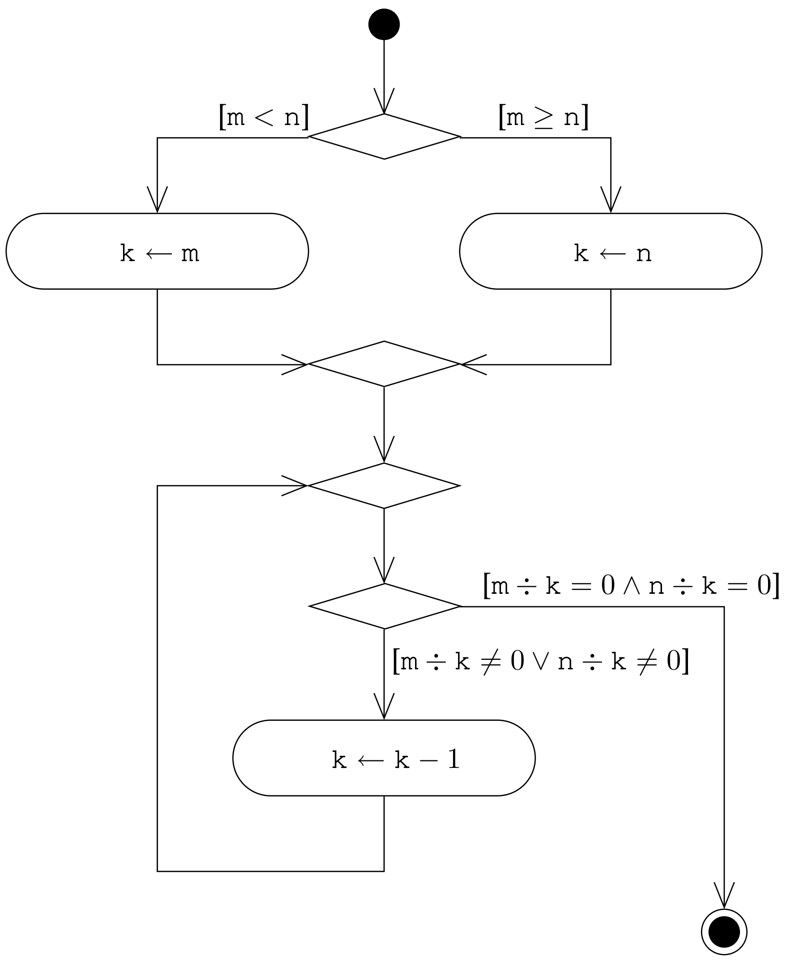


Figura 1.6: Diagrama de actividade correspondente ao algoritmo para cálculo do mdc.

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 CO, ou seja ($\mathbf{Q}$ significa qualquer que seja),

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

Finalmente, para o leitor mais interessado, fica a informação 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 $a = 0 \wedge b > 0$, então $\mathrm{mdc}(a, b) = b$.

Estas propriedades permitem, em particular, desenvolver o chamado algoritmo de Euclides.

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 a linguagem 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 e escrita da saída. 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.

```
#include <iostream>

using namespace std;
```

```
/// Este programa calcula o máximo divisor comum de dois números.
int main()
{
    cout << "Máximo divisor comum de dois números."
         << endl;
    cout << "Introduza dois inteiros positivos: ";
    int m, n;
    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(n > m)
        k = m;
    else
        k = n;
    // Neste momento sabe-se que k ≥ mdc(m, n).

    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) ∧ k ≥ mdc(m, n). Ou seja, k > mdc(m, n).
        --k; // Progresso. Afinal, o algoritmo tem de terminar...
        // Neste momento, k ≥ mdc(m, n) outra vez! É o invariante do ciclo!
    }

    // Como k ≥ mdc(m, n) (invariante do ciclo) e k divide m e n
    // (o ciclo terminou, não foi?), conclui-se que k = mdc(m, n)!
    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 (entre [] indica-se quem executa o respectivo passo):

1. Especificação do problema [humano].
2. Desenvolvimento de um algoritmo que resolve o problema [humano]. É neste passo que se faz mais uso da inteligência e criatividade.
3. Concretização do algoritmo na linguagem de programação: desenvolvimento do programa [humano]. Este passo é mecânico e relativamente pouco interessante.
4. Tradução do programa para linguagem máquina [computador, ou melhor, compilador].

5. Execução do programa para resolver um problema particular (e.g., cálculo de $\mathrm{mdc}(131, 47)$) [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 dramáticos, daí que seja 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 `1` em vez de `l` (letra 'l', por exemplo o nome de uma variável) pode ter consequências muito graves. 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

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 repetidamente ou alternativamente; ou
3. definem (ou declaram) entidades (variáveis, constantes, funções, etc.).

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

2.1 Introdução

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

```
/* 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>
```

[1]E não só, como se verá.

```
using namespace std;

/* A linha seguinte indica o ponto onde o programa vai começa 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 em aulas posteriores):
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 [10, pág.171], sendo usado aqui apenas para simplificar a apresentação sem tornar os programas inválidos (muitas vezes estas duas linhas iniciais não serão incluídas nos exemplos, devendo o leitor estar atento a esse facto se resolver compilar esses exemplos). Os espaços nominativos serão abordados no Capítulo 8.

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

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

```
#include <iostream>

using namespace std;

/// Este programa calcula o máximo divisor comum de dois números.
int main()
{
    cout << "Máximo divisor comum de dois números."
         << endl;
```

```
    cout << "Introduza dois inteiros positivos: ";
    int m, n;
    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(n > m)
        k = m;
    else
        k = n;
    // Neste momento sabe-se que k ≥ mdc(m, n).

    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) ∧ k ≥ mdc(m, n). Ou seja, k > mdc(m, n).
        --k; // Progresso. Afinal, o algoritmo tem de terminar...
        // Neste momento, k ≥ mdc(m, n) outra vez! É o invariante do ciclo!
    }

    // Como k ≥ mdc(m, n) (invariante do ciclo) e k divide m e n
    // (o ciclo terminou, não foi?), conclui-se que k = mdc(m, n)!
    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(n > m)
    k = m;
else
    k = n;
// Neste momento sabe-se que k ≥ mdc(m, n).

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) ∧ k ≥ mdc(m, n). Ou seja, k > mdc(m, n).
    --k; // Progresso. Afinal, o algoritmo tem de terminar...
    // Neste momento, k ≥ mdc(m, n) outra vez! É o invariante do ciclo!
}
```

As duas partes restantes resolvem dois problemas que foram subtilmente ignorados na 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. Tudo depende da aplicação que se pretende dar ao programa. Possibilidades seriam as entradas serem originadas:

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

para apresentar apenas alguns exemplos.

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.

2.1.1 Consola e canais

É típico os programas actuais usarem interfaces mais ou menos sofisticadas com o seu utilizador. Ao longo destas folhas, 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. Admite-se ainda que as entradas do programa serão feitas por um utilizador usando o teclado (ou que vêem de um ficheiro de texto no disco rígido do computador) e que as saídas serão escritas na própria consola de comandos (ou que são colocadas num ficheiro 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 caractere (símbolo gráfico) dos disponíveis na tabela de codificação usada na máquina em causa (ver explicação mais abaixo). Quando um programa é executado, o que se consegue normalmente escrevendo na consola o seu nome (precedido de `./`) depois do "pronto" (*prompt*), a grelha de células fica à disposição do programa a ser executado. As saídas do programa fazem-se escrevendo caracteres nessa grelha, i.e., inserindo-os no chamado *canal de saída*. Pelo contrário as entradas fazem-se lendo caracteres do teclado, i.e., extraindo-os do chamado *canal de entrada*[2]. 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

[2]Optou-se por traduzir *stream* por canal em vez de fluxo, com se faz noutros textos, por parecer uma abstracção mais apropriada.

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[3] 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.
```

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 pedir ao utilizador para introduzir dois números inteiros positivos, a segunda serve para definir as variáveis para onde as variáveis serão lidas e a terceira procede à leitura 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[4].

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:

[3] Para simplificar chamar-se-á muitas vezes "ecrã" à grelha de células que constitui a consola.

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

```
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`

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

o resultado é

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

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

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

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[5].

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

[5] 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);

    ...
}
```

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 é feito de cima para baixo e da esquerda para a direita ao longo do código 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 a veracidade de uma determinada condição. No programa existe uma instrução de selecção:

```
if(n > m)
    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 `n` $>$ `m`, 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 a condição entre parênteses logo após a palavra-chave `if`. Uma instrução de selecção pode seleccionar entre duas 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*.

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. Neste caso 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`, e consiste simplesmente na aplicação do operador de decrementação prefixa, que neste caso se limita a diminuir de um o valor guardado em `k`. A segunda é a expressão usada como condição do ciclo. Nesta expressão encontram-se três operadores:

1. O operador `or` que calcula a disjunção dos valores lógicos de duas expressões condicionais.
2. O operador `!=`, que tem valor $\mathcal{V}$ se os seus operandos forem diferentes e o valor $\mathcal{F}$ se eles forem iguais. A notação usada para a diferença deve-se à inexistência do símbolo $\neq$ na generalidade dos teclados (e tabelas de codificação).
3. O operador `%`, cujo valor é o resto da divisão inteira do primeiro operando pelo segundo. A notação usada (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 2.6.

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*[6], os famosos "zeros e uns", colocados na zona de memória atribuída a essa variável. O tipo de uma variável indica simplesmente como um dado padrão de *bits* deve ser interpretado.

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

Cada variável tem duas características estáticas, um nome e um tipo, e uma característica dinâmica, o seu valor[7], 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 criada, i.e., ficar associada a uma posição de memória (ou várias posições de memória), e de modo a que a forma de interpretar os padrões de *bits* nessa posição de memória fique estabelecida. Uma instrução onde se cria 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. Outras restrições são que a 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 que tem de conter sempre um valor, como se 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.

`a: int`
10

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 na memória o número de *bits* necessário para guardar um valor do tipo referido (essas reservas são feitas em múltiplos de uma unidade básica de memória, tipicamente com oito *bits*, ou seja, um octeto ou *byte*[8]). Se a variável não for explicitamente inicializada, essa posição de memória contém um padrão de *bits* arbitrário. Por exemplo, se se tivesse usado a definição

```
int a;
```

a variável `a` conteria um padrão de *bits* arbitrário e portanto um valor inteiro arbitrário. Para evitar esta arbitrariedade, que pode ter consequências nefastas num programa se não se tiver cuidado, ao definir uma variável deve-se, sempre que possível e razoável, atribuir-se-lhe um valor inicial como indicado na primeira definição. A atribuição de um valor a uma variável

[7] Mais tarde se aprenderá que existe um outro género de variáveis que não têm nome. São as variáveis dinâmicas introduzidas no Capítulo 9.

[8] Em rigor a dimensão dos *bytes* pode variar de máquina para máquina.

que acabou de ser definida é chamada a inicialização. Esta operação pode ser feita de várias formas[9]:

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

A sintaxe das definições de variáveis também permite que se definam 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 Nomes de variáveis

Os nomes de variáveis (e em geral de todos os identificadores em C++) podem ser constituídos por letras (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. O nome de uma variável 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 (nome), e não como o identificador `lê` seguido do identificador `Valor`.

Os nomes das variáveis devem ser tão auto-explicativos 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`, 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 de variáveis

As posições de memória correspondentes às variáveis contêm sempre um padrão de *bits*: não há posições "vazias". Assim, as variáveis têm sempre um valor qualquer. Depois de uma definição, se não for feita uma inicialização explícita conforme sugerido mais atrás, a variável definida contém um valor arbitrário[10]. 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[11]).

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

[10]Desde que seja de um tipo básico do C++ e pouco mais. A afirmação não é verdadeira para classes em geral (ver Capítulo 6).

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

Existem algumas circunstâncias nas quais as variáveis de tipos básicos do C++ que são inicializadas implicitamente com zero (ver Secção 3.2.12). Deve-se evitar fazer uso desta característica do C++ e inicializar sempre explicitamente.

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 do C++. Posteriormente estudar-se-ão duas filosofias de programação diferentes que passam por acrescentar à linguagem novos tipos mais apropriados para os problemas em causa: programação baseada em objectos (Capítulo 6) e programação orientada para objectos (Capítulo 10).

Nas tabelas seguintes são apresentados os tipos básicos existentes no C++ e a gama de valores que podem representar em Linux sobre Intel[12]. É muito importante notar que os computadores 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 [5])	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

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

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

[12] Ambas as tabelas se referem ao compilador de C++ `g++` da GNU Compiler Collection (GCC 2.91.66, egcs 1.1.2) num sistema Linux, distribuição Redhat 6.0, núcleo 2.2.5, correndo sobre a arquitectura Intel, localizado para a Europa Ocidental.

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 [5])	$2{,}2250738585072014 \times 10^{-308}$ a $1{,}7976931348623157 \times 10^{308}$ (e negativos)	64
`long double`	número racional (representação IEEE 754 [5])	$3{,}36210314311209350626 \times 10^{-4932}$ a $1{,}18973149535723176502 \times 10^{4932}$ (e negativos)	96

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, que muitas vezes 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 usados 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: considere-se que o valor representado é $(b_3 b_2 b_1 b_0)_2$, i.e., que os *bits* são dígitos de um número expresso na base binária (ou seja, na base 2)[13] . Por exemplo:

1	0	0	1

é o padrão de *bits* correspondente ao valor $(1001)_2$, ou seja 9 em decimal[14].

Suponha-se agora que a variável contém

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[15].

A extensão destas ideias para, por exemplo, 32 *bits* é trivial: nesse caso o (grande) relógio 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. Em particular os valores das variáveis do tipo `int` não podem crescer indefinidamente. Ao se atingir o topo do relógio volta-se a zero!

[13] O termo *bit* significa dígito binário (*binary digit*).

[14] 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. 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 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 $(d_{n-1} d_{n-2} \cdots d_1 d_0)_B$ 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 $(d_{n-1} d_{n-2} \cdots d_1 d_0)_B$ pode ser calculado como

$$\sum_{i=0}^{n-1} d_i B^i = d_{n-1} B^{n-1} + d_{n-2} B^{n-2} + \cdots + d_1 B^1 + d_0 B^0.$$

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.

[15] 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 231).

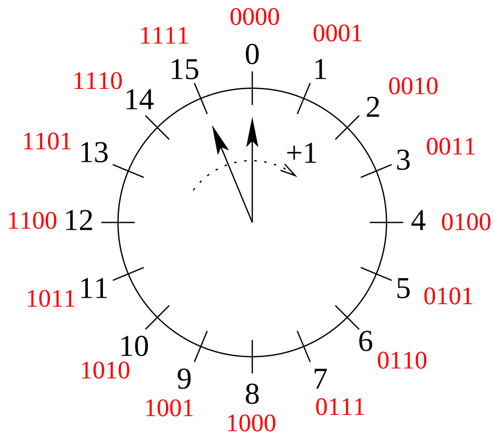


Figura 2.2: Representação de inteiros sem sinal com quatro *bits*.

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 $x + 1 = 0$, donde se conclui ser o padrão 1111 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.

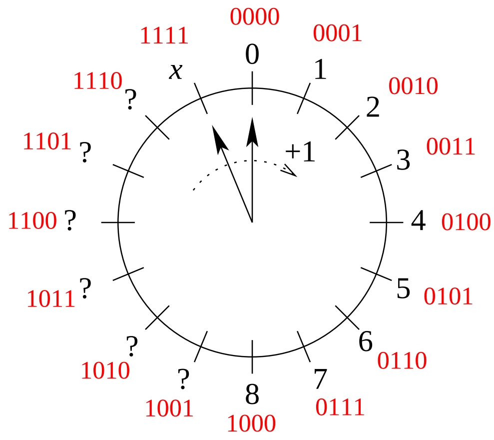


Figura 2.3: Como representar os inteiros com sinal?

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

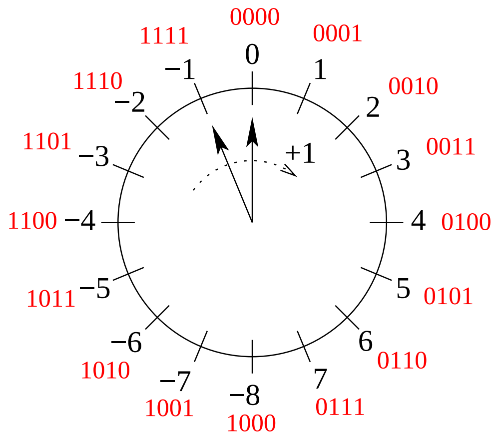


Figura 2.4: Representação de inteiros com sinal com quatro *bits*.

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 representada na tabela acima prende-se com o facto de que, dessa forma, a distinção entre não-negativos (positivos ou zero) e negativos se pode 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 as vantagens desta representação para simplificação do *hardware* do computador encarregue de fazer operações com valores inteiros ficarão claras.

Como saber, olhando para um padrão de *bits*, qual o valor representado? 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 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. Neste 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 admitir 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 2 e 3, um antes da vírgula entre um e dois depois. 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 9 e 10 dígitos decimais de precisão, cinco antes da vírgula e entre quatro e cinco depois.

A este tipo de representação chama-se vírgula fixa, por se colocar a vírgula numa posição fixa (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 3×10^{-5} e 3×10^5, 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 expoente. Na prática esta representação não usa 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 dos valores.

Na maior parte dos processadores utilizados hoje em dia usa-se uma das representações especificadas na norma IEEE 754 [5] 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);

ou seja,

s_0	m_{-1}	$\cdots$	m_{-23}	e_7	$\cdots$	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 [5]. 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[16])
	$e_7\cdots e_0 \neq 0\cdots 0 \wedge$ $e_7\cdots e_0 \neq 1\cdots 1$	$\pm\,(1,m_{-1}\cdots m_{-23})_2 \times 2^{(e_7\cdots e_0)_2-127}$ (valores normalizados[16])
$m_{-1}\cdots m_{-23} = 0\cdots 0$	$e_7\cdots e_0 = 1\cdots 1$	$\pm\infty$
$m_{-1}\cdots m_{-23} \neq 0\cdots 0$	$e_7\cdots e_0 = 1\cdots 1$	(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[16].

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

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

[16]Os valores normalizados do formato IEEE 754 [5] 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.

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.

A representação de valores de vírgula flutuante e pormenores de implementação de operações com essas representações serão estudados com maior pormenor na disciplina de Arquitectura de Computadores.

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 0.4 não são representáveis exactamente (experimente escrevê-lo em base 2)! Ao estudo da forma de lidar com estes tipos de erros sem surpresas dá-se o nome de análise numérica [2].

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 (`false`), todos os outros representando o valor verdadeiro (`true`). Os valores booleanos podem ser convertidos em inteiros. Nesse caso o valor `false` é convertido em 0 e o valor `true` 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 variada 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 $\quad$ A $\quad$ $\mathcal{A}$ $\quad$ A

É possível definir variáveis que guardam caracteres. O tipo `char` é usado 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-1 (ou melhor, ISO-8859-1, ver Apêndice G), que é uma extensão da tabela ASCII (*American Standard Code for Information Interchange*) que inclui os caracteres acentuados em uso na Europa Ocidental. 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)[17].

Não é necessário, conhecer os códigos dos caracteres de cor: para indicar o código do caractere correspondente à letra 'b' usa-se naturalmente `'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-1. Esta tradução de uma representação habitual, `'b'`, para um código específico, 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[18]:

```
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 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`), de -128 a 127, devendo-se usar `unsigned char` para forçar uma interpretação como inteiros sem sinal, de 0 a 255.

[17] 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.

[18] As instruções com a forma `cout << ...;` servem para escrever no ecrã. escrever `endl` no ecrã faz uma mudança de linha.

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-1)[19]:

```
#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 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). i.e., o resultado das seguintes instruções

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

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.

2.4 Valores literais

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

[19] 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.

```
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[20]:

```
0x1U   // o mesmo que 1U.
0x10FU // o mesmo que 271U, ou seja (00000000000000000000000100001111)_2.
077U   // o mesmo que 63U, ou seja (00000000000000000000000000111111)_2.
```

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. Basta para já saber que se podem usar para escrever texto no ecrã (ver Secção 2.1.1).

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 se especificados directamente como valores literais excepto usando sequências de escape. Existe um caractere chamado de escape que, quando ocorrer num valor literal do tipo `char`, 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, válidas também dentro de cadeias de caracteres.

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 nas variáveis. A definição de constantes faz-se colocando a palavra-chave[21] `const` após o tipo pretendido para a constante[22]. As constantes, justamente por o serem, têm obrigatoriamente de ser inicializadas no acto da definição, i.e., o seu valor estático tem de ser indicado na própria definição. Por exemplo:

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

[21] 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.

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

Tabela 2.4: Sequências de espace. 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.

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

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

```
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[23]

```
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;
   ```

[23] 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 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 parecem potencialmente mais úteis, pois agem sobre o valor de uma variável:

```
int i = 0;
i = 2;
i = 1 + 2 * 3;
```

Uma expressão é composta por valores (valores literais, valores de variáveis, de constantes, etc) e operações. Muitas vezes numa expressão existe um operador de atribuição = ou suas variantes[24] , ver Secção 2.6.5. A expressão

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

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

A expressão na última instrução de

```
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 comparação entre os valores variáveis `i` e `j`". Depois desta operação o valor de `são_iguais` é verdadeiro se `i` for igual a `j` e falso no caso contrário.

Os operadores 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.

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

2.6.1 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 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

```
const double 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! Por exemplo, o código acima deveria ser reescrito como:

```
const double 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 variável, então é preferível converter explicitamente um ou ambos os operandos para compatibilizar os seus tipos

```
const double 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, desastrosos.

2.6.2 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 ==.

2.6.3 Operadores lógicos

Os operadores lógicos (ou booleanos) aplicam-se operandos lógicos e são[25]

`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 $\mathcal{V}$.

A ordem de cálculo dos operandos de um operador não é, em geral, especificada. Os dois operadores binários `and` e `or` são uma excepção. Os seus operandos (que podem ser subexpressões) 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 2.5.

---

[25]Fez-se todo os esforço neste texto para usar a versão mais intuitiva destes operadores, uma introdução recente na linguagem. 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 8.2.1):

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

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

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

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

Figura 2.5: Tabelas de verdade das operações lógicas elementares.

2.6.4 Operadores *bit*-a-*bit*

Há alguns operadores do C++ que permitem fazer manipulações 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 `&`;

`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. Por exemplo, admitindo que o tipo `unsigned int` é representado com apenas 16 *bits*, para simplificar:

```
123U bitand 0xFU == 11U == 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, $(0000000000001011)_2 = 11 = (\mathrm{B})_{16}$.

Os deslocamentos simplesmente deslocam o padrão de *bits* correspondente ao 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 resulta em $1 = (0000000000010000)_2 = 16$ e $20 = (0000000000110100)_2$ 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 verá na Secção 2.1.1.

2.6.5 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[26]:

```
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 operando esquerdo da atribuição. Este facto, conjugado com a associatividade à direita deste tipo de operadores (ver Secção 2.6.7), permite escrever

[26] 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.

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

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

Ao contrário do que acontece com os operadores aritméticos e relacionais, por exemplo, 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`[27].

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

```
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;
```

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

2.6.6 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 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[28]:

```
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.

2.6.7 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 do C++ 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

[28] As razões para esta preferência ficarão claras quando na Secção 6.5.3.

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.

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

```
// 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.

2.6.8 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 atribuir o valor da expressão do lado direito à entidade (normalmente uma variável) do lado esquer.

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:

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

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

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

`++x;` 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) {
    ...
    cin >> x;
}
```

2.6.9 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[29]. 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++;
```

[29] Repare-se que é a ordem de cálculo dos operandos que é indefinida. A ordem de cálculo dos operadores 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.

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[30]. 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.

[30]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: funções e procedimentos

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 6, falar-se-á de um outro nível de modularização: as classes. Finalmente a modularização regressará a um nível ainda mais alto no Capítulo 8.

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. Um 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 ao conceito da modularização e da 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 [1]:

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

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 dividir em funções e 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(n > m)
    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(n > m)
        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(n > m)
        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 inutil. 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(n > m)
        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(n > m)
        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(n > m)
        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 roina 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(n > m)
        k = m;
    else
        k = n;

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

   return k;
}
```

Idealmente o corpo das roinas 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 dificil é reutilizar o seu código.

3.2.4 Contrato e documentação de uma rotina

A definição de uma rotina só fica realmente completa quando incluir um comentário indicando exactamente aquilo que a 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 $PC \equiv \mathtt{m} > 0 \wedge \mathtt{n} > 0$.
@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(n > m)
        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 ≡ m > 0 ∧ n > 0.
    @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(n > m)
        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 calcula-

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

do. 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 2.6.8). 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 ≡ m > 0 ∧ n > 0.
    @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(n > m)
        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 procedimen-
to 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:

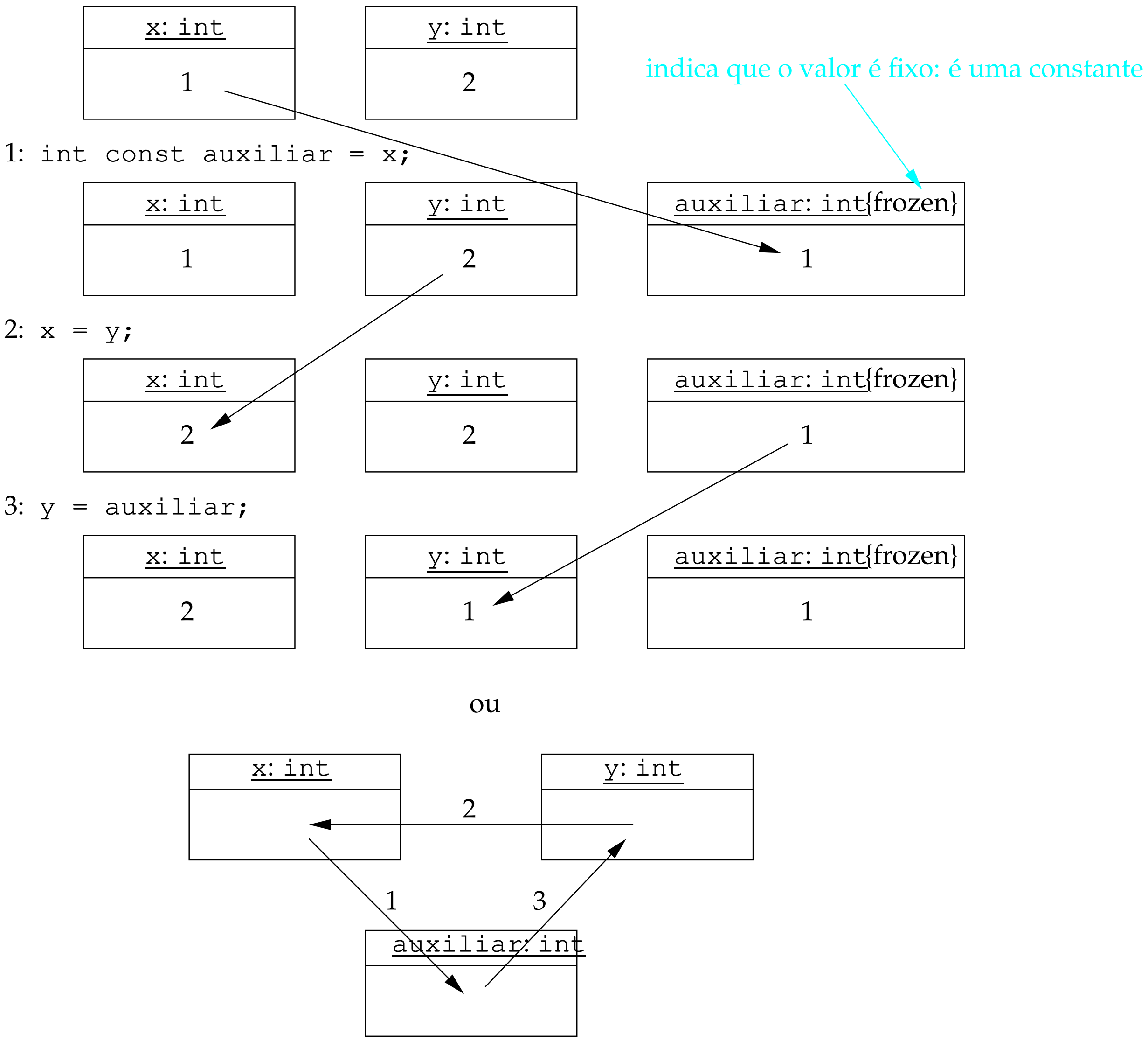


Figura 3.1: Algoritmo usual de troca de valores entre duas variáveis `x` e `y` através de uma constante auxiliar.

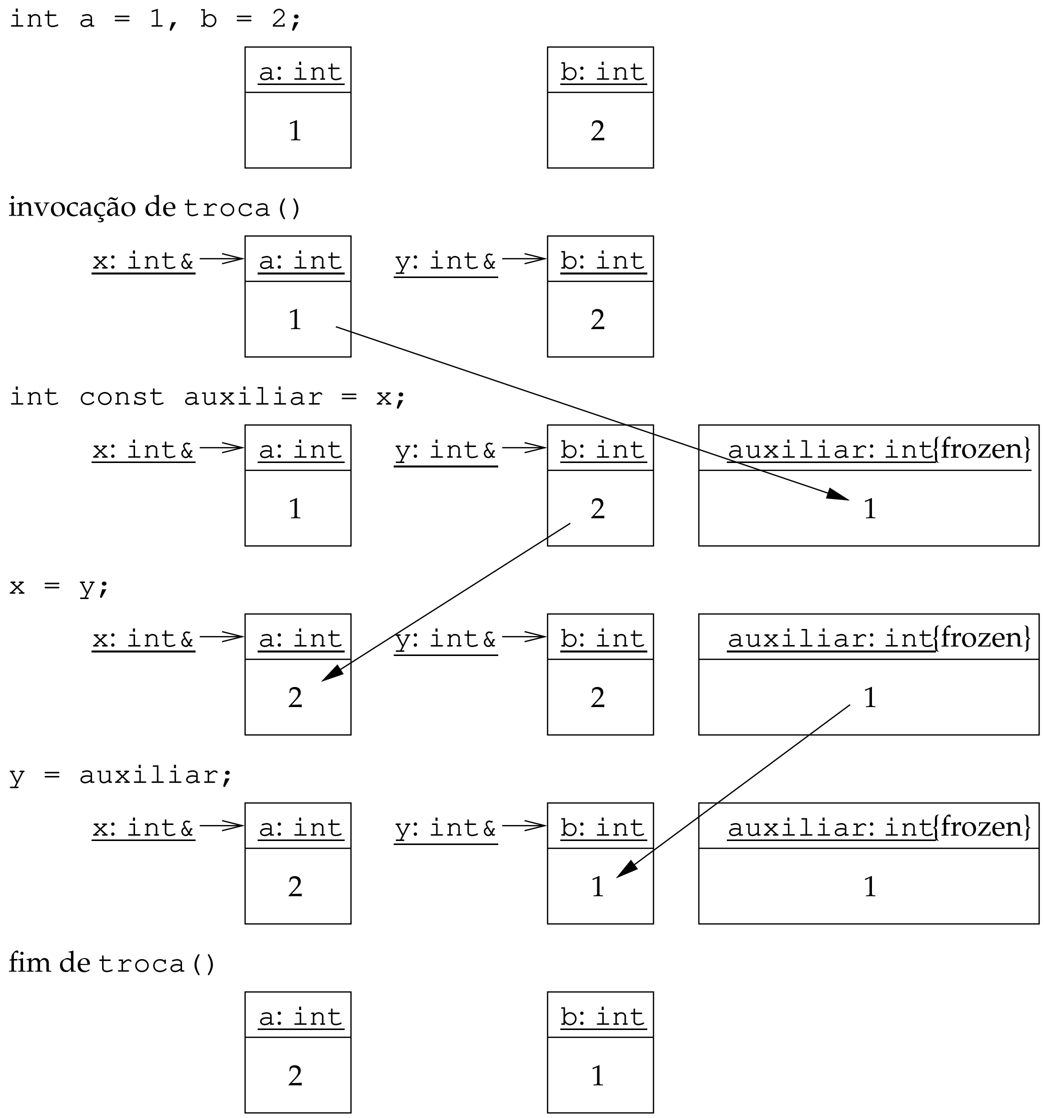


Figura 3.2: Diagramas com evolução do estado do programa que invoca o procedimento `troca()` entre cada instrução.

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 2.6.8.

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 ≡ m > 0 ∧ n > 0.
    @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(n > m)
        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 ∧ n > 0 ∧ d > 0
    @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 m e n 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 ≡ m > 0 ∧ n > 0.
    @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(n > m)
        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 ∧ n > 0 ∧ d > 0
    @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 8 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 8.

```
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 variável `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 8.6.2). 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 6, 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 pedeInteiro();
  ```

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 roina 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 $PC \equiv \mathtt{n} = n \land \mathtt{d} = d \land n > 0 \land d > 0$
@post $CO \equiv \frac{\mathtt{n}}{\mathtt{d}} = \frac{n}{d} \land \mathrm{mdc}(\mathtt{n}, \mathtt{d}) = 1$ */

```
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 $PC \equiv \mathtt{m} > 0 \land \mathtt{n} > 0$.
@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(n > m)
        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 roinas 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 ∧ n > 0 ∧ d > 0
    @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 ≡ m > 0 ∧ n > 0.
    @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(n > m)
            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 ∧ n > 0 ∧ d > 0
      @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 ≡ m > 0 ∧ n > 0.
      @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 12 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 ≡ m > 0 ∧ n > 0.
    @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: Asserti-
on '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 ≡ m > 0 ∧ n > 0.
    @post CO ≡ mdc = mdc(m, n). */
int mdc(int const m, int const n);
{
    assert(m > 0 and n > 0);

    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 ≡ m > 0 ∧ n > 0.
    @post  CO ≡ mdc = mdc(m, n). */
int mdc(int const m, int const n)
{
    assert(m > 0 and n > 0);

    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): Asserti-
on 'm > 0 and n > 0' 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 ∧ n > 0 ∧ d > 0
    @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 ≡ m > 0 ∧ n > 0.
    @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(n > 0 and d > 0);

    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(n > 0 and d > 0);

    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 ∧ n > 0 ∧ d > 0
    @post CO ≡ n/d = n/d ∧ mdc(n, d) = 1 */
void reduzFracção(int& n, int& d)
{
    assert(n > 0 and d > 0);

    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 [9], 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 ≡ m > 0 ∧ n > 0.
    @post  CO ≡ mdc = mdc(m, n). */
int mdc(int const m, int const n)
{
    assert(n > 0 and d > 0);

    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 ≡ m > 0 ∧ n > 0.
    @post  CO ≡ mdc = mdc(m, n). */
int mdc(int const m, int const n)
```

```
{
    assert(n > 0 and d > 0);

    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 8, 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 $m > 0 \wedge n \geq 0$.
2. $\mathrm{mdc}(m, n) = n$ se $m = 0 \wedge n > 0$.

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 ∧ d > 0 ∧ 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 ∧ n > 0.
    @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(d > 0 and mdc(n, d) == 1);
}

void escreveFracção(int const n, int const d)
{
    cout << n << '/' << d;
}

int mdc(int m, int n)
{
    assert(n > 0);

    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

$$n! = \begin{cases} n(n-1)! & \text{se } n > 0 \\ 1 & \text{se } n = 0 \end{cases},$$

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.

3.3.1 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

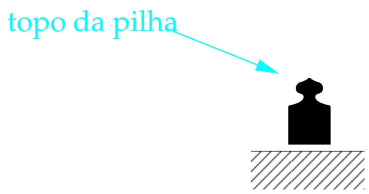
Suponha-se o seguinte programa, incluindo a função `mdc()`,

```
#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 ∧ n > 0.
    @post CO ≡ mdc = mdc(|m|, n). */
int mdc(int m, int n)
{
    assert(n > 0);

    if(m < 0)
        m = -m;
    while(m != 0) {
```

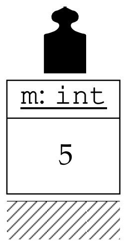
```
            int const auxiliar = n % m;
            n = m;
            m = auxiliar;
        }
        return n;
    }

    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.

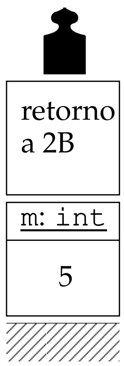
Considera-se, para simplificar, que pilha está vazia quando se começa a executar a função `main()`:



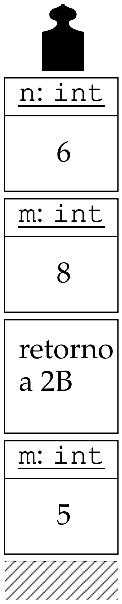
De seguida é executada a instrução 1, que constrói a variável `m`. Como essa variável é automática, é construída na pilha, que fica



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



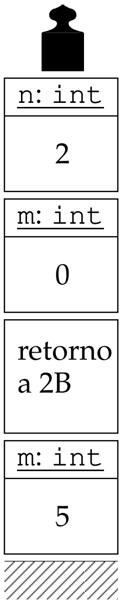
Em seguida são construídos na pilha os parâmetros da função. Cada parâmetro é inicializado com o valor do argumento respectivo:



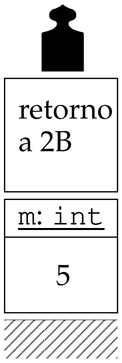
Neste momento existem na pilha duas variáveis de nome `m`: uma pertencente à função `main()` e outra à função `mdc()`. É fácil saber em cada instante quais são as variáveis automática visíveis: são todas as variáveis desde o topo da pilha até ao próximo endereço de retorno. Isto é, neste momento são 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 do 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:

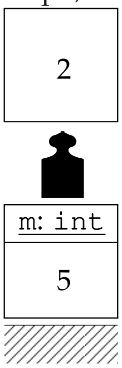
[10] Qualquer compilador minimamente inteligente evita este processo de construção e destruição repetitivas construindo a variável `auxiliar` na pilha logo no início da invocação da função.



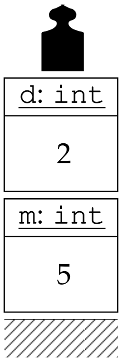
A instrução de retorno começa por calcular o valor a devolver (neste caso é o valor de `n`, i.e., 2), retira da pilha (destrói) todas as variáveis desde o topo até ao próximo endereço de retorno



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

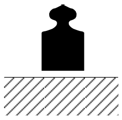


Em seguida a execução continua em 2B, que constrói a variável `divisor`, inicializando-a com o valor devolvido, colocado após o topo da pilha, que depois se deixa levar por uma "corrente de ar":



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.

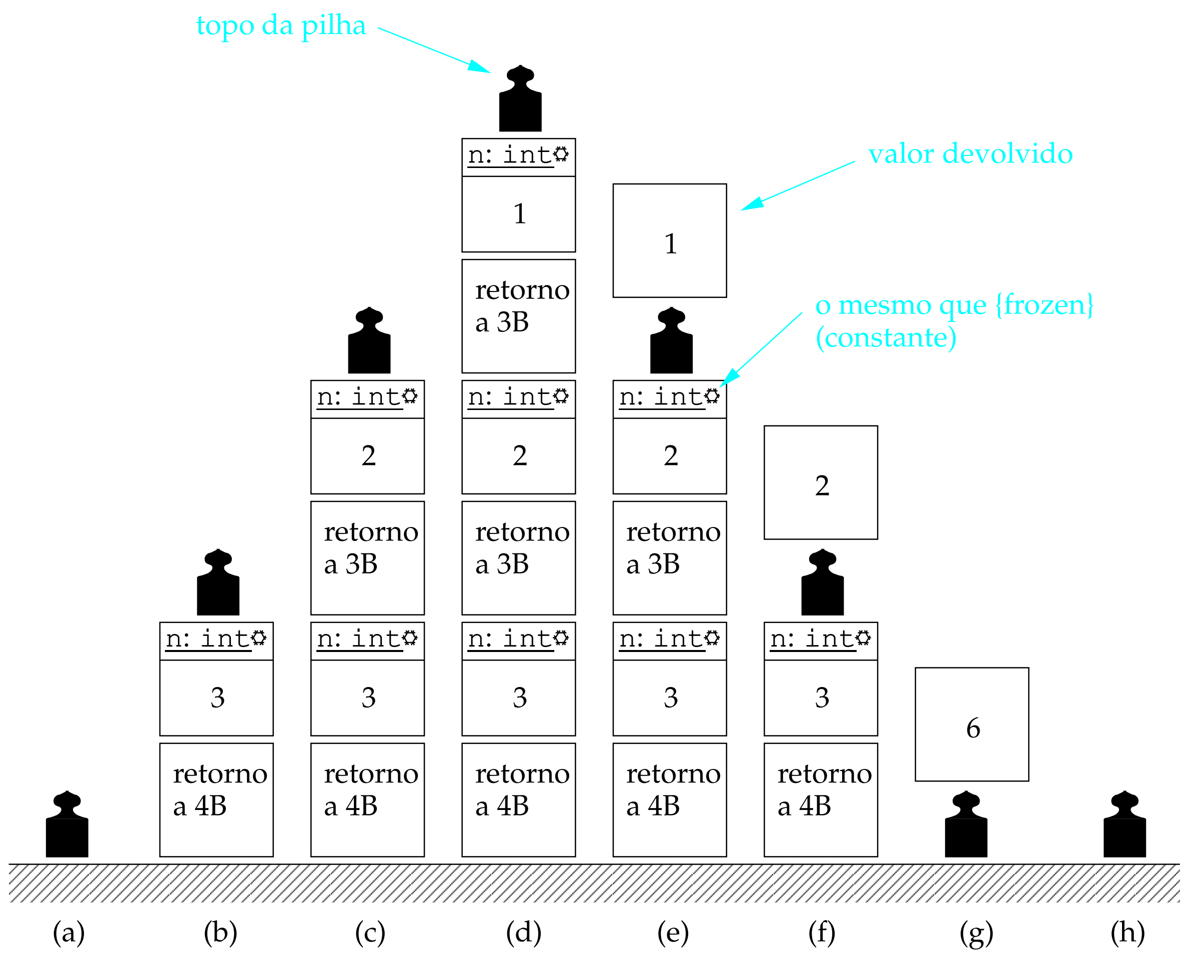


Figura 3.3: Evolução da pilha durante invocações recursivas da função `factorial()`. Admite-se que a pilha inicialmente está vazia, como indicado em (a).

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.

3.4 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 o mesmo número e tipo 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

```
cout << 1 / 2 << endl;
cout << 1.0 / 2.0 << endl;
```

têm como resultado no ecrã

```
0 0.5
```

[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.

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:

- `soma`, 2, `int`, `int`
- `soma`, 3, `int`, `int`, `int`
- `soma`, 2, `float`, `float`
- `soma`, 2, `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.

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,
              absoluto ≥ 0 ∧ (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 $\mathtt{x} \geq 0$. 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 $\mathtt{x} \geq 0$. Assim, numa instrução de selecção pode-se sempre inverter a ordem das instruções alternativas desde que se negue a condição. Assim,

```
if(x < 0)
    // G_1 ≡ x < 0
    x = 0;
else
    // G_2 ≡ x ≥ 0
    x = 1;
```

é equivalente a

```
if(x >= 0)
    // G_2 ≡ x ≥ 0
```

[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`.

```
    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.

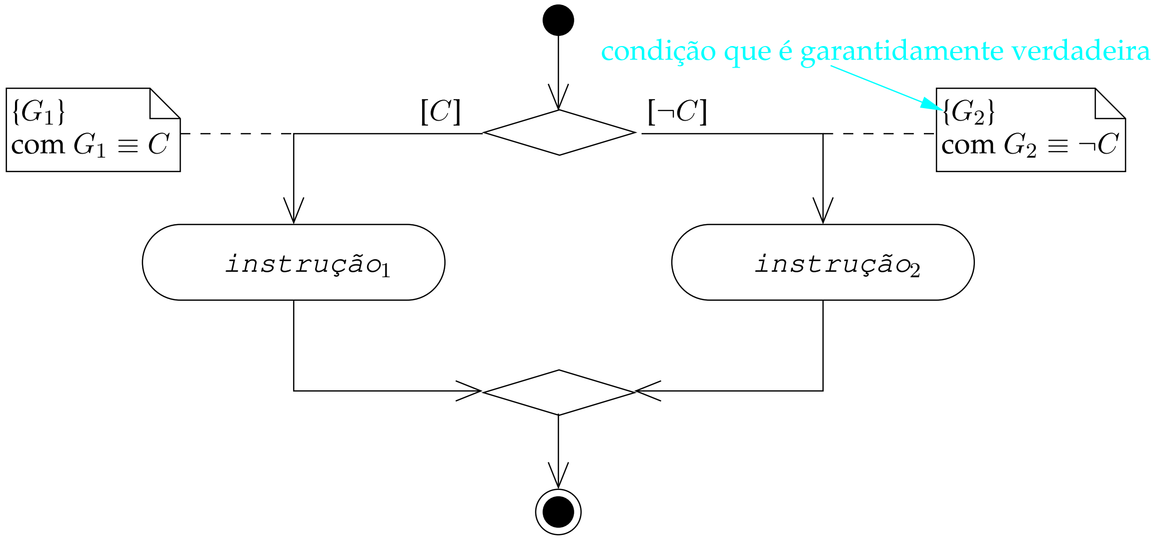


Figura 4.1: Diagrama de actividade de uma instrução de selecção genérica. Uma condição que tem de ser sempre verdadeira coloca-se num comentário entre chavetas.

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.

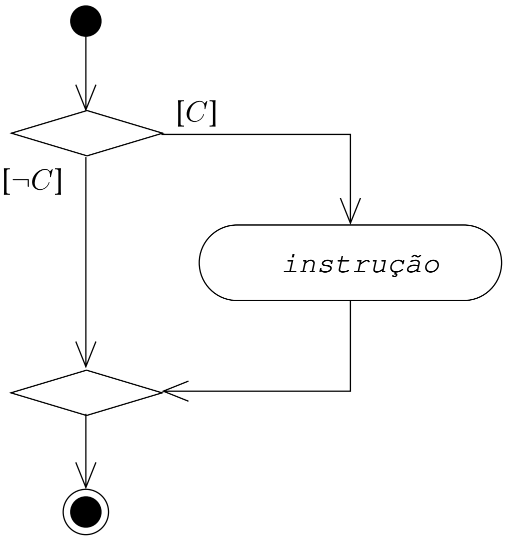


Figura 4.2: Diagrama de actividade de uma instrução condicional genérica.

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(x > y) {
```

```
        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)
    // 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 ≡ a ≥ mínimo ∧ a > máximo.
    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(C1)
    // G1 ≡ C1.
    ...
else if(C2)
    // G2 ≡ ¬G1 ∧ C2  (ou, ¬C1 ∧ C2).
    ...
else if(C3)
    // G3 ≡ ¬G1 ∧ ¬G2 ∧ C3  (ou, ¬C1 ∧ ¬C2 ∧ C3).
    ...
...
else if(Cn−1)
    // Gn−1 ≡ ¬G1 ∧ ··· ∧ ¬Gn−2 ∧ Cn−1  (ou, ¬C1 ∧ ··· ∧ ¬Cn−2 ∧ Cn−1).
    ...
```

```
else
    // $G_n \equiv \neg G_1 \wedge \cdots \wedge \neg G_{n-1}$  (ou, $\neg C_1 \wedge \cdots \wedge \neg C_{n-1}$).
    ...
```

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, com 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.

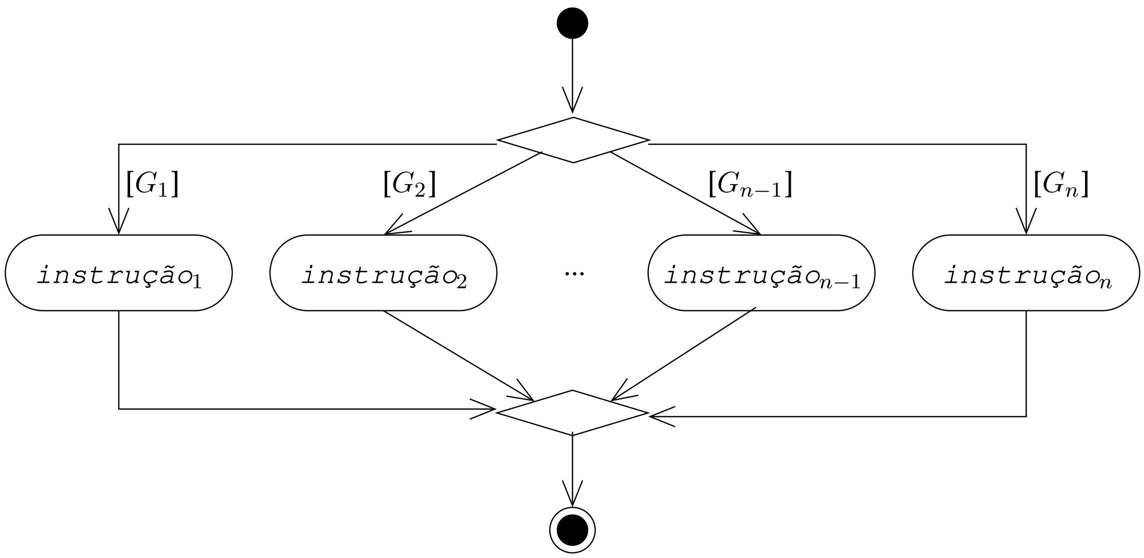


Figura 4.3: Diagrama de actividade de uma instrução de múltipla (sem correspondente directo na linguagem C++) equivalente a uma instrução de selecção encadeada. As guardas presumem-se mutuamente exclusivas.

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:

```
// n ≥ 0
++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

```
// n ≥ 0.
++n;
// n ≥ 1.
```

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

$$\texttt{x} = y \wedge \texttt{y} = x$$

e a PC como

$$\texttt{x} = x \wedge \texttt{y} = y$$

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 $P \equiv x > 0$ e $Q \equiv x = 1$, então P é mais fraco do que Q, pois o conjunto $\{1\}$ está contido no conjunto dos positivos $\{x : x > 0\}$. 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 2.6.8). 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 $x \geq 0$. Aplicando o método descrito conclui-se que

```
// −x ≥ 0, ou seja, x ≤ 0.
x = -x;
// x ≥ 0.
```

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;
// x ≥ 0.
```

Usando o método anterior conclui-se que:

```
// x < 0.
// −x ≥ 0, ou seja, x ≤ 0.
x = -x;
// x ≥ 0.
```

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 $\mathtt{x} \leq \mathtt{y}$,

```
if(x > y) {
    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(x > y) {
    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(x > y) {
    // G1 ≡ x > y.
    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(x > y) {
    // x > y ∧ 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(x > y) {
    // x > y ∧ x = x ∧ y = y.
    int const t = x;
    // x > y ∧ x = x ∧ y = y ∧ t > y ∧ t = x.
    x = y;
    // y = y ∧ t > y ∧ t = x ∧ x = y ∧ t > x.
    y = t;
    // t = x ∧ x = y ∧ t > x ∧ y = x ∧ y > x, 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, $\mathtt{y} > \mathtt{x}$, o que implica que $\mathtt{x} \leq \mathtt{y}$. Eliminando as asserções intermédias, úteis apenas durante a demonstração,

```
// PC ≡ x = x ∧ y = y.
if(x > y) {
    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(x > y) {
    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(x > y) {
    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(x > y) {
    // 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(x > y) {
    // 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 ∧ Gᵢ
   instruçãoᵢ
   // COᵢ
   ```

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:

   ```
   // PCᵢ
   instruçãoᵢ
   // 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,
              absoluto ≥ 0 ∧ (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,
              absoluto ≥ 0 ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
```

[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

$$CO \equiv \texttt{absoluto} = |x|\text{, ou seja, } \texttt{absoluto} \geq 0 \wedge (\texttt{absoluto} = -\texttt{x} \vee \texttt{absoluto} = \texttt{x}).$$

é

$$CO \equiv \texttt{r} = |x|\text{, ou seja, } \texttt{r} \geq 0 \wedge (\texttt{r} = -\texttt{x} \vee \texttt{r} = \texttt{x}).$$

No entanto, a instrução de retorno difere da instrução de atribuição pelo numa coisa: a instrução de retorno termina a execução da função.

```
    // PC ≡ V (ou seja, sem restrições).
    int r;
    ...
    // CO ≡ r = |x|, ou seja, r ≥ 0 ∧ (r = −x ∨ 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 [7] *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 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 ≡ r ≥ 0 ∧ (r = −x ∨ r = x).
```

Usando a regra da substituição discutida na Secção 4.2.3, chega-se a

```
// CO ≡ −x ≥ 0 ∧ (−x = −x ∨ −x = x), ou seja, x ≤ 0 ∧ (V ∨ x = 0), ou seja, x ≤ 0.
r = -x;
// CO ≡ r ≥ 0 ∧ (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 ≡ x ≥ 0 ∧ (x = −x ∨ x = x), ou seja, x ≥ 0 ∧ (x = 0 ∨ V), ou seja, x ≥ 0.
r = x;
// CO ≡ r ≥ 0 ∧ (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., $PC_2 \equiv$ `x` ≥ 0.

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,
              absoluto ≥ 0 ∧ (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 ≡ x ≥ 0.
        r = x;
    // CO ≡ r = |x|, ou seja, r ≥ 0 ∧ (r = −x ∨ 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 $PC \land G_i \Rightarrow PC_i$. 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 $PC \land G_i \Rightarrow PC_i$ 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,
              absoluto ≥ 0 ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    // PC ≡ V (ou seja, sem restrições).
    int r;
```

```
    if($C_1$)
        // $G_1 \equiv$ x $\leq 0$.
        r = -x;
    else
        // $G_2 \equiv$ x $\geq 0$.
        r = x;
    // $CO \equiv$ r $= |$x$|$, ou seja,  r $\geq 0 \wedge ($r $= -$x $\vee$ 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 $\mathtt{x} \leq 0 \vee \mathtt{x} \geq 0$, 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   $PC \equiv \mathcal{V}$  (ou seja, sem restrições).
    @post  $CO \equiv$ absoluto $= |$x$|$, ou seja,
               absoluto $\geq 0 \wedge ($absoluto $= -$x $\vee$ absoluto $=$ x$)$. */
int absoluto(int const x)
{
    // $PC \equiv \mathcal{V}$  (ou seja, sem restrições).
    int r;
    if(x <= 0)
        // $G_1 \equiv$ x $\leq 0$.
        r = -x;
    else
        // $G_2 \equiv$ x $> 0$.
        r = x;
    // $CO \equiv$ r $= |$x$|$, ou seja,  r $\geq 0 \wedge ($r $= -$x $\vee$ r $=$ x$)$.
    return r;
}
```

A segunda guarda foi alterada, pois de acordo com a Secção 4.2.4 $G_2 = \neg G_1 = \mathtt{x} > 0$. 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 e introduzir um instrução de asserção para verificar explicitamente a veracidade da condição objectivo:

```
/** Devolve o valor absoluto do argumento.
    @pre  PC ≡ V (ou seja, sem restrições).
    @post CO ≡ absoluto = |x|, ou seja,
               absoluto ≥ 0 ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    int r;
    if(x <= 0)
        // G1 ≡ x ≤ 0.
        r = -x;
    else
        // G2 ≡ x > 0.
        r = x;

    assert(r >= 0 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,
               absoluto ≥ 0 ∧ (absoluto = −x ∨ absoluto = x). */
int absoluto(int const x)
{
    if(x <= 0)
        // G1 ≡ x ≤ 0.
        return -x;
    else
        // G2 ≡ x > 0.
        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,
               absoluto ≥ 0 ∧ (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 $\texttt{instrução}_i$, 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.

[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.

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

   // $PC \wedge G_i$
   `instrução`$_i$
   // CO_i
   // CO

   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., `x` ≥ 0 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.

⋮

⋮

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 2.6.3).

A moral desta estória é que "quanto mais depressa, mais devagar"... O programador deve evitar as soluções rápidas, pouco pensadas e menos verificadas.

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 $[\texttt{mín}, \texttt{máx}]$, onde se assume que $\texttt{mín} \leq \texttt{máx}$. 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 [mín, máx].
    @pre  PC ≡ mín ≤ máx ∧ x = x.
    @post CO ≡ (x = mín ∧ x ≤ mín) ∨ (x = máx ∧ x ≥ máx)∨
               (x = x ∧ mín ≤ x ≤ 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

$$
\begin{array}{rcl}
PC_1 & \equiv & (\mathtt{x} = x \wedge \mathtt{mín} \leq x \leq \mathtt{máx}) \vee (\mathtt{x} = \mathtt{mín} \wedge x \leq \mathtt{mín}) \vee (\mathtt{x} = \mathtt{máx} \wedge x \geq \mathtt{máx}) \\
PC_2 & \equiv & (\mathtt{mín} = x \wedge \mathtt{mín} \leq x \leq \mathtt{máx}) \vee (\mathtt{mín} = \mathtt{mín} \wedge x \leq \mathtt{mín}) \vee (\mathtt{mín} = \mathtt{máx} \wedge x \geq \mathtt{máx}) \\
 & \equiv & (\mathtt{mín} = x \wedge \mathtt{mín} \leq x \leq \mathtt{máx}) \vee x \leq \mathtt{mín} \vee (\mathtt{mín} = \mathtt{máx} \wedge x \geq \mathtt{máx}) \\
PC_3 & \equiv & (\mathtt{máx} = x \wedge \mathtt{mín} \leq x \leq \mathtt{máx}) \vee (\mathtt{máx} = \mathtt{mín} \wedge x \leq \mathtt{mín}) \vee (\mathtt{máx} = \mathtt{máx} \wedge x \geq \mathtt{máx}) \\
 & \equiv & (\mathtt{máx} = x \wedge \mathtt{mín} \leq x \leq \mathtt{máx}) \vee (\mathtt{máx} = \mathtt{mín} \wedge x \leq \mathtt{mín}) \vee x \geq \mathtt{máx}
\end{array}
$$

A determinação das guardas faz-se de modo a que $PC \wedge G_i \Rightarrow PC_i$.

No primeiro caso tem-se da pré-condição PC que $\mathtt{x} = x$, pelo que a guarda mais fraca é

$$
\begin{array}{rcl}
G_1 & \equiv & (\mathtt{mín} \leq \mathtt{x} \wedge \mathtt{x} \leq \mathtt{máx}) \vee \mathtt{x} = \mathtt{mín} \vee \mathtt{x} = \mathtt{máx} \\
 & \equiv & \mathtt{mín} \leq \mathtt{x} \wedge \mathtt{x} \leq \mathtt{máx}
\end{array}
$$

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

$$
\begin{array}{rcl}
G_2 & \equiv & (\mathtt{mín} = \mathtt{x} \wedge \mathtt{mín} \leq \mathtt{x} \leq \mathtt{máx}) \vee \mathtt{x} \leq \mathtt{mín} \vee (\mathtt{mín} = \mathtt{máx} \wedge \mathtt{x} \geq \mathtt{máx}) \\
 & \equiv & \mathtt{mín} = \mathtt{x} \vee \mathtt{x} \leq \mathtt{mín} \vee (\mathtt{mín} = \mathtt{máx} \wedge \mathtt{x} \geq \mathtt{máx}) \\
 & \equiv & \mathtt{x} \leq \mathtt{mín} \vee (\mathtt{mín} = \mathtt{máx} \wedge \mathtt{x} \geq \mathtt{máx})
\end{array}
$$

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

$$G_3 \equiv (\mathtt{máx} = \mathtt{mín} \wedge \mathtt{x} \leq \mathtt{mín}) \vee \mathtt{x} \geq \mathtt{máx}$$

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 ∧ x ≥ máx).
    x = mín;
```

```
else if(C₂)
    // G₃ ≡ (máx = mín ∧ x ≤ mín) ∨ x ≥ máx.
    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 ≡ x ≥ máx.
    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 ≡ x > máx.
    x = máx;
else
    // G₁ ≡ mín ≤ x ∧ x ≤ máx.
    ;
```

As condições das instruções `if` são:

```
if(x < mín)
    // G₂ ≡ x < mín.
    x = mín;
else if(x > máximo)
    // G₃ ≡ x > máx.
    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 ∧ x ≥ máx)∨
               (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(x > máx)
        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,
               absoluto ≥ 0 ∧ (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).
```

[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...

```
    @post  CO ≡ absoluto = |x|, ou seja,
                absoluto ≥ 0 ∧ (absoluto = −x ∨ absoluto = x).  */
int absoluto(int const x)
{
    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 Secção 6.1), 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(x >= 1000)
        cout << "milhares";
    else if(x >= 100)
        cout << "centenas";
    else if(x >= 10)
        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 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:

[5]Demonstre-o!

```
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.

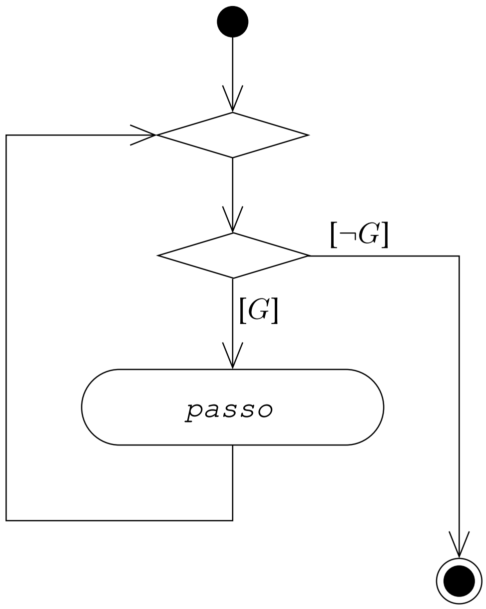


Figura 4.4: Diagrama de actividade da instrução de iteração `while`.

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 ≡ n ≥ 0.
    @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
    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.

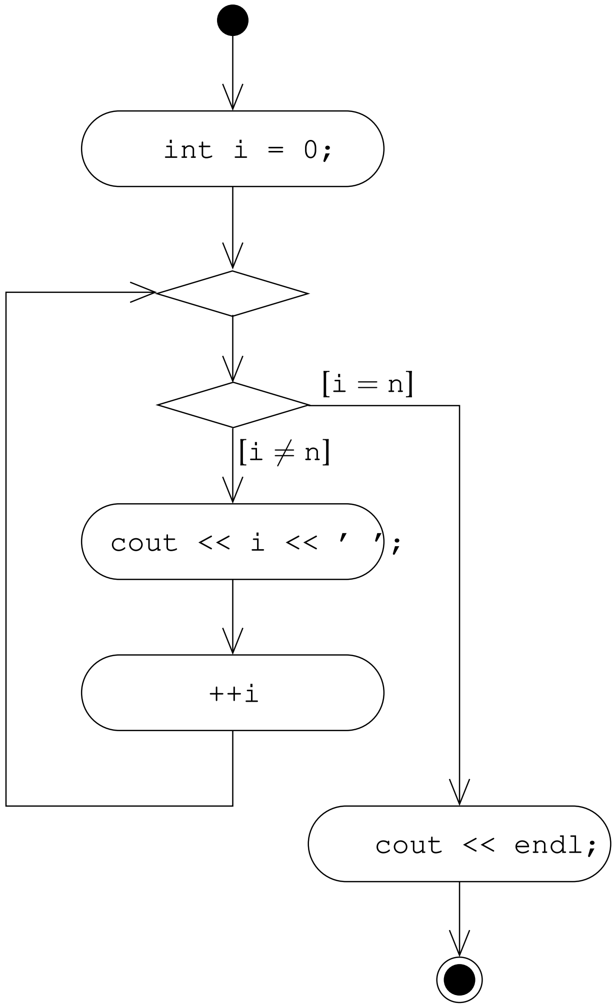


Figura 4.5: Diagrama de actividade do procedimento `escreveInteiros()`.

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 `i` $\neq$ `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

```
inic
while(G) {
    acção
    prog
}
```

onde *`inic`* são as instruções de *inicialização* que preparam as variáveis envolvidas no ciclo, *`prog`* é 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.

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 ≡ n ≥ 0.
    @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
```

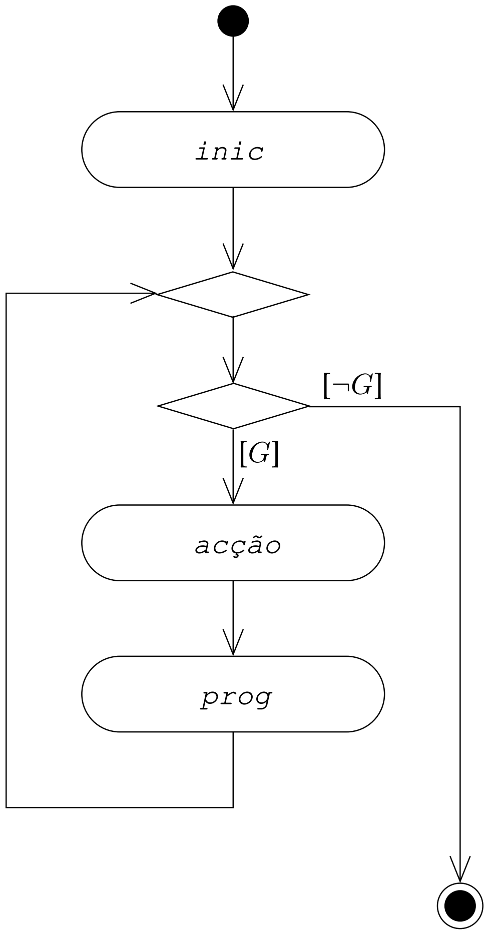


Figura 4.6: Diagrama de actividade de um `ciclo` típico.

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:

```
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.

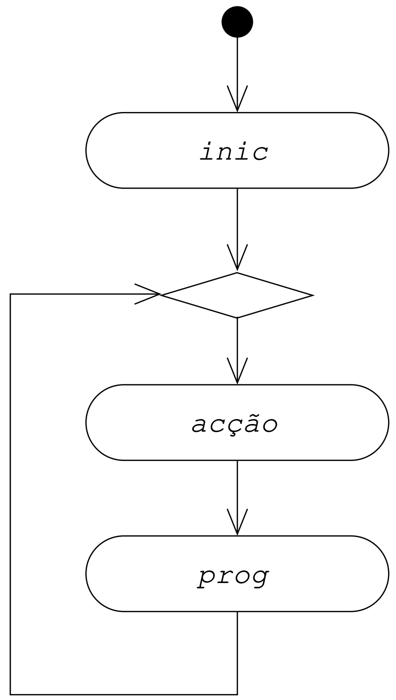


Figura 4.7: Diagrama de actividade de um laço.

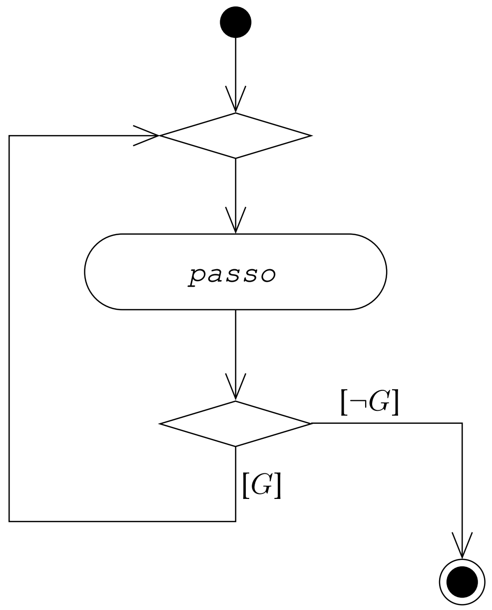


Figura 4.8: Diagrama de actividade da instrução de iteração `do while`.

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;
```

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 $\texttt{i} = 10$ 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:

```
/** Devolve um inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ inteiroPedidoDe0a100 ≤ 100. */
int inteiroPedidoDe0a100()
{
    cout << "Introduza valor (0 a 100): ";
    int n;
    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);

    return n;
}
```

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:

```
/** Devolve um inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ inteiroPedidoDe0a100 ≤ 100. */
int inteiroPedidoDe0a100()
{
    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);

    return n;
}
```

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`:

```
/** Devolve um inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ inteiroPedidoDe0a100 ≤ 100. */
int inteiroPedidoDe0a100()
{
    // Truque sujo: inicialização para a guarda ser inicialmente verdadeira. Má ideia!
    int 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);

    return n;
}
```

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 . 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.

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;
```

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*

```
/** Escreve um triângulo oco de asteriscos com a altura passada como argumento.
    @pre  PC ≡ altura ≥ 0.
    @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(altura >= 0);

    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 ≡ n ≥ 0.
    @post CO ≡ éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ n ≥ 2). */
bool éPrimo(int const n)
{
    assert(n >= 0);
```

```
    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., [10]), 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:

```
/** Devolve um inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ inteiroPedidoDe0a100 ≤ 100. */
int inteiroPedidoDe0a100()
{
    cout << "Introduza valor (0 a 100): ";
    int n;
    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);

    return n;
}
```

e

```
/** Devolve um inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ inteiroPedidoDe0a100 ≤ 100. */
int inteiroPedidoDe0a100()
{
    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);

    return n;
}
```

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`,

```
/** Devolve um inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ inteiroPedidoDe0a100 ≤ 100. */
int inteiroPedidoDe0a100()
{
    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);

    return n;
}
```

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

```
/** Devolve um inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ inteiroPedidoDe0a100 ≤ 100. */
int inteiroPedidoDe0a100()
{
    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);

    return n;
}
```

que é mais comum escrever como

```
/** Devolve um inteiro entre 0 e 100 pedido ao utilizador.
    @pre  PC ≡ V
    @post CO ≡ 0 ≤ inteiroPedidoDe0a100 ≤ 100. */
int inteiroPedidoDe0a100()
{
    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.

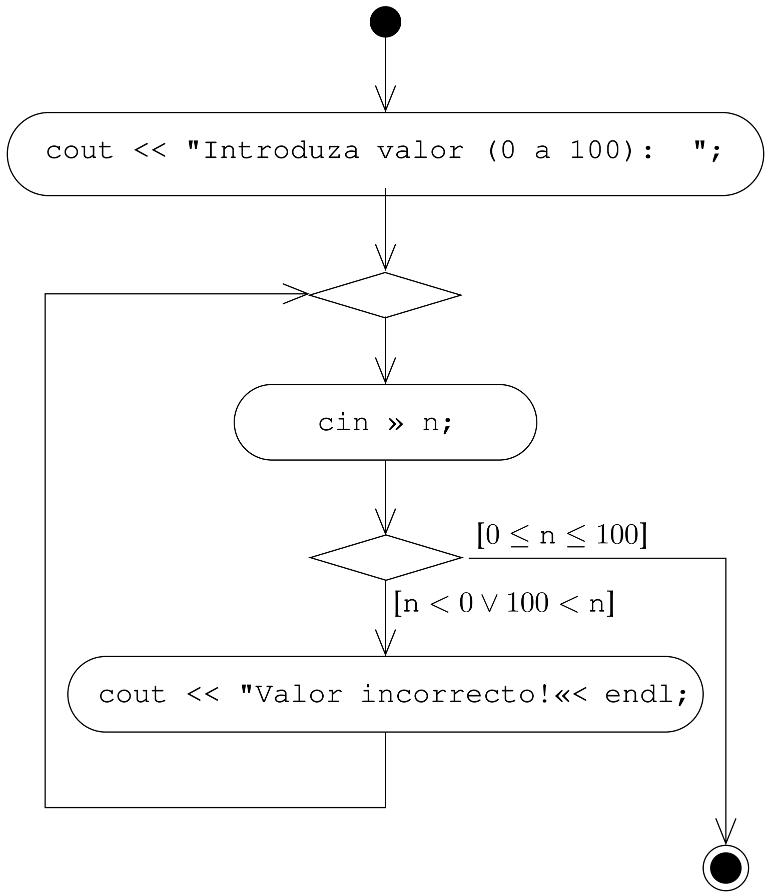


Figura 4.9: Diagrama de actividade da função `inteiroPedidoDe0a100()`.

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 pode ser escrita recorrendo a uma instrução de retorno:

```
/** 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;
    }
}
```

onde se passou a definição da variável `n` para o mais perto possível da sua primeira utilização[6].

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 errar a guarda do ciclo e escrever

```
for(int i = 0; i <= 10; ++i)
    cout << '*';
```

[6]Noutras linguagens existe uma instrução `loop` para este efeito, que pode ser simulada em C++ recorrendo ao pré-processador (ver Secção 8.2.1):

```
#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;
    }
}
```

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[7]. 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.

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").

[7] Se não terminar é mais fácil perceber que alguma coisa está errada no código!

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:

$$\left(\mathbf{Q}\, x \in \mathbb{R}\, :\, |x| \geq \sqrt{2}\, :\, x^2 - 2 \geq 0\right).$$

Um quantificador pode possuir mais do que uma variável muda. Por exemplo:

$$(\mathbf{S}\, i, j\, :\, 0 \leq i < m \land 0 \leq j < n\, :\, f(i, j))$$

$\text{predicado}(v)$ 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,

$$(\mathbf{S}\, j\, :\, 0 \leq j < n\, :\, j) = \frac{n(n-1)}{2} \text{ se } n > 0.$$

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 ≡ n ≥ 0.
    @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

$$n! = (\mathbf{P}\, j \,:\, 1 \leq j \leq n \,:\, j)\,.$$

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 ≡ n ≥ 0.
    @post CO ≡ factorial = n! = (P j : 1 ≤ j ≤ n : 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"[8]. 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[9],

$$(\mathbf{Q}\, j\ :\ m \leq j < n\ :\ \text{predicado}(j)) = \mathcal{V} \text{ se } n \leq m.$$

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

$$\text{primo}(n) = \begin{cases} (\mathbf{Q}\, j\ :\ 2 \leq j < n\ :\ n \div j \neq 0) & \text{se } n \geq 2, \text{ e} \\ \mathcal{F} & \text{se } 0 \leq n < 2, \end{cases}$$

ou seja,

$$\text{primo}(n) = ((\mathbf{Q}\, j\ :\ 2 \leq j < n\ :\ n \div j \neq 0) \wedge n \geq 2) \text{ para } \mathtt{n} \geq 0,$$

sendo $\div$ a operação resto da divisão inteira[10].

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:

[8] 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.

[9] 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.

[10] Como o operador `%` em C++.

```
/** Devolve V se n for um número primo e F no caso contrário.
    @pre  PC ≡ n ≥ 0.
    @post CO ≡ éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ n ≥ 2). */
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"[11]. 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[12],

$$(\mathbf{E}\, j\ :\ m \leq j < n\ :\ \mathrm{predicado}(j)) = \mathcal{F} \text{ se } n \leq m.$$

Este quantificador está estreitamente relacionado com o quantificador universal. É sempre verdade que

$$\neg\left(\mathbf{Q}\, j\ :\ m \leq j < n\ :\ \mathrm{predicado}(j)\right) = \left(\mathbf{E}\, j\ :\ m \leq j < n\ :\ \neg\mathrm{predicado}(j)\right),$$

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

$$\begin{aligned}\neg\mathrm{primo}(n) &= \neg\left((\mathbf{Q}\, j\ :\ 2 \leq j < n\ :\ n \div j \neq 0) \wedge n \geq 2\right)\\ &= \neg\left(\mathbf{Q}\, j\ :\ 2 \leq j < n\ :\ n \div j \neq 0\right) \vee n < 2\\ &= \left(\mathbf{E}\, j\ :\ 2 \leq j < n\ :\ n \div j = 0\right) \vee n < 2\end{aligned}$$

[11] 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.

[12] A afirmação "existe pelo menos um marciano com cabeça" é falsa, pois não existem marcianos.

para $n \geq 0$. 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 ≡ m ≥ 0.
    @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 ≡ n ≥ 1.
    @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 $m \geq 0$ e $n > 0$[13], é o único elemento do conjunto $\{0 \leq r < n \,:\, (\mathbf{E}\, q \,:\, q \geq 0 \,:\, m = qn + r)\}$. Claro que a definição está incompleta enquanto não se demonstrar que de facto o conjunto tem um único elemento, ou seja que, quando $m \geq 0$ e $n > 0$, se tem

$$(\mathbf{N}\, r \,:\, 0 \leq r < n \,:\, (\mathbf{E}\, q \,:\, q \geq 0 \,:\, m = qn + r)) = 1.$$

Deixa-se a demonstração como exercício para o leitor.

4.7 Desenvolvimento de ciclos

O desenvolvimento de programas usando ciclos é simultaneamente uma arte [8] e uma ciência [7]. 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 [7].

Suponha-se que se pretende desenvolver uma função para calcular a potência `n` de `x`, isto é, uma função que, sendo `x` e `n` os seus dois parâmetros, devolva $\mathtt{x}^{\mathtt{n}}$. A sua estrutura básica é

```
double potência(double x, int 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 `x`. Assim, uma solução passa por usar um ciclo que no seu passo faça cada uma dessas multiplicações:

```
double potência(double x, int 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.
```

[13]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:

$$\begin{array}{rll} 0 \leq r < n & \text{se} & m \geq 0 \wedge n > 0 \text{ (neste caso } q \geq 0) \\ 0 \leq r < -n & \text{se} & m \geq 0 \wedge n < 0 \text{(neste caso } q \leq 0) \\ -n < r \leq 0 & \text{se} & m \leq 0 \wedge n > 0 \text{(neste caso } q \leq 0) \\ n < r \leq 0 & \text{se} & m \leq 0 \wedge n < 0 \text{(neste caso } q \geq 0) \end{array}$$

```
    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 `i` $<$ `n` e não `i` $\leq$ `n`. Corrigindo a função:

```
double potência(double x, int 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 x, int 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. Aproveite-se também para indicar claramente que os valores dos parâmetros não devem poder ser alterados, i.e., que os parâmetros não são variáveis mas sim constantes:

```
/** Devolve a potência n de x.
    @pre PC ≡ n ≥ 0.
    @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;
}
```

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 $\texttt{i} < \texttt{n}$ para $\texttt{i} \neq \texttt{n}$. 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[14]. 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 ≡ n ≥ 0.
    @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 ≡ n ≥ 0.
    @post CO ≡ potência = xⁿ. */
double potência(double const x, int const n)
{
    assert(n >= 0);

    int i = 0;
    double r = 1.0;
    while(i != n) {
        r *= x;
        ++i;
    }
    return r;
}
```

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?

[14]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.

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[15]. 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 ≡ n ≥ 0.
    @post CO ≡ potência = x^n. */
double potência(double const x, int const n)
{
    assert(n >= 0);

    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`.

```
    }
```

5: Depois de se verificar que a guarda é falsa, imediatamente antes do retorno.

[15]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.

```
        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.

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
```

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.

Representando o ciclo na forma de um diagrama de actividade e colocando as asserções que se sabe serem verdadeiras em cada seta do diagrama, é mais fácil compreender a noção de invariante, como se vê na Figura 4.10.

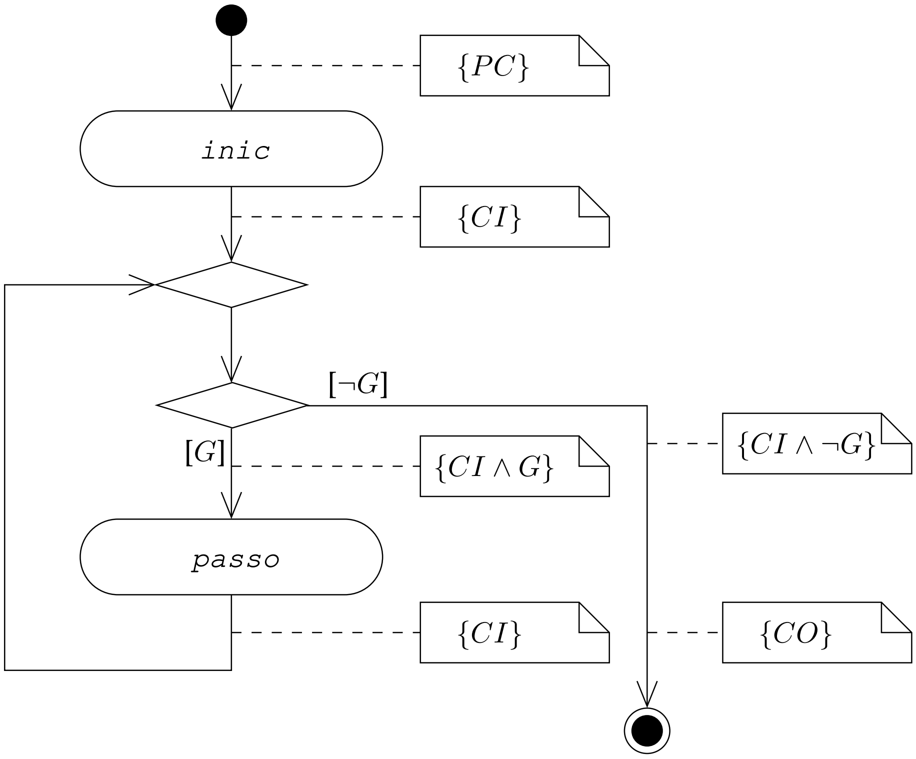


Figura 4.10: Diagrama de actividade de um ciclo mostrando as asserções nas transições entre instruções (actividades). Só as instruções podem alterar o estado do programa e portanto validar ou invalidar asserções.

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 n ≥ 0.
    @post potência = xⁿ. */
double potência(double const x, int const n)
{
    // PC ≡ n ≥ 0.
    assert(n >= 0);

    int i = 0;
    double r = 1.0;
    // CI ≡ r = x^i ∧ 0 ≤ i ≤ n.
    while(i != n) {
        r *= x;
        ++i;
    }
    // CO ≡ r = x^n.
    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 4.10). 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 ≡ n ≥ 0.
int i = 0;
double r = 1.0;
// CI ≡ r = x^i ∧ 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 4.10. 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[16]. 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.
```

[16] 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

$$\texttt{r} = \texttt{x}^{\texttt{i}} \times \texttt{x} \wedge -1 \leq \texttt{i} < \texttt{n}$$

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 $0 \leq \texttt{i} \Rightarrow -1 \leq \texttt{i}$ e $\texttt{r} = \texttt{x}^{\texttt{i}} \Rightarrow \texttt{r} \times \texttt{x} = \texttt{x}^{\texttt{i}} \times \texttt{x}$

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:

$$\begin{aligned} CI \wedge \neg G &\equiv \mathtt{r} = \mathtt{x}^{\mathtt{i}} \wedge 0 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i} = \mathtt{n} \\ &\equiv \mathtt{r} = \mathtt{x}^{\mathtt{i}} \wedge \mathtt{i} = \mathtt{n} \\ &\Rightarrow \mathtt{r} = \mathtt{x}^{\mathtt{n}} \equiv CO \end{aligned}$$

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 n ≥ 0.
    @post potência = xⁿ. */
double potência(double const x, int const n)
{
    // PC ≡ n ≥ 0.
    assert(n >= 0);

    int i = 0;
    double r = 1.0;
    // CI ≡ r = xⁱ ∧ 0 ≤ i ≤ n.
    while(i != n)
        ; // Instrução nula!
    // CO ≡ r = xⁿ.
    return r;
}
```

É fácil demonstrar a correcção parcial deste ciclo[17]! 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*) [7], 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 n ≥ 0.
    @post potência = xⁿ. */
double potência(double const x, int const n)
{
    // PC ≡ n ≥ 0.
    assert(n >= 0);

    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:

[17] 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 n ≥ 0 ∨ x ≠ 0.
    @post potência = xⁿ. */
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 n ≥ 0 ∨ x ≠ 0.
    @post potência = xⁿ. */
double potência(double const x, int const n)
{
    assert(n >= 0 or x != 0.0);

    int const exp = n < 0 ? -n : n;
    // PC ≡ exp ≥ 0.
    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 ≡ n ≥ 0 ∨ x ≠ 0.
    @post CO ≡ potência = xⁿ. */
double potência(double const x, int const n)
{
    assert(n >= 0 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 ≡ n ≥ 0 ∨ x ≠ 0.
    @post CO ≡ potência = x^n. */
double potência(double const x, int const n)
{
    assert(n >= 0 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 [7][4] 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 guarda.
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 ≡ n ≥ 0.
    @post somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(n >= 0);

    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 ≡ x ≥ 0.
    @post 0 ≤ raizInteira ≤ √x < raizInteira + 1, ou seja,
          0 ≤ raizInteira ∧ 0 ≤ raizInteira² ≤ x < (raizInteira + 1)². */
int raizInteira(int const x)
```

```
{
    assert(x >= 0);

    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 ambas os problemas podem 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 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 [4], 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 ≡ n ≥ 0.
```

```
    @post somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(n >= 0);

    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[18]. 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'

$$\neg G \equiv \texttt{i} = \texttt{n},$$

[18]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 ≡ n ≥ 0.
    @post somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(n >= 0);

    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 $\neg G \equiv \texttt{i} \geq \texttt{n}$, 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 ≡ x ≥ 0.
    @post 0 ≤ raizInteira ≤ √x < raizInteira + 1, ou seja,
          0 ≤ raizInteira ∧ 0 ≤ raizInteira² ≤ x < (raizInteira + 1)². */
int raizInteira(int const x)
{
    assert(x >= 0);

    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

$$CI \equiv \texttt{soma\_dos\_quadrados} = \left(\mathbf{S}\, j\, :\, 0 \leq j < \texttt{i}\, :\, j^2\right) \wedge 0 \leq \texttt{i} \leq \texttt{n}.$$

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,

$$\begin{aligned} CI &\equiv 0 = \left(\mathbf{S}\, j\, :\, 0 \leq j < 0\, :\, j^2\right) \wedge 0 \leq 0 \leq \texttt{n} \\ &\equiv 0 = 0 \wedge 0 \leq \texttt{n} \\ &\equiv 0 \leq \texttt{n} \end{aligned}$$

que é verdadeira desde que a pré-condição $PC \equiv \texttt{n} \geq 0$ o seja. A função neste momento é

```
/** Devolve a soma dos quadrados dos primeiros n inteiros não-negativos.
    @pre PC ≡ n ≥ 0.
    @post somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(n >= 0);

    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

$$CI \equiv 0 \leq \texttt{r} \wedge \texttt{r}^2 \leq \texttt{x}.$$

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 $PC \equiv \mathtt{x} \geq 0$ o seja. A função neste momento é

```
/** Devolve a melhor aproximação inteira por defeito da raiz quadrada de x.
    @pre PC ≡ x ≥ 0.
    @post 0 ≤ raizInteira ≤ √x < raizInteira + 1, ou seja,
          0 ≤ raizInteira ∧ 0 ≤ raizInteira² ≤ x < (raizInteira + 1)². */
int raizInteira(int const x)
{
    assert(n >= 0);

    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∧G ≡ soma_dos_quadrados = (S j : 0 ≤ j < i : j²)∧0 ≤ i ≤ n∧i ≠ n, ou seja,
// soma_dos_quadrados = (S j : 0 ≤ j < i : j²) ∧ 0 ≤ i < n
acção
++i;
// CI ≡ soma_dos_quadrados = (S j : 0 ≤ j < i : j²) ∧ 0 ≤ i ≤ 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∧G ≡ soma_dos_quadrados = (S j : 0 ≤ j < i : j²)∧0 ≤ i ≤ n∧i ≠ n, ou seja,
// soma_dos_quadrados = (S j : 0 ≤ j < i : j²) ∧ 0 ≤ i < n
acção
// soma_dos_quadrados = (S j : 0 ≤ j < i + 1 : j²) ∧ 0 ≤ i + 1 ≤ n, ou seja,
// soma_dos_quadrados = (S j : 0 ≤ j < i + 1 : j²) ∧ −1 ≤ i < n.
++i; // o mesmo que i = i + 1;
// CI ≡ soma_dos_quadrados = (S j : 0 ≤ j < i : j²) ∧ 0 ≤ i ≤ n.
```

Falta pois determinar uma acção tal que

```
// soma_dos_quadrados = (S j : 0 ≤ j < i : j²) ∧ 0 ≤ i < n
acção
// soma_dos_quadrados = (S j : 0 ≤ j < i + 1 : j²) ∧ −1 ≤ i < 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 = (S j : 0 ≤ j < i : j²) ∧ 0 ≤ i < n
acção
// soma_dos_quadrados = (S j : 0 ≤ j < i : j²) + i² ∧ 0 ≤ i < n
// soma_dos_quadrados = (S j : 0 ≤ j < i + 1 : j²) ∧ −1 ≤ i < 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 ≡ n ≥ 0.
    @post CO ≡ somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
```

```
int somaDosQuadrados(int const n)
{
    assert(n >= 0);

    int soma_dos_quadrados = 0;
    int i = 0;
    // CI ≡ soma_dos_quadrados = (S j : 0 ≤ j < i : j²) ∧ 0 ≤ i ≤ 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 ≡ n ≥ 0.
    @post CO ≡ somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(n >= 0);

    int soma_dos_quadrados = 0;
    // CI ≡ soma_dos_quadrados = (S j : 0 ≤ j < i : j²) ∧ 0 ≤ i ≤ 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 ≡ x ≥ 0.
    @post CO ≡ 0 ≤ raizInteira ≤ √x < raizInteira + 1, ou seja,
         0 ≤ raizInteira ∧ 0 ≤ raizInteira² ≤ x < (raizInteira + 1)². */
int raizInteira(int const x)
{
    assert(x >= 0);

    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 ≡ x ≥ 0.
    @post CO ≡ 0 ≤ raizInteira ≤ √x < raizInteira + 1, ou seja,
```

$0 \leq$ `raizInteira` $\wedge\, 0 \leq$ `raizInteira`$^2 \leq$ `x` $<$
$($`raizInteira`$+1)^2$. `*/`

```
int raizInteira(int const x)
{
    assert(x >= 0);

    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[19]

$$\frac{n(n-1)(2n-1)}{6}.$$

Não é necessário qualquer ciclo na função, que pode ser implementada simplesmente como[20]:

```
/** Devolve a soma dos quadrados dos primeiros n inteiros não-negativos.
    @pre PC ≡ n ≥ 0.
    @post CO ≡ somaDosQuadrados = (S j : 0 ≤ j < n : j²). */
int somaDosQuadrados(int const n)
{
    assert(n >= 0);

    return n * (n - 1) / 2 * (2 * n - 1) / 3;
}
```

[19] A demonstração faz-se utilizando a propriedade telescópica dos somatórios

$$\sum_{j=0}^{n-1}(f(j)-f(j-1)) = f(n-1)-f(-1)$$

com $f(j) = j^3$.

[20] 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++.

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 ≡ n ≥ 0.
    @post CO ≡ éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ n ≥ 2). */
bool éPrimo(int const n)
{
    assert(n >= 0);

    ...
}
```

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  n ≥ 0.
    @post éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ n ≥ 2). */
bool éPrimo(int const n)
{
    assert(n >= 0);

    if(n == 0 || n == 1)
        return false;

    // PC ≡ n ≥ 2.
    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 $n \geq 0$ é verdadeira, então forçosamente $n \geq 2$ 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 ≡ n ≥ 2.
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

$$\neg G \equiv \mathtt{i} = \mathtt{n},$$

ou seja,

$$G \equiv \mathtt{i} \neq \mathtt{n}.$$

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 ≡ n ≥ 2.
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 `n` >= 2 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 ≡ n ≥ 2.
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:

$$CI \wedge G \equiv \mathtt{é\_primo} = (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} \leq \mathtt{n} \wedge \mathtt{i} \neq \mathtt{n},$$

ou seja,

$$CI \wedge G \equiv \mathtt{é\_primo} = (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n}.$$

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 $\mathtt{i} \geq 2$, 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 2.6.7).

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:

$$
\begin{array}{ll}
 & \texttt{é\_primo} = (\mathbf{Q}\, j\,:\, 2 \le j < \texttt{i}\,:\, \texttt{n} \div j \ne 0) \wedge 2 \le \texttt{i} < \texttt{n} \\
\Rightarrow & (\texttt{é\_primo} \wedge \texttt{n} \div \texttt{i} \ne 0) = ((\mathbf{Q}\, j\,:\, 2 \le j < \texttt{i}\,:\, \texttt{n} \div j \ne 0) \wedge \texttt{n} \div \texttt{i} \ne 0) \wedge 2 \le \texttt{i} < \texttt{n}.
\end{array}
$$

Escrevendo agora o ciclo completo tem-se:

```
// PC ≡ n ≥ 2.
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[21].

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.
```

[21] 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:

// $CI \wedge G \equiv (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n} \wedge \mathtt{é\_primo}$.
acção
// $\mathtt{é\_primo} = (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i} + 1\, :\, \mathtt{n} \div j \neq 0) \wedge 1 \leq \mathtt{i} < \mathtt{n}$.
`++i;`
// $CI \equiv \mathtt{é\_primo} = (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} \leq \mathtt{n}$.

Por outro lado, se $\mathtt{i} \geq 2$, pode-se extrair o último termo da conjunção. Assim, reforçando a pré-condição do progresso,

// $CI \wedge G \equiv (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n} \wedge \mathtt{é\_primo}$.
acção
// $\mathtt{é\_primo} = ((\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge \mathtt{n} \div \mathtt{i} \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n}$.
// $\mathtt{é\_primo} = (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i} + 1\, :\, \mathtt{n} \div j \neq 0) \wedge 1 \leq \mathtt{i} < \mathtt{n}$.
`++i;`
// $CI \equiv \mathtt{é\_primo} = (\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} \leq \mathtt{n}$.

Assim sendo, a acção deve ser tal que

// $(\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n} \wedge \mathtt{é\_primo}$.
acção
// $\mathtt{é\_primo} = ((\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge \mathtt{n} \div \mathtt{i} \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n}$.

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:

// $(\mathtt{n} \div \mathtt{i} \neq 0) = ((\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge \mathtt{n} \div \mathtt{i} \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n}$.
`é_primo = n % i != 0;`
// $\mathtt{é\_primo} = ((\mathbf{Q}\, j\, :\, 2 \leq j < \mathtt{i}\, :\, \mathtt{n} \div j \neq 0) \wedge \mathtt{n} \div \mathtt{i} \neq 0) \wedge 2 \leq \mathtt{i} < \mathtt{n}$.

e observando que $CI \wedge G$ leva forçosamente à sua verificação:

$$\begin{array}{ll} & (\mathbf{Q}\, j\ :\ 2 \le j < \mathtt{i}\ :\ \mathtt{n} \div j \ne 0) \wedge 2 \le \mathtt{i} < \mathtt{n} \wedge \mathtt{é\_primo} \\ \Rightarrow & (\mathtt{n} \div \mathtt{i} \ne 0) = ((\mathbf{Q}\, j\ :\ 2 \le j < \mathtt{i}\ :\ \mathtt{n} \div j \ne 0) \wedge \mathtt{n} \div \mathtt{i} \ne 0) \wedge 2 \le \mathtt{i} < \mathtt{n}. \end{array}$$

Escrevendo agora o ciclo completo tem-se:

```
// PC ≡ n ≥ 2.
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 ≡ n ≥ 0.
    @post CO ≡ éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ n ≥ 2). */
bool éPrimo(int const n)
{
    assert(n >= 0);

    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 n >= 2);
```

Recorde-se que $A \Rightarrow B$ é o mesmo que $\neg A \vee B$. A expressão da asserção verifica portanto se é_primo $\Rightarrow$ n ≥ 2.

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
        (n >= 2 and ((n + 1) % 6 == 0 or (n + 5) % 6 == 0))));
```

que, expressa em notação matemática fica

$$((\mathrm{n} = 2 \vee \mathrm{n} = 3) \Rightarrow \texttt{é\_primo}) \wedge$$
$$(\texttt{é\_primo} \Rightarrow (\mathrm{n} = 2\mathrm{n} = 3 \vee (\mathrm{n} \geq 2 \wedge ((\mathrm{n}+1) \div 6 = 0 \vee (\mathrm{n}+5) \div 6 = 0))))$$

A função fica então:

```
/** Devolve V se n for um número primo e F no caso contrário.
    @pre PC ≡ n ≥ 0.
    @post CO ≡ éPrimo = ((Q j : 2 ≤ j < n : n ÷ j ≠ 0) ∧ n ≥ 2). */
bool éPrimo(int const n)
{
    assert(n >= 0);

    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
           (n >= 5 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
            (n >= 2 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 $\mathcal{V}$ se n for um número primo e $\mathcal{F}$ no caso contrário.
    @pre $PC \equiv$ n $\geq 0$.
    @post $CO \equiv$ éPrimo $= ((\mathbf{Q}\, j : 2 \leq j <$ n $:$ n $\div j \neq 0) \land$ n $\geq 2)$. */
bool éPrimo(int const n)
{
    assert(n >= 0);

    if(n == 0 || n == 1)
        return false;

    // $CI \equiv (\mathbf{Q}\, j : 2 \leq j <$ i $:$ n $\div j \neq 0) \land 2 \leq$ i $\leq$ 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 `n` ≥ 2 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., $\mathbf{C} = \{$`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 $\mathcal{V}$ se n for um número primo e $\mathcal{F}$ no caso contrário.
    @pre n $\geq 0$.
    @post éPrimo $= (\min\{2 \leq j \leq$ n $:$ n $\div j = 0\} =$ n $\land$ n $\geq 2)$. */
bool éPrimo(int const n)
{
```

```
    assert(n >= 0);

    if(n == 0 || n == 1)
        return false;

    // PC ≡ n ≥ 2.
    ...
    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 $\mathtt{i} = \mathtt{n}$. Então pode-se reescrever a função:

```
/** Devolve V se n for um número primo e F no caso contrário.
    @pre  n ≥ 0.
    @post éPrimo = (min {2 ≤ j ≤ n : n ÷ j = 0} = n ∧ n ≥ 2). */
bool éPrimo(int const n)
{
    assert(n >= 0);


    if(n == 0 || n == 1)
        return false;

    // PC ≡ n ≥ 2.
    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 ≡ n ≥ 2.
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 ≡ n ≥ 2.
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 ≡ n ≥ 2.
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 ≡ n ≥ 2.
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 n ≥ 0.
    @post éPrimo = (min {2 ≤ j ≤ n : n ÷ j = 0} = n ∧ n ≥ 2). */
bool éPrimo(int const n)
{
```

```
    assert(n >= 0);

    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
            (n >= 2 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 ≡ dividendo ≥ 0 ∧ divisor > 0.
    @post CO ≡ 0 ≤ r < divisor ∧ dividendo = q × divisor + r.
void divide(int const dividendo, int const divisor,
            int& q, int& r)
{
    assert(dividendo >= 0 and divisor > 0);

    ...

    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 ≡ dividendo ≥ 0 ∧ divisor > 0.
    @post CO ≡ 0 ≤ r < divisor ∧ dividendo = q × divisor + r.
void divide(int const dividendo, int const divisor,
            int& q, int& r)
{
    assert(dividendo >= 0 and divisor > 0);

    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 ≡ dividendo ≥ 0 ∧ divisor > 0.
    @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 8 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 a classe modelo `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").

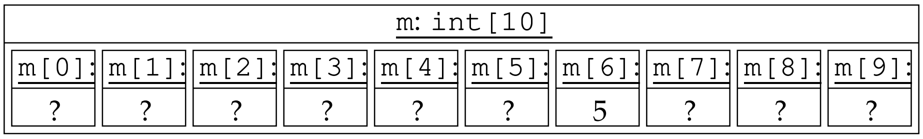


Figura 5.1: Matriz `m` definida por `int m[10];` depois das instruções `int a = 6;` e `m[a] = 5;`.

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 6.5.7). 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 Secção 6.1, é zero, mesmo que este valor não seja tomado por nenhum dos seus valores literais enumerados, e o das classes, ver Secção 6.5.7, 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};
```

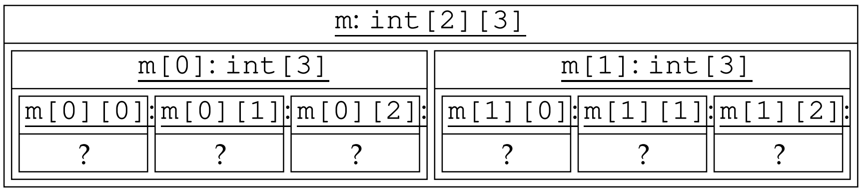
tem o mesmo efeito que as definições anteriores.

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>`. Este assunto será visto em maior pormenor no !!.

```
#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 nota[7] 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, $\mathtt{m} = m$, 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 9.

[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 decimais13.
    @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 nota14 abaixo.
```

[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 236: 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 extraír 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, ver !!.

```
    for(int i = 0; i != n; ++i)
        cin >> m[i]; // lê o i-ésimo elemento do teclado.
}

/** Devolve a média dos n primeiros elementos da matriz.
    @pre PC ≡ 1 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ média = (S j : 0 ≤ 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 ≡ 0 ≤ n ∧ n ≤ dim(m) ∧ divisor ≠ 0 ∧ m = m.
    @post CO ≡ (Q j : 0 ≤ j < n : m[j] = m[j] / divisor). */
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 ≡ 0 ≤ n ∧ n ≤ dim(m).
    @post CO ≡ 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 uma classe modelo chamada `vector`. Esta classe modelo é uma excelente alternativa às matrizes é apresentada 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 modelo, definido na biblioteca padrão do C++ e a que se acede colocando

```
#include <vector>
```

no início do programa. As noções de classe e classe modelo serão abordadas no Capítulo 6 e no Capítulo 11, 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 à classe modelo `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 Métodos

Os vectores são variáveis como qualquer outras. A diferença principal está em suportarem a invocação dos chamados *métodos*, um conceito que será visto em pormenor no Capítulo 6. Simplificando grosseiramente, os métodos são rotinas associadas a um tipo (classe) que se invocam para uma determinada variável desse tipo. A forma de invocação é um pouco diferente da usada para as rotinas "normais". Os métodos invocam-se usando o operador `.` de selecção de membro (no Capítulo 6 se verá o que é exactamente um *membro*). Este operador tem dois operandos. O primeiro é a variável à qual o método é aplicado. O segundo é o método a aplicar, seguido dos respectivos argumentos, se existirem.

Um método extremamente útil da classe modelo `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 pelo método `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 do método `size()`:

```
for(vector<double>::size_type i = 0; i != distâncias.size(); ++i)
    cout << distâncias[i] << endl;
```

Um outro método pode ser usado se se pretender apenas saber se um vector está ou não vazio, i.e., se se pretender saber se um vector tem dimensão zero. O método 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. A classe modelo vector fornece um método chamado `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. Este método 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 dois métodos que permitem aceder aos dois itens nas posições extremas do vector. O método `front()` devolve o primeiro item do vector. O método `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 ao método `resize()`. Este método 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 o método `clear()` como se segue:

```
distâncias.clear();
```

Os vectores não podem ter uma dimensão arbitrária. Para saber a dimensão máxima possível para um vector usar o método `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, a classe modelo fornece um método chamado `reserve()` para pré-reservar espaço para itens que se prevê venham a ser necessários no futuro. Este método 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. O método `capacity()` permite saber qual o número de itens para os quais há espaço reservado em cada momento. A secção seguinte demonstra a utilização deste método.

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 o método `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 o método `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));
```

O método `pop_back()` pode ser conjugado com o método `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?

A classe modelo vector não foi pensada 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 inicialmente 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 ≡ 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;
}
```

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 6.5.3.

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 [3]:
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 $\texttt{i} = \texttt{n}$, ou seja, $G \equiv \texttt{i} \neq \texttt{n}$:

```
// 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 $0 \leq \texttt{n}$ (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 i >= 0, 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
}
```
// $CO \equiv 0 \le \mathtt{i} < \mathtt{n} \land (\mathbf{Q}\, j : 0 \le j < \mathtt{n} : \mathtt{m}[j] \le \mathtt{m}[\mathtt{i}]) \land (\mathbf{Q}\, j : 0 \le j < \mathtt{i} : \mathtt{m}[j] < \mathtt{m}[\mathtt{i}])$.

Analisando os termos da condição invariante um a um, verifica-se que:

1. $0 \le \mathtt{i} < \mathtt{k}$ fica $0 \le 0 < 1$ que é verdadeiro;
2. $(\mathbf{Q}\, j : 0 \le j < \mathtt{k} : \mathtt{m}[j] \le \mathtt{m}[\mathtt{i}])$ fica $(\mathbf{Q}\, j : 0 \le j < 1 : \mathtt{m}[j] \le \mathtt{m}[0])$, que é o mesmo que $\mathtt{m}[0] \le \mathtt{m}[0]$, que é verdadeiro;
3. $(\mathbf{Q}\, j : 0 \le j < \mathtt{i} : \mathtt{m}[j] < \mathtt{m}[\mathtt{i}])$ fica $(\mathbf{Q}\, j : 0 \le j < 0 : \mathtt{m}[j] < \mathtt{m}[0])$ que, como existem zero termos no quantificador, tem valor verdadeiro por definição; e
4. $0 \le \mathtt{k} \le \mathtt{n}$ fica $0 \le 1 \le \mathtt{n}$, 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:

// $PC \equiv 1 \le \mathtt{n} \land \mathtt{n} \le \dim(\mathtt{m})$.
```
int i = 0;
int k = 1;
```
// $CI \equiv 0 \le \mathtt{i} < \mathtt{k} \land (\mathbf{Q}\, j : 0 \le j < \mathtt{k} : \mathtt{m}[j] \le \mathtt{m}[\mathtt{i}]) \land (\mathbf{Q}\, j : 0 \le j < \mathtt{i} : \mathtt{m}[j] < \mathtt{m}[\mathtt{i}]) \land$
// $\quad 0 \le \mathtt{k} \le \mathtt{n}$.
```
while(k != n) {
    acção
    ++k;
}
```
// $CO \equiv 0 \le \mathtt{i} < \mathtt{n} \land (\mathbf{Q}\, j : 0 \le j < \mathtt{n} : \mathtt{m}[j] \le \mathtt{m}[\mathtt{i}]) \land (\mathbf{Q}\, j : 0 \le j < \mathtt{i} : \mathtt{m}[j] < \mathtt{m}[\mathtt{i}])$.

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

// $CI \land G \equiv 0 \le \mathtt{i} < \mathtt{k} \land (\mathbf{Q}\, j : 0 \le j < \mathtt{k} : \mathtt{m}[j] \le \mathtt{m}[\mathtt{i}]) \land (\mathbf{Q}\, j : 0 \le j < \mathtt{i} : \mathtt{m}[j] < \mathtt{m}[\mathtt{i}]) \land$
// $\quad 0 \le \mathtt{k} \le \mathtt{n} \land \mathtt{k} \ne \mathtt{n}$, ou seja,
// $0 \le \mathtt{i} < \mathtt{k} \land (\mathbf{Q}\, j : 0 \le j < \mathtt{k} : \mathtt{m}[j] \le \mathtt{m}[\mathtt{i}]) \land (\mathbf{Q}\, j : 0 \le j < \mathtt{i} : \mathtt{m}[j] < \mathtt{m}[\mathtt{i}]) \land$
// $\quad 0 \le \mathtt{k} < \mathtt{n}$.
acção
```
++k;
```
// $CI \equiv 0 \le \mathtt{i} < \mathtt{k} \land (\mathbf{Q}\, j : 0 \le j < \mathtt{k} : \mathtt{m}[j] \le \mathtt{m}[\mathtt{i}]) \land (\mathbf{Q}\, j : 0 \le j < \mathtt{i} : \mathtt{m}[j] < \mathtt{m}[\mathtt{i}]) \land$
// $\quad 0 \le \mathtt{k} \le \mathtt{n}$.

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[k] > m[i].
    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[k]` $>$ `m[i]`. Falta, pois, falta determinar uma `instrução`$_2$ tal que:

// $0 \leq \texttt{i} < \texttt{k} \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \texttt{k}\, :\, \texttt{m}[j] \leq \texttt{m[i]}) \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \texttt{i}\, :\, \texttt{m}[j] < \texttt{m[i]}) \wedge$
// $\quad 0 \leq \texttt{k} < \texttt{n} \wedge \texttt{m[k]} > \texttt{m[i]}$.
$\texttt{instrução}_2$
// $0 \leq \texttt{i} \leq \texttt{k} \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \texttt{k}\, :\, \texttt{m}[j] \leq \texttt{m[i]}) \wedge \texttt{m[k]} \leq \texttt{m[i]} \wedge$
// $\quad (\mathbf{Q}\, j\, :\, 0 \leq j < \texttt{i}\, :\, \texttt{m}[j] < \texttt{m[i]}) \wedge 0 \leq \texttt{k} < \texttt{n}$.

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:

// $0 \leq \texttt{k} \leq \texttt{k} \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \texttt{k}\, :\, \texttt{m}[j] \leq \texttt{m[k]}) \wedge \texttt{m[k]} \leq \texttt{m[k]} \wedge$
// $\quad (\mathbf{Q}\, j\, :\, 0 \leq j < \texttt{k}\, :\, \texttt{m}[j] < \texttt{m[k]}) \wedge 0 \leq \texttt{k} < \texttt{n}$, ou seja,
// $(\mathbf{Q}\, j\, :\, 0 \leq j < \texttt{k}\, :\, \texttt{m}[j] < \texttt{m[k]}) \wedge 0 \leq \texttt{k} < \texttt{n}$.
`i = k;`
// $0 \leq \texttt{i} \leq \texttt{k} \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \texttt{k}\, :\, \texttt{m}[j] \leq \texttt{m[i]}) \wedge \texttt{m[k]} \leq \texttt{m[i]} \wedge$
// $\quad (\mathbf{Q}\, j\, :\, 0 \leq j < \texttt{i}\, :\, \texttt{m}[j] < \texttt{m[i]}) \wedge 0 \leq \texttt{k} < \texttt{n}$.

Falta pois verificar se

// $0 \leq \texttt{i} < \texttt{k} \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \texttt{k}\, :\, \texttt{m}[j] \leq \texttt{m[i]}) \wedge (\mathbf{Q}\, j\, :\, 0 \leq j < \texttt{i}\, :\, \texttt{m}[j] < \texttt{m[i]}) \wedge$
// $\quad 0 \leq \texttt{k} < \texttt{n} \wedge \texttt{m[k]} > \texttt{m[i]}$
// $\Rightarrow (\mathbf{Q}\, j\, :\, 0 \leq j < \texttt{k}\, :\, \texttt{m}[j] < \texttt{m[k]}) \wedge 0 \leq \texttt{k} < \texttt{n}$.

Eliminando os termos que não são necessários para verificar a implicação,

// $(\mathbf{Q}\, j\, :\, 0 \leq j < \texttt{k}\, :\, \texttt{m}[j] \leq \texttt{m[i]}) \wedge 0 \leq \texttt{k} < \texttt{n} \wedge \texttt{m[k]} > \texttt{m[i]}$
// $\Rightarrow (\mathbf{Q}\, j\, :\, 0 \leq j < \texttt{k}\, :\, \texttt{m}[j] < \texttt{m[k]}) \wedge 0 \leq \texttt{k} < \texttt{n}$.

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[k] > m[i].
    i = k;
```

que pode ser simplificada para uma instrução condicional mais simples

```
if(m[k] > m[i])
    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[k] > m[i])
        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[k] > m[i])
        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[k] > m[i])
            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 int k = 1; k != v.size(); ++k)
        if(v[k] > v[i])
            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:

$$CI \equiv (\mathbf{Q}\, j\, :\, 0 \le j < \mathtt{i}\, :\, \mathtt{mínimo} \le \mathtt{m}[j] \le \mathtt{máximo}) \wedge 0 \le \mathtt{i} \le \mathtt{n},$$

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

$$G \equiv \texttt{i} \neq \texttt{n},$$

então no final do ciclo tem-se $CI \wedge \neg G$, ou seja,

$$CI \wedge \neg G \equiv \left(\mathbf{Q}\, j \,:\, 0 \leq j < \texttt{i} \,:\, \texttt{mínimo} \leq \texttt{m}[j] \leq \texttt{máximo}\right) \wedge 0 \leq \texttt{i} \leq \texttt{n} \wedge \texttt{i} = \texttt{n},$$

que implica

$$\left(\mathbf{Q}\, j \,:\, 0 \leq j < \texttt{n} \,:\, \texttt{mínimo} \leq \texttt{m}[j] \leq \texttt{máximo}\right).$$

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
    // $G_2 \equiv \mathtt{m}[\mathtt{i}] < \mathtt{mínimo} \vee \mathtt{máximo} < \mathtt{m}[\mathtt{i}]$.
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,

$$\begin{aligned} CI \wedge G \wedge G_2 \equiv\;& (\mathbf{Q}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{mínimo} \leq \mathtt{m}[j] \leq \mathtt{máximo}) \wedge 0 \leq \mathtt{i} < \mathtt{n} \wedge \\ & (\mathtt{m}[\mathtt{i}] < \mathtt{mínimo} \vee \mathtt{máximo} < \mathtt{m}[\mathtt{i}])\,, \end{aligned}$$

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

$$\begin{aligned} CI \wedge G \wedge G_2 &\Rightarrow (\mathbf{E}\, j : 0 \leq j < \mathtt{i}+1 : \mathtt{m}[j] < \mathtt{mínimo} \vee \mathtt{máximo} < \mathtt{m}[j]) \wedge 0 \leq \mathtt{i} < \mathtt{n} \\ &\Rightarrow (\mathbf{E}\, j : 0 \leq j < \mathtt{n} : \mathtt{m}[j] < \mathtt{mínimo} \vee \mathtt{máximo} < \mathtt{m}[j]) \end{aligned}$$

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)
{
    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)
{
    // 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) ∧ (N j : 0 ≤ j < n : m[j] = k) ≥ 2.
    @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) ∧ (N j : 0 ≤ j < n : m[j] = k) ≥ 2.
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

$$CO \equiv \overbrace{(\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n}}^{CI} \wedge \overbrace{\mathtt{m}[\mathtt{i}] = \mathtt{k}}^{\neg G}$$

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) ∧ (N j : 0 ≤ j < n : m[j] = k) ≥ 2.
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 $CI \wedge G$, ou seja:

$$CI \wedge G \equiv (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n} \wedge \mathtt{m}[\mathtt{i}] \neq \mathtt{k}.$$

Mas

$$(\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i}\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge \mathtt{m}[\mathtt{i}] \neq \mathtt{k}$$

é o mesmo que

$$(\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i} + 1\, :\, \mathtt{m}[j] = \mathtt{k}) = 1,$$

pois, sendo $\mathtt{m}[\mathtt{i}] \neq \mathtt{k}$, pode-se estender a gama de valores tomados por j sem afectar a contagem de afirmações verdadeiras:

$$CI \wedge G \equiv (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i} + 1\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n}.$$

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

$$CI \wedge G \equiv (\mathbf{N}\, j\, :\, 0 \leq j < \mathtt{i} + 1\, :\, \mathtt{m}[j] = \mathtt{k}) = 1 \wedge 1 \leq \mathtt{i} < \mathtt{n} - 1.$$

O passo tem de ser escolhido de modo a garantir a invariância da condição invariante, ou seja, de modo a garantir que

```
// CI ∧ G ≡ (N j : 0 ≤ j < i + 1 : m[j] = k) = 1 ∧ 1 ≤ i < n − 1.
passo
// CI ≡ (N j : 0 ≤ j < i : m[j] = k) = 1 ∧ 1 ≤ i < 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

// $PC \equiv 2 \leq \mathtt{n} \wedge \mathtt{n} \leq \dim(\mathtt{m}) \wedge (\mathbf{N}\, j : 0 \leq j < \mathtt{n} : \mathtt{m}[j] = \mathtt{k}) \geq 2.$
`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) ∧ (N j : 0 ≤ j < n : m[j] = k) ≥ 2.
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) ∧ (N j : 0 ≤ j < n : m[j] = k) ≥ 2.
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) ∧ (N j : 0 ≤ j < n : m[j] = k) ≥ 2.
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) ∧ (N j : 0 ≤ j < n : m[j] = k) ≥ 2.
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) ∧ (N j : 0 ≤ j < n : m[j] = k) ≥ 2.
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) ∧ (N j : 0 ≤ j < n : m[j] = k) ≥ 2.
    @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) ∧ (N j : 0 ≤ j < n : m[j] = k) ≥ 1.
    @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) ∧ (N j : 0 ≤ j < n : m[j] = k) ≥ 2.
    @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 uma 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) ∧ (N j : 0 ≤ j < n : m[j] = k) ≥ 1.
    @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 ocorrênciasDe(m, n, k) >= 1);

    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) ∧ (N j : 0 ≤ j < n : m[j] = k) ≥ 2.
    @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 ocorrênciasDe(m, n, k) >= 2);

    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 $PC \equiv 0 \leq \mathtt{n} \wedge \mathtt{n} \leq \dim(\mathtt{m})$.
@post $CO \equiv ((\mathbf{Q}\, j : 0 \leq j < \mathtt{índiceDoPrimeiro} : \mathtt{m}[j] \neq \mathtt{k}) \wedge$
$0 \leq \mathtt{índiceDoPrimeiro} < \mathtt{n} \wedge \mathtt{m}[\mathtt{índiceDoPrimeiro}] = \mathtt{k}) \vee$
$((\mathbf{Q}\, j : 0 \leq j < \mathtt{n} : \mathtt{m}[j] \neq \mathtt{k}) \wedge \mathtt{índiceDoPrimeiro} = \mathtt{n})$. */

```
int índiceDoPrimeiro(int const m[], int const n, int const k)
{
    assert(0 <= n);

    int i = 0;

    // 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;
}
```

[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 [7] usam-se os nomes **cand** e **cor** com o mesmo objectivo. Em [9] 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.

```
/** 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). */
vectro<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 $PC \equiv \mathcal{V}$.
@post $CO \equiv ((\mathbf{Q}\, j : 0 \leq j < \texttt{índiceDoPrimeiro} : \texttt{v}[j] \neq \texttt{k}) \wedge$
$0 \leq \texttt{índiceDoPrimeiro} < \texttt{v.size()} \wedge \texttt{v}[\texttt{índiceDoPrimeiro}] = \texttt{k}) \vee$
$((\mathbf{Q}\, j : 0 \leq j < \texttt{v.size()} : \texttt{v}[j] \neq \texttt{k}) \wedge \texttt{índiceDoPrimeiro} = \texttt{v.size()})$. */

```
vector<int>::size_type índiceDoPrimeiro(vector<int> const& v,
                                         int const k)
{
    vector<int>::size_type i = 0;
```

// $CI \equiv (\mathbf{Q}\, j : 0 \leq j < \texttt{i} : \texttt{v}[j] \neq \texttt{k}) \wedge 0 \leq \texttt{i} \leq \texttt{v.size()}$.

```
    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 $PC \equiv \mathcal{V}$.
@post $CO \equiv ((\mathbf{N}\, j : 0 \leq j < \texttt{índiceDoSegundo} : \texttt{v}[j] = \texttt{k}) = 1 \wedge$
$1 \leq \texttt{índiceDoSegundo} < \texttt{v.size()} \wedge \texttt{v}[\texttt{índiceDoSegundo}] = \texttt{k}) \vee$
$((\mathbf{N}\, j : 0 \leq j < \texttt{v.size()} : \texttt{m}[j] = \texttt{k}) < 2 \wedge \texttt{índiceDoSegundo} = \texttt{v.size()})$. */

```
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 à classe modelo `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 métodos especializados 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 [10].

As cadeias de caracteres suportam todas as operações dos vectores descritas na Secção 5.2, com excepção dos métodos `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 o método `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 todos os métodos relativos a dimensões e capacidade dos vectores, adicionados dos métodos `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 os métodos `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 o método `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 do método `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 aos vários métodos `replace()`, substituir pedaços de uma cadeia por outras sequências de caractere. Os argumentos são os mesmos que para os métodos `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 (método `find()`) ou do fim para o início (método `rfind()`) da cadeia. Estes métodos 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 (método `find_first_of()`) ou do fim para o início (método `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 os métodos `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

$$
\begin{array}{rcl}
\texttt{'A'} & < & \texttt{'a'} \\
\texttt{'a'} & < & \texttt{'á'} \\
\texttt{'Z'} & < & \texttt{'a'} \\
\texttt{'z'} & < & \texttt{'a'}
\end{array}
$$

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 ao método `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 8

Modularização de alto nível

A modularização é um assunto recorrente em programação. De acordo com [3]:

> **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. Das funções e procedimentos membro públicos, apenas fazem parte da interface os respectivos cabeçalhos. A implementação está inacessível ao consumidor da classe[2]. Fazem parte da implementação de uma classe todos os membros privados (incluindo normalmente todos os atributos) e o corpo dos métodos públicos 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

[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.

superior ao da modularização física e, simultaneamente, evitar a colisão de nomes definidos em cada parte de um programa complexo. Assim, um espaço nominativo (ou melhor, um pacote) abarca tipicamente ferramentas definidas em 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 oum 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).

8.1 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., classes), 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, 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 vantagens não apenas em termos da estruturação do programa, como também por 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 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.

8.1.1 Constituição de um módulo

Um módulo físico de um programa é constituído normalmente por dois ficheiros fonte: o ficheiro de interface (*header file*) com extensão `.H` e o ficheiro de implementação com extensão `.C` (alternativamente `.cpp`, `.cc` ou mesmo `.c++`)[3]. 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.

Assim, o ficheiro de interface contém normalmente a declaração de rotinas e a definição das classes disponibilizadas pelo módulo. Cada definição de classe contém a declaração dos respectivos métodos, a definição dos atributos de instância, a declaração dos atributos de classe, e a declaração das classes e rotinas amigos da classe. 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.

8.2 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`) especificados por directivas `#include` e executando 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*.

[3]É boa ideia reservar as extensões `.h` e `.c` para ficheiros de interface escritos na linguagem C.

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, os ficheiros objecto do programa são fundidos (*linked*) num executável. Ao programa que funde os ficheiros objecto chama-se fusor (*linker*).

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.

8.2.1 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 o fazer, 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. O comando para pré-processar um ficheiro de implementação em Linux é[4]:

```
c++ -E nome.C -o nome.ii
```

onde *`nome`* é o nome do módulo a pré-processar e `-E` é uma opção que leva o programa `c++` a limitar-se a proceder à pré-compilação do ficheiro.

O pré-processador copia o código C++ do ficheiro de implementação para a unidade de tradução[5], mas sempre que encontra uma linha começada por um cardinal (`#`) interpreta-a. Estas

[4] O comando também pode ser `g++`.

[5] 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.

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*[6]). 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>
```

pois este é um ficheiro de interface oficial. Neste caso é um ficheiro que faz parte da norma do C++!!citar norma. 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 típico que os ficheiros de interface oficiais não tenham qualquer extensão no seu nome, embora também sejam comuns as extensões `.H` ou `.h`. 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 sistems operativos baseados no Unix há vários directórios com ficheiros de interface oficiais, sendo o mais usual o directório `/usr/include`.

[6]Optou-se por traduzir por *caminho* o inglês *path*.

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". É típico que os ficheiros de interface não-oficiais tenham a extensão `.H`.

Os ficheiros incluídos são ficheiros de interface com declarações de ferramentas úteis ao módulo em processamento.

!!!Fazer figura com exemplo?

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 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. 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 o texto *texto1...* é pré-processado e incluído na unidade de tradução, caso contrário 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`. 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...).

As únicas directivas presentes nas unidades de tradução (`.ii`) têm o seguinte formato:

`#` *`linha nome`* [*x*...]

onde:

linha é 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.

Veja-se um exemplo em Evitando inclusões múltiplas.

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 se a pré-condição da função se verifica. 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[7]:

```
/** 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) ∧
```

[7] 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...

```
              (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 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[8].

Mas pôr e tirar instruções desta forma é uma 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 a 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!"
             << endl;
        exit(1);
    }
```

[8]O modo de distribuição (*release*) é o modo do programa tal como distribuído aos seus utilizadores finais. O modo de depuração (*debug*) ou desenvolvimento é o modo do programa enquanto está em desenvolvimento e teste.

```
    #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!

Esta macro NDEBUG é usada exactamente da mesma forma que acima pelas instruções de asserção descritas na Secção 3.2.19. De resto, as instruções de asserção permitem escrever o código de uma forma mais clara (onde se aproveitou para colocar uma verificação parcial da condição objectivo)[9]:

```
/** 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 pré-compilação -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

[9] Note-se que assert() é, na realidade, uma macro com argumentos (noção que não se descreve neste texto).

```
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
```

8.2.2 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. O comando para compilar uma unidade de tradução nome em Linux 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[10]

   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, simplfica 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.
```

[10] "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.

8.2.3 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. 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. Verifica se existe uma função `main()`. De outra forma não se poderia criar o executável, cuja execução começa sempre nessa função.
3. 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.

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 8.2.4) 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`

```
#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 `iostream.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[11]:

```
/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
```

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:

[11] Estes erros foram obtidos num sistema Linux, distribuição RedHat 7.0, com o GCC versão 2.96, pelo que em sistemas com outra configuração as mensagens podem ser diferentes.

```
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 [**?**, Capitulo 7].

8.2.4 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 libbibliteca.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 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) 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.

8.3 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.

8.3.1 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 $PC \equiv 0 < \texttt{v.size()}$.
@post $CO \equiv (\mathbf{Q}\, j : 0 \leq j < \texttt{v.size()} : \texttt{m}[j] \leq \texttt{máximoDe}) \wedge$
$(\mathbf{E}\, j : 0 \leq j < \texttt{v.size()} : \texttt{m}[j] = \texttt{máximoDe})$. `*/`

```
double máximoDe(vector<double> const& v);
```

`/**` Devolve o índice do primeiro item com o máximo valor do vector `v`.
@pre $PC \equiv 1 \leq \texttt{v.size()}$.
@post $CO \equiv 0 \leq \texttt{índiceDoPrimeiroMáximoDe} < \texttt{v.size()} \wedge$
$(\mathbf{Q}\, j : 0 \leq j < \texttt{v.size()} : \texttt{v}[j] \leq \texttt{v}[\texttt{índiceDoPrimeiroMáximoDe}]) \wedge$
$(\mathbf{Q}\, j : 0 \leq j < \texttt{índiceDoPrimeiroMáximoDe} : \texttt{v}[j] < \texttt{v}[\texttt{índiceDoPrimeiroMáximoDe}])$. `*/`

```
int índiceDoPrimeiroMáximoDe(vector<int> const& v);
```

`/**` Ordena os valores do vector `v`.
@pre $PC \equiv \texttt{v} = v$.
@post $CO \equiv \mathrm{perm}(\texttt{v}, v) \wedge (\mathbf{Q}\, i, j : 0 \leq i \leq j < \texttt{v.size()} : \texttt{v}[i] \leq \texttt{v}[j])$.

```
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 $PC \equiv 0 < \texttt{v.size()}$.
@post $CO \equiv \texttt{éMáximoDe} = ((\mathbf{Q}\, j : 0 \leq j < \texttt{v.size()} : \texttt{m}[j] \leq \texttt{máximo}) \wedge$
$(\mathbf{E}\, j : 0 \leq j < \texttt{v.size()} : \texttt{m}[j] = \texttt{máximo}))$. `*/`

```
bool éMáximoDe(int const máximo, vector<int> const& v)
{
    assert(0 < v.size());

    bool existe = false;
```

`//` $CI \equiv \texttt{existe} = (\mathbf{E}\, j : 0 \leq j < \texttt{i} : \texttt{m}[j] = \texttt{máximo}) \wedge$
`//` $(\mathbf{Q}\, j : 0 \leq j < \texttt{i} : \texttt{m}[j] \leq \texttt{máximo})$.

```
    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 int k = 1; k != v.size(); ++k)
        if(v[k] > v[i])
```

```
            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 8.6.1 mais abaixo).

Assim, seria desejável que as duas funções de verificação tivessem ligação interna.

8.3.2 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 8.6.2). 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 $PC \equiv 0 <$ v.size().
        @post $CO \equiv$ éMáximoDe $= ((\mathbf{Q}\, j : 0 \leq j <$ v.size() $:$ m[$j$] $\leq$ máximo$) \wedge$
                                   $(\mathbf{E}\, j : 0 \leq j <$ v.size() $:$ m[$j$] $=$ máximo$))$. */
    bool éMáximoDe(int const máximo, vector<int> const& v)
    {
        assert(0 < v.size());

        bool existe = false;
        // $CI \equiv$ existe $= (\mathbf{E}\, j : 0 \leq j <$ i $:$ m[$j$] $=$ máximo$) \wedge$
        //       $(\mathbf{Q}\, j : 0 \leq j <$ i $:$ m[$j$] $\leq$ 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$) \wedge$
```

```
    //      (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 int k = 1; k != v.size(); ++k)
        if(v[k] > v[i])
            i = k;

    assert(0 <= i and i < n and éMáximoDe(v[i], v));

    return i;
}

void ordenaPorOrdemNãoDecrescente(int m[], int n)
{
    assert(n >= 0);

    ... // 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.

8.3.3 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 ≡ n ≥ 0 ∨ 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(n >= 0 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 funções ou procedimentos locais. 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 8.1.

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.

O especificador `extern`, embora útil no caso das variáveis e constantes, como se verá mais abaixo, é inútil no caso das rotinas[12]. 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()
{ ... }
```

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 6. As variáveis membro de classe (ou estáticas, ver Secção 6.5.5) 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.

[12]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[]);
```

define.C

```
void f4(); // externo

void f1()  // externo
{...}

void f2()  // externo
{...}

static void f3()  // interno
{...}

void f4()  // externo
{ f3(); }
```

usa.C

```
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 8.1: Exemplo de ligação para rotinas. Declarações em itálico e fontes de erros de compilação a vermelho.

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 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 8.2.

Ao fundir os dois módulos acima (e admitindo que se eliminaram os erros de compilação assinalados) o fusor gerará erros como os seguintes:

<table>
<tr><th><code>define.C</code></th><th><code>usa.C</code></th></tr>
<tr><td>
<code>extern int v4;</code> // externa

<code>int v1;</code> // externa

<code>int v2 = 1;</code> // externa

<code>static int v3 = 3;</code> // interna

<code>int v4;</code> // externa
</td><td>
<code>extern int v1;</code> // externa

<code>extern int v2;</code> // externa

<code>extern int v3;</code> // externa

<code>extern int v5;</code> // externa

<code>extern int v6;</code> // externa

/* Erro (ou aviso) de compilação! Declarada externa e depois definida interna: */
<code>static int v6 = 12;</code>

<code>static int v7;</code> // interna

<code>extern int v7;</code> // interna

<code>int main()</code>
<code>{</code>
   <code>int i = v1 +</code>
      <code>v2 +</code>
      <code>v3 +</code>
      // Erro de compilação, falta declaração:
      <code>v4 +</code>
      <code>v5 +</code>
      <code>v7;</code>
<code>}</code>
</td></tr>
</table>

Figura 8.2: Exemplo de ligação para variáveis. Declarações em itálico e fontes de erros de compilação a vermelho.

```
.../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 6. As constantes membro de classe (ou estáticas, ver Secção 6.5.5) têm ligação 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 porque j é de um ti-
po 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 8.3.

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.

8.3.4 Ligação de classes 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 é simples e semelhante sintacticamente distinção entre declaração e definição no caso das rotinas. A declaração no sentido estrito de uma classe 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:

<table>
<tr><th>define.C</th><th>usa.C</th></tr>
<tr><td>

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

</td><td>

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;
}
```

</td></tr>
</table>

Figura 8.3: Exemplo de ligação para constantes. Declarações em itálico e fontes de erros de compilação a vermelho.

```
class Racional; // declaração.

class Racional { // definição.
  public:
    ...

  private:
    ...
};
```

A declaração em sentido estrito de uma classe é útil em poucas situações. Como só a definição da classe 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. Assim, não é possível definir variáveis de uma classe sem que a classe tenha sido definida antes, embora se possam definir referências e ponteiros (a ver no Capítulo 9) para essa classe. A declaração em sentido estrito usa-se principalmente:

1. Quando se estão a definir duas classes 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 não pode ter atributos de instância da outra classe. 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 com outra embutida e se pretendem separar as duas definições por uma questão de clareza. Por exemplo:

```
class ListaInt {
  public:
```

```
        ...
        class Iterador;
        class IteradorConstante;
        ...
    };

    class ListaInt::Iterador {
        ...
    };

    class ListaInt::IteradorConstante {
        ...
    };
```

É importante também perceber que a definição de uma classe não inclui a definição de funções e procedimentos membro da classe, excepto se esta for feita internamente à classe. Assim, pode-se distinguir entre a *definição de uma classe* e a *definição completa de uma classe*, que inclui a definição de todos os seus membros.

Podem-se definir tipo enumerados ou mesmo classes 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(numero_de_valores != 0);
            return soma_dos_valores / numero_de_valores;
        }
      private:
          double soma_dos_valores;
          int numero_de_valores;
    };
```

```
    Média m;
    m += 3;
    m += 5;
    m += 1;
    cout << m << endl;
}
```

define uma classe `Média`[13] localmente à função `main()`[14]. 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 outras 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 rotinas membro que não são 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 tem 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.

8.4 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

[13] 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 +=.

[14] 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.

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.

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, como sugerido na Secção 8.3.2. 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.

8.4.1 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>
...
```

8.4.2 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:
    ...
}
```

8.4.3 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, as rotinas não-membro em linha podem ser definidas num terceiro ficheiro do módulo, como descrito na Secção 8.4.15.

8.4.4 Variáveis globais

As variáveis globais não devem ser usadas.

8.4.5 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.

8.4.6 Tipos enumerados não-membro

Os tipos enumerados têm de ser definidos no ficheiro de interface (`.H`).

***módulo*.H**

```
// Definições de enumerados:

enum Nome { ... };
```

8.4.7 Classes não-membro

As classes, 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.
};
```

8.4.8 Métodos (rotinas membro)

Os métodos 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, os métodos, quer de classe quer de instância, são sempre declarados dentro da classe e portanto no ficheiro de interface (`.H`). A sua definição será feita 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 8.4.15.

8.4.9 Variáveis e constantes membro de instância

As variáveis e constantes membro de instância são definidas dentro da própria classe e inicializadas pelos seus construtores, pelo que não requerem qualquer tratamento especial.

8.4.10 Variáveis membro de classe

As variáveis membro de classe declaram-se dentro da classe, 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:

tipo Nome::nome; // valor por omissão.

tipo Nome::nome = expressão;

tipo Nome::nome(argumentos);
```

8.4.11 Constantes membro de classe

As constantes membro de classe tratam-se exactamente como as variáveis membro de classe: declaram-se na classe, 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 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 limite = 100;
    ...
    int matriz[limite];
    ...
};
```

8.4.12 Classes membro (embutidas)

As classes membro de outras classes, ou seja, embutidas, podem ser definidas dentro da classe 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 envolvente. Aos membros de classes embutidas aplicam-se os mesmos comentários que aos membros da classe 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 {
        ...
    };
    ...
};
```

8.4.13 Enumerados membro

Os tipos enumerados membro têm de ser definidos dentro da classe 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
        };
        ...
    };
```

8.4.14 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 condicionada. 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.

8.4.15 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 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 11 para mais pormenores.

8.5 Construção automática do ficheiro executável

A construção do ficheiro excutável de um programa constituído por vários módulos exige uma sequência de comandos. 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, e `CXXFLAGS`, que guarda as opções de compilação desejadas. 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"
```

Estes comandos também podem ser colocados no ficheiro de configuração do interpretador de comandos: `~/.tcshrc`[15].

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...
```

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`,

[15]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.

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[16]:

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_;
};

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);
```

aluno.C

```
#include "aluno.H"

using namespace std;
```

[16]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.

```
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
```

8.6 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.

8.6.1 Colisão de nomes

Um problema comum em grandes projectos é a existência de entidades (classes, variáveis, funções ou procedimentos) 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, se se está a desenvolver uma aplicação de gestão, pode-se resolver 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[17]:

```
void consolida();
```

Simplificando, os ficheiros fornecidos por cada empresa são:

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

[17]Fica a cargo do leitor imaginar o que cada um desses dois procedimentos faz.

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.

8.6.2 Espaços nominativos

A solução é escolher fornecedores que garantam um mínimo de possibilidades 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();
}
```

8.6.3 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.

8.6.4 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.

8.6.5 Espaços nominativos e modularização física

Podem-se colocar vários módulos num mesmo espaço nominativo[18]. 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

`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()`.

[18] E vice-versa, mas não é tão útil...

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.

8.6.6 Ficheiros de interface

Os ficheiros de interface devem ter o conteúdo descrito anteriormente (ver Secção 8.4), 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.

8.6.7 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()
{
    ...
}
```

8.6.8 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).

8.7 Exemplo final

!!No exemplo pôr documentação. Gerar html e pdf com doxygen e colocar disponível na rede. O ideal eram links pdf...

Apresenta-se aqui o exemplo da Secção 8.5, 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, opcionalmen-
    te, 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_(numero), 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 de-
pois 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
```

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 "igual a" ou "equivalente a".

$\equiv$ significa "é o mesmo que".

$\neq$ 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):[2]

    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] Notação retirada de [7].

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:

$$\mathrm{perm}(v_1, v_2) \equiv \dim(v_1) = \dim(v_2) \wedge (\mathbf{Q}\, x :: (\mathbf{N}\, j : 0 \leq j < \dim(v_1) : v_1[j] = x) = (\mathbf{N}\, j : 0 \leq j < \dim(v_2) :$$

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, $(\mathbf{Q}\, i \in \mathbb{N} : i\,\mathrm{par} : \cdots)$: 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 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 tes-
te:
    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 modelo `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 modelo 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 modelo para permitir acesso completo, incluindo escritas. Essa classe modelo fica como exercício!

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`

Descrição	**Sintaxe** (itálico: partes variáveis da sintaxe)
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 |)

Descrição	**Sintaxe** (itálico: partes variáveis da sintaxe)
	§
conjunção	*expressão* `and` *expressão* (ou `&&`)
	§
disjunção	*expressão* `or` *expressão* (ou `||`)
	§
operador condicional	*expressão* `?` *expressão* `:` *expressão*
	§
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`)
	§
lançamento de excepção	`throw` *expressão*
sequenciamento	*expressão* `,` *expressão*

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 esclamação	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	¤
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 maiúsculo grave	224	à
a maiúsculo agudo	225	á
a maiúsculo circunflexo	226	â
a maiúsculo til	227	ã
a maiúsculo trema	228	ä
a maiúsculo círculo	229	å
ae ligado maiúsculo	230	æ
c maiúsculo cedilha	231	ç
e maiúsculo grave	232	è
e maiúsculo agudo	233	é
e maiúsculo circunflexo	234	ê
e maiúsculo trema	235	ë
i maiúsculo grave	236	ì
i maiúsculo agudo	237	í
i maiúsculo circunflexo	238	î
i maiúsculo trema	239	ï
eth maiúsculo	240	ð
n maiúsculo til	241	ñ
o maiúsculo grave	242	ò
o maiúsculo agudo	243	ó
o maiúsculo circunflexo	244	ô
o maiúsculo til	245	õ
o maiúsculo trema	246	ö
sinal de multiplicação	247	÷
o maiúsculo cortado	248	ø
u maiúsculo grave	249	ù
u maiúsculo agudo	250	ú
u maiúsculo circunflexo	251	û
u maiúsculo trema	252	ü
y maiúsculo agudo	253	ý
thorn maiúsculo	254	þ
y minúsculo trema	255	ÿ

Bibliografia

[1] Doug Bell, Ian Morrey, e John Pugh. *Software Engineering: A Programming Approach*. Prentice Hall, Nova Iorque, 2ª edição, 1992. 59

[2] Samuel D. Conte e Carl de Boor. *Elementary Numerical Analysis*. McGraw-Hill International, Auckland, 1983. 37

[3] Aurélio Buarque de Holanda Ferreira. *Novo Aurélio Século XXI: O Dicionário da Lingua Portuguesa*. Editora Nova Fronteira, Rio de Janeiro, 3ª edição, 1999. 253, 341

[4] Edsger Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976. 196, 198

[5] Irv Englander. *The Architecture of Computer Hardware and Systems Software: An Information Technology Approach*. John Wiley & Sons, Inc., Nova Iorque, 1996. 30, 31, 36

[6] Ronald L. Graham, Donald E. Knuth, e Oren Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley, 2ª edição, 1994. 297

[7] David Gries. *The Science of Programming*. Texts and Monographs in Computer Science. Springer-Verlag, Nova Iorque, 1981 [1989]. 143, 182, 193, 196, 277, 418

[8] Michael K Johnson e Erik W. Troan. *Linux Application Development*. Addison-Wesley, Reading, Massachusetts, 1998. 357

[9] Donald E. Knuth. *Fundamental algorithms*, volume 1 de *The Art of Computer Programming*. Addison-Wesley Publishing Company, Reading, Massachusetts, 2ª edição, 1973. 3, 4, 182

[10] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall PTR, Upper Saddle River, New Jersey, 2ª edição, 1997. 108, 277

[11] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 3ª edição, 1998. 20, 171, 283, 290