Capítulo 7

Tipos abstractos de dados e classes C++

In this connection it might be worthwhile to point out that the purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.

Edsger W Dijkstra, The Humble Programmer, *Communications of the ACM*, 15(10), 1972

Quando se fala de uma linguagem de programação, não se fala apenas da linguagem em si, com o seu léxico, sintaxe, gramática e semântica. Fala-se também de um conjunto de ferramentas acessíveis ao programador que, não fazendo parte da linguagem propriamente dita, estão acessíveis em qualquer ambiente de desenvolvimento de programas. Ao conjunto dessas ferramentas adicionais que se encontra em todos os ambientes de desenvolvimento chama-se biblioteca padrão (*standard library*). Da biblioteca padrão do C++ fazem parte, por exemplo, os canais `cin` e `cout`, que permitem leituras do teclado e escritas para o ecrã, o tipo `string` e o tipo genérico `vector`. Em rigor, portanto, o programador tem à sua disposição não a linguagem em si, mas a linguagem equipada com a biblioteca padrão. Para o programador, no entanto, tudo funciona como se a linguagem em si incluísse essas ferramentas. Isto é, para o programador em C++ o que está acessível não é o C++, mas um "C++ ++" de que fazem parte todas as ferramentas da biblioteca padrão.

A tarefa de um programador é resolver problemas usando (pelo menos) um computador. Fá-lo através da escrita de programas numa linguagem de programação dada. Depois de especificado o problema com exactidão, o programador inteligente começa por procurar, na linguagem básica, na biblioteca padrão e noutras quaisquer bibliotecas disponíveis, ferramentas que resolvam o problema na totalidade ou pelo menos parcialmente: esta procura evita as perdas de tempo associadas ao reinventar da roda infelizmente ainda tão em voga[1]. Se não existirem

[1]Por outro lado, é importante notar que se pede muitas vezes ao *estudante* que reinvente a roda. Fazê-lo é parte fundamental do treino na resolução de problemas concretos. Convém, portanto, que o estudante se disponha a essa tarefa que fora do contexto da aprendizagem é inútil. Mas convém também que não se deixe viciar na resolução por si próprio de todos os pequenos problemas que já foram resolvidos milhares de vezes. É importante saber fazer um equilíbrio entre a curiosidade intelectual de resolver esses problemas e o pragmatismo de procurar um solução já pronta. Durante a vida académica, a balança deve pender fortemente no sentido da curiosidade intelectual. Finda a vida académica, o equilíbrio deve pender mais para o pragmatismo.

ferramentas disponíveis, então há que construí-las. Ao fazê-lo, o programador está a expandir mais uma vez a linguagem disponível, que passa a dispor de ferramentas adicionais (digamos que "incrementa" de novo a linguagem para "C++ ++ ++").

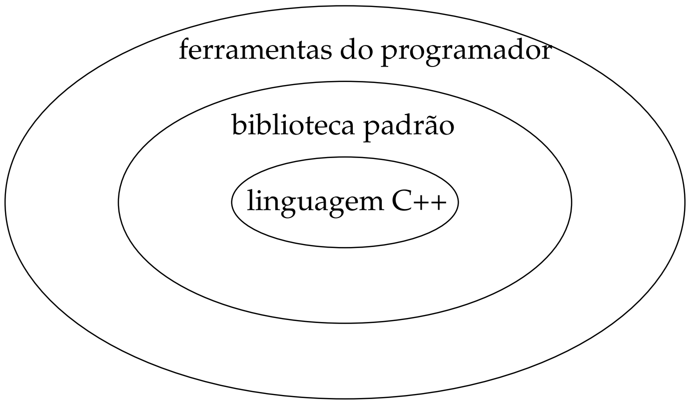


Figura 7.1: A biblioteca padrão do C++ e as ferramentas do programador como extensões à funcionalidade básica da linguagem C++.

Há essencialmente duas formas distintas de construir ferramentas adicionais para uma linguagem. A primeira passa por equipar a linguagem com operações adicionais, na forma de rotinas, mas usando os tipos existentes (`int`, `char`, `bool`, `double`, matrizes, etc.): é a chamada programação procedimental. A segunda passa por adicionar tipos à linguagem e engloba a programação centrada nos dados (ou programação baseada em objectos). Para que os novos tipos criados tenham algum interesse, é fundamental que tenham operações próprias, que têm de ser concretizadas pelo programador. Assim, a segunda forma de expandir a linguagem passa necessariamente pela primeira.

Neste capítulo ver-se-á a forma por excelência de acrescentar tipos, e respectivas operações, à linguagem. No capítulo anterior abordaram-se as simples e limitadas enumerações, neste ver-se-ão os tipos abstractos de dados, peça fundamental da programação centrada nos dados. A partir deste ponto, portanto, o ênfase será posto na construção de novos tipos. Neste capítulo construir-se-ão novos tipos relativamente simples e independentes uns dos outros. Quando se iniciar o estudo da programação orientada por objectos, em capítulos posteriores, ver-se-á como se podem desenhar classes e hierarquias de classes e quais as suas aplicações na resolução de problemas de maior escala.

7.1 De novo a soma de fracções

Na Secção 3.2.20 desenvolveu-se um pequeno programa para ler duas fracções do teclado e mostrar a sua soma. Neste capítulo desenvolver-se-á esse programa até construir uma pequena calculadora. Durante esse processo aproveitar-se-á para introduzir uma quantidade considerável de conceitos novos.

O programa apresentado na Secção 3.2.20 pode ser melhorado. Assim, apresenta-se abaixo uma versão melhorada nos seguintes aspectos:

- A noção de máximo divisor comum é facilmente generalizável a inteiros negativos ou nulos. O único caso complicado é o de $\mathrm{mdc}(0,0)$. Como é óbvio, todos os inteiros positivos são divisores comuns de zero, pelo que não existe este máximo divisor comum. No entanto, é de toda a conveniência estender a definição do máximo divisor comum, arbitrando o valor 1 como resultado de $\mathrm{mdc}(0,0)$. Ou seja, por definição $\mathrm{mdc}(0,0) = 1$. Assim, a função `mdc()` foi flexibilizada, tendo-se enfraquecido a respectiva pré-condição de modo a ser aceitar argumentos arbitrários. A utilidade da cobertura do caso $\mathrm{mdc}(0,0)$ será vista mais tarde.

- O enfraquecimento da pré-condição da função `mdc` permitiu enfraquecer também todas as restantes pré-condições, tornando o programa capaz de lidar com fracções com termos negativos.

- O ciclo usado na função `mdc()` foi optimizado, passando a usar-se um ciclo pouco ortodoxo, com duas possíveis saídas. Fica como exercício para o leitor demonstrar o seu correcto funcionamento e verificar a sua eficiência.

- Foram acrescentadas rotinas para a leitura e cálculo da soma de duas fracções.

- Nas rotinas lidando com fracções alterou-se o nome das variáveis para explicitar melhor aquilo que representam (e.g., `numerador` em vez de `n`).

- Para evitar código demasiado extenso para uma versão impressa deste texto, cada rotina é definida antes das rotinas que dela fazem uso, não se fazendo uma distinção clara entre declaração e definição. Mais tarde ser verá que esta não é forçosamente uma boa solução.

- Uma vez que a pré-condição e a condição objectivo são facilmente identificáveis pela sua localização na documentação das rotinas, após “@pre” e “@post” respectivamente, abandonou-se o hábito de nomear essas condições PC e CO.

- Protegeu-se de erros a leitura das fracções (ver Secção 7.14).

```
#include <iostream>
#include <cassert>

using namespace std;

/** Devolve o máximo divisor comum dos inteiros passados como argumento.
    @pre m = m ∧ n = n.
    @post mdc = { mdc(m, n)   m ≠ 0 ∨ n ≠ 0
                {     1       m = 0 ∧ n = 0   . */
int mdc(int m, int n)
{
    if(m == 0 and n == 0)
        return 1;

    if(m < 0)
        m = -m;
    if(n < 0)
```

```
        n = -n;

    while(true) {
        if(m == 0)
            return n;
        n = n % m;
        if(n == 0)
            return m;
        m = m % n;
    }
}
```

/** Reduz a fracção recebida como argumento.
@pre $\texttt{denominador} \neq 0 \wedge \texttt{denominador} = d \wedge \texttt{numerador} = n$.
@post $\texttt{denominador} \neq 0 \wedge \mathrm{mdc}(\texttt{numerador}, \texttt{denominador}) = 1 \wedge \frac{\texttt{numerador}}{\texttt{denominador}} = \frac{n}{d}$. */

```
void reduzFracção(int& numerador, int& denominador)
{
    assert(denominador != 0);

    int máximo_divisor_comum = mdc(numerador, denominador);

    numerador /= máximo_divisor_comum;
    denominador /= máximo_divisor_comum;

    assert(denominador != 0 and mdc(numerador, denominador) == 1);
}
```

/** Lê do teclado uma fracção, na forma de dois inteiros sucessivos.
@pre $\texttt{numerador} = n \wedge \texttt{denominador} = d$.
@post Se `cin` e `cin` tem dois inteiros n' e d' disponíveis para leitura, com $d' \neq 0$, então

$$0 < \mathit{denominador} \wedge \mathrm{mdc}(\texttt{numerador}, \texttt{denominador}) = 1 \wedge \frac{\texttt{numerador}}{\texttt{denominador}} = \frac{n'}{d'} \wedge \texttt{cin},$$

senão

$$\texttt{numerador} = n \wedge \texttt{denominador} = n \wedge \neg\texttt{cin}.$$ */

```
void lêFracção(int& numerador, int& denominador)
{
    int n, d;

    if(cin >> n >> d)
        if(d == 0)
            cin.setstate(ios_base::failbit);
        else {
            numerador = d < 0 ? -n : n;
            denominador = d < 0 ? -d : d;
```

```
            reduzFracção(numerador, denominador);

            assert(0 < denominador and mdc(numerador, denominador) == 1 and
                   numerador * d == n * denominador and cin);

            return;
        }

    assert(not cin);
}
```

/** Soma duas fracções.
@pre $\mathtt{denominador1} \neq 0 \wedge \mathtt{denominador2} \neq 0$.
@post $\frac{\mathtt{numerador}}{\mathtt{denominador}} = \frac{\mathtt{numerador1}}{\mathtt{denominador1}} + \frac{\mathtt{numerador2}}{\mathtt{denominador2}} \wedge$
$\mathtt{denominador} \neq 0 \wedge \mathrm{mdc}(\mathtt{numerador}, \mathtt{denominador}) = 1$. */

```
void somaFracção(int& numerador, int& denominador,
                 int const numerador1, int const denominador1,
                 int const numerador2, int const denominador2)
{
    assert(denominador1 != 0 and denominador2 != 0);

    numerador = numerador1 * denominador2 + numerador2 * denominador1;
    denominador = denominador1 * denominador2;

    reduzFracção(numerador, denominador);

    assert(denominador != 0 and mdc(numerador, denominador) == 1);
}
```

/** Escreve uma fracção no ecrã no formato usual.
@pre $\mathcal{V}$.
@post $\neg\mathtt{cout} \vee$ `cout` contém n/d (ou simplesmente n se $d = 1$)
sendo n e d os valores de `numerador` e `denominador`. */

```
void escreveFracção(int const numerador, int const denominador)
{
    cout << numerador;
    if(denominador != 1)
        cout << '/' << denominador;
}

int main()
{
    // Ler fracções:
    cout << "Introduza duas fracções (numerador denominador): ";
    int n1, d1, n2, d2;
```

```
    lêFracção(n1, d1);
    lêFracção(n2, d2);

    if(not cin) {
        cerr << "Opps!  A leitura das fracções falhou!" << endl;
        return 1;
    }

    // Calcular fracção soma reduzida:
    int n, d;
    somaFracção(n, d, n1, d1, n2, d2);

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

A utilização de duas variáveis inteiras independentes para representar cada fracção não permite a definição de uma função para proceder à soma, visto que as funções em C++ podem devolver um único valor. De facto, a utilização de múltiplas variáveis independentes para representar um único valor torna o código complexo e difícil de perceber. O ideal seria poder reescrever o código da mesma forma que se escreveria se o seu objectivo fosse ler e somar inteiros, e não fracções. Sendo as fracções representações dos números racionais, pretende-se escrever o programa como se segue:

```
...

int main()
{
    cout << "Introduza duas fracções (numerador denominador): ";
    Racional r1, r2;
    cin >> r1 >> r2;

    if(not cin) {
        cerr << "Opps!  A leitura dos racionais falhou!" << endl;
        return 1;
    }

    Racional r = r1 + r2;

    cout << "A soma de " << r1 << " com " << r2 << " é "
```

```
            << r << '.' << endl;
}
```

Este objectivo irá ser atingido ainda neste capítulo.

7.2 Tipos Abstractos de Dados e classes C++

Como representar cada número racional com uma variável apenas? É necessário definir um novo tipo que se comporte como qualquer outro tipo existente em C++. É necessário um TAD (Tipo Abstracto de Dados) ou tipo de primeira categoria[2]. Um TAD ou tipo de primeira categoria é um tipo definido pelo programador que se comporta como os tipos básicos, servindo para definir instâncias, i.e., variáveis ou constantes, que guardam valores sobre os quais se pode operar. A linguagem C++ proporciona uma ferramenta, as classes C++, que permite concretizar tipos de primeira categoria.

É importante notar aqui que o termo "classe" tem vários significados. Em capítulos posteriores falar-se-á de classes propriamente ditas, que servem para definir as características comuns de objectos dessa classe, e que se concretizam também usando as classes C++. Este capítulo, por outro lado, debruça-se sobre os TAD, que também se concretizam à custa de classes C++. Se se acrescentar que a fronteira entre TAD, cujo objectivo é definir instâncias, e as classes propriamente ditas, cujo objectivo é definir as características comuns de objectos independentes, percebe-se que é inevitável alguma confusão de nomenclatura. Assim, sempre que se falar simplesmente de classe, será na acepção de classe propriamente dita, enquanto que sempre que se falar do mecanismo da linguagem C++ que permite concretizar quer TAD quer classes propriamente ditas, usar-se-á sempre a expressão "classe C++"[3].

Assim:

TAD Tipo definido pelo utilizador que se comporta como qualquer tipo básico da linguagem. O seu objectivo é permitir a definição de instâncias que armazenam valores. O que distingue umas instâncias das outras é fundamentalmente o seu valor. Nos TAD o ênfase põe-se na igualdade, pelo que as cópias são comuns.

Classe propriamente dita Conceito mais complexo a estudar em capítulos posteriores. Representam as características comuns de objectos independentes. O seu objectivo é poder construir objectos independentes de cuja interacção e colaboração resulte o comportamento adequado do programa. O ênfase põe-se na identidade, e não na igualdade, pelo que as cópias são infrequentes, merecendo o nome de clonagens.

Classe C++ Ferramenta da linguagem que permite implementar quer TAD, quer classes propriamente ditas.

---

[2] Na realidade os tipos de primeira categoria são concretizações numa linguagem de programação de TAD, que são uma abstracção matemática. Como os TAD na sua acepção matemática estão fora (por enquanto) do âmbito deste texto, os dois termos usam-se aqui como sinónimos.

[3] Apesar do cuidado posto na redacção deste texto é provável que aqui e acolá ocorram violações a esta convenção. Espera-se que não sejam factor de distracção para o leitor.

7.2.1 Definição de TAD

É possível definir um novo tipo (um TAD) para representar números racionais (na forma de uma fracção), como se segue:

```
/** Representa números racionais. */
class Racional {
  public: // Isto é magia (por enquanto).
    int numerador;
    int denominador;
};
```

A sintaxe de definição de um TAD à custa de uma classe C++ é, portanto,

```
class nome_do_tipo {
    declaração_de_membros
};
```

sendo importante notar que este é um dos poucos locais onde a linguagem exige um terminador (`;`) depois de uma chaveta final[4].

A notação usada para representar a classe C++ `Racional` pode ser vista na Figura 7.2.

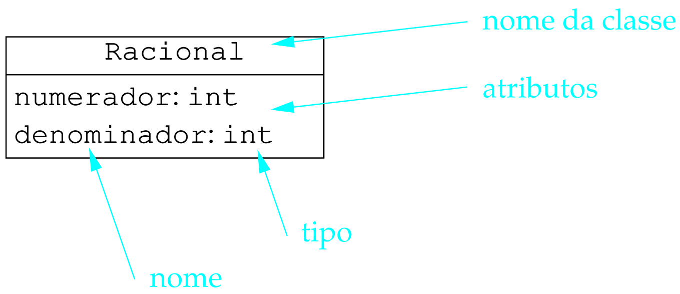


Figura 7.2: Notação usada para representar a classe C++ `Racional`.

A definição de uma classe C++ consiste na declaração dos seus *membros*. A definição da classe estabelece um modelo segundo o qual serão construídas as respectivas variáveis. No caso apresentado, as variáveis do tipo `Racional`, quando forem construídas, consistirão em dois membros: um numerador e um denominador do tipo `int`. Neste caso os membros são simples

[4] Ao contrário do que acontece na definição de rotinas e nos blocos de instruções em geral, o terminador é aqui imprescindível, pois a linguagem C++ permite a definição simultânea de um novo tipo de de variáveis desse tipo. Por exemplo:

```
class Racional {
    ...
} r; // Define o TAD Racional e uma variável r numa única instrução. Má ideia, mas possível.
```

Note-se que esta possibilidade deve ser evitada na prática.

variáveis, mas poderiam ser também constantes. Às variáveis e constantes membro de uma classe dá-se o nome de *atributos*.

Tal como as matrizes, as classes permitem guardar agregados de informação (ou seja, agregados de variáveis ou constantes, chamados elementos no caso das matrizes e membros no caso das classes), com a diferença de que, no caso das classes, essa informação pode ser de tipos diferentes.

As variáveis de um TAD definem-se como qualquer variável do C++:

```
TAD nome [= expressão];
```

ou

```
TAD nome[(expressão, ...)];
```

Por exemplo:

```
Racional r1, r2;
```

define duas variáveis `r1` e `r2` não inicializadas, i.e., contendo lixo (mais tarde se verá como se podem evitar construções sem inicialização em TAD).

Para classes C++ que representem meros agregados de informação é possível inicializar cada membro da mesma forma como se inicializam os elementos de uma matriz clássica do C++:

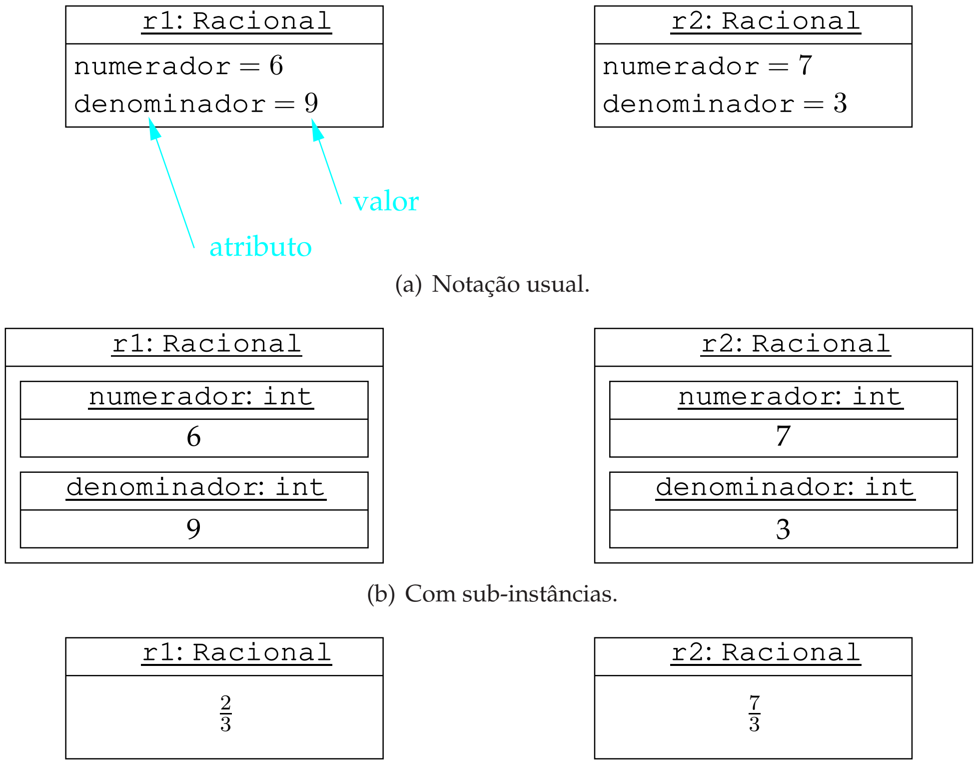
```
Racional r1 = {6, 9};

Racional r2 = {7, 3};
```

Note-se, no entanto, que esta forma de inicialização deixará de ser possível (e desejável) quando se equipar a classe C++ com um construtor, como se verá mais à frente.

As instruções apresentadas constroem duas novas variáveis do tipo `Racional`, `r1` e `r2`, cada uma das quais com versões próprias dos atributos `numerador` e `denominador`. Às variáveis de um TAD também é comum chamar-se objectos e instâncias, embora em rigor o termo objecto deva ser reservado para as classes propriamente ditas, a estudar em capítulos posteriores.

Para todos os efeitos, os atributos da classe `Racional` funcionam como variáveis guardadas quer dentro da variável `r1`, quer dentro da variável `r2`. A notação usada para representar instâncias de uma classe é a que se pode ver na Figura 7.3, onde fica claro que os atributos são parte das instâncias da classe. Deve-se comparar a Figura 7.2 com a Figura 7.3, pois na primeira representa-se a classe `Racional` e na segunda as variáveis `r1` e `r2` dessa classe.



(a) Notação usual.

(b) Com sub-instâncias.

(c) Como TAD com valor lógico representado.

Figura 7.3: Notações usadas para representar instâncias da classe C++ `Racional`.

7.2.2 Acesso aos membros

O acesso aos membros de uma instância de uma classe C++ faz-se usando o operador de selecção de membro, colocando como primeiro operando a instância a cujo membro se pretende aceder, depois o símbolo `.` e finalmente o nome do membro pretendido:

```
instância.membro
```

Por exemplo,

```
Racional r1, r2;

r1.numerador = 6;
r1.denominador = 9;

r2.numerador = 7;
r2.denominador = 3;
```

constrói duas novas variáveis do tipo `Racional` e atribui valores aos respectivos atributos.

Os nomes dos membros de uma classe só têm visibilidade dentro dessa classe, pelo que poderia existir uma variável de nome `numerador` sem que isso causasse qualquer problema:

```
Racional r = {6, 9};

int numerador = 1000;
```

7.2.3 Alguma nomenclatura

Às instâncias, i.e., variáveis ou constantes, de uma classe C++ é comum chamar-se *objectos*, sendo essa a razão para as expressões "programação baseada em objectos" e "programação orientada para os objectos". No entanto, reservar-se-á o termo objecto para classes C++ que sejam concretizações de classes propriamente ditas, e não para classes C++ que sejam concretizações de TAD.

Às variáveis e constantes membro de uma classe C++ também se chama *atributos*.

Podem também existir rotinas membro de uma classe C++. A essas funções ou procedimentos chama-se *operações*. No contexto das classes propriamente ditas, em vez de se dizer "invocar uma operação para uma instância (de uma classe)" diz-se por vezes "enviar uma mensagem a um objecto".

Como se verá mais tarde, quer os atributos, quer as operações podem ser de instância ou de classe, consoante cada instância da classe C++ possua conceptualmente a sua própria cópia do membro em causa ou exista apenas uma cópia desse membro partilhada entre todas as instâncias da classe.

Todas as rotinas têm uma interface e uma implementação, e as rotinas membro não são excepção. Normalmente o termo operação é usado para designar a rotina membro do ponto de vista da sua interface, enquanto o termo *método* é usado para designar a implementação da rotina membro. Para já, a cada operação corresponde um e um só método, mas mais tarde se verá que é possível associar vários métodos à mesma operação.

Ao conjunto dos atributos e das operações de uma classe C++ chama-se *características*, embora, como se verá, o que caracteriza um TAD seja apenas a sua interface, que normalmente não inclui quaisquer atributos.

7.2.4 Operações suportadas pelas classes C++

Ao contrário do que se passa com as matrizes, as variáveis de uma classe C++ podem-se atribuir livremente entre si. O efeito de uma atribuição é o de copiar todos os atributos (de instância) entre as variáveis em causa. Da mesma forma, é possível construir uma instância de uma classe a partir de outra instância da mesma classe, ficando a primeira igual à segunda. Por exemplo:

```
Racional r1 = {6, 9};
Racional r2 = r1; // r2 construída igual a r1.

Racional r3;

r3 = r1;          // o valor de r1 é atribuído a r3, ficando as variáveis iguais.
```

Da mesma forma, estão bem definidas as devoluções e a passagem de argumentos por valor para valores de uma classe C++: as instâncias de um TAD concretizado por intermédio de uma classe C++ podem ser usadas exactamente da mesma forma que as instâncias dos tipos básicos. É possível, por isso, usar uma função, e não um procedimento, para calcular a soma de dois racionais no programa em desenvolvimento. Antes de o fazer, no entanto, far-se-á uma digressão sobre as formas de representação de número racionais.

7.3 Representação de racionais por fracções

Qualquer número racional pode ser representado por uma fracção, que é um par ordenado de números inteiros (n, d), em que n e d são os termos da fracção[5]. Ao segundo termo dá-se o nome de *denominador* (é o que dá o nome à fracção) e ao primeiro *numerador* (diz a quantas fracções nos referimos). Por exemplo, $(3, 4)$ significa "três quartos". Normalmente os racionais representam-se graficamente usando uma notação diferente da anterior: n/d ou $\frac{n}{d}$.

Uma fracção $\frac{n}{d}$ só representa um número racional se $d \neq 0$. Por outro lado, é importante saber se fracções diferentes podem representar o mesmo racional ou se, pelo contrário, fracções diferentes representam sempre racionais diferentes. A resposta à questão inversa é evidente:

[5] Há representações alternativas para as fracções, ver [1][7].

racionais diferentes têm forçosamente representações diferentes. Mas $\frac{-4}{2}$, $\frac{2}{-1}$ e $\frac{-2}{1}$ são fracções que correspondem a um único racional, e que, por acaso, também é um inteiro. Para se obter uma representação em fracções que seja única para cada racional, é necessário introduzir algumas restrições adicionais.

Em primeiro lugar, é necessário usar apenas o numerador ou o denominador para conter o sinal do número racional. Como já se impôs uma restrição ao denominador, viz. $d \neq 0$, é natural impor uma restrição adicional: d deve ser não-negativo. Assim, $0 < d$. Mas é necessária uma restrição adicional. Para que a representação seja única, é também necessário que n e d não tenham qualquer divisor comum diferente de 1, i.e., que $\mathrm{mdc}(n, d) = 1$. Uma fracção nestas condições diz-se em termos mínimos e dos seus termos diz-se que são *mutuamente primos*. Dos três exemplos acima ($\frac{-4}{2}$, $\frac{2}{-1}$ e $\frac{-2}{1}$), apenas a última fracção verifica todas as condições enunciadas, ou seja, tem denominador positivo e numerador e denominador são mutuamente primos. Uma fracção $\frac{n}{d}$ que verifique estas condições, i.e., $0 < d \wedge \mathrm{mdc}(n, d) = 1$, diz-se no *formato canónico*.

7.3.1 Operações aritméticas elementares

As operações aritméticas elementares (adição, subtracção, multiplicação, divisão, simétrico e identidade) estão bem definidas para os racionais (com excepção da divisão por 0, ou melhor, por $\frac{0}{1}$). Assim, em termos da representação dos racionais como fracções, o resultado das operações aritméticas elementares pode ser expresso como

$$
\begin{aligned}
\frac{n_1}{d_1} + \frac{n_2}{d_2} &= \frac{n_1 \times d_2 + n_2 \times d_1}{d_1 \times d_2}, \\
\frac{n_1}{d_1} - \frac{n_2}{d_2} &= \frac{n_1 \times d_2 - n_2 \times d_1}{d_1 \times d_2}, \\
\frac{n_1}{d_1} \times \frac{n_2}{d_2} &= \frac{n_1 \times n_2}{d_1 \times d_2}, \\
\frac{n_1}{d_1} / \frac{n_2}{d_2} &= \frac{\frac{n_1}{d_1}}{\frac{n_2}{d_2}} = \frac{n_1 \times d_2}{d_1 \times n_2} \text{ se } n_2 \neq 0, \\
-\frac{n}{d} &= \frac{-n}{d}, \text{ e} \\
+\frac{n}{d} &= \frac{n}{d}.
\end{aligned}
$$

7.3.2 Canonicidade do resultado

Tal como definidas, algumas destas operações sobre fracções não garantem que o resultado esteja no formato canónico, mesmo que as fracções que servem de operandos o estejam. Este problema é fácil de resolver, no entanto, pois dada uma fracção $\frac{n}{d}$ que não esteja forçosamente no formato canónico, pode-se dividir ambos os termos pelo seu máximo divisor comum para obter uma fracção equivalente em termos mínimos, $\frac{n/\,\mathrm{mdc}(n,d)}{d/\,\mathrm{mdc}(n,d)}$, e, se o denominador for negativo, pode-se multiplicar ambos os termos por -1 para obter uma fracção equivalente com o numerador positivo, $\frac{-n}{-d}$.

7.3.3 Aplicação à soma de fracções

Voltando à classe C++ definida,

```
/** Representa números racionais. */
class Racional {
  public: // Isto é magia (por enquanto).
    int numerador;
    int denominador;
};
```

é muito importante estar ciente das diferenças entre a concretização do conceito de racional e o conceito em si: os valores representáveis num `int` são limitados, o que significa que não é possível representar qualquer racional numa variável do tipo `Racional`, tal como não era possível representar qualquer inteiro numa variável do tipo `int`. Os problemas causados por esta diferença serão ignorados durante a maior parte deste capítulo, embora na Secção 7.13 sejam, senão resolvidos, pelo menos mitigados.

Um mero agregado de dois inteiros, mesmo com um nome sugestivo, não só tem pouco interesse, como poderia representar muitas coisas diferentes. Para que esse agregado possa ser considerado a concretização de um TAD, é necessário definir também as operações que o novo tipo suporta. Uma das operações a implementar é a soma. Pode-se implementar a soma actualizando o procedimento do programa original para a seguinte função:

```
/** Devolve a soma de dois racionais.
    @pre r1.denominador ≠ 0 ∧ r2.denominador ≠ 0.
    @post somaDe = r1 + r2∧
          somaDe.denominador ≠
0 ∧ mdc(somaDe.numerador, somaDe.denominador) = 1. */
Racional somaDe(Racional const r1, Racional const r2)
{
    assert(r1.denominador1 != 0 and r2.denominador2 != 0);

    Racional r;

    r.numerador = r1.numerador * r2.denominador +
        r2.numerador * r1.denominador;
    r.denominador = r1.denominador * r2.denominador;

    reduz(r);

    assert(r.denominador != 0 and mdc(r.numerador, r.denominador) == 1);

    return r;
}
```

onde `reduz()` é um procedimento para reduzir a fracção que representa o racional, i.e., uma adaptação do procedimento `reduzFracção()`.

O programa pode agora ser reescrito ser na íntegra para usar a nova classe C++, devendo-se ter o cuidado de colocar a definição da classe C++ `Racional` antes da sua primeira utilização no programa. Pode-se aproveitar para alterar os nomes das rotinas, onde o sufixo `Fracção` se torna desnecessário, dado o tipo dos respectivos parâmetros:

```
#include <iostream>
#include <cassert>

using namespace std;

/** Devolve o máximo divisor comum dos inteiros passados como argumento.
    @pre m = m ∧ n = n.
    @post mdc = { mdc(m, n)   m ≠ 0 ∨ n ≠ 0
                {     1       m = 0 ∧ n = 0  . */
int mdc(int m, int n)
{
    ...
}
/** Representa números racionais. */
class Racional {
  public: // Isto é magia (por enquanto).
    int numerador;
    int denominador;
};

/** Reduz a fracção que representa o racional recebido como argumento.
    @pre r.denominador ≠ 0 ∧ r = r.
    @post r.denominador ≠ 0∧mdc(r.numerador, r.denominador) = 1∧r = r. */
void reduz(Racional& r)
{
    assert(r.denominador != 0);

    int k = mdc(r.numerador, r.denominador);

    r.numerador /= k;
    r.denominador /= k;

    assert(r.denominador != 0 and mdc(r.numerador, r.denominador) == 1);
}

/** Lê do teclado um racional, na forma de dois inteiros sucessivos.
    @pre r = r.
    @post Se cin e cin tem dois inteiros n′ e d′ disponíveis para leitura, com d′ ≠
0, então
```

$0 < \mathtt{r.denominador} \wedge \mathrm{mdc}(\mathtt{r.numerador}, \mathtt{r.denominador}) = 1 \wedge$
$\mathtt{r} = \frac{n'}{d'} \wedge \mathtt{cin}$,
senão
$\mathtt{r} = r \wedge \neg\mathtt{cin}$. */

```
void lê(Racional& r)
{
    int n, d;

    if(cin >> n >> d)
        if(d == 0)
            cin.setstate(ios_base::failbit);
        else {
            r.numerador = d < 0 ? -n : n;
            r.denominador = d < 0 ? -d : d;

            reduz(r);

            assert(0 < r.denominador and mdc(r.numerador, r. denominador) == 1 and
                   r.numerador * d == n * r.denominador and cin);

            return;
        }

    assert(not cin);
}
```

/** Devolve a soma de dois racionais.
@pre $\mathtt{r1.denominador} \neq 0 \wedge \mathtt{r2.denominador} \neq 0$.
@post $\mathtt{somaDe} = \mathtt{r1} + \mathtt{r2} \wedge$
$\mathtt{somaDe.denominador} \neq$
$0 \wedge \mathrm{mdc}(\mathtt{somaDe.numerador}, \mathtt{somaDe.denominador}) = 1$. */

```
Racional somaDe(Racional const r1, Racional const r2)
{
    assert(r1.denominador != 0 and r2.denominador != 0);

    Racional r;

    r.numerador = r1.numerador * r2.denominador +
        r2.numerador * r1.denominador;
    r.denominador = r1.denominador * r2.denominador;

    reduz(r);

    assert(r.denominador != 0 and mdc(r.numerador, r.denominador) == 1);

    return r;
```

```
}

/** Escreve um racional no ecrã no formato de uma fracção.
    @pre V.
    @post ¬cout∨ cout contém n/d (ou simplesmente n se d = 1)
          sendo n/d a fracção canónica correspondente ao racional r. */
void escreve(Racional const r)
{
    cout << r.numerador;
    if(r.denominador != 1)
        cout << '/' << r.denominador;
}

int main()
{
    // Ler fracções:
    cout << "Introduza duas fracções (numerador denominador): ";
    Racional r1, r2;
    lê(r1);
    lê(r2);

    if(not cin) {
        cerr << "Opps!  A leitura das fracções dos racionais falhou!" << endl;
        return 1;
    }

    // Calcular racional soma:
    Racional r = somaDe(r1, r2);

    // Escrever resultado:
    cout << "A soma de ";
    escreve(r1);
    cout << " com ";
    escreve(r2);
    cout << " é ";
    escreve(r);
    cout << '.' << endl;
}
```

Ao escrever este pedaço de código o programador assumiu dois papeis: produtor e consumidor. Quando definiu a classe C++ `Racional` e a função `somaDe()`, que opera sobre variáveis dessa classe C++, fez o papel de produtor. Quando escreveu a função `main()`, assumiu o papel de consumidor.

7.3.4 Encapsulamento e categorias de acesso

O leitor mais atento terá reparado que o código acima tem pelo menos um problema: a classe `Racional` não tem qualquer mecanismo que impeça o programador de colocar 0 (zero) no denominador de uma fracção:

```
Racional r1;
r.numerador = 6;
r.denominador = 0;
```

ou

```
Racional r1 = {6, 0};
```

Isto é claramente indesejável, e tem como origem o facto do produtor ter tornado públicos os membros `numerador` e `denominador` da classe: é esse o significado do especificador de acesso `public`. De facto, os membros de uma classe podem pertencer a uma de três categorias de acesso: público, protegido e privado. Para já apenas se descreverão a primeira e a última.

Membros públicos, introduzidos pelo especificador de acesso `public:`, são acessíveis sem qualquer restrição. Membros privados, introduzidos pelo especificador de acesso `private:`, são acessíveis apenas por membros da mesma classe (ou, alternativamente, por funções "amigas" da classe, que serão vistas mais tarde). Fazendo uma analogia de uma classe com um clube, dir-se-ia que há certas partes de um clube que estão abertas ao público e outras que estão à disposição apenas dos seus membros.

O consumidor de um relógio ou de um micro-ondas assume que não precisa de conhecer o funcionamento interno desses aparelhos, podendo recorrer apenas a uma interface. Assim, o produtor desses aparelhos normalmente esconde o seu mecanismo numa caixa, deixando no exterior apenas a interface necessária para o consumidor. Também o produtor da classe C++ `Racional` deveria ter escondido os pormenores de implementação da classe C++ do consumidor final.

Podem-se resumir estas ideias num princípio básico da programação:

Princípio do encapsulamento: O produtor deve esconder do consumidor final tudo o que puder ser escondido. I.e., os pormenores de implementação devem ser escondidos, devendo-se fornecer interfaces limpas e simples para a manipulação das entidades fabricadas (aparelhos de cozinha, relógios, rotinas C++, classes C++, etc.).

Isso consegue-se, no caso das classes C++, usando o especificador de acesso `private:` para esconder os membros da classe:

```
/** Representa números racionais. */
class Racional {
  private:
    int numerador;
    int denominador;
};
```

Ao se classificar os membros `numerador` e `denominador` como privados não se impede o programador consumidor de, usando mecanismos mais ou menos obscuros e perversos, aceder ao seu valor. O facto de um membro ser privado não coloca barreiras muito fortes quanto ao seu acesso. Pode-se dizer que funciona como um aviso, esse sim forte, de que o programador consumidor não deve aceder a eles, para seu próprio bem (o produtor poderia, por exemplo, decidir alterar os nomes dos membros para `n` e `d`, com isso invalidando código que fizesse uso directo dos membros da classe). O compilador encarrega-se de gerar erros de compilação por cada acesso ilegal a membros privados de uma classe.

Assim, é claro que os membros privados de uma classe C++ fazem parte da sua implementação, enquanto os membros públicos fazem parte da sua interface.

Tornados os atributos da classe privados, torna-se impossível no procedimento `lê()` atribuir valores directamente aos seus membros. Da mesma forma, todas as outras rotinas deixam de poder aceder aos atributos da classe. A inicialização típica dos agregados, por exemplo

```
Racional r1 = {6, 9};
```

também deixa de ser possível.

Que fazer?

7.3.5 Rotinas membro: operações e métodos

Uma vez que a membros privados têm acesso quaisquer outros membros da classe, a solução passa por tornar as rotinas existentes membros da classe C++ `Racional`. Começar-se-á por tornar o procedimento `escreve()` membro da classe, i.e., por transformá-lo de simples rotina em operação do TAD em concretização:

```
...

/** Representa números racionais. */
class Racional {
  public:
    /** Escreve o racional no ecrã no formato de uma fracção.
        @pre *this = r.
        @post *this = r∧(¬cout∨ cout contém n/d (ou simplesmente n se d = 1)
              sendo n/d a fracção canónica correspondente ao racional *this). */
    void escreve(); // Declaração da rotina membro: operação.

  private:
    int numerador;
    int denominador;
};

// Definição da rotina membro: método.
void Racional::escreve()
```

```
{
    cout << numerador;
    if(denominador != 1)
        cout << '/' << denominador;
}

...
```

São de notar quatro pontos importantes:

1. Para o consumidor da classe C++ poder invocar a nova operação, é necessário que esta seja pública. Daí o especificador de acesso `public:`, que coloca a nova operação `escreve()` na interface da classe C++.

2. Qualquer operação ou rotina membro de uma classe C++ tem de ser declarada dentro da definição dessa classe e definida fora ou, alternativamente, definida (e portanto também declarada) dentro da definição da classe. Recorda-se que à implementação de uma operação se chama método, e que por isso todos os métodos fazem parte da implementação de uma classe C++[6].

3. A operação `escreve()` foi declarada sem qualquer parâmetro.

4. Há um pormenor na definição do método `escreve()` que é novo: o nome do método é precedido de `Racional::`. Esta notação serve para indicar que `escreve()` é um método correspondente a uma operação da classe `Racional`, e não uma rotina vulgar.

Onde irá a operação `Racional::escreve()` buscar o racional a imprimir? De onde vêem as variáveis `numerador` e `denominador` usadas no corpo do método `Racional::escreve()`?

Em primeiro lugar, recorde-se que o acesso aos membros de uma classe se faz usando o operador de selecção de membro. Ou seja,

```
instância.nome_do_membro
```

em que *`instância`* é uma qualquer instância da classe em causa. Esta notação é tão válida para atributos como para operações, pelo que a instrução para escrever a variável `r` no ecrã, no programa em desenvolvimento, deve passar a ser:

```
r.escreve();
```

O que acontece é que instância através da qual a operação `Racional::escreve()` é invocada está explícita na própria invocação, mas está implícita durante a execução do respectivo método! Mais, essa instância que está implícita durante a execução pode ser modificada pelo

[6]Em capítulos posteriores se verá que as classes propriamente ditas podem ter mais do que um método associado a cada operação.

método, pelo menos se for uma variável. Tudo funciona como se a instância usada para invocar a operação fosse passada automaticamente por referência.

Durante a execução do método `Racional::escreve()`, `numerador` e `denominador` referem-se aos atributos da instância através da qual a respectiva operação foi invocada. Assim, quando se adaptar o final do programa em desenvolvimento para

```
int main()
{
    ...

    // Escrever resultado:
    ...
    r1.escreve();
    ...
    r2.escreve();
    ...
    r.escreve();
    ...
}
```

durante a execução do método `Racional::escreve()` as variáveis `numerador` e `denominador` referir-se-ão sucessivamente aos correspondentes atributos de `r1`, `r2`, e `r`. À instância que está implícita durante a execução de um método chama-se naturalmente *instância implícita* (ou *variável implícita* se for uma variável, ou *constante implícita* se for uma constante), pelo que no exemplo anterior a instância implícita durante a execução do método começa por ser `r1`, depois é `r2` e finalmente é `r`.

É possível explicitar a instância implícita durante a execução de um método da classe, ou seja, a instância através da qual a respectiva operação foi invocada. Para isso usa-se a construção `*this`[7]. Esta construção usou-se na documentação da operação `escreve()`, nomeadamente no seu contrato, para deixar claro que a invocação da operação não afecta a instância implícita. Mais tarde se verá uma forma mais elegante de garantir a constância da instância implícita durante a execução de um método, i.e, uma forma de garantir que a instância implícita é tratada como uma constante implícita, mesmo que na realidade seja uma variável.

Resolvemos o problema do acesso aos atributos privados para o procedimento `escreve()`, transformando-o em procedimento membro da classe C++. É necessário fazer o mesmo para todas as outras rotinas que acedem directamente aos atributos:

```
#include <iostream>
#include <cassert>

using namespace std;

/** Devolve o máximo divisor comum dos inteiros passados como argumento.
```

[7] O significado do operador `*` ficará claro em capítulos posteriores.

```
    @pre m = m ∧ n = n.
    @post mdc = { mdc(m, n)   m ≠ 0 ∨ n ≠ 0
                { 1           m = 0 ∧ n = 0 .  */
int mdc(int m, int n)
{
    ...
}
/** Representa números racionais. */
class Racional {
  public:
    /** Escreve o racional no ecrã no formato de uma fracção.
        @pre *this = r.
        @post *this = r∧(¬cout∨ cout contém n/d (ou simplesmente n se d = 1)
              sendo n/d a fracção canónica correspondente ao racional *this). */
    void escreve();

    /** Devolve a soma com o racional recebido como argumento.
        @pre denominador ≠ 0 ∧ r2.denominador ≠ 0 ∧ *this = r.
        @post *this = r ∧ somaCom = *this + r2 ∧ denominador ≠ 0∧
             somaCom.denominador ≠
0 ∧ mdc(somaCom.numerador, somaCom.denominador) = 1. */
    Racional somaCom(Racional const r2);

    /** Lê do teclado um novo valor para o racional, na forma de dois inteiros sucessi-
vos.
        @pre *this = r.
        @post Se cin e cin tem dois inteiros n' e d' disponíveis para lei-
tura, com d' ≠ 0, então
                 0 < denominador ∧ mdc(numerador, denominador) = 1∧
                 *this = n'/d' ∧ cin,
             senão
                 *this = r ∧ ¬cin. */
    void lê();

  private:
    int numerador;
    int denominador;
    /** Reduz a fracção que representa o racional.
        @pre denominador ≠ 0 ∧ *this = r.
        @post denominador ≠ 0 ∧ mdc(numerador, denominador) = 1∧
              *this = r. */
    void reduz();
};

void Racional::escreve()
{
```

```
    cout << numerador;
    if(denominador != 1)
        cout << '/' << denominador;
}

Racional Racional::somaCom(Racional const r2)
{
    assert(denominador != 0 and r2.denominador != 0);

    Racional r;
    r.numerador = numerador * r2.denominador +
        r2.numerador * denominador;
    r.denominador = denominador * r2.denominador;

    r.reduz();

    assert(denominador != 0 and
           r.denominador != 0 and mdc(r.numerador, r.denominador) == 1);

    return r;
}

void Racional::lê()
{
    int n, d;

    if(cin >> n >> d)
        if(d == 0)
            cin.setstate(ios_base::failbit);
        else {
            numerador = d < 0 ? -n : n;
            denominador = d < 0 ? -d : d;

            reduz();

            assert(0 < denominador and mdc(numerador, denominador) == 1 and
                   numerador * d == n * denominador and cin);

            return;
        }

    assert(not cin);
}

void Racional::reduz()
{
```

```
    assert(denominador != 0);

    int k = mdc(numerador, denominador);

    numerador /= k;
    denominador /= k;

    assert(denominador != 0 and mdc(numerador, denominador) == 1);
}

int main()
{
    // Ler fracções:
    cout << "Introduza duas fracções (numerador denominador): ";
    Racional r1, r2;
    r1.lê();
    r2.lê();

    if(not cin) {
        cerr << "Opps!  A leitura dos racionais falhou!" << endl;
        return 1;
    }

    // Calcular racional soma:
    Racional r = r1.somaCom(r2);

    // Escrever resultado:
    cout << "A soma de ";
    r1.escreve();
    cout << " com ";
    r2.escreve();
    cout << " é ";
    r.escreve();
    cout << '.' << endl;
}
```

Na operação `Racional::somaCom()`, soma-se a instância implícita com o argumento passado à operação. No programa acima, por exemplo, a variável `r1` da função `main()` funciona como instância implícita durante a execução do método correspondente à operação `Racional::somaCom()` e `r2` funciona como argumento.

O procedimento `reduz()` foi transformado em operação *privada* da classe C++ que representa o TAD em desenvolvimento. Tomou-se tal decisão por não haver qualquer necessidade de o consumidor do TAD se preocupar directamente com a representação em fracção dos racionais. O consumidor do TAD limita-se a preocupar-se com o comportamento exterior do tipo. Pelo contrário, para o produtor da classe C++ a representação dos racionais é fundamental, pois é ele que tem de garantir que todas as operações cumprem o respectivo contrato.

A invocação da operação `Racional::reduz()` no método `Racional::lê()` é feita sem necessidade de usar a sintaxe usual para a invocação de operações, i.e., sem indicar explicitamente a instância através da qual (e para a qual) essa invocação é feita. Isso deve-se ao facto de se pretender fazer a invocação para a instância implícita. Seria possível explicitar essa instância,

```
(*this).reduz();
```

tal como de resto poderia ter sido feito para os atributos,

```
(*this).numerador = n;
```

mas isso conduziria apenas a código mais denso. Note-se que os parênteses em volta de `*this` são fundamentais, pois o operador de selecção de membro tem maior precedência que o operador unário `*` (ou seja, o operador "conteúdo de", a estudar mais tarde).

É também importante perceber-se que não existe qualquer vantagem em tornar a função `mdc()` membro na nova classe C++. Em primeiro lugar, pode haver necessidade de calcular o máximo divisor comum de outros inteiros que não o numerador e o denominador. Aliás, tal necessidade surgirá ainda durante este capítulo. Em segundo lugar porque o cálculo do máximo divisor comum poderá ser necessário em contextos que nada tenham a ver com números racionais.

Finalmente, a notação usada para calcular a soma

```
Racional r = r1.somaCom(r2);
```

é horrenda, sem dúvida alguma. Numa secção posterior se verá como sobrecarregar o operador + de modo a permitir escrever

```
Racional r = r1 + r2;
```

7.4 Classes C++ como módulos

Das discussões anteriores, nomeadamente sobre o princípio do encapsulamento e as categorias de acesso dos membros de uma classe, torna-se claro que as classes C++ são uma unidade de modularização. De facto, assim é. Aliás, as classes são a unidade de modularização por excelência na linguagem C++ e na programação baseada em (e orientada para) objectos.

Como qualquer módulo que se preze, as classes C++ distinguem claramente interface e implementação. A interface de uma classe C++ corresponde aos seus membros públicos. Usualmente a interface de uma classe C++ consiste num conjunto de operações e tipos públicos. A implementação de uma classe C++ consiste, pelo contrário, nos membros privados e na definição das respectivas operações, i.e., nos métodos da classe. Normalmente a implementação de uma classe C++ contém os atributos da classe, particularmente as variáveis membro, e operações utilitárias, necessárias apenas para o programador produtor da classe.

É de toda a conveniência que os atributos de uma classe C++ (e em especial as suas variáveis membro) sejam privados. Só dessa forma se garante que um consumidor da classe não pode, perversa ou acidentalmente, alterar os valores dos atributos de tal forma que um instância da classe C++ deixe de estar num estado válido. Este assunto será retomado com maior pormenor mais abaixo, quando se falar da chamada *CIC* (Condição Invariante de Classe).

As classes C++ possuem também um "manual de utilização", correspondente ao contrato entre o seu produtor e os seus consumidores. Esse contrato é normalmente expresso através de um comentário de documentação para a classe em si e dos comentários de documentação de todas os seus membros públicos.

7.4.1 Construtores

Suponha-se o código

```
Racional a;
a.numerador = 1;
a.denominador = 3;
```

ou

```
Racional a = {1, 3};
```

A partir do momento em que os atributos da classe passaram a ser privados ambas as formas de inicialização[8] deixaram de ser possíveis. Como resolver este problema?

Para os tipos básicos da linguagem, a inicialização faz-se usando uma de duas possíveis sintaxes:

```
int a = 10;
```

ou

```
int a(10);
```

Se realmente se pretende que a nova classe C++ `Racional` represente um tipo de primeira categoria, é importante fornecer uma forma de os racionais poderem se inicializados de uma forma semelhante. Por exemplo,

```
Racional r(1, 3); // Pretende-se que inicialize r com o racional 1/3.
```

[8]Na realidade no primeiro troço de código não se faz uma inicialização. As operações de atribuição alteram os valores dos atributos já inicializados (ou melhor, a atributos deixados por inicializar pelas regras absurdas importadas da linguagem C, e por isso contendo lixo).

ou mesmo

```
Racional r = 2; // Pretende-se que inicialize r com o racional 2/1.
Racional r(3);  // Pretende-se que inicialize r com o racional 3/1.
```

Por outro lado, deveria ser possível evitar o comportamento dos tipos básicos do C++ e eliminar completamente as instâncias por inicializar, fazendo com que à falta de uma inicialização explícita, os novos racionais fossem inicializados com o valor zero, (0, representado pela fracção $\frac{0}{1}$). Ou seja,

```
Racional r;
Racional r(0);
Racional r(0, 1);
```

deveriam ser instruções equivalentes.

Finalmente, deveria haver alguma forma de evitar a inicialização de racionais com valores impossíveis, nomeadamente com denominador nulo. I.e., a instrução

```
Racional r(3, 0);
```

deveria de alguma forma resultar num erro.

Quando se constrói uma instância de uma classe C++, é chamado um procedimento especial que se chama construtor da classe C++. Esse construtor é fornecido implicitamente pela linguagem e é um construtor por omissão, i.e., é um construtor que se pode invocar sem lhe passar quaisquer argumento[9]. O construtor por omissão fornecido implicitamente constrói cada um dos atributos da classe invocando o respectivo construtor por omissão. Neste caso, como os atributos são de tipos básicos da linguagem, não são inicializados durante a sua construção, ao contrário do que seria desejável, contendo por isso lixo. Para evitar o problema, deve ser o programador produtor a declarar explicitamente um ou mais construtores (e, já agora, defini-los com o comportamento pretendido), pois nesse caso o construtor por omissão deixa de ser fornecido implicitamente pela linguagem.

Uma vez que se pretende que os racionais sejam inicializados por omissão com zero, tem de se fornecer um construtor por omissão explicitamente que tenha esse efeito:

```
/** Representa números racionais. */
class Racional {
  public:
    /** Constrói racional com valor zero. Construtor por omissão.
        @pre V.
        @post *this = 0∧
              0 < denominador ∧ mdc(numerador, denominador) = 1. */
```

[9]Nem sempre a linguagem fornece um construtor por omissão implicitamente. Isso acontece quando a classe tem atributos que são constantes, referências, ou que não têm construtores por omissão, entre outros casos.

```
        Racional();

        ...

      private:

        ...

    };

    Racional::Racional()
        : numerador(0), denominador(1)
    {
        assert(0 < denominador and mdc(numerador, denominador) == 1);
    }
```

Os construtores são operações de uma classe C++, mas são muito especiais, quer por por razões semânticas, quer por razões sintácticas. Do ponto de vista semântico, o que os distingue dos outros operadores é o facto de não serem invocados através de variáveis da classe preexistentes. Pelo contrário, os construtores são invocados justamente para construir uma nova variável. Do ponto de vista sintáctico os construtores têm algumas particularidades. A primeira é que têm o mesmo nome que a própria classe. Os construtores são como que funções membro, pois têm como resultado uma nova variável da classe a que pertencem. No entanto, não só não se pode indicar qualquer tipo de devolução no seu cabeçalho, como no seu corpo não é permitido devolver qualquer valor, pois este age sobre uma instância implícita em construção.

Quando uma instância de uma classe é construída, por exemplo devido à definição de uma variável dessa classe, é invocado o construtor da classe compatível com os argumentos usados na inicialização. I.e., é possível que uma classe tenha vários construtores sobrecarregados, facto de que se tirará partido em breve. Os argumentos são passados aos construtores colocando-os entre parênteses na definição das instâncias. Por exemplo, as instruções

```
    Racional r;
    Racional r(0);
    Racional r(0, 1);
```

deveriam todas construir uma nova variável racional com o valor zero, muito embora para já só a primeira instrução seja válida, pois a classe ainda não possui construtores com argumentos.

Note-se que as instruções

```
    Racional r;
    Racional r();
```

não são equivalentes! Esta irregularidade sintáctica do C++ deve-se ao facto de a segunda instrução ter uma interpretação alternativa: a de declarar uma função `r` que não tem parâmetros e devolve um valor `Racional`. Face a esta ambiguidade de interpretação, a linguagem optou por dar preferência à declaração de uma função...

Aquando da construção de uma instância de uma classe C++, um dos seus construtores é invocado. Antes mesmo de o seu corpo ser executado, no entanto, todos os atributos da classe são construídos. Se se pretender passar argumentos aos construtores dos atributos, então é obrigatória a utilização de listas de utilizadores, que se colocam na definição do construtor, entre o cabeçalho e o corpo, após o símbolo dois-pontos (`:`). Esta lista consiste no nome dos atributos pela mesma ordem pela qual estão definidos na classe C++, seguido cada um dos argumentos a passar ao respectivo construtor colocados entre parênteses. No caso da classe C++ `Racional`, pretende-se inicializar os atributos `numerador` e `denominador` respectivamente com os valores 0 e 1, pelo que a lista de inicializadores é

```
Racional::Racional()
    : numerador(0), denominador(1)
{
    ...
}
```

Uma vez que se pretendem mais duas formas de inicialização dos racionais, é necessário fornecer dois construtores adicionais. O primeiro constrói um racional a partir de um único inteiro, o que é quase tão simples como construir um racional com o valor zero. O segundo é um pouco mais complicado, pois, construindo um racional a partir do numerador e denominador de uma fracção, precisa de receber garantidamente um denominador não-nulo e tem de ter o cuidado de garantir que os seus atributos, `numerador` e `denominador`, estão no formato canónico das fracções:

```
/** Representa números racionais. */
class Racional {
  public:
    /** Constrói racional com valor zero. Construtor por omissão.
        @pre V.
        @post *this = 0∧
              0 < denominador ∧ mdc(numerador, denominador) = 1. */
    Racional();

    /** Constrói racional com valor inteiro.
        @pre V.
        @post *this = n∧
              0 < denominador ∧ mdc(numerador, denominador) = 1. */
    Racional(int const n);
    /** Constrói racional correspondente a n/d.
        @pre d ≠ 0.
        @post *this = n/d ∧
              0 < denominador ∧ mdc(numerador, denominador) = 1. */
    Racional(int const n, int const d);
```

```
    ...

  private:

    ...

};

Racional::Racional()
    : numerador(0), denominador(1)
{
    assert(0 < denominador and mdc(numerador, denominador) == 1 and
           numerador == 0);
}

Racional::Racional(int const n)
    : numerador(n), denominador(1)
{
    assert(0 < denominador and mdc(numerador, denominador) == 1 and
           numerador == n * denominador);
}

Racional::Racional(int const n, int const d)
    : numerador(d < 0 ? -n : n),
      denominador(d < 0 ? -d : d)
{
    assert(d != 0);

    reduz();

    assert(0 < denominador and mdc(numerador, denominador) == 1 and
           numerador * d == n * denominador);
}

...
```

Uma observação atenta dos três construtores revela que os dois primeiros são quase iguais, enquanto o terceiro é mais complexo, pois necessita verificar o sinal do denominador recebido no parâmetro `d` e, além disso, tem de se preocupar com a redução dos termos da fracção. Assim, surge naturalmente a ideia de condensar os dois primeiros construtores num único, não se fazendo o mesmo relativamente ao último construtor (à custa do qual poderiam ser obtidos os dois primeiros), por razões de eficiência.

A condensação dos dois primeiros construtores num único faz-se recorrendo aos parâmetros com argumentos por omissão, vistos na Secção 3.6:

```
/** Representa números racionais. */
class Racional {
  public:
     /** Constrói racional com valor inteiro. Construtor por omissão.
          @pre V.
          @post *this = n∧
               0 < denominador ∧ mdc(numerador, denominador) = 1. */
    Racional(int const n = 0);
    /** Constrói racional correspondente a n/d.
          @pre d ≠ 0.
          @post *this = n/d ∧
               0 < denominador ∧ mdc(numerador, denominador) = 1. */
    Racional(int const n, int const d);

    ...

  private:

    ...

};

Racional::Racional(int const n)
    : numerador(n), denominador(1)
{
    assert(0 < denominador and mdc(numerador, denominador) == 1 and
           numerador == n * denominador);
}

Racional::Racional(int const n, int const d)
    : numerador(d < 0 ? -n : n),
      denominador(d < 0 ? -d : d)
{
    assert(d != 0);

    reduz();

    assert(0 < denominador and mdc(numerador, denominador) == 1 and
           numerador * d == n * denominador);
}
...
```

A Figura 7.4 mostra a notação usada para representar a classe C++ `Racional` desenvolvida até aqui.

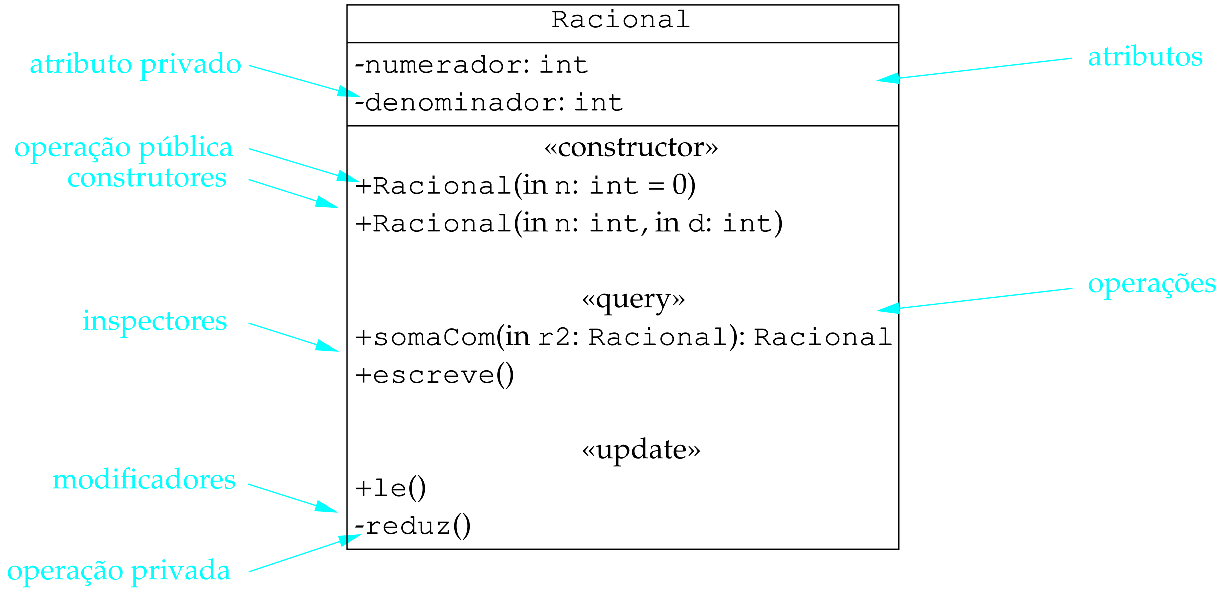


Figura 7.4: A classe C++ `Racional` agora também com operações. Note-se a utilização de + e - para indicar a características públicas e privadas da classe C++, respectivamente, e o termo "in" para indicar que o argumento da operação `Racional::somaCom()` é passado por valor, ou seja, apenas "para dentro" da operação.

7.4.2 Construtores por cópia

Viu-se que a linguagem fornece implicitamente um construtor por omissão para as classes, excepto quando estas declaram algum construtor explicitamente. Algo de semelhante se passa relativamente aos chamados *construtores por cópia*. Estes construtores são usados para construir uma instância de uma classe à custa de outra instância da mesma classe.

A linguagem fornece também implicitamente um construtor por cópia, desde que tal seja possível, para todas as classes C++ que não declarem explicitamente um construtor por cópia. O construtor por cópia fornecido implicitamente limita-se a invocar os construtores por cópia para construir os atributos da instância em construção à custa dos mesmos atributos na instância original, sendo a invocação realizada por ordem de definição dos atributos na definição da classe.

É possível, e muitas vezes desejável, declarar ou mesmo definir explicitamente um construtor por cópia para as classes. Este assunto será tratado com pormenor num capítulo posterior.

7.4.3 Condição invariante de classe

Na maior parte das classes C++ que concretizam um TAD, os atributos só estão num estado aceitável se verificarem um conjunto de restrições, expressos normalmente na forma de uma condição a que se dá o nome de *condição invariante de classe* ou CIC. A classe dos racionais possui uma condição invariante de classe que passa por exigir que os atributos `numerador` e `denominador` sejam o numerador e o denominador da fracção canónica representativa do racional correspondente, i.e.,

$$CIC \equiv 0 < \texttt{denominador} \wedge \mathrm{mdc}(\texttt{numerador}, \texttt{denominador}).$$

A vantagem da definição de uma condição invariante de classe é que todos os métodos correspondentes a operações públicas bem como todas as rotinas amigas da classe C++ (que fazem parte da interface da classe com o consumidor, Secção 7.15) poderem admitir que os atributos das variáveis da classe C++ com que trabalham verificam inicialmente a condição, o que normalmente os simplifica bastante. I.e., a condição invariante de classe pode ser vista como parte da pré-condição quer de métodos correspondentes a operações públicas, quer de rotinas amigas da classe C++. Claro que, para serem "bem comportadas", as rotinas, membro e não membro, também devem garantir que a condição se verifica para todas as variáveis da classe C++ criadas ou alteradas por essas rotinas. Ou seja, a condição invariante de classe para cada instância da classe criada ou alterada pelas mesmas rotinas pode ser vista também como parte da sua condição objectivo.

Tal como sucedia nos ciclos, em que durante a execução do passo a condição invariante muitas vezes não se verificava, embora se verificasse garantidamente antes e após o passo, também a condição invariante de classe pode não se verificar durante a execução dos métodos públicos ou das rotinas amigas da classe C++ em causa, embora se verifique garantidamente no seu início e no seu final. Durante os períodos em que a condição invariante de classe não é verdadeira, pode ser conveniente invocar alguma rotina auxiliar, que portanto terá de lidar com instâncias que não verificam a condição invariante de classe e que poderá também não garantir que a mesma condição se verifica para as instâncias por si criadas ou alteradas. Essas rotinas "mal comportadas" devem ser privadas, de modo a evitar utilizações erróneas por parte do consumidor final da classe C++ que coloquem alguma instância num estado inválido.

A definição de uma condição invariante de classe e a sua imposição à entrada e saída dos métodos públicos e de rotinas amigas de uma classe C++ não passa de um esforço inútil se as suas variáveis membro forem públicas, i.e., se o seu *estado* for alterável do exterior. Se o forem, o consumidor da classe C++ pode alterar o estado de uma variável da classe, por engano ou maliciosamente, invalidando a condição invariante de classe, com consequências potencialmente dramáticas no comportamento da classe C++ e no programa no seu todo. Essas consequências são normalmente graves, porque as rotinas que lidam com as variáveis membro da classe assumem que estas verificam a condição invariante de classe, não fazendo quaisquer garantias acerca do seu funcionamento quando ela não se verifica.

De todas as operações de uma classe C++, as mais importantes são porventura as operações construtoras[10]. São estas que garantem que as instâncias são criadas verificando imediatamente a condição invariante de classe. A sua importância pode ser vista na classe `Racional`, em que os construtores garantem, desde que as respectivas pré-condições sejam respeitadas, que a condição invariante da classe se verifica para as instâncias construídas.

Finalmente, é de notar que algumas classes C++ não têm condição invariante de classe. Tais classes C++ não são normalmente concretizações de nenhum TAD, sendo meros agregados de

[10] Note-se que construtor e operação construtora não significam forçosamente a mesma coisa. A noção de operação construtora é mais geral, e refere-se a qualquer operação que construa novas variáveis da classe C++. É claro que os construtores são operações construtoras, mas uma função membro pública que devolva um valor da classe C++ em causa também o é.

informação. É o caso, por exemplo, de um agregado que guarde nome e morada de utentes de um serviço qualquer. Essas classes C++ têm normalmente todas as suas variáveis membro públicas, e por isso usam normalmente a palavra-chave `struct` em vez de `class`. Note-se que estas palavras chave são quase equivalentes, pelo que a escolha de `class` ou `struct` é meramente convencional, escolhendo-se `class` para classes C++ que sejam concretizações de TAD ou classes propriamente ditas, e `struct` para classes C++ que sejam meros agregados de informação. A única diferença entre as palavras chave `struct` e `class` é que, com a primeira, todos os membros são públicos por omissão, enquanto com a segunda todos os membros são privados por omissão.

7.4.4 Porquê o formato canónico das fracções?

Qual a vantagem de manter todas as fracções que representam os racionais no seu formato canónico? I.e., qual a vantagem de impor

$$0 < \texttt{denominador} \wedge \mathrm{mdc}(\texttt{numerador}, \texttt{denominador})$$

como condição invariante de classe C++?

A verdade é que esta condição poderia ser consideravelmente relaxada: para o programador consumidor, a representação interna dos racionais é irrelevante, muito embora ele espere que a operação `escreve()` resulte numa representação canónica dos racionais. Logo, o problema poderia ser resolvido alterando apenas o método `escreve()`, de modo a reduzir a fracção, deixando o restante código de se preocupar com a questão. Ou seja, poder-se-ia relaxar a condição invariante de classe para

$$\texttt{denominador} \neq 0.$$

No entanto, a escolha de uma condição invariante de classe mais forte trará algumas vantagens.

A primeira vantagem tem a ver com a unicidade de representação garantida pela condição invariante de classe escolhida: a cada racional corresponde uma e uma só representação na forma de uma fracção canónica. Dessa forma é muito fácil comparar dois racionais: dois racionais são iguais se e só se as correspondentes fracções canónicas tiverem o mesmo numerador e o mesmo denominador.

A segunda vantagem tem a ver com as limitações dos tipos básicos do C++.

Sendo os valores do tipo `int` limitados em C++, como se viu no Capítulo 2, a utilização de uma representação em fracções não-canónicas põe alguns problemas graves de implementação. O primeiro tem a ver com a facilidade com que permite realizar algumas operações. Por exemplo, é muito fácil verificar a igualdade de dois racionais comparando simplesmente os seus numeradores e denominadores, coisa que só é possível fazer directamente se se garantir que as fracções que os representam estão no formato canónico. O segundo problema tem a ver com as limitações dos inteiros. Suponha-se o seguinte código:

```
int main()
{
    Racional x(50000, 50000), y(1, 50000);
    Racional z = x.soma(y);
    z.escreve();
    cout << endl;
}
```

No ecrã deveria aparecer

```
1/50000
```

Não se usando uma representação em fracções canónicas, ao se calcular o denominador do resultado, i.e., ao se multiplicar os dois denominadores, obtém-se $50000 \times 50000 = 2500000000$. Em máquinas em que os `int` têm 32 *bits*, esse valor não é representável, pelo que se obtém um valor errado (em Linux i386 obtém-se -1794967296), apesar de a fracção resultado ser perfeitamente representável! Este problema pode ser mitigado se se trabalhar sempre com fracções no formato canónico.

Mesmo assim, o problema não é totalmente resolvido. Suponha-se o seguinte código:

```
int main()
{
    Racional x(1, 50000), y(1, 50000);
    Racional z = x.soma(y);
    z.escreve();
    cout << endl;
}
```

No ecrã deveria aparecer

```
1/25000
```

mas ocorre exactamente o mesmo problema que anteriormente. É pois desejável não só usar uma representação canónica para os racionais, como também tentar garantir que os resultados de cálculos intermédios são tão pequenos quanto possível. Este assunto será retomado mais tarde (Secção 7.13).

7.4.5 Explicitação da condição invariante de classe

A condição invariante de classe é útil não apenas como uma ferramenta formal que permite verificar o correcto funcionamento de, por exemplo, um método. É útil como ferramenta de detecção de erros. Da mesma forma que é conveniente explicitar pré-condições e condições objectivo das rotinas através de instruções de asserção, também o é no caso da condição invariante de classe. A intenção é detectar as violações dessa condição durante a execução do programa e abortá-lo se alguma violação for detectada[11].

A condição invariante de classe é claramente uma noção de implementação: refere-se sempre aos atributos (que se presume serem privados) de uma classe. Uma das vantagens de se estabelecer esta distinção clara entre interface e implementação está em permitir alterações substanciais na implementação sem que a interface mude. De facto, é perfeitamente possível que o programador produtor mude substancialmente a implementação de uma classe C++ sem que isso traga qualquer problema para o programador consumidor, que se limita a usar a interface da classe C++. A mudança da implementação de uma classe implica normalmente uma alteração da condição invariante de classe, mas não do comportamento externo da classe. É por isso muito importante que pré-condição e condição objectivo de cada operação/método sejam claramente factorizadas em condições que dizem respeito apenas à implementação, e que devem corresponder à condição invariante de classe, e condições que digam respeito apenas ao comportamento externo da operação. Dito por outras palavras, apesar de do ponto de vista da implementação a condição invariante de classe fazer parte da pré-condição e da condição objectivo de todas as operações/métodos, como se disse na secção anterior, é preferível "pô-la em evidência", documentando-a claramente à parte das operações e métodos, e excluindo-a da documentação/contrato de cada operação. Ou seja, a condição invariante de classe fará parte do contrato de cada *método* (ponto de vista da implementação), mas não fará parte do contrato da correspondente operação (ponto de vista externo, da interface).

Quando a condição invariante de classe é violada, de quem é a culpa? Nesta altura já não deverão subsistir dúvidas: a culpa é do programador produtor da classe:

1. Violação da condição invariante de classe: culpa do programador produtor da classe.
2. Violação da pré-condição de uma operação: culpa do programador consumidor da classe.
3. Violação da condição objectivo de uma operação: culpa do programador produtor do respectivo método.

Como explicitar a condição invariante de classe? É apenas uma questão de usar instruções de asserção e comentários de documentação apropriados. Para simplificar, é conveniente definir uma operação privada da classe, chamada convencionalmente `cumpreInvariante()`, que devolve o valor lógico $\mathcal{V}$ se a condição invariante de classe se cumprir e falso no caso contrário.

```
/** Descrição da classe Classe.
    @invariant CIC.
class Classe {
  public:

    ...
```

[11] Mais tarde se verá que, dependendo da aplicação em desenvolvimento, abortar o programa em caso de erro de programação pode ou não ser apropriado.

```
    /** Descrição da operação operação().
          @pre PC.
          @post CO. */
    tipo operação(parâmetros);

  private:

    ...

    /** Descrição da operação operação_privada().
          @pre PC.
          @post CO. */
    tipo operação_privada(parâmetros);


    /** Indica se a condição invariante de classe (CIC) se verifica.
          @pre *this = v.
          @post cumpreInvariante = CIC ∧ *this = v. */
    bool cumpreInvariante();
};

...

// Implementação da operação operação(): método.
tipo Classe::operação(parâmetros)
{
    assert(cumpreInvariante() [and v.cumpreInvariante()]...);
    assert(PC);

    ... // Implementação.

    assert(cumpreInvariante() [and v.cumpreInvariante()]...);
    assert(CO);

    return ...;
}

...

bool Classe::cumpreInvariante()
{
    return CIC.
}

// Implementação da operação operação(): método.
tipo Classe::operação_privada(parâmetros)
```

```
{
    assert(PC);

    ... // Implementação.

    assert(CO);

    return ...;
}
```

São de notar os seguintes pontos importantes:

- A condição invariante de classe foi incluída na documentação da classe, que é parte da sua interface, apesar de antes se ter dito que esta condição era essencialmente uma questão de implementação. É de facto infeliz que assim seja, mas os programas que extraem automaticamente a documentação de uma classe (e.g., Doxygen) requerem este posicionamento[12].
- A documentação das operações não inclui a condição invariante de classe, visto que esta foi "posta em evidência", ficando na documentação da classe.
- A implementação das operações, i.e., o respectivo método, inclui instruções de asserção para verificar a condição invariante de classe para todas as instâncias da classe em jogo (que incluem a instância implícita[13], parâmetros, instâncias locais ao método, etc.), quer no início do método, quer no seu final.
- As instruções de asserção para verificar a veracidade da condição invariante de classe são anteriores quer à instrução de asserção para verificar a pré-condição da operação, quer à instrução de asserção para verificar a condição objectivo da operação. Esse posicionamento é importante, pois as verificações da pré-condição e da condição objectivo podem obrigar à invocação de outras operações públicas da classe, que por sua vez verificam a condição invariante de classe: se a ordem fosse outra, o erro surgiria durante a execução dessas outras operações.
- Separaram-se as instruções de asserção relativas a pré-condições, condições objectivo e condições invariantes de classe, de modo a ser mais óbvia a razão do erro no caso de o programa abortar.
- A função membro privada `cumpreInvariante()` não tem qualquer instrução de asserção. Isso deve-se ao facto de ter sempre pré-condição $\mathcal{V}$ e de poder operar sobre variáveis implícitas que não verificam a condição invariante de classe (como é óbvio, pois serve justamente para indicar se essa condição se verifica).

[12] Parece haver aqui uma contradição. Não será toda a documentação parte da interface? A resposta é simplesmente "não". Para uma classe, podem-se gerar três tipos de documentação. A primeira diz respeito de facto à interface, e inclui todos os membros públicos: é a documentação necessária ao programador consumidor. A segunda diz respeito à categoria de acesso `protected` e deixar-se-á para mais tarde. A terceira diz respeito à implementação, e inclui os membros de todas as categorias de acesso: é a documentação necessária ao programador produtor ou, pelo menos, à "assistência técnica", i.e., aos programadores que farão a manutenção do código existente. Assim, a condição invariante de classe deveria ser parte apenas da documentação de implementação.

[13] Excepto para operações de classe.

- Os métodos privados não têm instruções de asserção para a condição invariante de classe, pois podem ser invocados por outros métodos em instantes de tempo durante os quais as instâncias da classe (instância implícita, parâmetros, etc.) não verifiquem essa condição.

Aplicando estas ideias à classe `Racional` em desenvolvimento obtém-se:

```
#include <iostream>
#include <cassert>

using namespace std;
```

`/**` Devolve o máximo divisor comum dos inteiros passados como argumento.
    @pre $\mathtt{m} = m \wedge \mathtt{n} = n$.
    @post $\mathtt{mdc} = \begin{cases} \mathrm{mdc}(m,n) & m \neq 0 \vee n \neq 0 \\ 1 & m = 0 \wedge n = 0 \end{cases}$. `*/`

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

`/**` Representa números racionais.
    @invariant $0 < \mathtt{denominador} \wedge \mathrm{mdc}(\mathtt{numerador}, \mathtt{denominador}) = 1$. `*/`

```
class Racional {
  public:
```

`/**` Constrói racional com valor inteiro. Construtor por omissão.
    @pre $\mathcal{V}$.
    @post $\mathtt{*this} = \mathtt{n}$. `*/`

```
    Racional(int const n = 0);
```

`/**` Constrói racional correspondente a n/d.
    @pre $\mathtt{d} \neq 0$.
    @post $\mathtt{*this} = \frac{\mathtt{n}}{\mathtt{d}}$. `*/`

```
    Racional(int const n, int const d);
```

`/**` Escreve o racional no ecrã no formato de uma fracção.
    @pre $\mathtt{*this} = r$.
    @post $\mathtt{*this} = r \wedge (\neg\mathtt{cout} \vee$ `cout` contém n/d (ou simplesmente n se $d = 1$) sendo $\frac{n}{d}$ a fracção canónica correspondente ao racional `*this`). `*/`

```
    void escreve(); // Declaração da rotina membro: operação.
```

`/**` Devolve a soma com o racional recebido como argumento.
    @pre $\mathtt{*this} = r$.
    @post $\mathtt{*this} = r \wedge \mathtt{somaCom} = \mathtt{*this} + \mathtt{r2}$. `*/`

```
    Racional somaCom(Racional const r2);
```

`/**` Lê do teclado um novo valor para o racional, na forma de dois inteiros sucessivos.

```
        @pre *this = r.
        @post Se cin e cin tem dois inteiros n' e d' disponíveis para leitura, com d' ≠ 0, então
                *this = n'/d' ∧ cin,
            senão
                *this = r ∧ ¬cin. */
    void lê();

  private:
    int numerador;
    int denominador;
    /** Reduz a fracção que representa o racional.
        @pre denominador ≠ 0 ∧ *this = r.
        @post denominador ≠ 0 ∧ mdc(numerador, denominador) = 1∧
              *this = r. */
    void reduz();

    /** Indica se a condição invariante de classe se verifica.
        @pre *this = r.
        @post *this = r ∧ cumpreInvariante = (0 < denominador ∧ mdc(numerador, denominador) = 1). */
    bool cumpreInvariante();
};

Racional::Racional(int const n)
    : numerador(n), denominador(1)
{

    assert(cumpreInvariante());
    assert(numerador == n * denominador);
}

Racional::Racional(int const n, int const d)
    : numerador(d < 0 ? -n : n),
      denominador(d < 0 ? -d : d)
{
    assert(d != 0);

    reduz();

    assert(cumpreInvariante());
    assert(numerador * d == n * denominador);
}

void Racional::escreve()
{
```

```
    assert(cumpreInvariante());

    cout << numerador;
    if(denominador != 1)
        cout << '/' << denominador;

    assert(cumpreInvariante());
}

Racional Racional::somaCom(Racional const r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    Racional r;

    r.numerador = numerador * r2.denominador +
        r2.numerador * denominador;
    r.denominador = denominador * r2.denominador;

    r.reduz();

    assert(cumpreInvariante() and r.cumpreInvariante());

    return r;
}

void Racional::lê()
{
    assert(cumpreInvariante());

    int n, d;

    if(cin >> n >> d)
        if(d == 0)
            cin.setstate(ios_base::failbit);
        else {
            numerador = d < 0 ? -n : n;
            denominador = d < 0 ? -d : d;

            reduz();

            assert(cumpreInvariante());
            assert(numerador * d == n * denominador and cin);

            return;
        }
```

```
    assert(cumpreInvariante());
    assert(not cin);
}

void Racional::reduz()
{
    assert(denominador != 0);

    int k = mdc(numerador, denominador);

    numerador /= k;
    denominador /= k;

    assert(denominador != 0 and mdc(numerador, denominador) == 1);
}

bool Racional::cumpreInvariante()
{
    return 0 < denominador and mdc(numerador, denominador) == 1;
}

int main()
{
    // Ler fracções:
    cout << "Introduza duas fracções (numerador denominador): ";
    Racional r1, r2;
    r1.lê();
    r2.lê();

    if(not cin) {
        cerr << "Opps!  A leitura dos racionais falhou!" << endl;
        return 1;
    }

    // Calcular racional soma:
    Racional r = r1.somaCom(r2);

    // Escrever resultado:
    cout << "A soma de ";
    r1.escreve();
    cout << " com ";
    r2.escreve();
    cout << " é ";
    r.escreve();
    cout << '.' << endl;
}
```

Note-se que o compilador se encarrega de garantir que algumas instâncias não mudam de valor durante a execução de um método. É o caso das constantes. É evidente, pois, que se essas constantes cumprem inicialmente a condição invariante de classe, também a cumprirão no final no método, pelo que se pode omitir a verificação explícita através de uma instrução de asserção, tal como se fez para o método `somaCom()`.

A Figura 7.5 mostra a notação usada para representar a condição invariante da classe C++ `Racional`, bem como a pré-condição e a condição objectivo da operação `Racional::somaCom()`.

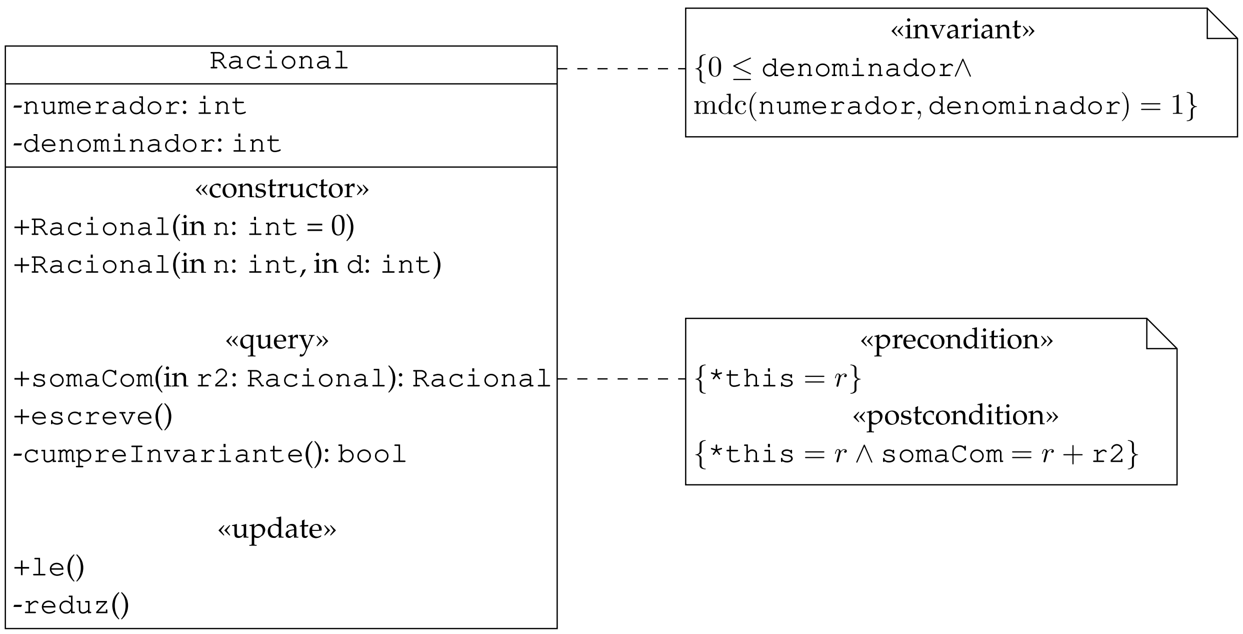


Figura 7.5: A classe C++ `Racional` agora também com condição invariante de instância, pré-condição e condição objectivo indicadas para a operação `Racional::somaCom()`.

7.5 Sobrecarga de operadores

Tal como definida, a classe C++ `Racional` obriga o consumidor a usar uma notação desagradável e pouco intuitiva para fazer operações com racionais. Como se viu, seria desejável que a função `main()`, no programa em desenvolvimento, se pudesse escrever simplesmente como:

```
int main()
{
    // Ler fracções:
    cout << "Introduza duas fracções (numerador denominador): ";
    Racional r1, r2;
    cin >> r1 >> r2;

    if(not cin) {
```

```
        cerr << "Opps!  A leitura dos racionais falhou!" << endl;
        return 1;
    }

    // Calcular racional soma:
    Racional r = r1 + r2;

    // Escrever resultado:
    cout << "A soma de " << r1 << " com " << r2
         << " é " << r << '.' << endl;
}
```

Se se pudesse escrever o programa como acima, claramente a classe `Racional`, uma vez equipada com os restantes operadores dos tipos aritméticos básicos, passaria a funcionar para o consumidor como qualquer outro tipo básico do C++: seria verdadeiramente um *tipo de primeira categoria*.

O C++ possibilita a sobrecarga dos operadores (`+`, `-`, `*`, `/`, `==`, etc.) de modo a poderem ser utilizados com TAD concretizados pelo programador na forma de classes C++. A solução para o problema passa então pela sobrecarga dos operadores do C++ de modo a terem novos significados quando aplicados ao novo tipo `Racional`, da mesma forma que se tinha visto antes relativamente aos tipos enumerados (ver Secção 6.1). Mas, ao contrário do que se fez então, agora as funções de sobrecarga têm de ser membros da classe `Racional`, de modo a poderem aceder aos seus membros privados (alternativamente poder-se-iam usar funções membro amigas da classe, Secção 7.15). Ou seja, a solução é simplesmente alterar o nome da operação `Racional::somaCom()` de `somaCom` para `operator+`:

```
...
/** Representa números racionais.
    @invariant 0 < denominador ∧ mdc(numerador, denominador) = 1. */
class Racional {
  public:

    ...

    /** Devolve a soma com o racional recebido como argumento.
        @pre *this = r.
        @post *this = r ∧ operator+ = *this + r2. */
    Racional operator+(Racional const r2);

    ...

  private:

    ...
```

```
};

...

Racional Racional::operator+(Racional const r2)
{
    ...
}

...
```

Tal como acontecia com a expressão `r1.somaCom(r2)`, a expressão `r1.operator+(r1)` invoca a operação `operator+()` da classe C++ `Racional` usando `r1` como instância (variável) implícita. Só que agora é possível escrever a mesma expressão de uma forma muito mais clara e intuitiva:

```
r1 + r2
```

De facto, sempre que se sobrecarregam operadores usando operações, o primeiro operando (que pode ser o único no caso de operadores unários, i.e., só com um operando) é sempre a instância implícita durante a execução do respectivo método, sendo os restantes operandos passados como argumento à operação.

Se `@` for um operador binário (e.g., `+`, `-`, `*`, etc.), então a sobrecarga do operador `@` pode ser feita:

- Para uma classe C++ *`Classe`*, definindo uma operação *`tipo_de_devolução`* *`Classe`*`::operator@(`*`tipo_do_segundo_operando`*`)`. Numa invocação deste operador, o primeiro operando, obrigatoriamente do tipo *`Classe`*, é usado como instância implícita e o segundo operando é passado como argumento.
- Através de uma rotina não-membro *`tipo_de_devolução`* `operator@(`*`tipo_do_primeiro_operando`*`,` *`tipo_do_segundo_operando`*`)`. Numa invocação deste operador, ambos os operandos são passados como argumentos.

A expressão `a @ b` pode portanto ser interpretada como

```
a.operator@(b)
```

ou

```
operator@(a, b)
```

consoante o operador esteja definido como membro da classe a que `a` pertence ou esteja definido como rotina normal, não-membro.

Se `@` for um operador unário (e.g., `+`, `-`, `++` prefixo, etc.), então a sobrecarga do operador `@` pode ser feita:

- Para uma classe C++ `Classe`, definindo uma operação `tipo_de_devolução Classe::operator@()`. Numa invocação deste operador, o seu único operando, obrigatoriamente do tipo `Classe`, é usado como instância implícita.
- Através de uma rotina não membro `tipo_de_devolução operator@(tipo_do_operando)`.

A expressão `@a` (ou `a@` se `@` for sufixo) pode portanto ser interpretada como

```
a.operator@()
```

ou

```
operator@(a)
```

consoante o operador esteja definido como membro da classe a que `a` pertence ou esteja definido como rotina normal, não-membro.

É importante notar que:

1. Quando a sobrecarga de um operador se faz por intermédio de uma operação (rotina membro) de uma classe C++, o primeiro operando (e único no caso de uma operação unária) numa expressão que envolva esse operador *não sofre nunca conversões implícitas de tipo*. Em todos os outros casos as conversões implícitas são possíveis.
2. Nunca se deve alterar a semântica dos operadores. Imagine-se os problemas que traria sobrecarregar o operador + para a classe C++ `Racional` como significando o produto!
3. Nem todos os operadores podem ser sobrecarregados por intermédio rotinas não-membro. Os operadores = (atribuição), `[]` (indexação), `()` (invocação) e `->` (selecção), só podem ser sobrecarregados por meio de operações (rotinas membro). Para todas as classes que não os redefinam, os operadores = (atribuição), `&` (unário, endereço de) e **,** (sequenciamento) são definidos implicitamente: por isso é possível atribuir instâncias de classes C++, como a classe `Racional`, sem para isso ter de sobrecarregar o operador de atribuição =).

Falta agora a tarefa algo penosa de sobrecarregar todos os operadores aplicáveis a racionais. Porquê? Porque, apesar de o programa da soma das fracções não necessitar senão dos operadores > > e < <, de extracção e inserção em canais, é instrutivo preparar a classe para utilizações futuras, ainda difíceis de antecipar.

Pretende-se, pois, equipar o TAD `Racional` com todos os operadores usuais para os tipos básicos do C++:

+, -, * e / Operadores aritméticos (binários): adição, subtracção, produto e divisão. Não têm efeitos laterais, i.e., não alteram os operandos.

+ e - Operadores aritméticos (unários): identidade e simétrico. Não têm efeitos laterais.

<, <=, >, >= Operadores relacionais (binários): menor, menor ou igual, maior e maior ou igual. Não têm efeitos laterais.

== e != Operadores de igualdade e diferença (binários). Não têm efeitos laterais.

++ e – Operadores de incrementação e decrementação prefixo (unários). Têm efeitos laterais, pois alteram o operando. Aliás, são eles a sua principal razão de ser.

++ e – Operadores de incrementação e decrementação sufixo (unários). Têm efeitos laterais.

+=, -=, *= e /= Operadores especiais de atribuição: adição e atribuição, subtracção e atribuição, produto e atribuição e divisão e atribuição (binários). Têm efeitos laterais, pois alteram o primeiro operando.

> > e < < Operadores de extracção e inserção de um canal (binários). Ambos alteram o operando esquerdo (que é um canal), mas apenas o primeiro altera o operando direito. Têm efeitos laterais.

7.6 Testes de unidade

Na prática, não é fácil a decisão de antecipar ou não utilizações futuras. Durante o desenvolvimento de uma classe deve-se tentar suportar utilizações futuras, difíceis de antecipar, ou deve-se restringir o desenvolvimento àquilo que é necessário em cada momento? Se o objectivo é preparar uma biblioteca de ferramentas utilizáveis por qualquer programador, então claramente devem-se tentar prever as utilizações futuras. Mas, se a classe está a ser desenvolvida para ser utilizada num projecto em particular, a resposta cai algures no meio destas duas opções. É má ideia, de facto, gastar esforço de desenvolvimento[14] a desenvolver ferramentas de utilização futura mais do que dúbia. Mas também é má ideia congelar o desenvolvimento de tal forma que aumentar as funcionalidades de uma classe C++, logo que tal se revele necessário, seja difícil. O ideal, pois, está em não desenvolver prevendo utilizações futuras, mas em deixar a porta aberta para futuros desenvolvimentos.

A recomendação anterior não se afasta muito do preconizado pela metodologia de desenvolvimento *eXtreme Programming* [1]. Uma excelente recomendação dessa metodologia é também o desenvolvimento dos chamados *testes de unidade*. Se se olhar com atenção para a definição da classe C++ `Racional` definida até agora, conclui-se facilmente que a maior parte das condições objectivo das operações não são testadas usando instruções de asserção. O problema é que a condição objectivo das operações está escrita em termos da noção matemática de número racional, e não é fácil fazer a ponte entre uma noção matemática e o código C++... Por exemplo, como explicitar em código a condição objectivo do operador + para racionais? Uma primeira tentativa poderia ser a tradução directa:

`/**` Devolve a soma com o racional recebido como argumento.
@pre `*this` $= r$.
@post `*this` $= r \wedge$ `operator+` $=$ `*this` $+$ `r2`. `*/`

[14] A não ser para efeitos de estudo e desenvolvimento pessoal, claro.

```
Racional Racional::operator+(Racional const r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    Racional r;

    r.numerador = numerador * r2.denominador +
        r2.numerador * denominador;
    r.denominador = denominador * r2.denominador;

    r.reduz();

    assert(cumpreInvariante() and r.cumpreInvariante());
    assert(r == *this + r2);

    return r;
}
```

Há dois problemas neste código. O primeiro é que o operador == ainda não está definido. Este problema resolver-se-á facilmente mais à frente neste capítulo. O segundo é muito mais importante: a asserção, tal como está escrita, recorre recursivamente ao próprio operador +! Claramente, o caminho certo não passa por aqui.

Os testes de unidade proporcionam uma alternativa interessante às instruções de asserção para as condições objectivo das operações. A ideia é que se deve escrever um conjunto exaustivo de testes para as várias operações da unidade e mantê-los durante toda a vida do código. Por unidade entende-se aqui uma unidade de modularização, tipicamente uma classe C++ e rotinas associadas que concretizam um TAD ou uma classe propriamente dita. Os testes de unidade só muito parcialmente substituem as instruções de asserção para as condições objectivo das operações da classe:

1. As instruções de asserções estão sempre activas e verificam a validade da condição objectivo sempre que o operação é invocada. Por outro lado, os testes de unidade apenas são executados de tempos e tempos, e de uma forma independente à do programa, ou dos programas, no qual a unidade testada está integrada.

2. As instruções de asserção verificam a validade da condição objectivo para todos os casos para os quais o programa, ou os programas, invocam a respectiva operação da classe C++. No caso dos testes de unidade, no entanto, é impensável testar exaustivamente as operações em causa.

3. As instruções de asserção estão activas durante o desenvolvimento e durante a exploração do programa desenvolvido[15], enquanto os testes de unidade são executados de tempos a tempos, durante o desenvolvimento ou manutenção do programa.

[15]Uma das características das instruções de asserção é que pode ser desactivadas facilmente, bastando para isso definir a macro `NDEBUG`. No entanto, não é muito boa ideia desactivar as instruções de asserção. Ver discussão sobre o assunto no !! citar capítulo sobre tratamento de erros.

Justificados que foram os testes de unidade, pode-se agora criar o teste de unidade para o TAD `Racional`:

```
#ifdef TESTE

#include <fstream>

/** Programa de teste do TAD Racional e da função mdc(). */
int main()
{
    assert(mdc(0, 0) == 1);
    assert(mdc(10, 0) == 10);
    assert(mdc(0, 10) == 10);
    assert(mdc(10, 10) == 10);
    assert(mdc(3, 7) == 1);
    assert(mdc(8, 6) == 2);
    assert(mdc(-8, 6) == 2);
    assert(mdc(8, -6) == 2);
    assert(mdc(-8, -6) == 2);

    Racional r1(2, -6);

    assert(r1.numerador() == -1 and r1.denominador() == 3);

    Racional r2(3);

    assert(r2.numerador() == 3 and r2.denominador() == 1);

    Racional r3;

    assert(r3.numerador() == 0 and r2.denominador() == 1);

    assert(r2 == 3);
    assert(3 == r2);
    assert(r3 == 0);
    assert(0 == r3);

    assert(r1 < r2);
    assert(r2 > r1);
    assert(r1 <= r2);
    assert(r2 >= r1);
    assert(r1 <= r1);
    assert(r2 >= r2);

    assert(r2 == +r2);
    assert(-r1 == Racional(1, 3));
```

```
    assert(++r1 == Racional(2, 3));
    assert(r1 == Racional(2, 3));

    assert(r1++ == Racional(2, 3));
    assert(r1 == Racional(5, 3));
    assert((r1 *= Racional(7, 20)) == Racional(7, 12));
    assert((r1 /= Racional(3, 4)) == Racional(7, 9));
    assert((r1 += Racional(11, 6)) == Racional(47, 18));
    assert((r1 -= Racional(2, 18)) == Racional(5, 2));

    assert(r1 + r2 == Racional(11, 2));
    assert(r1 - Racional(5, 7) == Racional(25, 14));
    assert(r1 * 40 == 100); assert(30 / r1 == 12);

    ofstream saída("teste");
    saída << r1 << ' ' << r2;
    saída.close();

    ifstream entrada("teste");
    Racional r4, r5;
    entrada >> r4 >> r5;

    assert(r1 == r4);
    assert(r2 == r5);
}

#endif // TESTE
```

São de notar os seguintes pontos:

- Alguma da sintaxe utilizada neste teste só será introduzida mais tarde. O leitor deve regressar a este teste quando o TAD `Racional` for totalmente desenvolvido.
- Cada teste consiste essencialmente numa instrução de asserção. Há melhores formas de escrever os testes de unidade, sem recorrer a asserções, nomeadamente recorrendo a bibliotecas de teste. Mas tais bibliotecas estão fora do âmbito deste texto.
- O teste consiste numa função `main()`. De modo a não entrar em conflito com a função `main()` do programa propriamente dito, envolveu-se a função `main()` de teste entre duas directivas de pré-compilação, `#ifdef TESTE` e `#endif // TESTE`. Isso faz com que toda a função só seja levada em conta pelo compilador quando estiver definida a macro `TESTE` (coisa que num compilador em Linux se consegue tipicamente com a opção de compilação `-DTESTE`). Este assunto será visto com rigor no Capítulo 9, onde se verá também como se pode preparar um TAD como o tipo `Racional` para ser utilizado em qualquer programa onde seja necessário trabalhar com racionais.

7.7 Devolução por referência

Começar-se-á o desenvolvimento dos operadores para o TAD `Racional` pelo operador de incrementação prefixo. Uma questão nesse desenvolvimento é saber o que é que devolve esse operador e, de uma forma mais geral, todos os operadores de incrementação e decrementação prefixos e especiais de atribuição.

7.7.1 Mais sobre referências

Na Secção 3.2.11 viu-se que se pode passar um argumento por referência a um procedimento se este tiver definido o parâmetro respectivo como uma referência. Por exemplo,

```
void troca(int& a, int& b)
{
    int auxiliar = a;
    a = b;
    b = auxiliar;
}
```

é um procedimento que troca os valores das duas variáveis passadas como argumento. Este procedimento pode ser usado no seguinte troço de programa

```
int x = 1, y = 2;
troca(x, y);
cout << x << ' ' << y << endl;
```

que mostra

```
2 1
```

no ecrã.

O conceito de referência pode ser usado de formas diferentes. Por exemplo,

```
int i = 1;
int& j = i; // a partir daqui j é sinónimo de i!
j = 3;
cout << i << endl;
```

mostra

```
3
```

no ecrã, pois alterar a variável `j` é o mesmo que alterar a variável `i`, já que `j` é sinónimo de `i`.

As variáveis que são referências, caso de `j` no exemplo anterior e dos parâmetros `a` e `b` do procedimento `troca()`, têm de ser inicializadas com a variável de que virão a ser sinónimos. Essa inicialização é feita explicitamente no caso de `j`, e implicitamente no caso das variáveis `a` e `b`, neste caso através da passagem de `x` e `y` como argumento na chamada de `troca()`.

Necessidade da devolução por referência

Suponha-se o código:

```
int i = 0;
++(++i);
cout << i << endl;
```

(Note-se que este código é muito pouco recomendável! Só que, como a sobrecarga dos operadores deve manter a mesma semântica que esses mesmos operadores possuem para os tipos básicos, é necessário conhecer bem "os cantos à casa", mesmo os mais obscuros, infelizmente.)

Este código resulta na dupla incrementação da variável `i`, como seria de esperar. Mas para isso acontecer, o operador `++`, para além de incrementar a variável `i`, tem de devolver a própria variável `i`, e não uma sua cópia, pois de outra forma a segunda aplicação do operador `++` levaria à incrementação da cópia, e não do original.

Para que este assunto fique mais claro, começar-se-á por escrever um procedimento `incrementa()` com o mesmo objectivo do operador de incrementação. Como este procedimento deve afectar a variável passada como argumento, neste caso `i`, deve receber o argumento por referência:

```
/** Incrementa o inteiro recebido como argumento e devolve-o.
    @pre i = i.
    @post i = i + 1. */
void incrementa(int& v)
{
    v = v + 1;
}
...
int i = 0;
incrementa(incrementa(i));
cout << i << endl;
```

Infelizmente este código não compila, pois a invocação mais exterior do procedimento recebe como argumento o resultado da primeira invocação, que é `void`. Logo, é necessário devolver um inteiro nesta rotina:

```
        /** Incrementa o inteiro recebido como argumento e devolve-o.
            @pre i = i.
            @post incrementa = i ∧ i = i + 1. */
        int incrementa(int& v)
        {
/* 1 */     v = v + 1;
/* 2 */     return v;
        }
        ...
```

```
        int i = 0;
/* 3 */             incrementa(i)
/* 4 */ incrementa(              );
/* 5 */ cout << i << endl;
```

Este código tem três problemas. O primeiro problema é que, dada a definição actual da linguagem, não compila, pois o valor temporário devolvido pela primeira invocação da rotina não pode ser passado por referência (não-constante) para a segunda invocação. A linguagem C++ proíbe a passagem por referência (não-constante) de valores, ou melhor, de variáveis temporárias[16]. O segundo problema é que, ao contrário do que se recomendou no capítulo sobre modularização, esta rotina não é um procedimento, pois devolve alguma coisa, nem uma função, pois afecta um dos seus argumentos. Note-se que continua a ser indesejável escrever este tipo de código. Mas a emulação do funcionamento do operador ++, que é um operador com efeitos laterais, obriga à utilização de uma função com efeitos laterais... O terceiro problema, mais grave, é que, mesmo que fosse possível a passagem de uma variável temporária por referência, o código acima ainda não faria o desejado, pois nesse caso a segunda invocação da rotina `incrementa()` acabaria por alterar apenas essa variável temporária, e não a variável `i`, como se pode ver na Figura 7.6. Para resolver este problema, a rotina deverá devolver não uma cópia de `i`, mas a própria variável `i`, que é como quem diz, um sinónimo da variável `i`. Ou seja, a rotina deverá devolver `i` por referência.

```
        /** Incrementa o inteiro recebido como argumento e devolve-o.
            @pre i = i.
            @post incrementa ≡ i ∧ i = i + 1. */
        int& incrementa(int& v)
        {
/* 1 */     v = v + 1;
/* 2 */     return v; // ou simplesmente return v = v + 1;
        }
        ...
        int i = 0;
/* 3 */             incrementa(i)
/* 4 */ incrementa(              );
/* 5 */ cout << i << endl;
```

Esta versão da rotina `incrementa()` já leva ao resultado pretendido, usando para isso uma *devolução por referência*. Repare-se na condição objectivo desta rotina e compare-se com a usada para a versão anterior da mesma `rotina()`, em que se devolvia por valor: neste caso é necessário dizer que o que se devolve é não apenas igual a `i`, mas também é *idêntico* a `i`, ou seja, é o próprio `i`, como se pode ver na Figura 7.7[17]. Para isso usou-se o símbolo $\equiv$ em vez do usual $=$.

Para se compreender bem a diferença entre a devolução por valor e devolução por referência, comparem-se as duas rotinas abaixo:

[16] Esta restrição é razoavelmente arbitrária, e está em discussão a sua possível eliminação numa próxima versão da linguagem C++.

[17] É necessário clarificar a diferença entre igualdade e identidade. Pode-se dizer que dois gémeos são iguais, mas não que são idênticos, pois são indivíduos diferentes. Por outro lado, pode-se dizer que Fernando Pessoa e Alberto Caeiro não são apenas iguais, mas também idênticos, pois são nomes que se referem à mesma pessoa.

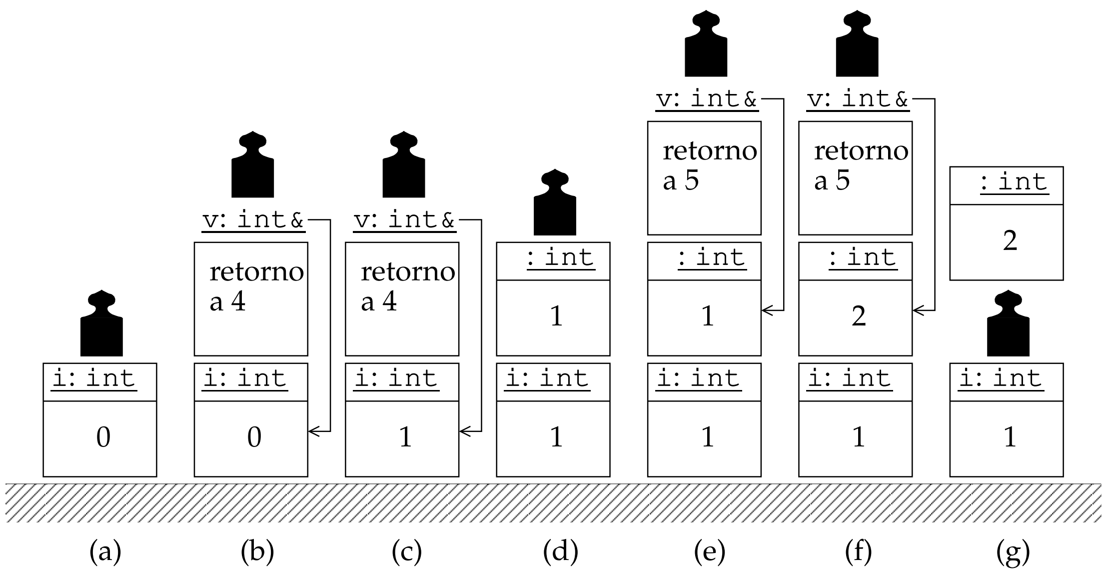


Figura 7.6: Estado da pilha durante a execução. (a) Imediatamente antes das invocações nas linhas 3 e 4; (b) imediatamente antes da instrução 1, depois de invocada a rotina pela primeira vez, já com o parâmetro `v` na pilha; (c) entre as instruções 1 e 2, já depois de incrementado `v` e portanto `i` (pois `v` é sinónimo de `i`); (d) imediatamente após a primeira invocação da rotina e imediatamente antes da sua segunda invocação, já com os parâmetros retirados da pilha, sendo de notar que o valor devolvido está guardado numa variável temporária, sem nome, no topo da pilha (a variável está abaixo do "pisa-papeis" que representa o topo da pilha, e não acima, como é habitual com os valores devolvidos, por ir ser construída uma referência para alea, que obriga a que seja preservada mais tempo); (e) imediatamente antes da instrução 1, depois de invocada a rotina pela segunda vez; (f) entre as instruções 1 e 2, já depois de incrementado `v` e portanto já depois de incrementada a variável temporária, de que `v` é sinónimo; e (g) imediatamente antes da instrução 5, depois de a rotina retornar. Vê-se claramente que a variável incrementada da segunda vez não foi a variável `i`, como se pretendia, mas uma variável temporária, entretanto destruída.

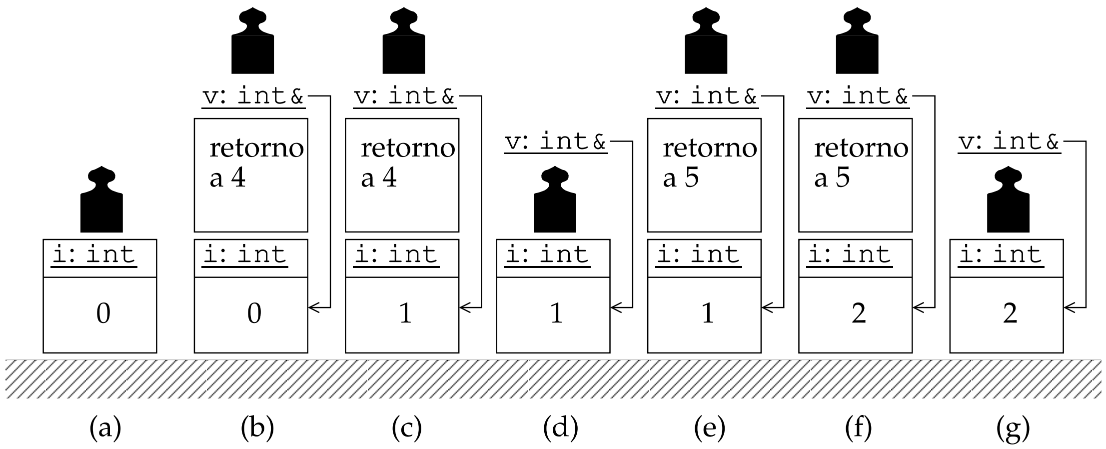


Figura 7.7: Estado da pilha durante a execução. (a) Imediatamente antes das invocações nas linhas 3 e 4; (b) imediatamente antes da instrução 1, depois de invocada a rotina pela primeira vez, já com o parâmetro `v` na pilha; (c) entre as instruções 1 e 2, já depois de incrementado `v` e portanto `i` (pois `v` é sinónimo de `i`); (d) imediatamente após a primeira invocação da rotina e imediatamente antes da sua segunda invocação, já com os parâmetros retirados da pilha, sendo de notar que a referência devolvida se encontra no topo da pilha; (e) imediatamente antes da instrução 1, depois de invocada a rotina pela segunda vez, tendo a referência `v` sido inicializada à custa da referência no topo da pilha, e portanto sendo `v` sinónimo de `i`; (f) entre as instruções 1 e 2, já depois de incrementado `v` e portanto já depois de incrementada a variável `i` pela segunda vez; e (g) imediatamente antes da instrução 5, depois de a rotina retornar. Vê-se claramente que a variável incrementada da segunda vez foi exactamente a variável `i`, como se pretendia.

```
int cópia(int v)
{
    return v;
}

int& mesmo(int& v)
{
    return v;
}
```

A primeira rotina devolve uma cópia do parâmetro `v`, que por sua vez é já uma cópia do argumento passado à rotina. Ou seja, devolve uma cópia do argumento, coisa que aconteceria mesmo que o argumento fosse passado por referência. A segunda rotina, pelo contrário, recebe o seu argumento por referência e devolve o seu parâmetro também por referência. Ou seja, o que é devolvido é um sinónimo do próprio parâmetro `v`, que por sua vez é um sinónimo do argumento passado à rotina. Ou seja, a rotina devolve um sinónimo do argumento. Uma questão filosófica é que esse sinónimo ... não tem nome! Ou melhor, é a própria expressão de invocação da rotina que funciona como sinónimo do argumento. Isso deve ser claro no seguinte código:

```
int main()
{
    int valor = 0;

    cópia(valor) = 10; // erro!
    mesmo(valor) = 10;
}
```

A instrução envolvendo a rotina `cópia()` está errada, pois a rotina devolve um valor temporário, que não pode surgir do lado esquerdo de uma atribuição. Na terminologia da linguagem C++ diz-se que `cópia(valor)` não é um *valor esquerdo* (*lvalue* ou *left value*). Pelo contrário a expressão envolvendo a rotina `mesmo()` está perfeitamente correcta, sendo absolutamente equivalente a escrever:

```
valor = 10;
```

Na realidade, ao se devolver por referência numa rotina, está-se a dar a possibilidade ao consumidor desse procedimento de colocar a sua invocação do lado esquerdo da atribuição. Por exemplo, definido a rotina `incrementa()` como acima, é possível escrever

```
int a = 11;
incrementa(a) = 0;   // possível (mas absurdo), incrementa e depois atri-
bui zero a  a.
incrementa(a) /= 2; // possível (mas má ideia), incrementa e depois di-
vide a por dois.
```

Note-se que a devolução de referências implica alguns cuidados adicionais. Por exemplo, a rotina

```
int& mesmoFalhado(int v)
{
    return v;
}
```

contém um erro grave: devolve uma referência (ou sinónimo) para uma variável local, visto que o parâmetro `v` não é uma referência, mas sim uma variável local cujo valor é uma cópia do argumento respectivo. Como essa variável local é destruída exactamente aquando do retorno da rotina, a referência devolvida fica a referir-se a ... coisa nenhuma!

Uma digressão pelo operador `[]`

Uma vez que o operador de indexação `[]`, usado normalmente para as matrizes e vectores, pode ser sobrecarregado por tipos definidos pelo programador, a devolução de referências permite, por exemplo, definir a classe `VectorDeInt` abaixo, que se comporta aproximadamente como a classe `vector<int>` descrita na Secção 5.2, embora com verificação de erros de indexação:

```
#include <iostream>
#include <vector>
#include <cstdlib>

using namespace std;

class VectorDeInt {
  public:
    VectorDeInt(vector<int>::size_type const dimensão_inicial = 0,
                int const valor_inicial_dos_itens = 0);

    ...

    int& operator[](vector<int>::size_type const índice);

    ...

  private:
    vector<int> itens;
};

VectorDeInt::VectorDeInt(vector<int>::size_type const dimensão_inicial,
                         int const valor_inicial_dos_itens)
    : v(dimensão_inicial, valor_inicial_dos_itens)
```

```
{
}

...

int& VectorDeInt::operator[](vector<int>::size_type const índice)
{
    assert(0 <= índice and índice < itens.size());

    return itens[índice];
}

int main()
{
    VectorDeInt v(10);

    v[0] = 1;
    v[10] = 3; // índice errado! aborta com asserção falhada.
}
```

7.7.2 Operadores ++ e −− prefixo

O operador ++ prefixo necessita de alterar o seu único operando. Assim, é conveniente sobrecarregá-lo na forma de uma operação da classe C++ `Racional`. Uma vez que tem um único operando, este será usado como instância, neste caso variável, implícita durante a execução do respectivo método, pelo que a operação não tem qualquer parâmetro. É importante perceber que a incrementação de um racional pode ser feita de uma forma muito mais simples do que recorrendo à soma de racionais em geral: a adição de um a um racional representado por uma fracção canónica $\frac{n}{d}$ é

$$\frac{n+d}{d},$$

que também é uma fracção no formato canónico[18], pelo que o código é simplesmente

[18]Seja $\frac{n}{d}$ o racional guardado numa instância da classe C++ `Racional`, e que portanto verifica a condição invariante dessa classe, ou seja, $0 < d \wedge \mathrm{mdc}(n, d) = 1$. É óbvio que

$$\frac{n}{d} + 1 = \frac{n+d}{d}.$$

Mas será que a fracção

$$\frac{n+d}{d}$$

verifica a condição invariante de classe? Claramente o denominador d é positivo, pelo que resta verificar se $\mathrm{mdc}(n+d, d) = 1$. Suponha-se que existe um divisor $1 < k$ comum ao numerador e ao denominador. Nesse caso existem n' e d' tais que $kn' = n + d$ e $kd' = d$, de onde se conclui facilmente que $kn' = n + kd'$, ou seja, $n = k(n' - d')$. Só que isso significaria que $1 < k \leq \mathrm{mdc}(n, d)$, o que é contraditório com a hipótese de partida de que $\mathrm{mdc}(n, d) = 1$. Logo, não existe divisor comum ao numerador e denominador superior à unidade, ou seja, $\mathrm{mdc}(n+d, d) = 1$ como se queria demonstrar.

```
/** Representa números racionais.
    @invariant 0 < denominador ∧ mdc(numerador, denominador) = 1. */
class Racional {
  public:

    ...

    /** Incrementa e devolve o racional.
        @pre *this = r.
        @post operador++ ≡ *this ∧ *this = r + 1. */
    Racional& operator++();

    ...

};

...

Racional& Racional::operator++()
{
    assert(cumpreInvariante());

    numerador += denominador;

    assert(cumpreInvariante());

    return ?;
}

...
```

não se necessitando de reduzir a fracção depois desta operação.

Falta resolver um problema, que é o que devolver no final do método. Depois da discussão anterior com a rotina `incrementa()`, deve já ser claro que se deseja devolver o próprio operando do operador, que neste caso corresponde à variável implícita. Como se viu atrás, é possível explicitar a variável implícita usando a construção `*this`, pelo que o código fica simplesmente:

```
Racional& Racional::operator++()
{
    assert(cumpreInvariante());

    numerador += denominador;

    assert(cumpreInvariante());
```

```
    return *this;
}
```

A necessidade de devolver a própria variável implícita ficará porventura mais clara se se observar um exemplo semelhante ao que se usou mais atrás, mas usando racionais em vez de inteiros:

```
Racional r(1, 2);
++ ++r; // o mesmo que ++(++r);
r.escreve();
cout << endl;
```

Este código é absolutamente equivalente ao seguinte, que usa a notação usual de invocação de operações de uma classe C++:

```
Racional r(1, 2);
(r.operator++()).operator++();
r.escreve();
cout << endl;
```

Aqui torna-se perfeitamente clara a necessidade de devolver a própria variável implícita, para que esta possa ser usada par invocar pela segunda vez o mesmo operador.

Quanto ao operador -- prefixo, a sua definição é igualmente simples:

```
/** Representa números racionais.
    @invariant 0 < denominador ∧ mdc(numerador, denominador) = 1. */
class Racional {
  public:

    ...

    /** Decrementa e devolve o racional.
        @pre *this = r.
        @post operador-- ≡ *this ∧ *this = r − 1. */
    Racional& operator--();

    ...

};

...

inline Racional& Racional::operator--()
```

```
{
    assert(cumpreInvariante());

    numerador -= denominador;

    assert(cumpreInvariante());

    return *this;
}

...
```

7.7.3 Operadores ++ e −− sufixo

Qual é a diferença entre os operadores de incrementação e decrementação prefixo e sufixo? Como já foi referido no Capítulo 2, a diferença está no que devolvem. As versões prefixo devolvem o próprio operando, já incrementado, e as versões sufixo devolvem uma cópia do valor do operando *antes de incrementado*. Para que tal comportamento fique claro, convém comparar cuidadosamente os seguintes troços de código:

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

e

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

Enquanto o primeiro troço de código inicializa a variável `j` como valor de `i` já incrementado, i.e., com 1, o segundo troço de código inicializa a variável `j` com o valor de `i` *antes de incrementado*, ou seja, com 0. Em ambos os casos a variável `i` é incrementada, ficando com o valor 1.

Clarificada esta diferença, há agora que implementar os operadores de incrementação e decrementação sufixo para a classe C++ `Racional`. A primeira questão, fundamental, é sintáctica: sendo os operadores prefixo e sufixo ambos unários, como distingui-los na definição dos operadores? Se forem definidos como operações da classe C++ `Racional`, então ambos terão o mesmo nome e ambos não terão nenhum parâmetro, distinguindo-se apenas no tipo de devolução, visto que as versões prefixo devolvem por referência e as versões sufixo devolvem por valor. O mesmo se passa se os operadores forem definidos como rotinas normais, não-membro. O problema é que o tipo de devolução não faz parte da assinatura das rotinas, membro ou não, pelo que o compilador se queixará de uma dupla definição do mesmo operador...

Face a esta dificuldade, os autores da linguagem C++ tomaram uma das decisões mais arbitrárias que poderiam ter tomado. Arbitraram que para as assinaturas entre os operadores de incrementação e decrementação prefixo serem diferentes das respectivas versões sufixo, estas últimas teriam como que um operando adicional, inteiro, implícito, e cujo valor deve ser ignorado. É um pouco como se os operadores sufixo fossem binários...

Por razões que ficarão claras mais à frente, definir-se-ão os operadores de incrementação e decrementação sufixo como rotinas normais, não-membro.

Comece-se pelo operador de incrementação sufixo. Sendo sufixo, a sua definição assume que o operador é binário, tendo como primeiro operando o racional a incrementar e como segundo operando um inteiro cujo valor deve ser ignorado. Como o operador será sobrecarregado através de uma rotina normal, ambos os operandos correspondem a parâmetros da rotina, sendo o primeiro, corresponde ao racional a incrementar, passado por referência:

```
/** Incrementa o racional recebido como argumento, devolvendo o seu valor an-
tes de incrementado.
    @pre *this = r.
    @post operator++ = r ∧ *this = r + 1. */
Racional operator++(Racional& r, int valor_a_ignorar)
{
    Racional const cópia = r;

    ++r;

    return cópia;
}
```

Como a parâmetro `valor_a_ignorar` é arbitrário, servindo apenas para compilador perceber que se está a sobrecarregar o operador sufixo, e não o prefixo, não é necessário sequer dar-lhe um nome, pelo que a definição pode ser simplifica para

```
/** Incrementa o racional recebido como argumento, devolvendo o seu valor an-
tes de incrementado.
    @pre *this = r.
    @post operator++ = r ∧ *this = r + 1. */
Racional operator++(Racional& r, int)
{
    Racional const cópia = r;

    ++r;

    return cópia;
}
```

É interessante notar como se recorre ao operador de incrementação prefixo, que já foi definido, na implementação do operador sufixo. Ao contrário do que pode parecer, tal não ocorre simplesmente porque se está a sobrecarregar o operador sufixo como uma rotina não-membro da classe `Racional`. De facto, mesmo que o operador fosse definido como membro da classe

```
/* A sobrecarga também se poderia fazer à custa de uma operação da classe! */
Racional Racional::operator++(int)
{
    Racional const cópia = *this;

    ++*this;

    return cópia;
}
```

continuaria a ser vantajoso fazê-lo: é que o código de incrementação propriamente dito fica concentrado numa única rotina, pelo que, se for necessário mudar a representação dos racionais, apenas será necessário alterar a implementação do operador prefixo.

Repare-se como, em qualquer dos casos, é necessário fazer uma cópia do racional antes de incrementado e devolver essa cópia por valor, o que implica realizar ainda outra cópia. Finalmente compreende-se a insistência, desde o início deste texto, em usar a incrementação prefixo em detrimento da versão sufixo, mesmo onde teoricamente ambas produzem o mesmo resultado, tal como em incrementações ou decrementações isoladas (por exemplo no progresso de um ciclo): é que a incrementação ou decrementação sufixo é quase sempre menos eficiente do que a respectiva versão prefixo.

O operador de decrementação sufixo define-se exactamente de mesma forma:

```
/** Decrementa o racional recebido como argumento, devolvendo o seu valor an-
tes de decrementado.
    @pre *this = r.
    @post operator-- = r ∧ *this = r − 1. */
Racional operator--(Racional& r, int)
{
    Racional const cópia = r;

    --r;

    return cópia;
}
```

Como é óbvio, tendo-se devolvido por valor em vez de por referência, não é possível escrever

```
Racional r;
r++ ++; // erro!
```

que de resto já era uma construção inválida para os tipos básicos do C++.

7.8 Mais operadores para o TAD **Racional**

Falta ainda sobrecarregar muitos operadores para o TAD Racional. Um facto curioso, como se verificará em breve, é que os operadores aritméticos sem efeitos laterais se implementam facilmente à custa dos operadores aritméticos com efeitos laterais, e que a versão alternativa, em que se implementam os operadores com efeitos laterais à custa dos que não os têm, conduz normalmente a menores eficiências, pois estes últimos operadores implicam frequentemente a realização de cópias. Assim, tendo-se já sobrecarregado os operadores de incrementação e decrementação, o próximo passo será o de sobrecarregar os operadores de atribuição especiais. Depois definir-se-ão os operadores aritméticos normais. Aliás, no caso do operador + será uma re-implementação.

7.8.1 Operadores de atribuição especiais

Começar-se-á pelo operador *=, de implementação muito simples.

Tal como os operadores de incrementação e decrementação, também os operadores de atribuição especiais são mal comportados. São definidos à custas de rotinas que são mistos de função e procedimento, ou funções com efeitos laterais. O operador *= não é excepção. Irá ser sobrecarregado à custa de uma operação da classe C++ Racional, pois necessita de alterar os atributos da classe. Como o operador *= tem dois operandos, o primeiro será usado com instância (aliás, variável) implícita, e o segundo será passado como argumento. A operação terá, pois um único parâmetro. Todos os operadores de atribuição especiais devolvem uma referência para o primeiro operando, tal como os operadores de incrementação e decrementação prefixo. É isso que permite escrever o seguinte pedaço de código, muito pouco recomendável, mas idêntico ao que se poderia também escrever para variáveis dos tipos básicos do C++:

```
Racional a(4), b(1, 2);
(a *= b) *= b;
```

Deve ser claro que este código multiplica a por $\frac{1}{2}$ duas vezes, ficando a com o valor 1.

A implementação do operador produto e atribuição é simples::

```
...

class Racional {
  public:

    ...

    /** Multiplica por um racional.
        @pre *this = r.
        @post operator*= ≡ *this ∧ *this = r × r2. */
    Racional& operator*=(Racional r2);
```

```
    ...
};

...

Racional& Racional::operator*=(Racional const r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    numerador *= r2.numerador;
    denominador *= r2.denominador;

    reduz();

    assert(cumpreInvariante());

    return *this;
}

...
```

O corpo do método definido limita-se a efectuar o produto da forma usual para as fracções, i.e., o numerador do produto é o produto dos numeradores e o denominador do produto é o produto dos denominadores. Como os denominadores são ambos positivos, o seu produto também o será. Para que o resultado cumpra a condição invariante de classe falta apenas garantir que no final do método $\mathrm{mdc}(n, d) = 1$. Como isso não é garantido (pense-se, por exemplo, o produto de $\frac{1}{2}$ por 2), é necessário reduzir a fracção resultado. Tal como no caso dos operadores de incrementação e decrementação prefixo, também aqui se termina devolvendo a variável implícita, i.e., o primeiro operando.

O operador `/=` sobrecarrega-se da mesma forma, embora tenha de haver o cuidado de garantir que o segundo operando não é zero:

```
...

class Racional {
  public:

    ...

    /** Divide por um racional.
        @pre *this = r ∧ r2 ≠ 0.
        @post operator/= ≡ *this ∧ *this = r/r2. */
    Racional& operator/=(Racional r2);

    ...
```

```
};

...

Racional& Racional::operator/=(Racional const r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    assert(r2 != 0);

    if(r2.numerador < 0) {
        numerador *= -r2.denominador;
        denominador *= -r2.numerador;
    } else {
        numerador *= r2.denominador;
        denominador *= r2.numerador;
    }

    reduz();

    assert(cumpreInvariante());

    return *this;
}

...
```

Há neste código algumas particularidades que é preciso estudar.

A divisão por zero é impossível, pelo que a pré-condição obriga `r2` a ser diferente de zero. A instrução de asserção reflecte isso mesmo, embora contenha um erro: por ora não é possível comparar dois racionais através do operador `!=`, quanto mais um racional e um inteiro (0 é um do tipo `int`). Pede-se ao leitor que seja paciente, pois dentro em breve este problema será resolvido sem ser necessário alterar em nada este método!

O cálculo da divisão é muito simples: o numerador da divisão é o numerador do primeiro operando multiplicado pelo denominador do segundo operando, e vice-versa. Uma versão simplista do cálculo da divisão seria:

```
numerador *= r2.denominador;
denominador *= r2.numerador;
```

Este código, no entanto, não só não garante que o resultado esteja reduzido, e daí a invocação de `reduz()` no código mais acima. (tal como acontecia para o operador `*=`) como também não garante que o denominador resultante seja positivo, visto que o numerador de `r2` pode perfeitamente ser negativo. Prevendo esse caso o código fica

```
if(r2.numerador < 0) {
    numerador *= -r2.denominador;
    denominador *= -r2.numerador;
} else {
    numerador *= r2.denominador;
    denominador *= r2.numerador;
}
```

tal como se pode encontrar no no método acima.

Relativamente ao operador += é possível resolver o problema de duas formas. A mais simples neste momento é implementar o operador += à custa do operador +, pois este já está definido. Nesse caso a solução é:

```
Racional& Racional::operator+=(Racional const r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    *this = *this + r2

    assert(cumpreInvariante());

    return *this;
}
```

Esta solução, no entanto, tem o inconveniente de obrigar à realização de várias cópias entre racionais, além de exigir a construção de um racional temporário para guardar o resultado da adição antes de este ser atribuído à variável implícita. Como se verá, a melhor solução é desenvolver o operador += de raiz e implementar o operador + à sua custa.

Os operadores += e −= sobrecarregam-se de forma muito semelhante:

```
...

class Racional {
  public:

    ...

    /** Adiciona de um racional.
        @pre *this = r.
        @post operator+= ≡ *this ∧ *this = r + r2. */
    Racional& operator+=(Racional r2);

    /** Subtrai de um racional.
        @pre *this = r.
```

```
        @post operator-= ≡ *this ∧ *this = r − r2. */
    Racional& operator-=(Racional r2);

    ...
};

...

Racional& Racional::operator+=(Racional const r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    numerador = numerador * r2.denominador +
        r2.numerador * denominador;
    denominador *= r2.denominador;

    reduz();

    assert(cumpreInvariante());

    return *this;
}

Racional& Racional::operator-=(Racional const r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    numerador = numerador * r2.denominador -
        r2.numerador * denominador;
    denominador *= r2.denominador;

    reduz();

    assert(cumpreInvariante());

    return *this;
}

...
```

7.8.2 Operadores aritméticos

Os operadores aritméticos usuais podem ser facilmente implementados à custa dos operadores especiais de atribuição. Implementar-se-ão aqui como rotinas normais, não-membro, por razões que serão clarificadas em breve. Começar-se-á pelo operador `*`. A ideia é criar uma variável local temporária cujo valor inicial seja uma cópia do primeiro operando, e em seguida usar o operador *= para proceder à soma:

```
/** Produto de dois racionais.
    @pre V.
    @post operator* = r1 × r2. */
Racional operator*(Racional const r1, Racional const r2)
{
    Racional auxiliar = r1;
    auxiliar *= r2;
    return auxiliar;
}
```

Observando cuidadosamente este código, conclui-se facilmente que o parâmetro `r1`, desde que deixe de ser constante, pode fazer o papel da variável `auxiliar`, visto que a passagem se faz por valor:

```
/** Produto de dois racionais.
    @pre r1 = r₁.
    @post operator* = r₁ × r2. */
Racional operator*(Racional r1, Racional const r2)
{
    r1 *= r2;
    return r1;
}
```

Finalmente, dado que o operador *= devolve o primeiro operando, podem-se condensar as duas instruções do método numa única instrução idiomática:

```
/** Produto de dois racionais.
    @pre r1 = r₁.
    @post operator* = r₁ × r2. */
Racional operator*(Racional r1, Racional const r2)
{
    return r1 *= r2;
}
```

A implementação dos restantes operadores aritméticos faz-se exactamente da mesma forma:

```
/** Divisão de dois racionais.
    @pre r1 = r₁ ∧ r2 ≠ 0.
    @post operator/ = r₁/r2. */
Racional operator/(Racional r1, Racional const r2)
{
    assert(r2 != 0);
```

```
    return r1 /= r2;
}
```

/** Adição de dois racionais.
@pre r1 $= r_1$.
@post operator+ $= r_1 +$ r2. */

```
Racional operator+(Racional r1, Racional const r2)
{
    return r1 += r2;
}
```

/** Subtracção de dois racionais.
@pre r1 $= r_1$.
@post operator- $= r_1 -$ r2. */

```
Racional operator-(Racional r1, Racional const r2)
{
    return r1 -= r2;
}
```

Para além da vantagem já discutida de implementar um operador à custa de outro, agora deve já ser clara a vantagem de ter implementado o operador `*` à custa do operador `*=` e não o contrário: a operação `*=` tornou-se muito mais eficiente, pois não obriga a copiar ou construir qualquer racional, enquanto a operação `*` continua a precisar a sua dose de cópias e construções...

Mas, porquê definir estes operadores como rotinas normais, não-membro? Há uma razão de peso, que tem a ver com as conversões implícitas.

7.9 Construtores: conversões implícitas e valores literais

7.9.1 Valores literais

Já se viu que a definição de classes C++ concretizando TAD permite a acrescentar à linguagem C++ novos tipos que funcionam praticamente como os seus tipos básicos. Mas haverá equivalente aos valores literais? Recorde-se que, num programa em C++, `10` e `100.0` são valores literais dos tipos `int` e `double`, respectivamente. Será possível especificar uma forma para, por exemplo, escrever valores literais do novo tipo `Racional`? Infelizmente isso é impossível em C++. Por exemplo, o código

```
Racional r;
r = 1/3;
```

redunda num programa aparentemente funcional mas com um comportamento inesperado. Acontece que a expressão `1/3` é interpretada como a divisão inteira, que neste caso tem resultado zero. Esse valor inteiro é depois convertido implicitamente para o tipo `Racional` e atribuída à variável `r`. Logo, `r`, depois da atribuição, conterá o racional zero!

Existe uma alternativa elegante aos inexistentes valores literais para os racionais. É proporcionada pelos construtores da classe, e funciona quase como se de valores literais se tratasse: os construtores podem ser chamados explicitamente para criar um novo valor dessa classe. Assim, o código anterior deveria ser corrigido para:

```
Racional r;
r = Racional(1, 3);
```

7.9.2 Conversões implícitas

Se uma classe `A` possuir um construtor que possa ser invocado passando um único argumento do tipo `B` como argumento, então está disponível uma conversão implícita do tipo `B` para a classe `A`. Por exemplo, o primeiro construtor da classe `Racional` (ver Secção 7.4.1) pode ser chamado com apenas um argumento do tipo `int`, o que significa que, sempre que o compilador esperar um `Racional` e encontrar um `int`, converte o `int` implicitamente para um valor `Racional`. Por exemplo, estando definido um operator + com operandos do tipo `Racional`, o seguinte pedaço de código

```
Racional r1(1, 3);
Racional r2 = r1 + 1;
```

é perfeitamente legal, tendo o mesmo significado que

```
Racional r1(1, 3);
Racional r2 = r1 + Racional(1);
```

e colocando em `r2` o racional $\frac{4}{3}$.

Em casos em que esta conversão implícita de tipos é indesejável, pode-se preceder o respectivo construtor da palavra-chave `explicit`. Assim, se a classe `Racional` estivesse definida como

```
...

class Racional {
  public:
     /** Constrói racional com valor inteiro. Construtor por omissão.
         @pre V.
         @post *this = n∧
               0 < denominador ∧ mdc(numerador, denominador) = 1. */
    explicit Racional(int const n = 0);
    ...

};

...
```

o compilador assinalaria erro ao encontrar a expressão `r1 + 1`. Neste caso, no entanto, a conversão implícita de `int` para `Racional` é realmente útil, pelo que o qualificador `explicit` é desnecessário.

7.9.3 Sobrecarga de operadores: operações ou rotinas?

Suponha-se por um instante que o operador + para a classe C++ Racional é sobrecarregado através de uma operação. Isto é, regresse-se à versão do operador + apresentada na Secção 7.5. Nesse caso o seguinte código

```
Racional r(1, 3);
Racional s = r + 3;
```

é válido, pois o valor inteiro 3 é convertido implicitamente para `Racional` e seguidamente é invocado o operador + definido.

Ou seja, o código acima é equivalente a

```
Racional r(1, 3);
Racional s = r + Racional(3);
```

Porém, o código

```
Racional r(1, 3);
Racional s = 3 + r;
```

é inválido, pois a linguagem C++ proíbe conversões na instância através da qual se invoca um método. Se o operador tivesse sido sobrecarregado à custa de uma normal rotina não-membro, todos os seus argumentos poderiam sofrer conversões implícitas, o que resolveria o problema. Mas foi exactamente isso que se fez nas secções anteriores! Logo, o código acima é perfeitamente legal e equivalente a

```
Racional r(1, 3);
Racional s = Racional(3) + r;
```

Este facto será utilizado para implementar alguns dos operadores em falta para a classe C++ `Racional`.

7.10 Operadores igualdade, diferença e relacionais

Os operadores de igualdade, diferença e relacionais serão desenvolvidos usando algumas das técnicas já apresentadas. Estes operadores serão sobrecarregados usando rotinas não-membro, de modo a se tirar partido das conversões implícitas de `int` para `Racional`, e tentar-se-á que sejam implementados à custa de outros módulos preexistentes, por forma a minimizar o impacte de possíveis alterações na representação (interna) dos números racionais.

Os primeiros operadores a sobrecarregar serão o operador igualdade, ==, e o operador diferença, `!=`. Viu-se na Secção 7.4.4 que o facto de as instâncias da classe C++ `Racional` cumprirem a condição invariante de classe, i.e., de $\frac{\texttt{numerador}}{\texttt{denominador}}$ ser uma fracção no formato canónico, permitia simplificar muito a comparação entre racionais. De facto, assim é, pois dois racionais são iguais se e só se tiverem representações em fracções canónicas iguais. Assim, uma primeira tentativa de definir o operador == poderia ser:

```
/** Indica se dois racionais são iguais.
    @pre V.
    @post operator== = (r1 = r2). */
bool operator==(Racional const r1, Racional const r2)
{
    return r1.numerador == r2.numerador and
           r1.denominador == r2.denominador;
}
```

O problema deste código é que, sendo o operador uma rotina não-membro, não tem acesso aos membros privados da classe C++. Por outro lado, se o operador fosse uma operação da classe C++, embora o problema do acesso aos membros se resolvesse, deixariam de ser possíveis conversões implícitas do primeiro operando do operador. Como resolver o problema?

Há duas soluções para este dilema. A primeira passa por tornar a rotina que sobrecarrega o operador == *amigo* da classe C++ `Racional` (ver Secção 7.15). Esta solução é desaconselhável, pois há uma alternativa simples que não passa por amizades (e, por isso, não está sujeita a introduzir quaisquer promiscuidades): deve-se explorar o facto de a rotina precisar de saber os valores do numerador e denominador da fracção canónica correspondente ao racional, *mas não precisar de alterar o seu valor*.

7.10.1 Inspectores e interrogações

Se se pensar cuidadosamente nas possíveis utilizações do TAD `Racional`, conclui-se facilmente que o programador consumidor pode necessitar de conhecer a fracção canónica correspondente ao racional. Se assim for, convém equipar a classe C++ com duas funções membro que se limitam a devolver o valor do numerador e denominador dessa fracção canónica. Como a representação de um `Racional` é justamente feita à custa de uma fracção canónica, conclui-se que as duas funções membro são muito fáceis de implementar[19]:

---

[19] A condição objectivo da operação `denominador()` é algo complexa, pois evita colocar na interface da classe referências à sua implementação, como seria o caso se se referisse ao atributo `denominador_`. Assim, usa-se a definição de denominador de fracção canónica. O valor devolvido `denominador` tem de ser tal que exista um numerador n tal que

1. a fracção $\frac{n}{\texttt{denominador}}$ é igual à instância implícita e
2. a fracção $\frac{n}{\texttt{denominador}}$ está no formato canónico,

```
...

class Racional {
  public:

    ...

    /** Devolve numerador da fracção canónica correspondente ao racional.
        @pre *this = r.
        @post *this = r ∧ numerador/denominador() = *this. */
    int numerador();

    /** Devolve denominador da fracção canónica correspondente ao racional.
        @pre *this = r.
        @post *this = r ∧
(E n : V : n/denominador = *this ∧ 0 < denominador ∧ mdc(n, denominador) = 1). */
    int denominador();

    ...

  private:
    int numerador_;
    int denominador_;

    ...
};

...

int Racional::numerador()
{
    assert(cumpreInvariante());

    assert(cumpreInvariante());
    return numerador_;
}

int Racional:: denominador()
{
    assert(cumpreInvariante());
```

ou seja, o valor devolvido é o denominador da fracção canónica correspondente à instância implícita.

A condição objectivo da operação `numerador()` é mais simples, pois recorre à definição da operação `denominador()` para dizer que se o valor devolvido for o numerador de uma fracção cujo denominador é o valor devolvido pela operação `denominador()`, então essa fracção é igual à instância implícita. Como o denominador usado é o denominador da fracção canónica igual à instância implícita, conclui-se que o valor devolvido é de facto o numerador dessa mesma fracção canónica.

```
        assert(cumpreInvariante());

        return denominador_;
    }

    ...
```

Às operações de uma classe C++ que se limitam a devolver propriedades das suas instâncias chama-se *inspectores*. Invocá-las também se diz *interrogar* a instância. Os inspectores permitem obter os valores de propriedades de um instância sem que se exponham as suas partes privadas à manipulação pelo público em geral.

É de notar que a introdução destes novos operadores trouxe um problema prático. As novas operações têm naturalmente o nome que antes tinham os atributos da classe. Repare-se que não se decidiu dar outros nomes às operações para evitar os conflitos: que produz uma classe deve estar preparado para, em nome do fornecimento de uma interface tão clara e intuitiva quanto possível, fazer alguns sacrifícios. Neste caso o sacrifício é o de alterar o nome dos atributos, aos quais é comum acrescentar um sublinhado (_) para os distinguir de operações com o mesmo nome, e, sobretudo, alterar os nomes desses atributos em todas as operações entretanto definidas (e note-se que já são algumas...).

7.10.2 Operadores de igualdade e diferença

Os inspectores definidos na secção anterior são providenciais, pois permitem resolver facilmente o problema do acesso aos atributos. Basta recorrer a eles para comparar os dois racionais:

```
/** Indica se dois racionais são iguais.
    @pre V.
    @post operator== = (r1 = r2). */
bool operator==(Racional const r1, Racional const r2)
{
    return r1.numerador() == r2.numerador() and
           r1.denominador() == r2.denominador();
}
```

O operador != sobrecarrega-se de uma forma ainda mais simples: negando o resultado de uma invocação ao operador == definido acima:

```
/** Indica se dois racionais são diferentes.
    @pre V.
    @post operator!= = (r1 ≠ r2). */
bool operator!=(Racional const r1, Racional const r2)
{
    return not (r1 == r2);
}
```

7.10.3 Operadores relacionais

O operador < pode ser facilmente implementado para a classe C++ `Racional`, bastando recorrer ao mesmo operador para os inteiros. Suponha-se que se pretende saber se

$$\frac{n_1}{d_1} < \frac{n_2}{d_2},$$

em que $\frac{n_1}{d_1}$ e $\frac{n_2}{d_2}$ são fracções no formato canónico. Como $0 < d_1$ e $0 < d_2$, a desigualdade acima é equivalente a

$$n_1 d_2 < n_2 d_1.$$

Logo, a sobrecarga do operador < pode ser feita como se segue:

```
/** Indica se o primeiro racional é menor que o segundo.
    @pre V.
    @post operator< = (r1 < r2). */
bool operator<(Racional const r1, Racional const r2)
{
    return r1.numerador() * r2.denominador() <
           r2.numerador() * r1.denominador();
}
```

Os restantes operadores relacionais podem ser definidos todos à custa do operador <. É instrutivo ver como, sobretudo no caso desconcertantemente simples do operador >:

```
/** Indica se o primeiro racional é maior que o segundo.
    @pre V.
    @post operator> = (r1 > r2). */
bool operator>(Racional const r1, Racional const r2)
{
    return r2 < r1;
}

/** Indica se o primeiro racional é menor ou igual ao segundo.
    @pre V.
    @post operator<= = (r1 ≤ r2). */
bool operator<=(Racional const r1, Racional const r2)
{
    return not (r2 < r1);
}

/** Indica se o primeiro racional é maior ou igual ao segundo.
    @pre V.
    @post operator>= = (r1 ≥ r2). */
```

```
bool operator>=(Racional const r1, Racional const r2)
{
    return not (r1 < r2);
}
```

Curiosamente (ou não), também os operadores `==` e `!=` se podem implementar à custa apenas do operador `<`. Fazê-lo fica como exercício para o leitor.

7.11 Constância: verificando erros durante a compilação

Uma boa linguagem de programação permite ao programador escrever programas com um mínimo de erros. Um bom programador que tira partido das ferramentas que a linguagem possui para reduzir ao mínimo os seus próprios erros.

Há três formas importantes de erros:

1. Erros lógicos. São erros devidos a um raciocínio errado do programador: a sua resolução do problema, incluindo algoritmos e estruturas de dados, ainda que correctamente implementados, não leva ao resultado pretendido, ou seja, na realidade não resolve o problema. Este tipo de erro é o mais difícil de corrigir. A facilidade ou dificuldade da sua detecção varia bastante conforme os casos, mas é comum que ocorram erros lógicos de difícil detecção.

2. Erros de implementação. Ao implementar a resolução do problema idealizada, foram cometidos erros não-sintácticos (ver abaixo), i.e., o programa não é uma implementação do algoritmo idealizado. Erros deste tipo são fáceis de corrigir, desde que sejam detectados. A detecção dos erros tanto pode ser fácil como muito difícil, por exemplo, quando os erros ocorrem em casos fronteira que raramente ocorrem na prática, ou quando o programa produz resultados que, sendo errados, parecem plausíveis.

3. Erros sintácticos. São "gralhas". O próprio compilador se encarrega de os detectar. São fáceis de corrigir.

Antes de um programa ser disponibilizado ao utilizador final, é testado. Antes de ser testado, é compilado. Antes de ser compilado, é desenvolvido. Com excepção dos erros durante o desenvolvimento, é claro que quanto mais cedo no processo ocorrerem os erros, mais fáceis serão de detectar e corrigir, e menor o seu impacte. Assim, uma boa linguagem é aquela que permite que os (inevitáveis) erros sejam sobretudo de compilação, detectados facilmente pelo compilador, e não de implementação ou lógicos, detectados com dificuldade pelo programador ou pelos utilizadores do programa.

Para evitar os erros lógicos, uma linguagem deve possuir uma boa biblioteca, que liberte o programador da tarefa ingrata, e sujeita a erros, de desenvolver algoritmos e estruturas de dados bem conhecidos. Mas como evitar os erros de implementação? Há muitos casos em que a linguagem pode ajudar. É o caso da possibilidade de usar constantes em vez de variáveis, que permite ao compilador detectar facilmente tentativas de alterar o seu valor, enquanto que a utilização de uma variável para o mesmo efeito impediria do compilador de detectar o erro, deixando esse trabalho nas mãos do programador. Outro caso é o do encapsulamento. A categorização de membros de uma classe C++ como privados permite ao compilador detectar tentativas erróneas de acesso a esses membros, coisa que seria impossível se os membros fossem públicos, recaindo sobre os ombros do programador consumidor da classe a responsabilidade de não aceder a determinados membros, de acordo com as especificações do programador produtor. Ainda outro caso é a definição das variáveis tão perto quando possível da sua primeira utilização, que permite evitar utilizações erróneas dessa variável antes do local onde é realmente necessária, e onde, se a variável for de um tipo básico, toma um valor arbitrário (lixo).

Assim, é conveniente usar os mecanismos da linguagem de programação que permitem exprimir no próprio código determinadas opções de implementação e condições de utilização, e que permitem que seja o próprio compilador a verificar do seu cumprimento, tirando esse peso dos ombros do programador, que pode por isso dedicar mais atenção a outros assuntos mais importantes. É o caso da classificação de determinadas instâncias como constantes, estudada nesta secção no âmbito da classe `Racional`.

7.11.1 Passagem de argumentos

Até agora viram-se duas formas de passagem de argumentos: por valor e por referência. Com a utilização da palavra-chave `const` as possibilidades passam a quatro, ou melhor, a três e meia...

A forma mais simples de passagem de argumentos é por valor. Neste caso os parâmetros são variáveis locais à rotina, inicializadas à custa dos argumentos respectivos. Ou seja, os parâmetros são *cópias* dos argumentos:

```
// Declaração:
TipoDeDevolução rotina(TipoDoParâmetro parâmetro);

// Definição:
TipoDeDevolução rotina(TipoDoParâmetro parâmetro)
{
    ...
}
```

É também possível que os parâmetros sejam constantes:

```
// Declaração:
TipoDeDevolução rotina(TipoDoParâmetro const parâmetro);

// Definição:
TipoDeDevolução rotina(TipoDoParâmetro const parâmetro)
{
    ...
}
```

No entanto, a diferença entre um parâmetro variável ou constante, no caso da passagem de argumentos por valor, não tem qualquer espécie de impacte sobre o código que invoca a rotina. Ou seja, para o programador consumidor da rotina é irrelevante se os parâmetros são variáveis ou constantes: o que lhe interessa é que serão cópias dos argumentos, que por isso não serão afectados pelas alterações que os parâmetros possam ou não sofrer[20]: a interface da rotina não é afectada, e as declarações

```
TipoDeDevolução rotina(TipoDoParâmetro parâmetro);
```

e

```
TipoDeDevolução rotina(TipoDoParâmetro const parâmetro);
```

são idênticas, pelo que se sói usar apenas a primeira forma, excepto quando for importante deixar clara a constância do parâmetro devido ao facto de ele ocorrer na condição objectivo da rotina, i.e., quando se quiser dizer que o parâmetro usado na condição objectivo tem o valor original, à entrada da rotina.

Já do ponto de vista do programador programador, ou seja, durante a definição da rotina, faz toda a diferença que o parâmetro seja constante: se o for, o compilador detectará tentativas de o alterar no corpo da rotina, protegendo o programador dos seus próprios erros no caso de a alteração do valor do parâmetro ser de facto indesejável.

Finalmente, note-se que a palavra-chave `const`, no caso da passagem de argumentos por valor, é eliminada automaticamente da assinatura da rotina, pelo que é perfeitamente possível que surja apenas na sua definição (implementação), sendo eliminada da declaração (interface):

```
// Declaração:
TipoDeDevolução rotina(TipoDoParâmetro parâmetro);

// Definição:
TipoDeDevolução rotina(TipoDoParâmetro const parâmetro)
{
    ... // Alteração de parâmetro proibida!
}
```

No caso da passagem por referência a palavra-chave `const` faz toda a diferença em qualquer caso, quer do ponto de vista da interface, quer do ponto de vista da implementação. Na passagem de argumentos por referência,

[20]Curiosamente é possível criar classes cujo construtor por cópia (ver Secção 7.4.2) altere o original! É normalmente muito má ideia fazê-lo, pois perverte a semântica usual da cópia, mas em alguns casos poderá ser uma prática justificada. É o caso do tipo genérico `auto_ptr`, da biblioteca padrão do C++. Mas mesmo no caso de uma classe C++ ter um construtor por cópia que altere o original, tal alteração ocorre durante a passagem de um argumento dessa classe C++ por valor, seja ou não o parâmetro respectivo constante, o que só vem reforçar a irrelevância para a interface de uma rotina de se usar a palavras chave `const` para qualificar parâmetros que não sejam referências.

```
// Declaração:
TipoDeDevolução rotina(TipoDoParâmetro& parâmetro);

// Definição:
TipoDeDevolução rotina(TipoDoParâmetro& parâmetro)
{
    ...
}
```

os parâmetros funcionam como *sinónimos* dos argumentos (ou *referências [variáveis]* para os argumentos). Assim, qualquer alteração de um parâmetro repercute-se sobre o argumento respectivo. Como neste tipo de passagem de argumentos não é realizada qualquer cópia, ela tende a ser mais eficiente que a passagem por valor, pelo menos para tipos em que as cópias são onerosas computacionalmente, o que não é o caso dos tipos básicos da linguagem. Por outro lado, este tipo de passagem de argumentos proíbe a passagem como argumento de constantes, como é natural, mas também de variáveis temporárias, tais como resultados de expressões que não sejam *lvalues* (ver Secção 7.7.1). Este facto impede a passagem de argumentos de tipos diferentes dos parâmetros mas para os quais exista uma conversão implícita.

Quando a passagem de argumentos se faz por referência constante,

```
// Declaração:
TipoDeDevolução rotina(TipoDoParâmetro const& parâmetro);

// Definição:
TipoDeDevolução rotina(TipoDoParâmetro const& parâmetro)
{
    ...
}
```

os parâmetros funcionam como *sinónimos constantes* dos argumentos (ou *referências constantes* para os parâmetros). Sendo constantes, as alterações aos parâmetros são proibidas. Por um lado, a passagem de argumentos por referência constante é semelhante à passagem por valor, pois não só impossibilita alterações aos argumentos como permite que sejam passados valores constantes e temporários como argumentos e que estes sofram conversões implícitas: uma referência constante pode ser sinónimo tanto de uma variável como de uma constante, pois uma variável pode sempre ser tratada como uma constante (e não o contrário), e pode mesmo ser sinónimo de uma variável ou constante *temporária*. Por outro lado, este tipo de passagem de argumentos é semelhante à passagem de argumentos por referência simples, pois não obriga à realização de cópias.

Ou seja, a passagem de argumentos por referência constante tem a vantagem das passagens por referência, ou seja, a sua maior eficiência na passagem de tipos não básicos, e a vantagens da passagem por valor, ou seja, a impossibilidade de alteração do argumento através do respectivo parâmetro e a possibilidade de passar instâncias (variáveis ou constantes) temporárias ou não. Assim, como regra geral, é sempre recomendável a passagem de argumentos por referência constante, em detrimento da passagem por valor, quando estiverem em causa tipos não básicos e quando não houver necessidade por alguma razão de alterar o valor do parâmetro durante a execução da rotina em causa.

Esta regra deve ser aplicada de forma sistemática às rotinas membro e não-membro desenvolvidas, no caso deste capítulo às rotinas associadas ao TAD `Racional` em desenvolvimento. A título de exemplo mostra-se a sua utilização na sobrecarga dos operadores `+=`, `/=` e `+` para a classe C++ `Racional`:

```
...

class Racional {
  public:

    ...

    /** Adiciona de um racional.
        @pre *this = r.
        @post operator+= ≡ *this ∧ *this = r + r2. */
    Racional& operator+=(Racional const& r2);

    /** Divide por um racional.
        @pre *this = r ∧ r2 ≠ 0.
        @post operator/= ≡ *this ∧ *this = r/r2. */
    Racional& operator/=(Racional const& r2);

   ...
};

...

Racional& Racional::operator+=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    numerador = numerador * r2.denominador +
        r2.numerador * denominador;
    denominador *= r2.denominador;

    reduz();

    assert(cumpreInvariante());

    return *this;
}

Racional& Racional::operator/=(Racional const& r2)
{
```

```
    assert(cumpreInvariante() and r2.cumpreInvariante());

    assert(r2 != 0);

    int numerador2 = r2.numerador_;
    if(r2.numerador_ < 0) {
        numerador_ *= -r2.denominador_;
        denominador_ *= -numerador2;
    } else {
        numerador_ *= r2.denominador_;
        denominador_ *= numerador2;
    }

    reduz();

    assert(cumpreInvariante());

    return *this;
}

...

/** Adição de dois racionais.
    @pre r1 = r1.
    @post operator+ = r1 + r2. */
Racional operator+(Racional r1, Racional const& r2)
{
    return r1 += r2;
}

...
```

Preservou-se a passagem por valor do primeiro argumento do operador + por ser desejável que nesse caso o parâmetro seja uma cópia do argumento, de modo a sobre ele se poder utilizar o operador +=.

É de notar uma alteração importante à definição da sobrecarga do operador `/=`: passou a ser feita um cópia do numerador do segundo operando, representado pelo parâmetro `r2`. É fundamental fazê-lo para que o código tenha o comportamento desejável no caso de se invocar o operador da seguinte forma:

```
r /= r;
```

Fica como exercício para o leitor verificar que o resultado estaria longe do desejado se esta alteração não tivesse sido feita (dica: a variável implícita e a variável da qual `r2` é sinónimo são a mesma variável).

7.11.2 Constantes implícitas: operações constantes

É possível definir constantes de um TAD concretizado à custa de uma classe C++. Por exemplo, para a classe C++ `Racional` é possível escrever o código

```
Racional const um_terço(1, 3);
```

que define uma constante `um_terço`. O problema está em que, tal como a classe C++ `Racional` está definida, esta constante praticamente não se pode usar. Por exemplo, o código

```
cout << "O denominador é " << um_terço.denominador() << endl;
```

resulta num erro de compilação.

A razão para o erro é simples: o compilador assume que as operações da classe C++ `Racional` alteram a instância implícita, ou seja, assume que as operações têm sempre uma variável e não uma constante implícita. Assim, como o compilador assume que há a possibilidade de a constante `um_terço` ser alterada, o que é um contra-senso, simplesmente proíbe a invocação da operação inspectora `Racional::denominador()`.

Note-se que o mesmo problema já existia no código desenvolvido: repare-se na rotina que sobrecarrega o operador ==, por exemplo:

```
/** Indica se dois racionais são iguais.
    @pre V.
    @post operator== = (r1 = r2). */
bool operator==(Racional const& r1, Racional const& r2)
{
    return r1.numerador() == r2.numerador() and
           r1.denominador() == r2.denominador();
}
```

Os parâmetros `r1` e `r2` desta rotina funcionam como sinónimos constantes dos respectivos argumentos (que podem ser constantes ou não). Logo, o compilador assinala um erro no corpo desta rotina ao se tentar invocar os inspectores `Racional::numerador()` e `Racional::denominador()` através das duas constantes: o compilador não adivinha que uma operação não altera a instância implícita. Aliás, nem o poderia fazer, pois muitas vezes no código que invoca a operação o compilador não tem acesso ao respectivo método, como se verá no 9, pelo que não pode verificar se de facto assim é.

Logo, é necessário indicar explicitamente ao compilador quais as operações que não alteram a instância implícita, ou seja, quais as operações que tratam a instância implícita como uma constante implícita. Isso consegue-se acrescentando a palavra-chave `const` ao cabeçalho das operações em causa e respectivos métodos, pois esta palavra-chave passará a fazer parte da respectiva assinatura (o que permite sobrecarregar uma operação com o mesmo nome e lista de parâmetros onde a única coisa que varia é a constância da instância implícita). Por exemplo, o inspector `Racional::numerador()` deve ser qualificado como não alterando a instância implícita:

```
...

class Racional {
  public:

    ...

    /** Devolve numerador da fracção canónica correspondente ao racional.
        @pre V.
        @post numerador / denominador() = *this. */
    int numerador() const;

    ...
};

...

int Racional::numerador() const
{
    assert(cumpreInvariante());

    assert(cumpreInvariante());
    return numerador_;
}

...
```

É importante perceber que o compilador verifica se no método correspondente a uma *operação constante*, que é o nome que se dá a uma operação que garante a constância da instância implícita, se executa alguma instrução que possa alterar a constante implícita. Isso significa que o compilador proíbe a invocação de operações não-constantes através da constante implícita e também que proíbe a alteração dos atributos, pois os atributos de uma constante assumem-se também constantes!

É o facto de a constância da instância implícita ser agora claramente indicada através do qualificador `const` e garantida pelo compilador que permitiu deixar de explicitar essa constância através de um termo extra na pré-condição e na condição objectivo: a constância da instância implícita continua a estar expressa no contrato destas operações, mas agora não na pré-condição e na condição objectivo mas na própria sintaxe do cabeçalho das operações[21].

Todas as operações inspectoras são naturalmente operações constantes. Embora também seja comum dizer-se que as operações constantes são inspectoras, neste texto reserva-se o nome inspector para as operações que devolvam propriedades da instância para a qual são invocados. Pelo contrário, às operações que alteram a instância implícita, ou que a permitem alterar

[21]Exactamente da mesma forma que as pré-condições não se referem normalmente ao tipo dos parâmetros (e.g., "o primeiro parâmetro tem de ser um `int`"), pois esse facto é expresso na própria linguagem C++ e garantido pelo compilador (bem, quase sempre, como se verá quando se distinguir tipo estático de tipo dinâmico...).

indirectamente, chama-se normalmente *operações modificadoras*, embora também seja possível distinguir entre várias categorias de operações não-constantes.

Resta, pois, qualificar como constantes todas as operações e respectivos métodos que garantem a constância da instância implícita:

```
...

class Racional {
  public:

    ...

    /** Devolve numerador da fracção canónica correspondente ao racional.
        @pre V.
        @post numerador / denominador() = *this. */
    int numerador() const;

    /** Devolve denominador da fracção canónica correspondente ao racional.
        @pre V.
        @post (E n : V : n / denominador = *this ∧ 0 < denominador ∧ mdc(n, denominador) = 1). */
    int denominador() const;

   /** Escreve o racional no ecrã no formato de uma fracção.
        @pre V.
        @post ¬cout∨ cout contém n/d (ou simplesmente n se d = 1)
              sendo n/d a fracção canónica correspondente ao racional *this. */
    void escreve() const;

    ...

  private:

    ...

    /** Indica se a condição invariante de classe se verifica.
        @pre V.
        @post cumpreInvariante = (0 <
denominador_ ∧ mdc(numerador_, denominador_) = 1). */
    bool cumpreInvariante() const;

    ...

};

...
```

```
int Racional::numerador() const
{
    assert(cumpreInvariante());

    return numerador_;
}

int Racional:: denominador() const
{
    assert(cumpreInvariante());

    return denominador_;
}

void Racional::escreve() const
{
    assert(cumpreInvariante());

    cout << numerador_;
    if(denominador_ != 1)
        cout << '/' << denominador_;
}

...

bool Racional::cumpreInvariante() const
{
    return 0 < denominador_ and mdc(numerador_, denominador_) == 1;
}

...
```

Note-se que nas operações que garantem a constância da instância implícita, tendo-se verificado a veracidade da condição invariante de classe no seu início, não é necessário voltar a verificá-la no seu final. Note-se também que, pela sua natureza, a operação que indica se a condição invariante de instância se verifica, tipicamente chamada `cumpreInvariante()`, é uma operação constante.

É interessante verificar que uma classe C++ tem duas interface distintas. A primeira, mais pequena, é a interface disponível para utilização com constantes dessa classe, e consiste no conjunto das operações que garantem a constância da instância implícita. A segunda, que engloba a primeira, é a interface disponível para utilização com variáveis da classe.

Finalmente, é muito importante pensar logo nas operações de uma classe como sendo ou não constantes, ou melhor, como garantindo ou não a constância da instância implícita, e não fazê-lo à posteriori, como neste capítulo! O desenvolvimento do TAD `Racional` feito neste capítulo não é feito pela ordem mais apropriada na prática (para isso ver o próximo capítulo), mas sim pela ordem que se julgou mais conveniente pedagogicamente para introduzir os muitos conceitos associados a classes C++ que o leitor tem de dominar para as desenhar com proficiência.

7.11.3 Devolução por valor constante

Outro assunto relacionado com a constância é a devolução de constantes. A ideia de devolver constantes pode parecer estranha à primeira vista, mas repare-se no seguinte código:

```
Racional r1(1, 2), r2(3, 2);
++(r1 + r2);
```

Que faz ele? Define duas variáveis `r1` e `r2`, soma-as, e finalmente *incrementa a variável temporária* devolvida pelo operador `+`. Tal código é mais provavelmente fruto de erro do programador do que algo desejado. Além disso, semelhante código seria proibido se em vez de racionais as variáveis fossem do tipo `int`. Como se pretende que o TAD `Racional` possa ser usado como qualquer tipo básico da linguagem, é desejável encontrar uma forma de proibir a invocação de operações modificadoras através de instâncias temporárias.

À luz da discussão na secção anterior, é fácil perceber que o problema se resolve se as funções que devolvem instâncias temporárias de classes C++, i.e., as funções que devolvem instâncias de classes C++ por valor, forem alteradas de modo a devolverem constantes temporárias, e não variáveis. No caso do TAD em desenvolvimento, são apenas as rotinas que sobrecarregam os operadores aritméticos usuais e os operadores de incrementação e decrementação sufixo que precisam de ser alteradas:

```
/** Adição de dois racionais.
    @pre r1 = r₁∧.
    @post operator+ = r₁ + r2. */
Racional const operator+(Racional r1, Racional const& r2)
{
    return r1 += r2;
}

/** Subtracção de dois racionais.
    @pre r1 = r₁∧.
    @post operator- = r₁ − r2. */
Racional const operator-(Racional r1, Racional const& r2)
{
    return r1 -= r2;
}

/** Produto de dois racionais.
    @pre r1 = r₁.
    @post operator* = r₁ × r2. */
```

```
Racional const operator*(Racional r1, Racional const& r2)
{
    return r1 *= r2;
}

/** Divisão de dois racionais.
    @pre r1 = r_1 ∧ r2 ≠ 0.
    @post operator/ = r_1/r2. */
Racional const operator/(Racional r1, Racional const& r2)
{
    assert(r2 != 0);

    return r1 /= r2;
}

/** Incrementa o racional recebido como argumento, devolvendo o seu valor an-
tes de incrementado.
    @pre *this = r.
    @post operator++ = r ∧ *this = r + 1. */
Racional const operator++(Racional& r, int valor_a_ignorar)
{
    Racional const cópia = r;

    ++r;

    return cópia;
}

/** Decrementa o racional recebido como argumento, devolvendo o seu valor an-
tes de decrementado.
    @pre *this = r.
    @post operator- = r ∧ *this = r − 1. */
Racional const operator--(Racional& r, int)
{
    Racional const cópia = r;

    --r;

    return cópia;
}

...
```

Ficaram a faltar ao TAD `Racional` os operadores + e − unários. Começar-se-á pelo segundo.

O operador − unário pode ser sobrecarregado quer através de uma operação da classe C++ `Racional`

```
...

class Racional {
  public:

    ...

    /** Devolve simétrico do racional.
        @pre V.
        @post operator- = −*this. */
    Racional const operator-() const;

    ...
};

...

Racional const Racional::operator-() const
{
    assert(cumpreInvariante());

    Racional r;
    r.numerador_ = -numerador_;
    r.denominador_ = denominador_;

    assert(r.cumpreInvariante());

    return r;
}

...
```

quer através de uma função normal

```
Racional const operator-(Racional const& r)
{
    return Racional(-r.numerador(), r.denominador());
}
```

Embora a segunda versão seja muito mais simples, ela implica a invocação do construtor mais complicado da classe C++, que verifica o sinal do denominador e reduz a fracção correspondente ao numerador e denominador passados como argumento. Neste caso essas verificações são inúteis, pois o denominador não varia, mantendo-se positivo, e mudar o sinal do numerador mantém numerador e denominador mutuamente primos. Assim, é preferível a primeira versão, onde se constrói um racional usando o construtor por omissão, que é muito eficiente, e em seguida se alteram directamente e sem mais verificações os valores do numerador e denominador. Em qualquer dos casos é devolvido um racional por valor e, por isso, constante.

7.11.4 Devolução por referência constante

Em alguns casos também é possível utilizar devolução por referência constante. Esta têm a vantagem de ser mais eficiente do que a devolução por valor, podendo ser utilizada quando o valor a devolver não for uma variável local à função, nem uma instância temporária construída dentro da função, pois tal redundaria na devolução de um sinónimo constante de uma instância entretanto destruída... É o caso do operador + unário que, por uma questão de simetria, se sobrecarrega por intermédio de uma operação da classe C++ `Racional`:

```
...

class Racional {
  public:

    ...

    /** Devolve versão constante do racional.
        @pre V.
        @post operator+ ≡ *this. */
    Racional const& operator+() const;

    ...
};

...

Racional const& Racional::operator+() const
{
    assert(cumpreInvariante());

    return *this;
}

...
```

Como contra exemplo, suponha-se que a rotina que sobrecarrega o operador ++ sufixo devolvia por referência constante:

```
/** Incrementa o racional recebido como argumento, devolvendo o seu valor an-
tes de incrementado.
    @pre *this = r.
    @post operator++ = r ∧ *this = r + 1. */
Racional const& operator++(Racional& r, int)
{
    Racional const cópia = r;
```

```
    ++r;

    return cópia; // Erro! Devolução de referência para variável local!
}
```

Seria claramente um erro fazê-lo, pois seria devolvida uma referência para uma instância local, que é destruída logo que a função retorna.

7.12 Reduzindo o número de invocações com `inline`

O mecanismo de invocação de rotinas (membro ou não) implica tarefas de "arrumação da casa" algo morosas, como se viu na Secção 3.4: é necessário colocar na pilha o endereço de retorno e os respectivo argumentos, executar as instruções do corpo da rotina, depois retirar os argumentos da pilha, e retornar, eventualmente devolvendo o resultado no seu topo. Logo, a invocação de rotinas pode ser, em alguns casos, um factor limitador da eficiência dos programas. Suponha-se as instruções:

```
Racional r(1, 3);
Racional s = r + 2;
```

Quantas invocações de rotinas são feitas neste código? A resposta é surpreendente, mesmo ignorando as instruções de asserção (que aliás podem ser facilmente "desligadas"):

1. O construtor para construir `r`, que invoca
2. a operação `Racional::reduz()`, cujo método invoca
3. a função `mdc()`.
4. O construtor para converter implicitamente o valor literal 2 num racional.
5. O construtor por cópia (ver Secção 7.4.2) para copiar o argumento `r` para o parâmetro `r1` durante a invocação d'
6. a função `operator+`, que invoca
7. a operação `Racional::operator+=`, cujo método invoca
8. a operação `Racional::reduz()`, cujo método invoca
9. a função `mdc()`.
10. O construtor por cópia para devolver `r1` por valor na função `operator+`.
11. O construtor por cópia para construir a variável `s` à custa da constante temporária devolvida pela função `operator+`.

Mesmo tendo em conta que o compilador pode eventualmente optimizar algumas destas invocações, 11 invocações para duas inocentes linhas de código parece demais. Não será lento? Como evitá-lo?

A linguagem C++ fornece uma forma simples de reduzir o peso da "arrumação da casa" aquando da invocação de uma rotina: rotinas muito simples, tipicamente não fazendo uso de ciclos e consistindo em apenas duas ou três linhas (excluindo instruções de asserção), podem qualificadas como *em-linha* ou *inline*. A palavra-chave `inline` pode ser usada para este efeito, qualificando-se com ela as definições das rotinas que se deseja que sejam em-linha.

Mas o que significa a definição de uma rotina ser em-linha? Que o compilador, se lhe parecer apropriado (e o compilador pode-se recusar a fazê-lo) em vez de traduzir o código da rotina em linguagem máquina, colocá-lo num único local do programa executável e chamá-lo quando necessário, coloca o código da rotina em linguagem máquina directamente nos locais onde ela deveria ser invocada.

Por exemplo, é natural que o código máquina produzido por

```
inline int soma(int const& a, int const& b)
{
    return a + b;
}

int main()
{
    int x1 = 10;
    int x2 = 20;
    int x3 = 30;
    int r = 0;

    r = soma(x1, x2); r = soma(r, x3);
}
```

seja idêntico ao produzido por

```
int main()
{
    int x1 = 10;
    int x2 = 20;
    int x3 = 30;
    int r = 0;

    r = x1 + x2;
    r = r + x3;
}
```

Para melhor compreender o que foi dito, é boa ideia fazer uma digressão pela linguagem *assembly*, aliás a única nestas folhas. Para isso recorrer-se-á à máquina MAC-1, desenvolvida por Andrew Tanenbaum para fins pedagógicos e apresentada em [13, Secção 4.3] (ver também MAC-1 asm `http://www.daimi.aau.dk/~bentor/html/useful/asm.html`).

A tradução para o *assembly* do MAC-1 do programa original é:

Se não levasse em conta o qualificador `inline`, um compilador de C++ para *assembly* MAC-1 poderia gerar:

```
        jump main

        # Variáveis:
        x1 = 10
        x2 = 20
        x3 = 30
        r = 0
main:   # Programa principal:
        lodd x1     # Carrega variável x1 no acumulador.
        push        # Coloca acumulador no topo da pilha.
        lodd x2     # Carrega variável x2 no acumulador.
        push        # Coloca acumulador no topo da pilha.
        # Aqui a pilha tem os dois argumentos x1 e x2:
        call soma   # Invoca a função soma().
        insp 2      # Repõe a pilha.
        # Aqui o acumulador tem o valor devolvido.
        stod r      # Guarda o acumulador na variável r.

        lodd r
        push
        lodd x3
        push
        # Aqui a pilha tem os dois argumentos r e x3:
        call soma
        insp 2
        # Aqui o acumulador tem o valor devolvido.
        stod r

        halt

soma:   # Função soma():
        lodl 1      # Carrega no acumulador o segundo parâmetro.
        addl 2      # Adiciona primeiro parâmetro ao acumulador.
        retn        # Retorna, devolvendo resultado no acumulador.
```

Se levasse em conta o qualificador `inline`, um compilador de C++ para *assembly* MAC-1 provavelmente geraria:

```
        jump main
```

```
      # Variáveis:

      x1 = 10
      x2 = 20
      x3 = 30
      r = 0

main: # Programa principal:
      lodd x1    # Carrega variável x1 no acumulador.
      addd x2    # Adiciona variável x2 ao acumulador.
      stod r     # Guarda o acumulador na variável r.

      lodd r
      addd x3
      stod r

      halt
```

A diferença entre os dois programas em *assembly* é notável. O segundo é claramente mais rápido, pois evita todo o mecanismo de invocação de funções. Mas também é mais curto, ou seja, ocupa menos espaço na memória do computador! Embora normalmente haja sempre um ganho em termos do número de instruções a efectuar, se o código a colocar em-linha for demasiado extenso, o programa pode-se tornar mais longo, o que pode inclusivamente levar ao esgotamento da memória física, levando à utilização da memória virtual do sistema operativo, que tem a lamentável característica de ser ordens de grandeza mais lenta. Assim, é necessário usar o qualificador `inline` com conta, peso e medida.

Para definir uma operação como em-linha, pode-se fazer uma de duas coisas:

1. Ao definir a classe C++, definir logo o método (em vez de a declarar apenas a respectiva operação).
2. Ao definir o método correspondente à operação declarada na definição da classe, preceder o seu cabeçalho do qualificador `inline`.

Em geral a segunda alternativa é preferível à primeira, pois torna mais evidente a separação entre a interface e a implementação da classe, separando claramente operações de métodos.

A definição de uma rotina, membro ou não-membro, como em-linha não altera a semântica da sua invocação, tendo apenas consequências em termos da tradução do programa para código máquina.

No caso do código em desenvolvimento neste capítulo, relativo ao TAD `Racional`, todas as rotinas são suficientemente simples para que se justifique a utilização do qualificador `inline`, com excepção apenas da função `mdc()`, por envolver um ciclo, e da operação `Racional::lê()`, por ser demasiado extensa. Exemplificando apenas para o primeiro construtor da classe:

```
...

class Racional {
  public:
    /** Constrói racional com valor inteiro. Construtor por omissão.
        @pre V.
        @post *this = n. */
    Racional(int const n = 0);

    ...
};

inline Racional::Racional(int const n)
    : numerador_(n), denominador_(1)
{

    assert(cumpreInvariante());
    assert(numerador_ == n * denominador_);
}

...
```

7.13 Optimização dos cálculos com racionais

Um dos problemas com a representação escolhida para a classe C++ `Racional` é o facto de os atributos `numerador_` e `denominador_` serem do tipo `int`, que tem limitações devidas à sua representação na memória do computador. Essa foi parte da razão pela qual se insistiu em que os racionais fossem sempre representados pelo numerador e denominador de uma fracção no formato canónico, i.e., com denominador positivo e formando uma fracção reduzida. No entanto, esta escolha não é suficiente. Basta olhar para a definição da função que sobrecarrega o operador < para a classe C++ `Racional`

```
/** Indica se o primeiro racional é menor que o segundo.
    @pre V.
    @post operator< = (r1 < r2). */
inline bool operator<(Racional const& r1, Racional const& r2)
{
    return r1.numerador() * r2.denominador() <
           r2.numerador() * r1.denominador();
}
```

para se perceber imediatamente que, mesmo que os racionais sejam representáveis, durante cálculos intermédios que os envolvam podem ocorrer transbordamentos. No entanto, embora seja impossível eliminar totalmente a possibilidade de transbordamentos (excepto eventualmente abandonando o tipo `int` e usando um TAD representando números inteiros de dimensão arbitrária), é possível minorar o seu impacte. Por exemplo, no caso do operador < é possível encontrar divisores comuns aos numeradores e aos denominadores dos racionais a comparar e usá-los para reduzir ao máximo a magnitude dos inteiros a comparar:

```
/** Indica se o primeiro racional é menor que o segundo.
    @pre V.
    @post operator< = (r1 < r2). */
inline bool operator<(Racional const& r1, Racional const& r2)
{
    int dn = mdc(r1.numerador(), r2.numerador());
    int dd = mdc(r1.denominador(), r2.denominador());

    return (r1.numerador() / dn) * (r2.denominador() / dd) <
           (r2.numerador() / dn) * (r1.denominador() / dd);
}
```

As mesmas ideias podem ser aplicadas a outras operações, pelo que se discutem nas secções seguintes. Durante estas secções admite-se que as fracções originais ($\frac{n}{d}$, $\frac{n_1}{d_1}$ e $\frac{n_2}{d_2}$) estão no formato canónico. Recorda-se também que se admite uma extensão da função mdc de tal forma que $\mathrm{mdc}(0,0) = 1$.

7.13.1 Adição e subtracção

O resultado da soma de fracções dado por

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 \times d_2 + n_2 \times d_1}{d_1 \times d_2},$$

embora tenha forçosamente o denominador positivo, pode não estar no formato canónico. Se $k = \mathrm{mdc}(d_1, d_2)$ e $l = \mathrm{mdc}(n_1, n_2)$, então, dividindo ambos os termos da fracção resultado por k e pondo l em evidência,

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{l \times (n_1' \times d_2' + n_2' \times d_1')}{k \times d_1' \times d_2'},$$

onde $d_1' = d_1/k$ e $d_2' = d_2/k$ são mutuamente primos, i.e., $\mathrm{mdc}(d_1', d_2') = 1$, e $n_1' = n_1/l$ e $n_2' = n_2/l$ são mutuamente primos, i.e., $\mathrm{mdc}(n_1', n_2') = 1$.

Este novo resultado, apesar da divisão por k de ambos os termos da fracção, pode ainda não estar no formato canónico, pois pode haver divisores não-unitários comuns ao numerador e ao denominador. Repare-se no exemplo

$$\frac{1}{10} + \frac{1}{15},$$

em que $k = \mathrm{mdc}(10, 15) = 5$. Aplicando a equação acima obtém-se

$$\frac{1}{10} + \frac{1}{15} = \frac{1 \times 3 + 1 \times 2}{5 \times 2 \times 3} = \frac{5}{30}.$$

Neste caso, para reduzir a fracção aos termos mínimos é necessário dividir ambos os termos da fracção por 5.

Em vez de tentar reduzir a fracção resultado tomando quer o numerador quer o denominador como um todo, é preferível verificar primeiro se é possível haver divisores comuns entre os respectivos factores. Considerar-se-ão dois factores para o numerador (l e $n'_1 \times d'_2 + n'_2 \times d'_1$) e dois factores para o denominador (k e $d'_1 \times d'_2$), num total de quatro combinações onde é possível haver divisores comuns.

Será que podem haver divisores não-unitários comuns a l e a k? Suponha-se que existe um divisor $1 < i$ comum a l e a k. Nesse caso, dado que $d_1 = d'_1 \times k$ e $n_1 = n'_1 \times l$, ter-se-ia de concluir que $i \leq \mathrm{mdc}(n_1, d_1)$, ou seja, $i \leq 1$, o que é uma contradição. Logo, l e a k não têm divisores comuns não-unitários.

Será que pode haver divisores não-unitários comuns a l e a $d'_1 d'_2$? Suponha-se que existe um divisor $1 < i$ comum a l e a $d'_1 d'_2$. Nesse caso, existe forçosamente um divisor $1 < j$ comum a l e a d'_1 ou a d'_2. Se j for divisor comum a l e a d'_1, então j é também divisor comum a n_1 e a d_1, ou seja, $j \leq \mathrm{mdc}(n_1, d_1)$, donde se conclui que $j \leq 1$, o que é uma contradição. O mesmo argumento se aplica se j for divisor comum a l e a d'_2. Logo, l e $d'_1 d'_2$ não têm divisores comuns não-unitários.

Será que podem haver divisores não-unitários comuns a $n'_1 \times d'_2 + n'_2 \times d'_1$ e a $d'_1 \times d'_2$? Suponha-se que existe um divisor $1 < h$ comum $n'_1 \times d'_2 + n'_2 \times d'_1$ e de $d'_1 \times d'_2$. Nesse caso, existe forçosamente um divisor $1 < i$ comum a $n'_1 \times d'_2 + n'_2 \times d'_1$ e a d'_1 ou a d'_2. Seja então $1 < i$ um divisor comum a $n'_1 \times d'_2 + n'_2 \times d'_1$ e a d'_1. Nesse caso tem de existir um divisor $1 < j$ comum a d'_1 e a n'_1 ou a d'_2. Isso implicaria que $j \leq \mathrm{mdc}(n_1, d_1)$ ou que $j \leq \mathrm{mdc}(d'_1, d'_2) \neq 1$. Em qualquer dos casos conclui-se que $j \leq 1$, o que é uma contradição. O mesmo argumento se aplica se $1 < i$ for divisor comum a $n'_1 \times d'_2 + n'_2 \times d'_1$ e a d'_2. Logo, $n'_1 \times d'_2 + n'_2 \times d'_1$ e $d'_1 \times d'_2$ não têm divisores comuns não-unitários.

Assim, a existirem divisores não-unitários comuns ao denominador e numerador da fracção

$$\frac{l \times (n'_1 \times d'_2 + n'_2 \times d'_1)}{k \times d'_1 \times d'_2},$$

eles devem-se à existência de divisores não-unitários comuns a $n'_1 \times d'_2 + n'_2 \times d'_1$ e a k. Assim, sendo

$$m = \mathrm{mdc}(n'_1 \times d'_2 + n'_2 \times d'_1, k),$$

a fracção

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{l \times ((n'_1 \times d'_2 + n'_2 \times d'_1)/m)}{(k/m) \times d'_1 \times d'_2},$$

está no formato canónico.

Qual foi a vantagem de factorizar l e k e proceder aos restantes cálculos face à alternativa, mais simples, de calcular a fracção como

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{(n_1 \times d_2 + n_2 \times d_1)/h}{(d_1 \times d_2)/h},$$

com $h = \mathrm{mdc}(n_1 \times d_2 + n_2 \times d_1, d_1 \times d_2)$?

A vantagem é meramente computacional. Apesar de os cálculos propostos exigirem mais operações, os valores intermédios dos cálculos são em geral mais pequenos, o que minimiza a possibilidade de existirem valores intermédios que não sejam representáveis em valores do tipo `int`, evitando-se assim transbordamentos.

A fracção canónica correspondente à adição pode ser portanto calculada pela equação acima. A fracção canónica correspondente à subtracção pode ser calculada por uma equação semelhante

$$\frac{n_1}{d_1} - \frac{n_2}{d_2} = \frac{l \times ((n'_1 \times d'_2 - n'_2 \times d'_1)/m)}{(k/m) \times d'_1 \times d'_2}.$$

Pode-se agora actualizar a definição dos métodos `Racional::operator+=` e `Racional::operator-=` para:

```
Racional& Racional::operator+=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    int dn = mdc(numerador_, r2.numerador_);
    int dd = mdc(denominador_, r2.denominador_);

    // Devido a r += r:
    int d2 = r2.denominador_;

    numerador_ /= dn;
    denominador_ /= dd;

    numerador_ = numerador_ * (d2 / dd) +
        r2.numerador_ / dn * denominador_;

    dd = mdc(numerador_, dd);

    numerador_ = dn * (numerador_ / dd);
    denominador_ *= d2 / dd;

    assert(cumpreInvariante());

    return *this;
}
```

```
Racional& Racional::operator-=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    int dn = mdc(numerador_, r2.numerador_);
    int dd = mdc(denominador_, r2.denominador_);

    // Devido a r -= r:
    int d2 = r2.denominador_;

    numerador_ /= dn;
    denominador_ /= dd;

    numerador_ = numerador_ * (d2 / dd) -
        r2.numerador_ / dn * denominador_;

    dd = mdc(numerador_, dd);

    numerador_ = dn * (numerador_ / dd);
    denominador_ *= d2 / dd;

    assert(cumpreInvariante());

    return *this;
}
```

Uma vez que ambos os métodos ficaram bastante extensos, decidiu-se retirar-lhes o qualificador `inline`.

7.13.2 Multiplicação

Relativamente à multiplicação de fracções,

$$\frac{n_1}{d_1} \times \frac{n_2}{d_2} = \frac{n_1 \times n_2}{d_1 \times d_2},$$

apesar de o denominador ser forçosamente positivo, é possível que o resultado não esteja no formato canónico, bastando para isso que existam divisores não-unitários comuns a n_1 e d_2 ou a d_1 e n_2. É fácil verificar que, sendo $k = \mathrm{mdc}(n_1, d_2)$ e $l = \mathrm{mdc}(n_2, d_1)$, a fracção

$$\frac{n_1}{d_1} \times \frac{n_2}{d_2} = \frac{(n_1/k) \times (n_2/l)}{(d_1/l) \times (d_2/k)}$$

está, de facto, no formato canónico.

Pode-se agora actualizar a definição do método `Racional::operator*=` para:

```
inline Racional& Racional::operator*=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    int n1d2 = mdc(numerador_, r2.denominador_);
    int n2d1 = mdc(r2.numerador_, denominador_);

    numerador_ = (numerador_ / n1d2) * (r2.numerador_ / n2d1);
    denominador_ = (denominador_ / n2d1) * (r2.denominador_ / n1d2);

    assert(cumpreInvariante());

    return *this;
}
```

7.13.3 Divisão

O caso da divisão de fracções,

$$\frac{n_1}{d_1} / \frac{n_2}{d_2} = \frac{n_1 \times d_2}{d_1 \times n_2} \text{ se } n_2 \neq 0,$$

é muito semelhante ao da multiplicação, sendo mesmo possível usar os métodos acima para a calcular. Em primeiro lugar é necessário garantir que $n_2 \neq 0$. Se $n_2 = 0$ a divisão não está definida. Admitindo que $n_2 \neq 0$, então a divisão é equivalente a uma multiplicação:

$$\frac{n_1}{d_1} / \frac{n_2}{d_2} = \frac{n_1}{d_1} \times \frac{d_2}{n_2}.$$

No entanto, é necessário verificar se n_2 é positivo, pois de outra forma o resultado da multiplicação não estará no formato canónico, uma vez que terá denominador negativo. Se $0 < n_2$, a divisão é calculada multiplicando as fracções canónicas $\frac{n_1}{d_1}$ e $\frac{d_2}{n_2}$. Se $n_2 < 0$, multiplicam-se as fracções canónicas $\frac{n_1}{d_1}$ e $\frac{-d_2}{-n_2}$. Para garantir que o resultado está no formato canónico usa-se a mesma técnica que para a multiplicação.

Pode-se agora actualizar a definição do método `Racional::operator/=` para:

```
inline Racional& Racional::operator/=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    assert(r2 != 0);

    int dn = mdc(numerador_, r2.numerador_);
    int dd = mdc(denominador_, r2.denominador_);

    if(r2.numerador_ < 0) {
```

```
            numerador_ = (numerador_ / dn) * (-r2.denominador_ / dd);
            denominador_ = (denominador_ / dd) * (-r2.numerador_ / dn);
        } else {
            numerador_ = (numerador_ / dn) * (r2.denominador_ / dd);
            denominador_ = (denominador_ / dd) * (r2.numerador_ / dn);
        }

        assert(cumpreInvariante());

        return *this;
    }
```

7.13.4 Simétrico e identidade

O caso das operações de cálculo do simétrico e da identidade,

$$\begin{aligned} -\frac{n}{d} &= \frac{-n}{d} \text{ e} \\ +\frac{n}{d} &= \frac{n}{d}, \end{aligned}$$

não põe qualquer problema, pois os resultados estão sempre no formato canónico.

7.13.5 Operações de igualdade e relacionais

Sendo dois racionais r_1 e r_2 e as respectivas representações na forma de fracções canónicas $r_1 = \frac{n_1}{d_1}$ e $r_2 = \frac{n_2}{d_2}$, é evidente que $r_1 = r_2$ se e só se $n_1 = n_2 \wedge d_1 = d_2$. Da mesma forma, $r_1 \neq r_2$ se e só se $n_1 \neq n_2 \vee d_1 \neq d_2$.

Relativamente aos mesmos dois racionais, a expressão $r_1 < r_2$ é equivalente a $\frac{n_1}{d_1} < \frac{n_2}{d_2}$ ou ainda a $n_1 d_2 < n_2 d_1$, pois ambos os denominadores são positivos. Assim, é possível comparar dois racionais usando apenas comparações entre inteiros. Os inteiros a comparar podem ser reduzidos calculando $k = \mathrm{mdc}(d_1, d_2)$ e $l = \mathrm{mdc}(n_1, n_2)$ e dividindo os termos apropriados da expressão:

$$(n_1/l)\,(d_2/k) < (n_2/l)\,(d_1/k)\,.$$

Da mesma forma podem-se reduzir todas as comparações entre racionais (com $<$, $>$, $\geq$ ou $\leq$) às correspondentes comparações entre inteiros.

Pode-se agora actualizar a definição da rotina `operator<`:

```
/** Indica se o primeiro racional é menor que o segundo.
    @pre V.
    @post operator< = (r1 < r2). */
inline bool operator<(Racional const& r1, Racional const& r2)
{
    int dn = mdc(r1.numerador(), r2.numerador());
```

```
    int dd = mdc(r1.denominador(), r2.denominador());

    return (r1.numerador() / dn) * (r2.denominador() / dd) <
           (r2.numerador() / dn) * (r1.denominador() / dd);
}
```

7.13.6 Operadores especiais

O TAD `Racional` tal como concretizado até agora, suporta operações simultâneas entre racionais e inteiros, sendo para isso fundamental a conversão implícita entre valores do tipo `int` e o tipo `Racional` fornecida pelo primeiro construtor da respectiva classe C++. No entanto, é instrutivo seguir a ordem dos acontecimentos quando se calcula, por exemplo, a soma de um racional com um inteiro:

```
Racional r(1, 2);

cout << r + 1 << endl;
```

A soma implica as seguintes invocações:

1. Construtor da classe C++ `Racional` para converter o inteiro 1 no correspondente racional.
2. Rotina `operator+()`.
3. Operação `Racional::operator+=()`.

Será possível evitar a conversão do inteiro em racional e, sobretudo, evitar calcular a soma de um racional com um inteiro recorrendo à complicada maquinaria necessária para somar dois racionais? Certamente. Basta fornecer versões especializadas para operandos inteiros das sobrecargas dos operadores em causa:

```
...

class Racional {
  public:

    ...

    /** Adiciona de um inteiro.
        @pre *this = r.
        @post operator+= ≡ *this ∧ *this = r + n. */
    Racional& operator+=(int const n);

    ...
```

```
};

...

Racional& Racional::operator+=(int const i)
{
    assert(cumpreInvariante());

    numerador_ += i * denominador_;

    assert(cumpreInvariante());

    return *this;
}

/** Adição de um racional e um inteiro.
    @pre r = r.
    @post operator+ = r + i. */
inline Racional const operator+(Racional r, int const i)
{
    return r += i;
}

/** Adição de um inteiro e um racional.
    @pre r = r.
    @post operator+ = i + r. */
inline Racional const operator+(int const i, Racional r)
{
    return r += i;
}
```

Fica como exercício para o leitor desenvolver as sobrecargas de operadores especializadas em todos os outros casos em que se possam fazer operações conjuntas entre inteiros e racionais.

7.14 Operadores de inserção e extracção

É possível e desejável sobrecarregar o operador << de inserção num canal e o operador >> de extracção de um canal. Aliás, estas sobrecargas são, digamos, a cereja sobre o bolo. São o toque final que permite escrever o programa da soma de racionais exactamente da mesma forma como se faria se se pretendesse somar inteiros:

```
int main()
{
    // Ler fracções:
```

```
    cout << "Introduza duas fracções (numerador denominador): ";
    Racional r1, r2;
    cin >> r1 >> r2;

    if(not cin) {
        cerr << "Opps!  A leitura dos racionais falhou!" << endl;
        return 1;
    }

    // Calcular racional soma:
    Racional r = r1 + r2;

    // Escrever resultado:
    cout << "A soma de " << r1 << " com " << r2
         << " é " << r << '.' << endl;
}
```

7.14.1 Sobrecarga do operador <<

Começar-se-á pelo operador de inserção, por ser mais simples. O operador << é binário, tendo por isso dois operandos. Por exemplo, se se pretender escrever um valor inteiro no ecrã pode-se usar a instrução

```
cout << 10;
```

onde o primeiro operando, `cout`, é um canal de saída, ligado normalmente ao ecrã, e 10 é um valor literal inteiro. O efeito da operação é fazer surgir o valor 10 no ecrã. O segundo operando é claramente do tipo `int`, mas, qual é o tipo do primeiro operando? Aliás, o que é `cout`? A resposta a estas perguntas é muito importante para a sobrecarga do operador << desejada: o primeiro operando, `cout` é uma variável global do tipo `ostream`, ou seja, *canal de saída*. Ambos, variável `cout` e tipo `ostream`, estão declarados no ficheiro de interface[22] `iostream`. Por outro lado, o operador << é um operador binário como qualquer outro, e por isso tem associatividade à esquerda. Isso quer dizer que a instrução

```
cout << 10 << ' ';
```

é interpretada como

```
(cout << 10) << ' ';
```

A primeira operação com o operador << faz-se, por isso, com o primeiro operando do tipo `ostream` e o segundo do tipo `char`. A segunda operação com o operador << faz-se claramente com o segundo operando do tipo `char`. De que tipo será o primeiro operando nesse

[22]Um ficheiro de interface é usado na modularização física para permitir a um módulo físico (ficheiro) usar ferramentas definidas noutro módulo físico. Estes assuntos são matéria do Capítulo 9.

caso? E que valor possui? As respostas são evidentes se se lembrar que a instrução acima é equivalente a

```
cout << 10;
cout << ' ';
```

É claro que o primeiro operando da segunda operação com o operador << tem de ser não apenas do tipo `ostream`, para que o segundo operando seja inserido num canal, mas deve ser exactamente o canal `cout`, para que a inserção se faça no local correcto.

A sobrecarga do operador << não se pode fazer à custa de uma operação da classe `Racional`, pois o primeiro operando do operador deve ser do tipo `ostream`. Sendo este tipo uma classe, quando muito o operador << poderia ser sobrecarregado por uma operação dessa classe. Mas como a classe está pré-definida na biblioteca padrão do C++, não é possível fazê-lo. Assim, a sobrecarga será feita usando uma rotina normal, e por isso com dois parâmetros. O primeiro parâmetro corresponderá ao canal onde se deve realizar a operação de inserção, e o segundo ao racional a inserir. Como o canal é certamente modificado pela operação de inserção, terá de ser passado por referência:

```
tipo_de_devolução operator<<(ostream& saída, Racional const& r);
```

O tipo de devolução, para que o canal seja passado sucessivamente em instruções de inserção encadeadas tais como

```
Racional r1, r2;

...

cout << r1 << ' ' << r2 << endl;
```

terá naturalmente de ser `ostream&`, ou seja, o canal terá de ser devolvido por referência. Aliás, a justificação para o fazer é idêntica à que se usou para os operadores de incrementação e decrementação prefixo e para os operadores especiais de atribuição. Ou seja, a definição da rotina `operator<<` deverá tomar a forma:

```
/** Insere o racional no canal de saída no formato de uma fracção.
    @pre V.
    @post ¬saída∨ saída contém n/d (ou simplesmente n se d = 1)
          sendo n/d a fracção canónica correspondente ao racional r. */
ostream& operator<<(ostream& saída, Racional const& r)
{
    ...

    return saída;
}
```

Dada a existência de operações de inspecção que permitem obter o numerador e o denominador da fracção canónica correspondente a um racional, seria perfeitamente possível eliminar a operação `Racional::escreve()` da classe C++ `Racional` e usar o seu corpo, com adaptações, como corpo da rotina `operator<<`. No entanto, adoptar-se-á uma solução diferente. Manter-se-á a operação `Racional::escreve()`, embora com um nome mais apropriado, e implementar-se-á a rotina `operator<<` à sua custa. Para isso é fundamental tornar a operação mais genérica, de modo a inserir o racional num canal arbitrário:

```
...

class Racional {
  public:
    ...

   /** Insere o racional no canal no formato de uma fracção.
        @pre V.
        @post ¬saída∨ saída contém n/d (ou simplesmente n se d = 1)
              sendo n/d a fracção canónica correspondente ao racional *this. */
    void insereEm(ostream& saída) const;

    ...
};

...

...
inline void Racional::insereEm(ostream& saída) const
{
    assert(cumpreInvariante());

    saída << numerador_;
    if(denominador_ != 1)
        saída << '/' << denominador_;
}

...

/** Insere o racional no canal de saída no formato de uma fracção.
    @pre V.
    @post ¬saída∨ saída contém n/d (ou simplesmente n se d = 1)
          sendo n/d a fracção canónica correspondente ao racional r. */
ostream& operator<<(ostream& saída, Racional const& r)
{
    r.insereEm(saída);

    return saída;
```

```
}

...
```

7.14.2 Sobrecarga do operador > >

O caso do operador > > é muito semelhante, embora neste caso o primeiro operando do operador seja do tipo `istream`, ou seja, *canal de entrada*, e se deva passar o racional por referência, para permitir a sua alteração:

```
...

class Racional {
  public:
    ...

    /** Extrai do canal um novo valor para o racional, na forma de dois inteiros sucessivos.
        @pre *this = r.
        @post Se entrada e entrada tem dois inteiros n' e d' disponíveis para leitura, com d' ≠ 0, então
                *this = n'/d' ∧ entrada,
              senão
                *this = r ∧ ¬entrada. */
    void extraiDe(istream& entrada);

    ...
};

...

...
void Racional::extraiDe(istream& entrada)
{
    assert(cumpreInvariante());

    int n, d;

    if(entrada >> n >> d)
        if(d == 0)
            entrada.setstate(ios_base::failbit);
        else {
            numerador_ = d < 0 ? -n : n;
            denominador_ = d < 0 ? -d : d;
```

```
            reduz();

            assert(cumpreInvariante());
            assert(numerador_ * d == n * denominador_ and cin);

            return;
        }

    assert(cumpreInvariante());
    assert(not entrada);
}

...

/** Extrai do canal um novo valor para o racional, na forma de dois inteiros sucessivos.
    @pre r = r.
    @post Se entrada e entrada tem dois inteiros n′ e d′ disponíveis para lei-
tura, com d′ ≠ 0, então
            r = n′/d′ ∧ entrada,
        senão
            r = r ∧ ¬entrada. */
istream& operator>>(istream& entrada, Racional& r)
{
    r.extraiDe(entrada);

    return entrada;
}

...
```

Neste caso a vantagem de implementar a rotina `operator>>` à custa de uma operação correspondente na classe C++ fica mais clara. Como a rotina `operator>>` não pode ser membro da classe e, no entanto, necessita de alterar os atributos da classe, a solução foi delegar a tarefa da extracção ou leitura para uma operação da classe, que não tem quaisquer restrições de acesso.

7.14.3 Lidando com erros

No início deste capítulo apresentou-se a primeira versão da rotina de leitura de racionais, então vistos simplesmente como fracções, sem mais explicações. Chegou agora a altura de explicar o código dessa rotina, entretanto convertida na operação `Racional::extraiDe()`, apresentada abaixo sem instruções de asserção, i.e., reduzida ao essencial:

```
void Racional::extraiDe(istream& entrada)
{
     int n, d;
```

```
        if(entrada >> n >> d)
            if(d == 0)
                entrada.setstate(ios_base::failbit);
            else {
                numerador_ = d < 0 ? -n : n;
                denominador_ = d < 0 ? -d : d;

                reduz();
            }
    }
```

Em primeiro lugar, note-se que a extracção do numerador e do denominador se faz, não directamente para os respectivos atributos, mas para duas variáveis criadas para o efeito. De outra forma, se a extracção do numerador tivesse sucesso, mas a extracção do denominador falhasse, a extracção do racional como um todo teria falhado e este teria mudado de valor, o que, para além de ser má ideia, violaria o estabelecido no contrato da operação.

Usa-se o idioma do C++

```
if(entrada >> n >> d)
    ...
```

para fazer simultaneamente a extracção dos dois valores inteiros canal de entrada e verificar se essa leitura teve sucesso. Esta instrução poderia (e talvez devesse...) ser decomposta em duas:

```
entrada >> n >> d;
if(entrada)
    ...
```

A primeira destas instruções serve para fazer as duas extracções. Tal como se viu para o operador < <, a primeira instrução é interpretada como

```
(entrada >> n) >> d;
```

devido à associatividade esquerda do operador > >. Que acontece quando a primeira extracção falha, por exemplo quando no canal não se encontra uma sequência de dígitos interpretável como um número inteiro, mas sim, por exemplo, caracteres alfabéticos? Nesse caso a primeira extracção não tem qualquer efeito e o canal `entrada` fica em estado de erro. Nesse caso, a segunda falhará também, pois todas as operações de inserção e extracção realizadas sobre um canal em estado de erro estão condenadas ao fracasso. Ou seja, se a primeira operação falhar, a segunda não tem qualquer efeito, o que é equivalente a não ser realizada. Como o valor de um canal interpretado como um valor booleano é verdadeiro se o canal não estiver em estado de erro e falso se estiver, as instruções acima são equivalentes a

```
if(not entrada.fail()) {
    entrada >> n;
    if(not entrada.fail()) {
        entrada >> d;
        if(not entrada.fail())
            ...
    }
}
```

onde `istream::fail()` é uma operação da classe `istream` que indica se o canal está em estado de erro.

Uma vez realizadas as duas extracções com sucesso, é necessário verificar ainda assim se os valores lidos são aceitáveis. Neste caso isso corresponde a verificar se o denominador é nulo. Se for, a leitura deve falhar. I.e., o racional deve manter o valor original *e o canal deve ficar em estado de erro*, de modo a que a falha de extracção possa ser detectada mais tarde. Só dessa forma será possível, por exemplo, escrever o seguinte código:

```
Racional r;

r.extraiDe(cin);

if(not cin){
    cerr << "Opps!  A leitura falhou!" << endl;
    ...
}
```

ou mesmo

```
Racional r;

cin >> r;

if(not cin){
    cerr << "Opps!  A leitura falhou!" << endl;
    ...
}
```

utilizando o operador >> já sobrecarregado para os racionais. Pretende-se de novo que o comportamento de um programa usando racionais seja tão semelhante quanto possível do ponto de vista semântico com o mesmo programa usando inteiros.

Assim, no caso de se extrair um denominador nulo, deve-se colocar o canal `entrada` em estado de erro. Para o fazer usa-se a instrução[23]

[23]Também se pode limpar o estado de erro de um canal, usando-se para isso a operação `istream::clear()`:

```
entrada.clear();
```

```
entrada.setstate(ios_base::failbit);
```

A parte restante do código é auto-explicativa.

7.14.4 Coerência entre os operadores << e >>

Tal como sobrecarregados, os operadores << e >> para racionais, ou melhor, as operações `Racional::insereEm()` e `Racional::extraiDe()` não são coerentes: a extracção não suporta o formato usado pela inserção, nomeadamente não espera o símbolo '/' entre os termos da fracção, nem admite a possibilidade de o racional não ter denominador explícito, nomeadamente quando é também um valor inteiro. É excelente ideia que o operador de extracção consiga extrair racionais em qualquer dos formatos produzidos pelo operador de inserção. Para o conseguir é necessário complicar um pouco o método `Racional::extraiDe()`, e indicar claramente o novo formato admissível no contrato das rotinas envolvidas:

```
...

class Racional {
  public:
    ...

    /** Extrai do canal um novo valor para o racional, na forma de uma fracção.
        @pre *this = r.
        @post Se entrada e entrada contém n'/d' (ou apenas n', assumindo-
se d' = 1), em
            que n' e d' são inteiros com d' ≠ 0, então
                *this = n'/d' ∧ entrada,
            senão
                *this = r ∧ ¬entrada. */
    void extraiDe(istream& entrada);

    ...
};

...

...
void Racional::extraiDe(istream& entrada)
{
    assert(cumpreInvariante());

    int n;
    int d = 1;

    if(entrada >> n) {
```

```
        if(entrada.peek() != '/') {
            numerador_ = n;
            denominador_ = 1;
        } else {
            if(entrada.get() and isdigit(entrada.peek()) and
               entrada >> d and d != 0)
            {
                numerador_ = d < 0 ? -n : n;
                denominador_ = d < 0 ? -d : d;

                reduz();
            } else if(entrada)
                entrada.setstate(ios_base::failbit);
        }
    }

    assert(cumpreInvariante());

    assert(numerador_ * d == n * denominador_ or not entrada.good());
}

...

/** Extrai do canal um novo valor para o racional, na forma de uma fracção.
    @pre r = r.
    @post Se entrada e entrada contém n′/d′ (ou apenas n′, assumindo-
se d′ = 1), em
        que n′ e d′ são inteiros com d′ ≠ 0, então
            r = n′/d′ ∧ entrada,
        senão
            r = r ∧ ¬entrada. */
istream& operator>>(istream& entrada, Racional& r)
{
    r.extraiDe(entrada);

    return entrada;
}

...
```

São de notar os seguintes pontos:

- A operação `istream::peek()` devolve o valor do próximo caractere no canal *sem o extrair*!
- A operação `istream::get()` extrai o próximo caractere do canal (e devolve-o).

- A função `isdigit()`[24] indica se o caractere passado como argumento é um dos dígitos decimais.

Fica como exercício para o leitor o estudo pormenorizado do método `Racional::extraiDe()`.

7.14.5 Leitura e escrita de ficheiros

Uma vez sobrecarregados os operadores << e >> para a classe C++ `Racional`, é possível usá-los para fazer extracções e inserções não apenas do teclado e para o ecrã, mas de e para onde quer que um canal esteja estabelecido. É possível estabelecer canais de entrada e saída para ficheiros, por exemplo. Para isso usam-se as classes `ifstream` e `ofstream`, compatíveis com `istream` e `ostream`, respectivamente, e que ficam disponíveis se se incluir a seguinte linha no início do programa:

```
#include <fstream>
```

Escrita em ficheiros

Para se poder escrever num ficheiro é necessário estabelecer um canal de escrita ligado a esse ficheiro e inserir nele os dados a escrever no ficheiro. O estabelecimento de um canal de escrita para um ficheiro é feito de uma forma muito simples. A instrução

```
ofstream saída("nome do ficheiro");
```

constrói um novo canal chamado `saída` que está ligado ao ficheiro da nome "*`nome do ficheiro`*". Note-se que `saída` é uma variável como outra qualquer, só que representa um canal de escrita para um dado ficheiro. A instrução acima é possível porque a classe `ofstream` possui um construtor que recebe uma cadeia de caracteres clássica do C++. Isso significa que é possível usar cadeias de caracteres para especificar o nome do ficheiro ao qual o canal deve estar ligado, desde que se faça uso da operação `string::c_str()`, que devolve a cadeia de caracteres clássica corresponde a uma dada cadeia de caracteres:

```
cout << "Diga o nome do ficheiro: ";
string nome_do_ficheiro;
cin >> nome_do_ficheiro;

ofstream saída(nome_do_ficheiro.c_str());
```

[24] Acrescentar

```
#include <cctype>
```

no início do programa para usar esta função.

O estabelecimento de um canal de saída para um dado ficheiro é uma operação destrutora: se o ficheiro já existir, é esvaziado antes. Dessa forma a escrita começa sempre do nada. Claro está que o estabelecimento de um canal de escrita pode falhar. Por exemplo, o disco rígido pode estar cheiro, pode não haver permissões para escrever no directório em causa, ou pode existir já um ficheiro com o mesmo nome protegido para escrita. Se o estabelecimento do canal falhar durante a sua construção, este fica em estado de erro, sendo muito fácil verificar essa situação tratando o canal como se de um booleano se tratasse:

```
cout << "Diga o nome do ficheiro: ";
string nome_do_ficheiro;
cin >> nome_do_ficheiro;

ofstream saída(nome_do_ficheiro.c_str());

if(not saída) {
    cerr << "Opps...  Não consegui criar ficheiro \""
         << nome_do_ficheiro << "\"!" << endl;
    ...
}
```

É ainda possível estabelecer e encerrar um canal usando as operações `ofstream::open()`, que recebe o nome do ficheiro como argumento, e `ofstream::close()`, que não tem argumentos. Só se garante que todos os dados inseridos num canal ligado a um ficheiro já nele foram escritos se:

1. o canal tiver sido explicitamente encerrado através da operação `ostream::close()`;
2. o canal tiver sido destruído da forma usual (e.g., no final do bloco onde a variável que o representa está definida);
3. tiver sido invocada a operação `ostream::flush()`;
4. tiver sido inserido no canal o manipulador `flush` (i.e., `saída << flush`); ou
5. tiver sido inserido no canal o manipulador `endl` (i.e., `saída << endl`), que coloca um fim-de-linha no canal e invoca automaticamente a operação `ostream::flush()`.

Ou seja, tudo funciona como se o canal tivesse um certa capacidade para dados, tal como uma mangueira tem a capacidade de armazenar um pouco de água. Tal como numa mangueira só se pode garantir que toda a água que nela entrou saiu pela outra ponta tomando algumas diligências, também num canal de saída só se pode garantir que os dados chegaram ao seu destino, e não estão ainda em circulação no canal se se diligenciar como indicado acima.

A partir do momento em que um canal de saída está estabelecido, pode-se usá-lo da mesma forma que ao canal `cout`. Por exemplo:

```
cout << "Diga o nome do ficheiro: ";
string nome_do_ficheiro;
cin >> nome_do_ficheiro;

ofstream saída(nome_do_ficheiro.c_str());

if(not saída) {
    cerr << "Opps...  Não consegui criar ficheiro \""
         << nome_do_ficheiro << "\"!" << endl;
    …
}

Racional r(1, 3);

saída << r << endl;
```

Leitura de ficheiros

Para se poder ler de um ficheiro é necessário estabelecer um canal de leitura ligado a esse ficheiro e extrair dele os dados a ler do ficheiro. O estabelecimento de um canal de leitura para um ficheiro é feito de uma forma muito simples. A instrução

```
ifstream entrada("nome do ficheiro");
```

constrói um novo canal chamado `entrada` que está ligado ao ficheiro da nome "*`nome do ficheiro`*". Mais uma vez `entrada` é uma variável como outra qualquer, só que representa um canal de leitura para um dado ficheiro. É também possível usar cadeias de caracteres para especificar o nome do ficheiro ao qual o canal deve estar ligado:

```
cout << "Diga o nome do ficheiro: ";
string nome_do_ficheiro;
cin >> nome_do_ficheiro;

ifstream entrada(nome_do_ficheiro.c_str());
```

O estabelecimento de um canal de entrada pode falhar. Por exemplo, se o ficheiro não existir, ou se estiver protegido para leitura. Se o estabelecimento do canal falhar durante a sua construção, este fica em estado de erro, sendo de novo muito fácil verificar essa situação:

```
cout << "Diga o nome do ficheiro: ";
string nome_do_ficheiro;
cin >> nome_do_ficheiro;

ifstream entrada(nome_do_ficheiro.c_str());
```

```
if(not entrada) {
    cerr << "Opps...  Não consegui ligar a ficheiro \""
         << nome_do_ficheiro << "\"!" << endl;
    ...
}
```

Tal como para os canais de saída, é também possível estabelecer e encerrar um canal usando as operações `ofstream::open()`, que recebe o nome do ficheiro como argumento, e `ofstream::close()`, que não tem argumentos.

A partir do momento em que um canal de entrada está estabelecido, pode-se usá-lo da mesma forma que ao canal `cin`. Por exemplo:

```
cout << "Diga o nome do ficheiro: ";
string nome_do_ficheiro;
cin >> nome_do_ficheiro;

ofstream entrada(nome_do_ficheiro.c_str());

if(not entrada) {
    cerr << "Opps...  Não consegui ligar a ficheiro \""
         << nome_do_ficheiro << "\"!" << endl;
    ...
}

Racional r;
entrada >> r;
```

Se num mesmo programa se escreve e lê de um mesmo ficheiro, é possível usar canais que permitem simultaneamente inserções e extracções. Porém, a forma mais simples é alternar escritas e leituras no mesmo ficheiro usando canais diferentes, desde que se garanta que todos os dados inseridos no respectivo canal de saída foram já escritos no respectivo ficheiro antes de os tentar extrair a partir de um canal de entrada ligado ao mesmo ficheiro. Repare-se nas linhas finais do teste de unidade do TAD `Racional`:

```
...

Racional r1(2, -6);

...

Racional r2(3);

...
```

```
ofstream saída("teste");
saída << r1 << ' ' << r2;
saída.close();

// Só o fecho explícito garante que a extracção do canal entrada tem sucesso.

ifstream entrada("teste");
Racional r4, r5;
entrada >> r4 >> r5;

...
```

7.15 Amizades e promiscuidades

7.15.1 Rotinas amigas

Há casos em que pode ser conveniente definir rotinas normais (i.e., não-membro) que tenham acesso aos membros privados de uma classe C++. Por exemplo, suponha-se que se pretendia definir a rotina `operator>>` para a classe C++ Racional, como se fez mais atrás, mas sem delegar o trabalho na operação `Racional::extraiDe()`. Nesse caso podia-se tentar definir a rotina como:

```
void Racional::extraiDe(istream& entrada)
{
}

...

/** Extrai do canal um novo valor para o racional, na forma de uma fracção.
    @pre r = $r$.
    @post Se entrada e entrada contém $n'/d'$ (ou apenas $n'$, assumindo-se $d' =$
1), em
        que $n'$ e $d'$ são inteiros com $d' \neq 0$, então
            r $= \frac{n'}{d'} \wedge$ entrada,
         senão
            r $= r \wedge \neg$entrada. */
istream& operator>>(istream& entrada, Racional& r)
{
    assert(r.cumpreInvariante());

    int n;
    int d = 1;

    if(entrada >> n) {
        if(entrada.peek() != '/') {
```

```
                r.numerador_ = n;
                r.denominador_ = 1;
            } else {
                if(entrada.get() and isdigit(entrada.peek()) and
                   entrada >> d and d != 0)
                {
                    r.numerador_ = d < 0 ? -n : n;
                    r.denominador_ = d < 0 ? -d : d;

                    reduz();
                } else if(entrada)
                    entrada.setstate(ios_base::failbit);
            }
        }

        assert(r.cumpreInvariante());

        assert(r.numerador_ * d == n * r.denominador_ or not canal);

        return entrada;
    }
```

Como a rotina definida não é membro da class C++ `Racional`, não tem acesso aos seus membros privados. Para resolver este problema pode-se usar o conceito de amizade: se a classe C++ `Racional` declarar que é amiga, e por isso confia, nesta rotina, ela passa a ter acesso total às suas "partes íntimas". Para o conseguir basta colocar o cabeçalho da rotina em qualquer ponto da definição da classe C++[25], precedendo-a do qualificador `friend`:

```
...

class Racional {
  public:

    ...

  private:

    ...

    friend istream& operator>>(istream& entrada, Racional& r);
};

...
```

[25]Para o compilador é irrelevante se a declaração de amizade é feita na parte pública ou na parte privada da classe. No entanto, sendo as amizades uma questão de implementação, é desejável colocar a declaração na parte privada da classe, tipicamente no seu final.

Qualquer rotina não-membro que faça uso de uma classe C++ é conceptualmente parte das operações que concretizam o comportamento desejado para o correspondente TAD. No entanto, se uma rotina não-membro não for amiga da classe C++, embora seja parte da implementação do TAD, não é parte da implementação da correspondente classe C++. Isso deve-se ao facto de não ter acesso às suas partes privadas, e por isso não ter nenhuma forma directa de afectar o estado das suas instâncias. Porém, a partir do momento em que uma rotina se torna amiga de uma classe C++, passa a fazer parte da implementação dessa classe. É por isso que a rotina `operator>>` definida acima se preocupa com o cumprimento da condição invariante de classe: se pode afectar directamente o estado das instâncias da classe é bom que assim seja. É que o produtor de uma rotina não-membro amiga de uma classe deve ser visto como produtor da classe C++ respectiva, enquanto o produtor de uma rotina não-membro e não-amiga que faça uso de uma classe não pode ser visto como produtor dessa classe C++, mas sim como seu consumidor.

7.15.2 Classes amigas

Da mesma forma que no caso das rotinas, também se podem declarar classes inteiras como amigas de uma classe C++. Nesse caso todos os métodos da classe declarada como amiga têm acesso às partes privadas da classe C++ que declarou a amizade. Este tipo de amizade pode ser muito útil em algumas circunstâncias, como se verá no Capítulo 10, embora em geral sejam de evitar. A sintaxe da declaração de amizade de classes é semelhante à usada para as rotinas. Por exemplo,

```
class B {
    ...
};

class A {
  public:

    ...

  private:

    ...

    // Declaração da classe C++ B como amiga da classe A.
    friend class B;
};
```

7.15.3 Promiscuidades

Em que casos devem ser usadas rotinas ou classes amigas de uma classe C++? A regra geral é o mínimo possível. Se se puder evitar usar rotinas e classes amigas tanto melhor, pois evita-se introduzir uma excepção à regra de que só os membros de uma classe C++ têm acesso à suas partes privadas e à regra de que a implementação de uma classe corresponde às suas partes privadas e aos respectivos métodos. Todas as excepções são potencialmente geradoras de erros. Como mnemónica do perigo das amizades em C++, fica a frase "amizades normalmente trazem promiscuidades". Assim, e respeitando a regra geral enunciada, manter-se-á a implementação da rotina `operator>>` à custa da operação `Racional::extraiDe()`.

7.16 Código completo do TAD **`Racional`**

O TAD `Racional` foi concretizado na forma de uma classe C++ com o mesmo nome *e de rotinas (não-membro) associadas*. Conceptualmente o TAD é definido pelas respectivas operações, pelo que a classe C++ não é suficiente para o concretizar: faltam as rotinas associadas, que não são membro da classe. Ou seja, as rotinas que operam com racionais fazem logicamente parte do TAD, mesmo não pertencendo à classe C++ que o concretiza.

Note-se que se concentraram as declarações das rotinas no início do código, pois fazem logicamente parte da interface do TAD, tendo-se colocado a respectiva definição no final do código, junto com a restante implementação do TAD. Esta organização será útil aquando da organização do código em módulos físicos, ver 9, de modo a permitir a fácil utilização do TAD desenvolvido em qualquer programa.

```
#include <iostream>
#include <cctype>
#include <cassert>

using namespace std;

/** Devolve o máximo divisor comum dos inteiros passados como argumento.
    @pre V.
    @post mdc = { mdc(m, n)   m ≠ 0 ∨ n ≠ 0
                {    1        m = 0 ∧ n = 0 . */
int mdc(int const m, int const n);

/** Representa números racionais.
    @invariant 0 < denominador_ ∧ mdc(numerador_, denominador_) = 1. */
class Racional {
  public:
    /** Constrói racional com valor inteiro. Construtor por omissão.
        @pre V.
        @post *this = n. */
    Racional(int const n = 0);

    /** Constrói racional correspondente a n/d.
        @pre d ≠ 0.
        @post *this = n/d. */
    Racional(int const n, int const d);
```

```
/** Devolve numerador da fracção canónica correspondente ao racional.
    @pre 𝒱.
    @post numerador / denominador() = *this. */
int numerador() const;

/** Devolve denominador da fracção canónica correspondente ao racional.
    @pre 𝒱.
    @post (E n : 𝒱 : n / denominador = *this ∧ 0 < denominador ∧ mdc(n, denominador) = 1). */
int denominador()  const;

/** Devolve versão constante do racional.
    @pre 𝒱.
    @post operator+ ≡ *this. */
Racional const& operator+() const;

/** Devolve simétrico do racional.
    @pre 𝒱.
    @post operator- = -*this. */
Racional const operator-() const;

/** Insere o racional no canal no formato de uma fracção.
    @pre 𝒱.
    @post ¬saída∨ saída contém n/d (ou simplesmente n se d = 1)
          sendo n/d a fracção canónica correspondente ao racional *this. */
void insereEm(ostream& saída) const;

/** Extrai do canal um novo valor para o racional, na forma de uma fracção.
    @pre *this = r.
    @post Se entrada e entrada contém n′/d′ (ou apenas n′, assumindo-
se d′ = 1), em
          que n′ e d′ são inteiros com d′ ≠ 0, então
              *this = n′/d′ ∧ entrada,
          senão
              *this = r ∧ ¬entrada. */
void extraiDe(istream& entrada);

/** Adiciona de um racional.
    @pre *this = r.
    @post operator+= ≡ *this ∧ *this = r + r2. */
Racional& operator+=(Racional const& r2);

/** Subtrai de um racional.
    @pre *this = r.
    @post operator-= ≡ *this ∧ *this = r − r2. */
Racional& operator-=(Racional const& r2);
```

```
    /** Multiplica por um racional.
        @pre *this = r.
        @post operator*= ≡ *this ∧ *this = r × r2. */
    Racional& operator*=(Racional const& r2);

    /** Divide por um racional.
        @pre *this = r ∧ r2 ≠ 0.
        @post operator/= ≡ *this ∧ *this = r/r2. */
    Racional& operator/=(Racional const& r2);

   /** Incrementa e devolve o racional.
        @pre *this = r.
        @post operador++ ≡ *this ∧ *this = r + 1. */
    Racional& operator++();

/** Decrementa e devolve o racional.
        @pre *this = r.
        @post operador- ≡ *this ∧ *this = r − 1. */
    Racional& operator--();

  private:
    int numerador_;
    int denominador_;
    /** Reduz a fracção que representa o racional.
        @pre denominador_ ≠ 0 ∧ *this = r.
        @post denominador_ ≠ 0 ∧ mdc(numerador_, denominador_) = 1∧
              *this = r. */
    void reduz();

    /** Indica se a condição invariante de classe se verifica.
        @pre V.
        @post cumpreInvariante = (0 <
denominador_ ∧ mdc(numerador_, denominador_) = 1). */
    bool cumpreInvariante() const;
};

/** Adição de dois racionais.
    @pre V.
    @post operator+ = r1 + r2. */
Racional const operator+(Racional const r1, Racional const& r2);

/** Subtracção de dois racionais.
    @pre V.
    @post operator- = r1 − r2. */
Racional const operator-(Racional const r1, Racional const& r2);
```

/** Produto de dois racionais.
@pre $\mathcal{V}$.
@post `operator*` $=$ `r1` $\times$ `r2`. */

```
Racional const operator*(Racional const r1, Racional const& r2);
```

/** Divisão de dois racionais.
@pre `r2` $\neq 0$.
@post `operator/` $=$ `r1`$/$`r2`. */

```
Racional const operator/(Racional const r1, Racional const& r2);
```

/** Incrementa o racional recebido como argumento, devolvendo o seu valor antes de incrementado.
@pre `*this` $= r$.
@post `operator++` $= r \wedge$ `*this` $= r + 1$. */

```
Racional const operator++(Racional& r, int);
```

/** Decrementa o racional recebido como argumento, devolvendo o seu valor antes de decrementado.
@pre `*this` $= r$.
@post `operator-` $= r \wedge$ `*this` $= r - 1$. */

```
Racional const operator--(Racional& r, int);
```

/** Indica se dois racionais são iguais.
@pre $\mathcal{V}$.
@post `operator==` $=$ (`r1` $=$ `r2`). */

```
bool operator==(Racional const& r1, Racional const& r2);
```

/** Indica se dois racionais são diferentes.
@pre $\mathcal{V}$.
@post `operator==` $=$ (`r1` $\neq$ `r2`). */

```
bool operator!=(Racional const& r1, Racional const& r2);
```

/** Indica se o primeiro racional é menor que o segundo.
@pre $\mathcal{V}$.
@post `operator<` $=$ (`r1` $<$ `r2`). */

```
bool operator<(Racional const& r1, Racional const& r2);
```

/** Indica se o primeiro racional é maior que o segundo.
@pre $\mathcal{V}$.
@post `operator>` $=$ (`r1` $>$ `r2`). */

```
bool operator>(Racional const& r1, Racional const& r2);
```

/** Indica se o primeiro racional é menor ou igual ao segundo.
@pre $\mathcal{V}$.
@post `operator<=` $=$ (`r1` $\leq$ `r2`). */

```
bool operator<=(Racional const& r1, Racional const& r2);

/** Indica se o primeiro racional é maior ou igual ao segundo.
    @pre V.
    @post operator>= = (r1 ≥ r2). */
bool operator>=(Racional const& r1, Racional const& r2);

/** Insere o racional no canal de saída no formato de uma fracção.
    @pre V.
    @post ¬saída∨ saída contém n/d (ou simplesmente n se d = 1)
             sendo n/d a fracção canónica correspondente ao racional r. */
ostream& operator<<(ostream& saída, Racional const& r);

/** Extrai do canal um novo valor para o racional, na forma de uma fracção.
    @pre r = r.
    @post Se entrada e entrada contém n′/d′ (ou apenas n′, assumindo-
se d′ = 1), em
            que n′ e d′ são inteiros com d′ ≠ 0, então
                r = n′/d′ ∧ entrada,
            senão
                r = r ∧ ¬entrada. */
istream& operator>>(istream& entrada, Racional& r);

int mdc(int m, int n)
{
    if(m == 0 and n == 0)
        return 1;

    if(m < 0)
        m = -m;
    if(n < 0)
        n = -n;

    while(true) {
        if(m == 0)
            return n;
        n = n % m;
        if(n == 0)
            return m;
        m = m % n;
    }
}

inline Racional::Racional(int const n)
    : numerador_(n), denominador_(1)
{
```

```
    assert(cumpreInvariante());
    assert(numerador_ == n * denominador_);
}

inline Racional::Racional(int const n, int const d)
    : numerador_(d < 0 ? -n : n),
      denominador_(d < 0 ? -d : d)
{
    assert(d != 0);

    reduz();

    assert(cumpreInvariante());
    assert(numerador_ * d == n * denominador_);
}

inline int Racional::numerador() const
{
    assert(cumpreInvariante());

    return numerador_;
}

inline int Racional::denominador() const
{
    assert(cumpreInvariante());

    return denominador_;
}

inline Racional const& Racional::operator+() const
{
    assert(cumpreInvariante());

    return *this;
}

inline Racional const Racional::operator-() const
{
    assert(cumpreInvariante());

    Racional r;
    r.numerador_ = -numerador_;
    r.denominador_ = denominador_;
```

```
    assert(r.cumpreInvariante());

    return r;
}

inline void Racional::insereEm(ostream& saída) const
{
    assert(cumpreInvariante());

    saída << numerador_;
    if(denominador_ != 1)
        saída << '/' << denominador_;
}

void Racional::extraiDe(istream& entrada)
{
     assert(cumpreInvariante());

    int n;
    int d = 1;

    if(entrada >> n) {
        if(entrada.peek() != '/') {
            numerador_ = n;
            denominador_ = 1;
        } else {
            if(entrada.get() and isdigit(entrada.peek()) and
               entrada >> d and d != 0)
            {
                numerador_ = d < 0 ? -n : n;
                denominador_ = d < 0 ? -d : d;

                reduz();
            } else if(entrada)
                entrada.setstate(ios_base::failbit);
        }
    }

    assert(cumpreInvariante());

    assert(numerador_ * d == n * denominador_ or not entrada);
}

Racional& Racional::operator+=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());
```

```
    int dn = mdc(numerador_, r2.numerador_);
    int dd = mdc(denominador_, r2.denominador_);

    // Devido a r += r:
    int n2 = r2.numerador_;
    int d2 = r2.denominador_;

    numerador_ /= dn;
    denominador_ /= dd;

    numerador_ = numerador_ * (d2 / dd) + n2 / dn * denominador_;

    dd = mdc(numerador_, dd);

    numerador_ = dn * (numerador_ / dd);
    denominador_ *= d2 / dd;

    assert(cumpreInvariante());

    return *this;
}

Racional& Racional::operator-=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    int dn = mdc(numerador_, r2.numerador_);
    int dd = mdc(denominador_, r2.denominador_);

    // Devido a r += r:
    int n2 = r2.numerador_;
    int d2 = r2.denominador_;

    numerador_ /= dn;
    denominador_ /= dd;

    numerador_ = numerador_ * (d2 / dd) - n2 / dn * denominador_;

    dd = mdc(numerador_, dd);

    numerador_ = dn * (numerador_ / dd);
    denominador_ *= d2 / dd;

    assert(cumpreInvariante());
```

```
    return *this;
}

inline Racional& Racional::operator*=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    int n1d2 = mdc(numerador_, r2.denominador_);
    int n2d1 = mdc(r2.numerador_, denominador_);

    numerador_ = (numerador_ / n1d2) * (r2.numerador_ / n2d1);
    denominador_ = (denominador_ / n2d1) * (r2.denominador_ / n1d2);

    assert(cumpreInvariante());

    return *this;
}

inline Racional& Racional::operator/=(Racional const& r2)
{
    assert(cumpreInvariante() and r2.cumpreInvariante());

    assert(r2 != 0);

    int dn = mdc(numerador_, r2.numerador_);
    int dd = mdc(denominador_, r2.denominador_);

    if(r2.numerador_ < 0) {
        numerador_ = (numerador_ / dn) * (-r2.denominador_ / dd);
        denominador_ = (denominador_ / dd) * (-r2.numerador_ / dn);
    } else {
        numerador_ = (numerador_ / dn) * (r2.denominador_ / dd);
        denominador_ = (denominador_ / dd) * (r2.numerador_ / dn);
    }

    assert(cumpreInvariante());

    return *this;
}

inline Racional& Racional::operator++()
{
    assert(cumpreInvariante());

    numerador_ += denominador_;
```

```
    assert(cumpreInvariante());

    return *this;
}

inline Racional& Racional::operator--()
{
    assert(cumpreInvariante());

    numerador_ -= denominador_;

    assert(cumpreInvariante());

    return *this;
}

inline void Racional::reduz()
{
    assert(denominador_ != 0);

    int k = mdc(numerador_, denominador_);

    numerador_ /= k;
    denominador_ /= k;

    assert(denominador_ != 0 and mdc(numerador_, denominador_) == 1);
}

inline bool Racional::cumpreInvariante() const
{
    return 0 < denominador_ and mdc(numerador_, denominador_) == 1;
}

inline Racional const operator+(Racional r1, Racional const& r2)
{
    return r1 += r2;
}

inline Racional const operator-(Racional r1, Racional const& r2)
{
    return r1 -= r2;
}

inline Racional const operator*(Racional r1, Racional const& r2)
{
    return r1 *= r2;
```

```
}

inline Racional const operator/(Racional r1, Racional const& r2)
{
    assert(r2 != 0);

    return r1 /= r2;
}

inline Racional const operator++(Racional& r, int)
{
    Racional const cópia = r;

    ++r;

    return cópia;
}

inline Racional const operator--(Racional& r, int)
{
    Racional const cópia = r;

    --r;

    return cópia;
}

inline bool operator==(Racional const& r1, Racional const& r2)
{
    return r1.numerador() == r2.numerador() and
           r1.denominador() == r2.denominador();
}

inline bool operator!=(Racional const& r1, Racional const& r2)
{
    return not (r1 == r2);
}

inline bool operator<(Racional const& r1, Racional const& r2)
{
    int dn = mdc(r1.numerador(), r2.numerador());
    int dd = mdc(r1.denominador(), r2.denominador());

    return (r1.numerador() / dn) * (r2.denominador() / dd) <
           (r2.numerador() / dn) * (r1.denominador() / dd);
}
```

```
inline bool operator>(Racional const& r1, Racional const& r2)
{
    return r2 < r1;
}

inline bool operator<=(Racional const& r1, Racional const& r2)
{
    return not (r2 < r1);
}

inline bool operator>=(Racional const& r1, Racional const& r2)
{
    return not (r1 < r2);
}

ostream& operator<<(ostream& saída, Racional const& r)
{
    r.insereEm(saída);

    return saída;
}

istream& operator>>(istream& entrada, Racional& r)
{
    r.extraiDe(entrada);

    return entrada;
}

#ifdef TESTE

#include <fstream>

/** Programa de teste do TAD Racional e da função mdc(). */
int main()
{
    assert(mdc(0, 0) == 1);
    assert(mdc(10, 0) == 10);
    assert(mdc(0, 10) == 10);
    assert(mdc(10, 10) == 10);
    assert(mdc(3, 7) == 1);
    assert(mdc(8, 6) == 2);
    assert(mdc(-8, 6) == 2);
    assert(mdc(8, -6) == 2);
    assert(mdc(-8, -6) == 2);
```

```
Racional r1(2, -6);

assert(r1.numerador() == -1 and r1.denominador() == 3);

Racional r2(3);

assert(r2.numerador() == 3 and r2.denominador() == 1);

Racional r3;

assert(r3.numerador() == 0 and r2.denominador() == 1);

assert(r2 == 3);
assert(3 == r2);
assert(r3 == 0);
assert(0 == r3);

assert(r1 < r2);
assert(r2 > r1);
assert(r1 <= r2);
assert(r2 >= r1);
assert(r1 <= r1);
assert(r2 >= r2);

assert(r2 == +r2);
assert(-r1 == Racional(1, 3));

assert(++r1 == Racional(2, 3));
assert(r1 == Racional(2, 3));

assert(r1++ == Racional(2, 3));
assert(r1 == Racional(5, 3));
assert((r1 *= Racional(7, 20)) == Racional(7, 12));
assert((r1 /= Racional(3, 4)) == Racional(7, 9));
assert((r1 += Racional(11, 6)) == Racional(47, 18));
assert((r1 -= Racional(2, 18)) == Racional(5, 2));

assert(r1 + r2 == Racional(11, 2));
assert(r1 - Racional(5, 7) == Racional(25, 14));
assert(r1 * 40 == 100); assert(30 / r1 == 12);

ofstream saída("teste");
saída << r1 << ' ' << r2;
saída.close();
```

```
    ifstream entrada("teste");
    Racional r4, r5;
    entrada >> r4 >> r5;

    assert(r1 == r4);
    assert(r2 == r5);
}

#else // TESTE

int main()
{
    // Ler fracções:
    cout << "Introduza duas fracções (numerador denominador): ";
    Racional r1, r2;
    cin >> r1 >> r2;

    if(not cin) {
        cerr << "Opps!  A leitura dos racionais falhou!" << endl;
        return 1;
    }

    // Calcular racional soma:
    Racional r = r1 + r2;

    // Escrever resultado:
    cout << "A soma de " << r1 << " com " << r2
         << " é " << r << '.' << endl;
}

#endif // TESTE
```

O código acima representa o resultado final de um longo périplo, começado no início deste capítulo. Será que valeu a pena o esforço posto no desenvolvimento do TAD `Racional`? Depende. Se o objectivo era simplesmente somar duas fracções, certamente que não. Se o objectivo, por outro lado, era permitir a escrita simples de programas usando números racionais, então valeu a pena. Em geral, quanto mais esforço for dispendido pelo programador produtor no desenvolvimento de uma módulo, menos esforço será exigido do programador consumidor. No entanto, não se deve nunca perder o pragmatismo de vista e desenhar um módulo tentando prever todas as suas possíveis utilizações. Em primeiro lugar porque essas previsões provavelmente falharão, conduzindo a código inútil, e em segundo porque todo esse código inútil irá servir apenas como fonte de complexidade e erros desnecessários. Neste caso, no entanto, o negócio foi bom: o desenvolvimento serviu não apenas para fazer surgir naturalmente muitas particularidades associadas ao desenvolvimento de TAD em C++, servindo assim como ferramenta pedagógica, como também redundou numa classe que é realmente útil[26], ou que

[26] É de tal forma assim que se está a estudar a inclusão de uma classe genérica semelhante na biblioteca padrão

pelo menos está no bom caminho para o vir a ser.

7.17 Outros assuntos acerca de classes C++

Ao longo deste capítulo, e a pretexto do desenvolvimento de um TAD `Racional`, introduziram-se muitos conceitos importantes relativos às classes C++. Nenhum exemplo prático esgota todos os assuntos, pelo que se reservou esta secção final para introduzir alguns conceitos adicionais sobre classes C++.

7.17.1 Constantes membro

Suponha que se pretende definir uma classe C++ para representar vectores num espaço de dimensão fixa e finita. De modo a ser fácil alterar a dimensão do espaço onde se encontra o vector sem grande esforço, é tentador definir um atributo constante para representar a dimensão desse espaço. Depois, as coordenadas do vector poderiam ser guardadas numa matriz clássica do C++, membro da classe, com a dimensão dada. Ou seja:

```
class Vector {
  public:
    int const dimensão = 3; // erro!
    ...

  private:
    double coordenadas[dimensão];
    ...
};
```

Infelizmente, não é possível definir e inicializar uma constante dentro de uma classe. A razão para a proibição é simples. Sendo `dimensão` uma constante membro (de instância), cada instância da classe C++ possuirá a sua própria cópia dessa constante. Mas isso significa que, para ser verdadeiramente útil, essa constante deverá poder tomar valores diferentes para cada instância da classe C++, i.e., para cada variável dessa classe C++ que seja construída. Daí que não se possam inicializar atributos constantes (de instância) na sua própria definição. Seria possível, por exemplo, inicializar a constante nos construtores da classe:

```
class Vector {
  public:
    int const dimensão;

    Vector();

    ...
```

do C++ (ver `http://www.boost.org`).

```
  private:
    double coordenadas[dimensão];
    ...
};

Vector::Vector()
    : dimensão(3), ...
{
    ...
}
```

As listas de inicializadores são extremamente úteis, sendo utilizadas para inicializar não só atributos constantes, como também atributos que sejam referências e atributos que sejam de uma classe sem construtor por omissão, i.e., que exijam uma inicialização explícita. Mas neste caso a solução a que chegámos não é a mais adequada. Se todos os vectores devem ter a mesma dimensão, porquê fornecer cada instância da classe C++ com a sua própria constante `dimensão`? O ideal seria partilhar essa constante entre todas as instâncias da classe.

7.17.2 Membros de classe

Até agora viu-se apenas como definir membros de instância, i.e., membros dos quais cada instância da classe C++ possui uma versão própria. Como fazer para definir um atributo do qual exista apenas uma versão, comum a todas as instâncias da classe C++? A solução é simples: basta declarar o atributo com sendo um atributo *de classe* e não *de instância*. Isso consegue-se precedendo a sua declaração do qualificador `static`:

```
class Vector {
  public:
    static int const dimensão;

    Vector();

    ...

  private:
    double coordenadas[dimensão]; // Opps... Não funciona...

    ...
};
```

Um atributo de classe não pode ser inicializado nas listas de inicializadores dos construtores. Nem faria qualquer sentido, pois um atributo de classe têm apenas uma instância, partilhada entre todas as instâncias da classe de que é membro, não fazendo sentido ser inicializada senão uma vez, antes do programa começar. Assim, é necessário definir esse atributo fora da classe, sendo durante essa definição que se procede à respectiva inicialização:

```
int const Vector::dimensão = 3;
```

Infelizmente esta solução não compila correctamente. É que em C++ só se pode especificar a dimensão de uma matriz clássica usando um constante *com valor conhecido pelo compilador*. Como um atributo de classe só é inicializado na sua definição, que está fora da classe, o atributo `dimensão` é inicializado tarde demais: o compilador "engasga-se" na definição da matriz membro, dizendo que não sabe o valor da constante.

Este problema não é, em geral, resolúvel, excepto quando o atributo de classe em definição for de um tipo básico inteiro do C++. Nesse caso é possível fazer a definição (como inicialização) dentro da própria classe:

```
class Vector {
  public:
    static int const dimensão = 3;

    Vector();

    ...

  private:
    double coordenadas[dimensão]; // Ah! Já funciona...

    ...
};
```

onde a definição externa à classe deixa de ser necessária.

Da mesma forma, é possível declarar operações da classe C++ que não precisam de ser invocadas através de nenhuma instância: são as chamadas operações de classe. Suponha-se, por absurdo, que se queria que a classe C++ `Vector` mantivesse uma contagem do número de instâncias de si própria existente em cada instante. Uma possibilidade seria:

```
class Vector {
  public:
    static int const dimensão = 3;

    Vector();

    static int númeroDeInstâncias();

    ...

  private:
    double coordenadas[dimensão]; // Ah! Já funciona...
```

```
    static int número_de_instâncias;

    ...
};

Vector::Vector()
    : ...
{
    ...

    ++número_de_instâncias;
}

inline int Vector::númeroDeInstâncias()
{
    return número_de_instâncias;
}

int Vector::número_de_instâncias = 0;
```

Os membros de classe (e não de instância) são precedidos do qualificador `static` aquando da sua declaração dentro da definição da classe. O atributo de classe `número_de_instâncias` é declarado durante a definição da classe e só é definido (i.e., construída de facto com o valor inicial 0) depois da classe C++, tal como acontece com as rotinas membro.

O contador de instâncias acima tem dois problemas. O primeiro é que a contabilização falha se algum vector for construído por cópia a partir de outro vector:

```
Vector v1; // Ok, incrementa contador.

Vector v2(v1); // Erro! Não incrementa contador.
```

Para já ignorar-se-á este problema, até porque só se falará do fornecimento explícito de construtores por cópia, que substituem o construtor por cópia fornecido implicitamente pela linguagem, num capítulo posterior.

O segundo problema é que a contabilização falha quando se destrói alguma instância da classe.

7.17.3 Destrutores

Da mesma forma que os construtores de uma classe C++ são usados para inicializar as suas instâncias quando estas são construídas, podem-se usar destrutores, i.e., código que deve é executado quando a instância é destruída, para "arrumar a casa" no final do tempo de vida das instâncias de uma classe C++. Os destrutores são extremamente úteis, particularmente quando se utilizam variáveis dinâmicas ou, em geral, quando os construtores da classe C++ reservam algum recurso externo para uso exclusivo da instância construída. Utilizações mais interessantes do conceito ver-se-ão mais tarde, bastando para já apresentar um exemplo ingénuo da sua utilização no âmbito da classe C++ `Vector`. Os destrutores declaram-se e definem-se como os construtores, excepto que se coloca o símbolo ~ antes do seu nome (que é também o nome da classe):

```
class Vector {
  public:
    static int const dimensão = 3;

    Vector();

    ~Vector();

    static int númeroDeInstâncias();

    ...

  private:
    double coordenadas[dimensão]; // Ah! Já funciona...

    static int número_de_instâncias;

    ...
};

Vector::Vector()
    : ...
{
    ...

    ++número_de_instâncias;
}

Vector::~Vector()
{
    --número_de_instâncias;

    ...
}

inline int Vector::númeroDeInstâncias()
{
    return número_de_instâncias;
}
```

```
int Vector::número_de_instâncias = 0;
```

Note-se que a linguagem fornece implicitamente um destrutor para as classes definidas sempre que este não seja definido explicitamente pelo programador fabricante.

7.17.4 De novo os membros de classe

Suponha-se o seguinte código usando a classe desenvolvida:

```
Vector a;

int main()
{
    {
        Vector b;

        for(int i = 0; i != 3; ++i) {
            Vector c;
            cout << "Existem " << Vector::númeroDeInstâncias()
                 << " instâncias." << endl;
            static Vector d;
        }

        cout << "Existem " << Vector::númeroDeInstâncias()
             << " instâncias." << endl;
    }

    cout << "Existem " << C::númeroDeInstâncias()
         << " instâncias." << endl;
}
```

É de notar que a invocação da operação de classe C++ `Vector::númeroDeInstâncias()` faz-se não através do operador de selecção de membro `.`, o que implicaria a invocação da operação através de uma qualquer instância da classe C++, mas através do operador de resolução de âmbito `::`. No entanto, é também possível, se bem que inútil, invocar operações de classe através do operador de selecção de membro. As mesmas observações se podem fazer no que diz respeito aos atributos de classe.

A execução do programa acima teria como resultado:

```
Existem 3 instâncias.
Existem 4 instâncias.
Existem 4 instâncias.
Existem 3 instâncias.
Existem 2 instâncias.
```

De aqui em diante utilizar-se-á a expressão "membro" como significando "membro de instância", i.e., membros dos quais cada instância da classe C++ a que pertencem possui uma cópia própria, usando-se sempre a expressão "membro de classe" para os membros partilhados entre todas as instâncias da classe C++.

Para que o resultado do programa acima seja claro, é necessário recordar que:

- instâncias automáticas (i.e., variáveis e constantes locais sem o qualificador `static`) são construídas quando a instrução da sua definição é executada e destruídas quando o bloco de instruções na qual foram definidas termina;
- instâncias estáticas globais são construídas antes de o programa começar e destruídas depois do seu final (depois de terminada a função `main()`); e
- instâncias estáticas locais são construídas quando a instrução da sua definição é executada pela primeira vez e destruídas depois do final do programa (depois de terminada a função `main()`).

7.17.5 Construtores por omissão

Todas as classes C++ têm construtores. A um construtor que possa ser invocado sem qualquer argumento (porque não tem qualquer parâmetro ou porque todos os parâmetros têm valores por omissão) chama-se construtor *por omissão*. Se o programador não declarar qualquer construtor, é fornecido implicitamente, sempre que possível, um construtor por omissão. Por exemplo, no código

```
class C {
  private:
    Racional r1;
    Racional r2;
    int i;
    int j;
};

C c; // nova instância, construtor por omissão invocado.
```

é fornecido implicitamente pelo compilador um construtor por omissão para a classe C++ `C`. Este construtor invoca os construtores por omissão de todos os atributos, com excepção dos pertencentes a tipos básicos do C++ que, infelizmente, não são inicializados implicitamente. Neste caso, portanto, o construtor por omissão da classe `C` constrói os atributos `r1` r `r2` com o valor racional zero, deixando os tributos `i` e `j` por inicializar.

O construtor por omissão fornecido implicitamente pode ser invocado explicitamente,

```
C c = C(); // ou C c(C());
```

embora neste caso seja também invocado o construtor por cópia, também fornecido implicitamente, que constrói a variável `c` à custa da instância temporária construída pelo construtor por omissão. Para que esse facto fique claro, repare-se no código

```
cout << Racional() << endl;
```

que mostra no ecrã o valor da instância temporária da classe C++ `Racional` construída pelo respectivo construtor por omissão, i.e., o valor zero. Neste caso o construtor por cópia, da classe C++ `Racional`, não é invocado.

Se o programador declarar algum construtor explicitamente, então o construtor por omissão deixa de ser fornecido implicitamente. Por exemplo, se a classe C++ `C` fosse

```
class C {
  public:
    C(int i, int j);

  private:
    Racional r1;
    Racional r2;
    int i;
    int j;
};
```

o código

```
C c; // ops... falta o construtor por omissão!
```

resultaria num erro de compilação.

No exemplo seguinte, o construtor por omissão faz exactamente o mesmo papel que o construtor por omissão fornecido implicitamente para a classe C++ `C` no exemplo original:

```
class C {
  public:
    C();

  private:
    Racional r1;
    Racional r2;
    int i;
    int j;
};

C::C()
    : r1(), r2()
{
}
```

Sempre não sejam colocados na lista de inicialização de um construtor, os atributos que não sejam de tipos básicos do C++ são inicializados implicitamente através do respectivo construtor por omissão (se ele existir, bem entendido). Assim, o construtor no exemplo acima pode-se simplificar para:

```
C::C()
{
}
```

Antes de ser executado o corpo do construtor envolvido numa construção, todos os atributos da classe C++ são construídos *pela ordem da sua definição na classe*, sendo passados aos construtores os argumentos indicados na lista de inicializadores entre parênteses após o nome do atributo, se existirem, ou os construtores por omissão na falta do nome do atributo na lista, com excepção dos atributos de tipos básicos do C++, que têm de ser inicializados explicitamente, caso contrário ficam por inicializar. Por exemplo, no código

```
class C {
  public:
    C(int n, int d, int i);

  private:
    Racional r1;
    Racional r2;
    int i;
    int j;
};

C::C(int const n, int const d, int const i)
    : r1(n, d), i(i), j()
{
    r2 = 3;
}

C c(2, 10, 1);
```

ao ser construída a variável `c` é invocado o seu construtor, o que resultará nas seguintes operações:

1. Construção de `r1` por invocação do construtor da classe C++ `Racional` com argumentos `n` e `d` (que neste caso inicializa `r1` com $\frac{1}{5}$).

2. Construção de `r2` por invocação implícita do construtor da classe C++ `Racional` que pode ser invocado sem argumentos (que inicializa `r2` com o racional zero ou $\frac{0}{1}$).

3. Construção do atributo `i` a partir do parâmetro `i` (que neste caso fica com 1). É usado o construtor por cópia dos `int`.

4. Construção do atributo `j` através do construtor por omissão dos `int`, que é necessário invocar explicitamente (e que inicializa `j` com o valor zero).

5. Execução do corpo do construtor da classe C++ `C`:

   (a) Conversão do valor literal 3 de `int` para `Racional`.

   (b) Atribuição do racional $\frac{3}{1}$ a `r2`.

É de notar que a variável membro `r2` é construída e inicializada com o valor zero e só depois lhe é atribuído o racional $\frac{3}{1}$, o que é uma perda de tempo. Seria preferível incluir `r2` na lista de inicializadores.

7.17.6 Matrizes de classe

É possível definir matrizes clássicas do C++ tendo como elementos valores de TAD, por exemplo da classe C++ `Racional`:

```
Racional m1[10];
Racional m2[10] = {1, 2, Racional(3), Racional(), Racional(1, 3)};
```

Os 10 elementos de `m1` e os últimos cinco elementos de `m2` são construídos usando o construtor por omissão da classe `Racional` (que os inicializa com o racional zero). Os dois primeiros elementos da matriz `m2` são inicializados a partir de inteiros implicitamente convertidos para racionais usando o construtor com um único argumento da classe C++ `Racional` (i.e., o segundo construtor, usando-se um segundo argumento tendo valor por omissão um). Essa conversão é explicitada no caso do terceiro elemento de `m2`. Já para o quarto e o quinto elementos, eles são construídos por cópia a partir de um racional construído usando o construtor por omissão, no primeiro caso, e o construtor completo, com dois argumentos, no segundo caso. Note-se que se a classe C++ em causa não possuir construtores por omissão, é obrigatório inicializar todos os elementos da matriz explicitamente.

7.17.7 Conversões para outros tipos

Viu-se já que ao se definir numa classe C++ um construtor passível de ser invocado com um único argumento, se fornece uma forma de conversão implícita de tipo (ver Secção 7.9.2). Por exemplo, a classe C++ `Racional` fornece um construtor que pode ser invocado com um único argumento inteiro, o que possibilita a conversão implícita de tipo entre valores do tipo `int` e valores da classe `Racional`,

```
Racional r(1, 3);

...

if(r < 1) // 1 convertido implicitamente de int para Racional.
```

sem haver necessidade de explicitar essa conversão:

```
Racional r(1, 3);

...

if(r < Racional(1)) // não é necessário...
```

E se se pretendesse equipar a classe C++ `Racional` com uma conversão implícita desse tipo para `double`, i.e., se se pretendesse tornar o seguinte código válido?

```
Racional r(1, 2);
double x = r;
cout << r << ' ' << x << endl; // mostra: 1/2 0.5
cout << double(r) << endl; // mostra: 0.5
```

A solução poderia passar por alterar a classe C++ `double`, de modo a ter um construtor que aceitasse um racional. O problema é que nem o tipo `double` é uma classe, nem, se por absurdo o fosse, estaria nas mãos do programador alterá-la.

A solução passa por alterar a classe C++ `Racional` indicando que se pode converter implicitamente um racional num `double`. De facto, a linguagem C++ permite fazê-lo. É possível numa classe C++ definir uma conversão implícita de um tipo para essa classe, mas também o contrário: definir uma conversão implícita da classe para um outro tipo. Isso consegue-se sobrecarregando um operador de conversão para o tipo pretendido, neste caso `double`. Esse operador chama-se `operator double`. Este tipo de operadores, de conversão de tipo, têm algumas particularidades:

1. Só podem ser sobrecarregados através de rotinas membro da classe C++ a converter.
2. Não incluem tipo de devolução no seu cabeçalho, pois este é indicado pelo próprio nome do operador.

Ou seja, no caso em apreço

```
...
class Racional {
  public:

    ...

    /** Devolve o racional visto como um double.
        @pre V.
        @post O valor devolvido é uma aproximação tão boa quanto
              possível do racional representado por *this. */
```

```
    operator double() const;

...

};

...

Racional::operator double() const
{
    return double(numerador()) / double(denominador());
}
```

A divisão do numerador pelo denominador é feita depois de ambos serem convertidos para `double`. De outra forma seria realizada a divisão inteira, cujo resultado não é bem aquilo que se pretendia...

O problema deste tipo de operadores de conversão de tipo, que devem ser usados com moderação, é que levam frequentemente a ambiguidades. Por exemplo, definido o operador de conversão para `double` de valores da classe C++ `Racional`, como deve ser interpretado o seguinte código?

```
Racional r(1,3);

...

if(r == 1)
    ...
```

O compilador pode interpretar a guarda da instrução de selecção ou condicional como

```
double(r) == double(1)
```

ou como

```
r == Racional(1)
```

mas não sabe qual escolher.

As regras do C++ para resolver este tipo de ambiguidades são algo complicadas (ver [12, Secção 7.4]) e não resolvem todos os casos. No exemplo dado o programa é de facto ambíguo e portanto resulta num erro de compilação. Como é muito mais natural e frequente a conversão implícita de um `int` num `Racional` do que a conversão implícita de um `Racional` num `double`, a melhor solução é simplesmente não sobrecarregar o operador de conversão para `double`.

7.17.8 Uma aplicação mais útil das conversões

Há casos em que os operadores de conversão podem ser muito úteis. Suponha-se que se pretende definir um tipo `Palavra` que se comporta como um `int` em quase tudo, mas fornece algumas possibilidades adicionais, tais como acesso individualizado aos seus *bits* (ver Secção 2.7.4). O novo tipo poderia ser concretizado à custa de uma classe C++:

```
class Palavra {
  public:
    Palavra(int const valor = 0);
    operator int() const;
    bool bit(int const n) const;

  private:
    int valor;
};

inline Palavra::Palavra(int const valor)
    : valor(valor)
{
}

inline Palavra::operator int() const
{
    return valor;
}

inline bool Palavra::bit(int const n) const
{
    return (valor & (1 << n)) != 0;
}
```

Esta definição tornaria possível escrever

```
Palavra p = 99996;

p = p + 4;

std::cout << "Valor é: " << p << std::endl;

std::cout << "Em binário: ";

for(int i = 32; i != 0; --i)
    std::cout << p.bit(i - 1);
std::cout << std::endl;
```

que resultaria em

```
Valor é: 100000
Em binário: 00000000000000011000011010100000
```

Esta classe C++, para ser verdadeiramente útil, deveria proporcionar outras operações, que ficam como exercício para o leitor.