Capítulo 3

Modularização: rotinas

Methods are more important than facts.

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

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

3.1 Introdução à modularização

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2 Funções e procedimentos: rotinas

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

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

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

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

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

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

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

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

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

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

3.2.1 Abordagens descendente e ascendente

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

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

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

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

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

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

```
#include <iostream>
```

```
using namespace std;

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

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

```
#include <iostream>

using namespace std;

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

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

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

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

```
#include <iostream>

using namespace std;

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

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

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

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

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

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

```
A soma de 6/9 com 7/3 é 3/1.
```

Então o problema pode ser resolvido como se segue:

```
#include <iostream>

using namespace std;

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

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

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

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

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

```
#include <iostream>

using namespace std;

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

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

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

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

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

pelo que o programa fica:

```
#include <iostream>

using namespace std;

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

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

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

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

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

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

```
#include <iostream>

using namespace std;

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

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

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

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

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

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

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

```
#include <iostream>

using namespace std;
```

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

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

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

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

```
#include <iostream>

using namespace std;

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

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

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

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

3.2.2 Definição de rotinas

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
    return k;
}
```

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

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

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

    return k;
}
```

3.2.3 Sintaxe das definições de funções

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

```
tipo_de_devolução nome(lista_de_parâmetros)
```

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

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

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

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

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

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

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

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

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

   return k;
}
```

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

3.2.4 Contrato e documentação de uma rotina

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

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

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

    return k;
}
```

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

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

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

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

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

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

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

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

3.2.5 Integração da função no programa

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

```
#include <iostream>

using namespace std;

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

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

    return k;
}

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

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

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

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

3.2.6 Sintaxe e semântica da invocação ou chamada

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

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

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

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

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

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

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

Que acontece quando o código apresentado é executado?

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

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

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

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

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

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

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

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

3.2.7 Parâmetros

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

3.2.8 Argumentos

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

3.2.9 Retorno e devolução

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

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

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

```
return k;
```

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

3.2.10 Significado de `void`

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

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

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

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

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

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

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

O programa completo é então:

```
#include <iostream>

using namespace std;

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

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

    return k;
}

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

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

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

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

3.2.11 Passagem de argumentos por valor e por referência

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

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

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

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

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

1. São construídas as variáveis `x` e `y`.
2. Sendo parâmetros do procedimento, a variável `x` é inicializada com o valor 1 (valor de `a`) e a variável `y` é inicializada com o valor 2 (valor de `b`). Assim, os parâmetros são *cópias* dos argumentos.
3. Durante a execução do procedimento os valores guardados em `x` e `y` são trocados, ver Figura 3.1.
4. Antes de o procedimento terminar, as variáveis `x` e `y` têm valores 2 e 1 respectivamente.
5. Quando termina a execução do procedimento, as variáveis `x` e `y` são destruídas (ver explicação mais à frente)

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

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

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

Ao invocar como anteriormente, ver Figura 3.2:

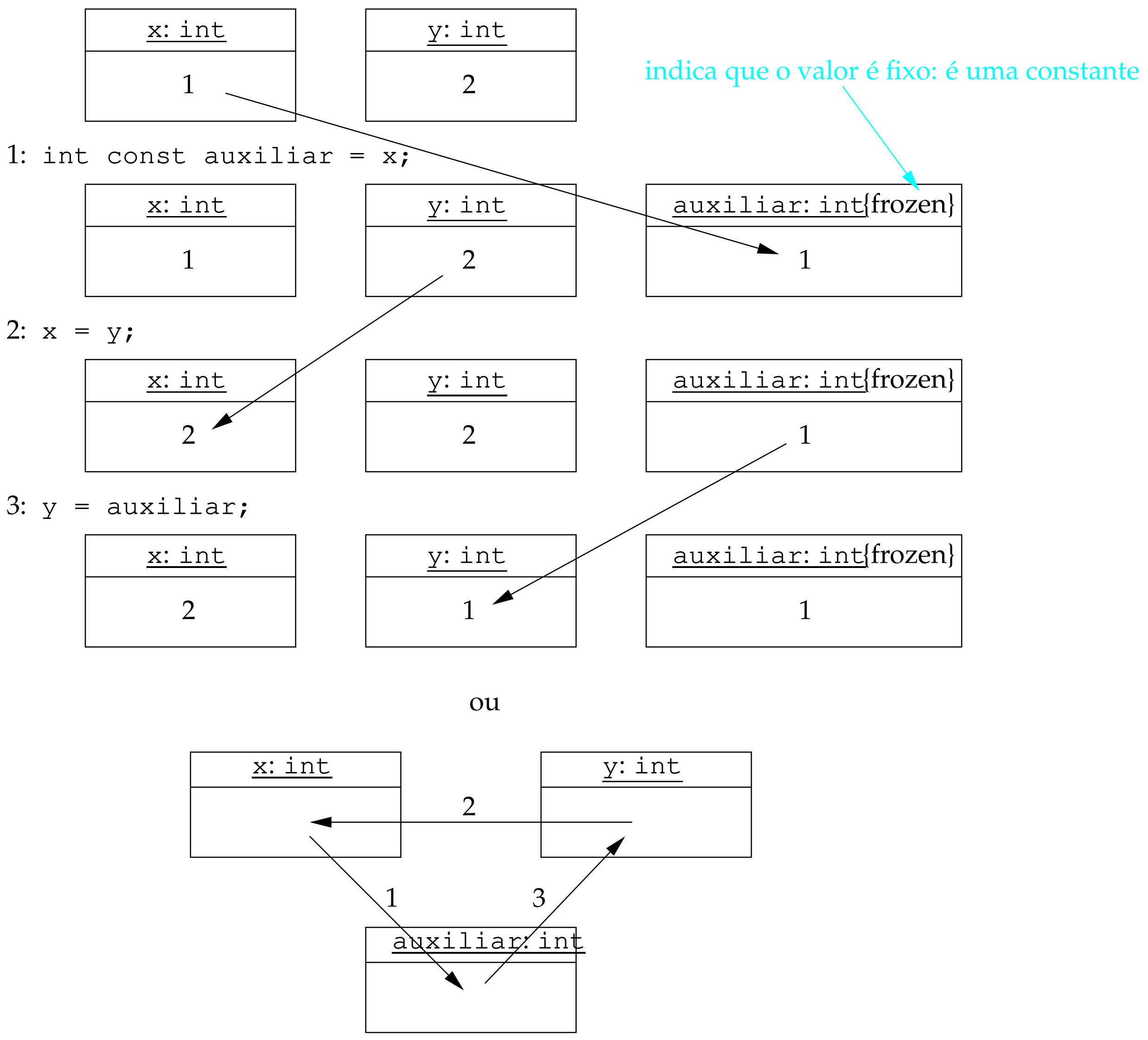


Figura 3.1: Algoritmo usual de troca de valores entre duas variáveis `x` e `y` através de uma constante auxiliar.

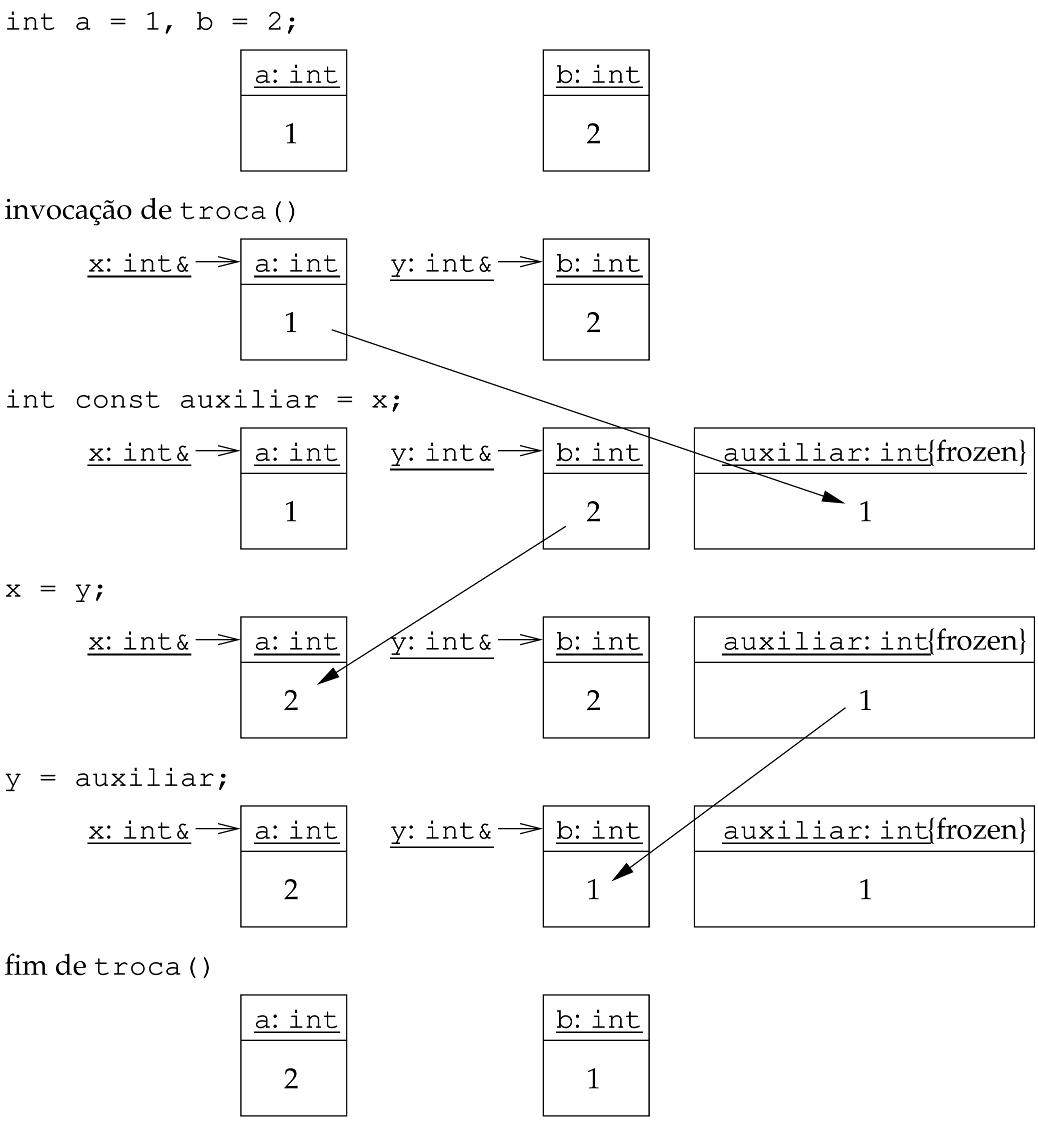


Figura 3.2: Diagramas com evolução do estado do programa que invoca o procedimento `troca()` entre cada instrução.

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

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

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

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

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

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

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

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

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

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

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

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

```
#include <iostream>

using namespace std;

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

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

    return k;
}

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

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

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

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

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

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

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

O programa completo é então:

```
#include <iostream>

using namespace std;

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

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

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

    return k;
}

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

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

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

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

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

3.2.12 Variáveis locais e globais

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

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

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

3.2.13 Blocos de instruções ou instruções compostas

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

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

as três instruções

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

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

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

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

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

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

3.2.14 Âmbito ou visibilidade de variáveis

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

Por exemplo:

```
#include <iostream>
```

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

```
using namespace std;

double const pi = 3.1416;

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

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

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

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

Neste código:

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

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

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

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

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

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

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

Em cada contexto `n` é uma variável diferente.

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

```
double f = 1.0;

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

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

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

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

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

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

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

3.2.15 Duração ou permanência de variáveis

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

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

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

Inicialização

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

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

3.2.16 Nomes de rotinas

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

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

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

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

A linguagem C++ é imperativa, i.e., os programas consistem em sequências de instruções. Assim, o corpo de uma função diz como se calcula qualquer coisa, mas não diz *o que* se calcula. É de toda a conveniência que, usando as regras indicadas, esse *o que* fique o mais possível explícito no nome da função, passando-se o mesmo quanto aos procedimentos. Isto, claro está,

[7]No caso de métodos de classes, ver Capítulo 7, o sujeito é o objecto em causa, ou seja, o objecto para o qual o método foi invocado.

não excluindo a necessidade de comentar funções e procedimentos com as respectivas PC e CO.

Finalmente, é de toda a conveniência que se use um estilo de programação uniforme. Tal facilita a compreensão do código escrito por outros programadores ou pelo próprio programador depois de passados uns meses sobre a escrita do código. Assim, sugerem-se as seguintes regras adicionais:

- Os nomes de variáveis e constantes devem ser escritos em minúsculas usando-se o sublinhado _ para separar as palavras. Por exemplo:

```
int const máximo_de_alunos_por_turma = 50;
int alunos_na_turma = 10;
```

- Os nomes de funções ou procedimentos devem ser escritos em minúsculas usando-se letras maiúsculas iniciais em todas as palavras excepto a primeira:

```
int númeroDeAlunos();
```

Recomendações mais gerais sobre nomes e formato de nomes em C++ podem ser encontradas no Apêndice D.

3.2.17 Declaração vs. definição

Antes de se poder invocar uma rotina é necessário que esta seja declarada, i.e., que o compilador saiba que ela existe e qual a sua interface. Declarar uma rotina consiste pois em dizer qual o seu nome, qual o tipo do valor devolvido, quantos parâmetros tem e de que tipo são esses parâmetros. Ou seja, declarar consiste em especificar o cabeçalho da rotina. Por exemplo,

```
void imprimeValorLógico(bool b);
```

ou simplesmente, visto que o nome dos parâmetros é irrelevante numa declaração,

```
void imprimeValorLógico(bool);
```

são possíveis declarações da função `imprimeValorLógico()` que se define abaixo:

```
/** Imprime “verdadeiro” ou “falso” consoante o valor lógico do argumento.
    @pre  PC ≡ V.
    @post CO ≡ surge escrito no ecrã “verdadeiro” ou “falso” conforme
               o valor de b. */
void imprimeValorLógico(bool b)
{
    if(b)
        cout << "verdadeiro";
    else
        cout << "falso";
}
```

Como se viu, na declaração não é necessário indicar os nomes dos parâmetros. Na prática é conveniente fazê-lo para mostrar claramente ao leitor o significado das entradas da função.

A sintaxe das declarações em sentido estrito é simples: o cabeçalho da rotina (como na definição) mas seguido de `;` em vez do corpo. O facto de uma rotina estar declarada não a dispensa de ter de ser definida mais cedo ou mais tarde: todas as rotinas têm de estar definidas em algum lado. Por outro lado, uma definição, um vez que contém o cabeçalho da rotina, também serve de declaração. Assim, após uma definição, as declarações em sentido estrito são desnecessárias.

Note-se que, mesmo que já se tenha procedido à declaração prévia de uma rotina, é necessário voltar a incluir o cabeçalho na definição.

O facto de uma definição ser também uma declaração foi usado no programa da soma de fracções para, colocando as definições das rotinas antes da função `main()`, permitir que elas fossem usadas no corpo da função `main()`. É possível inverter a ordem das definições através de declarações prévias:

```
#include <iostream>

using namespace std;

/** Calcula e escreve a soma em termos mínimos de duas fracções
    (positivas) lidas do teclado: */
int main()
{
    // Declaração dos procedimentos necessários. Estas declarações são visíveis
    // apenas dentro da função main().
    void reduzFracção(int& n, int& d);
    void escreveFracção(int n, int d);

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

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

    // Escrita do resultado:
    cout << "A soma de ";
    escreveFracção(n1, d1);
    cout << " com ";
    escreveFracção(n2, d2);
```

```
    cout << " é ";
    escreveFracção(n, d);
    cout << '.' << endl;
}

/** Reduz a fracção passada com argumento na forma de dois inteiros positivos.
    @pre PC ≡ n = n ∧ d = d ∧ 0 < n ∧ 0 < d
    @post CO ≡ n/d = n/d ∧ mdc(n, d) = 1 */
void reduzFracção(int& n, int& d)
{
    // Declaração da função necessária. Esta declaração é visível apenas
    // dentro desta função.
    int mdc(int m, int n);

    int const k = mdc(n, d);
    n /= k;
    d /= k;
}

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

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

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

    return k;
}
```

A vantagem desta disposição é que aparecem primeiro as rotinas mais globais e só mais tarde os pormenores, o que facilita a leitura do código.

Poder-se-ia alternativamente ter declarado as rotinas fora das funções e procedimentos em que são necessários. Nesse caso o programa seria:

```
#include <iostream>

using namespace std;

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

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

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

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

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

    // Escrita do resultado:
    cout << "A soma de ";
    escreveFracção(n1, d1);
```

```
    cout << " com ";
    escreveFracção(n2, d2);
    cout << " é ";
    escreveFracção(n, d);
    cout << '.' << endl;
}

void reduzFracção(int& n, int& d)
{
    int const k = mdc(n, d);
    n /= k;
    d /= k;
}

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

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

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

    return k;
}
```

Esta disposição é mais usual que a primeira. Repare-se que neste caso a documentação das rotinas aparece junto com a sua declaração. É que essa documentação é fundamental para saber o que a rotina faz, e portanto faz parte da interface da rotina. Como uma declaração é a especificação completa da interface, é natural que seja a declaração a ser documentada, e não a definição.

3.2.18 Parâmetros constantes

É comum que os parâmetros de uma rotina não mudem de valor durante a sua execução. Nesse caso é bom hábito explicitá-lo, tornando os parâmetros constantes. Dessa forma, enganos do programador serão assinalados prontamente: se alguma instrução da rotina tentar alterar o valor de um parâmetro constante, o compilador assinalará o erro. O mesmo argumento pode ser aplicado não só a parâmetros mas a qualquer variável: se não é suposto que o seu valor mude depois de inicializada, então deveria ser uma constante, e não uma variável.

No entanto, do ponto de vista do consumidor de uma rotina, a constância de um parâmetro correspondente a uma passagem de argumentos por valor é perfeitamente irrelevante: para o consumidor da rotina é suficiente saber que ela não alterará o argumento. A alteração ou não do parâmetro é um pormenor de implementação, e por isso importante apenas para o produtor da rotina. Assim, é comum indicar-se a constância de parâmetros associados a passagens de argumentos por valor *apenas na definição das rotinas e não na sua declaração*.

Estas ideias aplicadas ao programa da soma de fracções conduzem a

```
#include <iostream>

using namespace std;

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

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

/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre  PC ≡ 0 < m ∧ 0 < n.
    @post CO ≡ mdc = mdc(m, n). */
int mdc(int m, int n);

/** Devolve o menor de dois inteiros passados como argumentos.
    @pre  PC ≡ V (ou seja, nenhuma pré-condição).
    @post CO ≡ (mínimo = a ∧ a ≤ b) ∨ (mínimo = b ∧ b ≤ a). */
int mínimo(int a, int b);

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

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

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

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

void reduzFracção(int& n, int& d)
{
    int const k = mdc(n, d);
    n /= k;
    d /= k;
}

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

int mdc(int const m, int const n)
{
    int k = mínimo(m, n);

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

    return k;
}

int mínimo(int const a, int const b)
{
    if(a < b)
        return a;
    else
        return b;
```

```
}
```

Repare-se que se aproveitou para melhorar a modularização do programa acrescentando uma função para calcular o mínimo de dois valores.

3.2.19 Instruções de asserção

Que deve suceder quando o contrato de uma rotina é violado? A violação de um contrato decorre sempre, sem excepção, de um *erro de programação*. É muito importante perceber-se que assim é.

Em primeiro lugar, tem de se distinguir claramente os papéis dos vários intervenientes no processo de escrita e execução de um programa:

[programador] produtor é aquele que escreveu uma ferramenta, tipicamente um módulo (e.g., uma rotina).

[programador] consumidor é aquele que usa uma ferramenta, tipicamente um módulo (e.g., uma rotina), com determinado objectivo.

utilizador do programa é aquele que faz uso do programa.

A responsabilidade pela violação de um contrato nunca é do utilizador do programa. A responsabilidade é sempre de um programador. Se a pré-condição de uma rotina for violada, a responsabilidade é do programador consumidor da rotina. Se a condição objectivo de uma rotina for violada, e admitindo que a respectiva pré-condição não o foi, a responsabilidade é do programador fabricante dessa rotina.

Se uma pré-condição de uma rotina for violada, o contrato assinado entre produtor e consumidor não é válido, e portanto o produtor é livre de escolher o que a rotina faz nessas circunstâncias. Pode fazer o que entender, desde devolver lixo (no caso de uma função), a apagar o disco rígido e escrever uma mensagem perversa no ecrã: tudo é válido.

Claro está que isso não é desejável. O ideal seria que, se uma pré-condição ou uma condição objectivo falhassem, esse erro fosse assinalado claramente. No Capítulo 14 ver-se-á que o mecanismo das excepções é o mais adequado para lidar com este tipo de situações. Para já, à falta de melhor, optar-se-á por abortar imediatamente a execução do programa escrevendo-se uma mensagem de erro apropriada no ecrã.

Considere-se de novo a função para cálculo do máximo divisor comum:

```
/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre  PC ≡ 0 < m ∧ 0 < n.
    @post CO ≡ mdc = mdc(m, n). */
int mdc(int const m, int const n);
{
```

```
    int k = mínimo(m, n);

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

    return k;
}
```

Que sucede se a pré-condição for violada? Suponha-se que a função é invocada com argumentos 10 e -6. Então a variável `k` será inicializada com o valor -6. A guarda do ciclo é inicialmente verdadeira, pelo que o valor de `k` passa para -7. Para este valor de `k` a guarda será também verdadeira. E sê-lo-á também para -8, etc. Tem-se portanto um ciclo infinito? Não exactamente. Como se viu no Capítulo 2, as variáveis de tipos inteiros guardam uma gama de valores limitados. Quando se atingir o limite inferior dessa gama, o valor de `k` passará para o maior inteiro representável. Daí descerá, lentamente, inteiro por inteiro, até ao valor 2, que levará finalmente à falsidade da guarda, à consequente terminação do ciclo, e à devolução do valor correcto! Isso ocorrerá muito, mesmo muito tempo depois de chamada a função, visto que exigirá exactamente 4294967288 iterações do ciclo numa máquina onde os inteiros tenham 32 *bits*...

É certamente curioso este resultado. Mesmo que a pré-condição seja violada, o algoritmo, em algumas circunstâncias, devolve o resultado correcto. Há duas lições a tirar deste facto:

1. A função, tal como definida, é demasiado restritiva. Uma vez que faz sentido calcular o máximo divisor comum de quaisquer inteiros, mesmo negativos, desde que não sejam ambos nulos, a função deveria tê-lo previsto desde o início e a pré-condição deveria ter sido consideravelmente relaxada. Isso será feito mais abaixo.

2. Se o contrato é violado qualquer coisa pode acontecer, incluindo uma função devolver o resultado correcto... Quando isso acontece há normalmente uma outra característica desejável que não se verifica. Neste caso é a eficiência.

Claro está que nem sempre a violação de um contrato leva à devolução do valor correcto. Aliás, isso raramente acontece. Repare-se no que acontece quando se invoca a função `mdc()` com argumentos 0 e 10, por exemplo. Nesse caso o valor inicial de `k` é 0, o que leva a que a condição da instrução iterativa `while` tente calcular uma divisão por zero. Logo que isso sucede o programa aborta com uma mensagem de erro pouco simpática e semelhante à seguinte:

```
Floating exception (core dumped)
```

Existe uma forma padronizada de explicitar as condições que devem ser verdadeiras nos vários locais de um programa, obrigando o computador a verificar a sua validade durante a execução do programa: as chamadas instruções de asserção (afirmação). Para se poder usar instruções de asserção tem de ser incluir o ficheiro de interface apropriado:

```
#include <cassert>
```

As instruções de asserção têm o formato

```
assert(condição);
```

onde *condição* é uma condição que deve ser verdadeira no local onde a instrução se encontra para que o programa se possa considerar correcto. Se a condição for verdadeira quando a instrução de asserção é executada, nada acontece: a instrução não tem qualquer efeito. Mas se a condição for falsa o programa aborta e é mostrada uma mensagem de erro com o seguinte aspecto:

```
ficheiro_executável: ficheiro_fonte:linha: cabeçalho: Assertion 'condição' failed.
```

onde:

ficheiro_executável Nome do ficheiro executável onde ocorreu o erro.

ficheiro_fonte Nome do ficheiro em linguagem C++ onde ocorreu o erro.

linha Número da linha do ficheiro C++ onde ocorreu o erro.

cabeçalho Cabeçalho da função ou procedimento onde ocorreu o erro (só em alguns compiladores).

condição Condição que deveria ser verdadeira mas não o era.

Deve-se usar uma instrução de asserção para verificar a veracidade da pré-condição da função `mdc()`:

```
/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre   PC ≡ 0 < m ∧ 0 < n.
    @post  CO ≡ mdc = mdc(m, n). */
int mdc(int const m, int const n);
{
    assert(0 < m and 0 < n);

    int k = mínimo(m, n);

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

    return k;
}
```

Suponha-se o seguinte programa usando a função `mdc()`:

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

using namespace std;

/** Devolve o menor de dois inteiros passados como argumentos.
    @pre  PC ≡ V (ou seja, nenhuma pré-condição).
    @post CO ≡ (mínimo = a ∧ a ≤ b) ∨ (mínimo = b ∧ b ≤ a). */
int mínimo(int const a, int const b)
{
    if(a < b)
        return a;
    else
        return b;
}

/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre  PC ≡ 0 < m ∧ 0 < n.
    @post CO ≡ mdc = mdc(m, n). */
int mdc(int const m, int const n)
{
    assert(0 < m and 0 < n);

    int k = mínimo(m, n);

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

    return k;
}

int main()
{
    cout << mdc(0, 10) << endl;
}
```

A execução deste programa leva à seguinte mensagem de erro:

```
teste: teste.C:23: int mdc(int, int): Assertion `0 < m and 0 < n' failed.
Abort (core dumped)
```

É claro que este tipo de mensagens é muito mais útil para o programador que o simples abortar do programa ou, pior, a produção pelo programa de resultados errados.

Suponha-se agora o programa original, para o cálculo da soma de fracções, mas já equipado com instruções de asserção verificando as pré-condições dos vários módulos que o compõem (exceptuando os que não impõem qualquer pré-condição):

```
#include <iostream>

using namespace std;

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

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

/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre  PC ≡ 0 < m ∧ 0 < n.
    @post CO ≡ mdc = mdc(m, n). */
int mdc(int m, int n);

/** Devolve o menor de dois inteiros passados como argumentos.
    @pre  PC ≡ V (ou seja, nenhuma pré-condição).
    @post CO ≡ (mínimo = a ∧ a ≤ b) ∨ (mínimo = b ∧ b ≤ a). */
int mínimo(int a, int b);

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

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

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

void reduzFracção(int& n, int& d)
{
    assert(0 < n and 0 < d);

    int const k = mdc(n, d);
    n /= k;
    d /= k;
}

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

int mdc(int const m, int const n)
{
    assert(0 < m and 0 < n);

    int k = mínimo(m, n);

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

    return k;
}

int mínimo(int const a, int const b)
{
    if(a < b)
        return a;
    else
        return b;
}
```

Que há de errado com este programa? Considere-se o que acontece se o seu utilizador introduzir fracções negativas, por exemplo:

```
-6 7 15 7
```

Neste caso o programa aborta com uma mensagem de erro porque a pré-condição do procedimento `reduzFracção()` foi violada. Um programa não deve abortar nunca. Nunca mesmo. De quem é a responsabilidade disso acontecer neste caso? Do utilizador do programa, que desobedeceu introduzindo fracções negativas apesar de instruído para não o fazer, ou do programador produtor da função `main()` e consumidor do procedimento `reduzFracção()`? A resposta correcta é a segunda: a culpa nunca é do utilizador, é do programador. Isto tem de ficar absolutamente claro: o utilizador tem sempre razão.

Como resolver o problema? Há duas soluções. A primeira diz que deve ser o programador consumidor a garantir que os valores passados nos argumentos de `reduzFracção()` têm de ser positivos, conforme se estabeleceu na sua pré-condição. Em geral é este o caminho certo, embora neste caso se possa olhar para o problema com um pouco mais de cuidado e reconhecer que a redução de fracções só deveria ser proibida se o denominador fosse nulo. Isso implica, naturalmente, refazer o procedimento `reduzFracção()`, relaxando a sua pré-condição. O que ressalta daqui é que quanto mais leonina (forte) uma pré-condição, menos trabalho tem o programador produtor do módulo e mais trabalho tem o programador consumidor. Pelo contrário, se a pré-condição for fraca, isso implica mais trabalho para o produtor e menos para o consumidor.

Em qualquer dos casos, continua a haver circunstâncias nas quais as pré-condições do procedimento podem ser violadas. É sempre responsabilidade do programador consumidor garantir que isso não acontece. A solução mais simples seria usar um ciclo para pedir de novo ao utilizador para introduzir as fracções enquanto estas não verificassem as condições pretendidas. Tal solução não será ensaiada aqui, por uma questão de simplicidade. Adoptar-se-á a solução algo simplista de terminar o programa sem efectuar os cálculos no caso de haver problemas com os valores das fracções:

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

    if(n1 < 0 or d1 < 0 or n2 < 0 or d2 < 0)
        cout << "Termos negativos.  Nada feito." << endl;
    else {
        reduzFracção(n1, d1);
        reduzFracção(n2, d2);

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

```
            reduzFracção(n, d);

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

Neste caso é evidente que não haverá violação de nenhuma pré-condição. Muitos terão a tentação de perguntar para que servem as asserções neste momento, e se não seria apropriado eliminá-las. Há várias razões para as asserções continuarem a ser indispensáveis:

1. O programador, enquanto produtor, não deve assumir nada acerca dos consumidores (muito embora em muitos casos produtor e consumidor sejam uma e a mesma pessoa). O melhor é mesmo colocar a asserção: o diabo tece-as.
2. O produtor deve escrever uma rotina pensando em possíveis reutilizações futuras. Pode haver erros nas futuras utilizações, pelo que o mais seguro é mesmo manter a asserção.
3. Se alguém fizer alterações no programa pode introduzir erros. A asserção nesse caso permitirá a sua rápida detecção e correcção.

Parece ter faltado algo em toda esta discussão: a condição objectivo. Tal como se deve usar asserções para verificar as pré-condições, também se deve usar asserções para verificar as condições objectivo. A falsidade de uma condição objectivo, sabendo que a respectiva pré-condição é verdadeira, é também devida a um erro de programação, só que desta vez o responsável pelo erro é o programador produtor. Assim, as asserções usadas para verificar as pré-condições servem para o produtor de uma rotina facilitar a detecção de erros do programador consumidor, enquanto as asserções usadas para verificar as condições objectivo servem para o produtor de uma rotina se proteger dos seus próprios erros.

Transformar as condições objectivo em asserções é, em geral, uma tarefa mais difícil que no caso das pré-condições, que tendem a ser mais simples. As maiores dificuldades surgem especialmente se a condição objectivo contiver quantificadores (somatórios, produtos, quaisquer que seja, existe uns, etc.), se estiverem envolvidas inserções ou extracções de canais (entradas e saídas) ou se a condição objectivo fizer menção aos valores originais das variáveis, i.e., ao valor que as variáveis possuiam no início da rotina em causa.

Considere-se cada uma das rotinas do programa.

O caso da função `mínimo()` é fácil de resolver, pois a condição objectivo traduz-se facilmente para C++. É necessário, no entanto, abandonar os retornos imediatos e guardar o valor a devolver numa variável a usar na asserção[8]:

```
/** Devolve o menor de dois inteiros passados como argumentos.
    @pre   PC ≡ V (ou seja, nenhuma pré-condição).
    @post  CO ≡ (mínimo = a ∧ a ≤ b) ∨ (mínimo = b ∧ b ≤ a). */
int mínimo(int const a, int const b)
{
    int mínimo;

    if(a < b)
        mínimo = a;
    else
        mínimo = b;

    assert((mínimo == a and a <= b) or (mínimo == b and b <= a));

    return mínimo;
}
```

Quanto ao procedimento `reduzFracção()`, é fácil verificar o segundo termo da condição objectivo.

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

    int const k = mdc(n, d);
    n /= k;
    d /= k;

    assert(mdc(n, d) == 1);
}
```

Mas, e o primeiro termo, que se refere aos valores originais de `n` e `d`? Há linguagens, como Eiffel [11], nas quais as asserções correspondentes às condições objectivo podem fazer recurso aos valores das variáveis no início da respectiva rotina, usando-se para isso uma notação especial. Em C++ não é possível fazê-lo, infelizmente. Por isso o primeiro termo da condição objectivo ficará por verificar.

O procedimento `escreveFracção()` tem um problema: o seu objectivo é escrever no ecrã. Não é fácil formalizar uma condição objectivo que envolve alterações do ecrã, como se pode

[8] É comum usar-se o truque de guardar o valor a devolver por uma função numa variável com o mesmo nome da função, pois nesse caso a asserção tem exactamente o mesmo aspecto que a condição-objectivo.

ver pela utilização de português vernáculo na condição objectivo. É ainda menos fácil escrever uma instrução de asserção nessas circunstâncias. Este procedimento fica, pois, sem qualquer instrução de asserção.

Finalmente, falta a função `mdc()`. Neste caso a condição objectivo faz uso de uma função matemática mdc. Não faz qualquer sentido escrever a instrução de asserção como se segue:

```
/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre  PC ≡ 0 < m ∧ 0 < n.
    @post CO ≡ mdc = mdc(m, n). */
int mdc(int const m, int const n)
{
    assert(0 < n and 0 < d);

    int k = mínimo(m, n);

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

    assert(k == mdc(m, n)); // absurdo!

    return k;
}
```

Porquê? Simplesmente porque isso implica uma invocação recursiva interminável à função (ver Secção 3.3). Isso significa que se deverá exprimir a condição objectivo numa forma menos compacta:

$$CO \equiv \mathtt{m} \div \mathtt{mdc} = 0 \wedge \mathtt{n} \div \mathtt{mdc} = 0 \wedge (\mathbf{Q}\, j\ :\ \mathtt{mdc} < j\ :\ \mathtt{m} \div j \neq 0 \vee \mathtt{n} \div j \neq 0)\,.$$

Os dois primeiros termos da condição objectivo têm tradução directa para C++. Mas o segundo recorre a um quantificador que significa "qualquer que seja j maior que o valor devolvido pela função, esse j não pode ser divisor comum de `m` e `n`" (os quantificadores serão abordados mais tarde). Não há nenhuma forma simples de escrever uma instrução de asserção recorrendo a quantificadores, excepto invocando uma função que use um ciclo para verificar o valor do quantificador. Mas essa solução, além de complicada, obriga à implementação de uma função adicional, cuja condição objectivo recorre de novo ao quantificador. Ou seja, não é solução... Assim, o melhor que se pode fazer é reter os dois primeiros termos da condição objectivo reescrita:

```
/** Calcula e devolve o máximo divisor comum de dois inteiros positivos passados
    como argumentos.
    @pre  PC ≡ 0 < m ∧ 0 < n.
    @post CO ≡ mdc = mdc(m, n). */
int mdc(int const m, int const n)
```

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

    int k = mínimo(m, n);

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

    assert(m % k == 0 and n % k == 0);

    return k;
}
```

Estas dificuldades não devem levar ao abandono pura e simples do esforço de expressar pré-condições e condições objectivo na forma de instruções de asserção. A vantagem das instruções de asserção por si só é enorme, além de que o esforço de as escrever exige uma completa compreensão do problema, o que leva naturalmente a menos erros na implementação da respectiva resolução.

A colocação criteriosa de instruções de asserção é, pois, um mecanismo extremamente útil para a depuração de programas. Mas tem uma desvantagem aparente: as verificações da asserções consomem tempo. Para quê, então, continuar a fazer essas verificações quando o programa já estiver liberto de erros? O mecanismo das instruções de asserção é interessante porque permite evitar esta desvantagem com elegância: basta definir uma macro de nome `NDEBUG` (*no debug*) para que as asserções deixem de ser verificadas e portanto deixem de consumir tempo, não sendo necessário apagá-las do código. As macros serão explicadas no Capítulo 9, sendo suficiente para já saber que a maior parte dos compiladores para Unix (ou Linux) permitem a definição dessa macro de uma forma muito simples: basta passar a opção `-DNDEBUG` ao compilador.

3.2.20 Melhorando módulos já produzidos

Uma das vantagens da modularização, como se viu, é que se pode melhorar a implementação de qualquer módulo sem com isso comprometer o funcionamento do sistema e sem obrigar a qualquer outra alteração. Na versão do programa da soma de fracções que se segue utiliza-se uma função de cálculo do mdc com um algoritmo diferente, mais eficiente. É o algoritmo de Euclides, que decorre naturalmente das seguintes propriedades do mdc (lembra-se das sugestões no final do Capítulo 1?):

1. $\mathrm{mdc}(m,n) = \mathrm{mdc}(n \div m, m)$ se $0 < m \wedge 0 \leq n$.
2. $\mathrm{mdc}(m,n) = n$ se $m = 0 \wedge 0 < n$.

O algoritmo usado deixa de ser uma busca exaustiva do mdc para passar a ser uma redução sucessiva do problema até à trivialidade: a aplicação sucessiva da propriedade1 vai reduzindo os valores até um deles ser zero. A demonstração da sua correcção faz-se exactamente da mesma forma que no caso da busca exaustiva, e fica como exercício para o leitor. Regresse-se a este algoritmo depois de ter lido sobre metodologias de desenvolvimentos de ciclos no Capítulo 4.

Aproveitou-se ainda para relaxar as pré-condições da função, uma vez que o algoritmo utilizado permite calcular o mdc de dois inteiros `m` e `n` qualquer que seja `m` desde que `n` seja positivo. Este relaxar das pré-condições permite que o programa some convenientemente fracções negativas, para o que foi apenas necessário alterar o procedimento `reduzFracção()` de modo a garantir que o denominador é sempre positivo.

```
#include <iostream>

using namespace std;

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

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

/** Calcula e devolve o máximo divisor comum de dois inteiros passados
    como argumentos (o segundo inteiro tem de ser positivo).
    @pre PC ≡ m = m ∧ n = n ∧ 0 < n.
    @post CO ≡ mdc = mdc(|m|, n). */
int mdc(int m, int n);

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

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

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

void reduzFracção(int& n, int& d)
{
    assert(d != 0);

    if(d < 0) {
        n = -n;
        d = -d;
    }
    int const k = mdc(n, d);
    n /= k;
    d /= k;

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

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

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

    if(m < 0)
        m = -m;
    while(m != 0) {
        int const auxiliar = n % m;
        n = m;
        m = auxiliar;
    }

    return n;
}
```

3.3 Rotinas recursivas

O C++, como a maior parte das linguagens de programação imperativas, permite a definição daquilo a que se chama rotinas recursivas. Diz-se que uma rotina é recursiva se o seu corpo incluir chamadas à própria rotina[9]. Por exemplo

```
/** Devolve o factorial do inteiro passado como argumento.
    @pre   PC ≡ 0 ≤ n.
    @post  CO ≡ factorial = n! (ou ainda factorial = ∏_{i=1}^{n} i). */
int factorial(int const n)
{
    if(n == 0 or n == 1)
        return 1;
    return n * factorial(n - 1);
}
```

é uma função recursiva que calcula o factorial e que foi obtida de uma forma imediata a partir da definição recorrente do factorial

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

usando-se adicionalmente o facto de que 1! é também 1.

Este tipo de rotinas pode ser muito útil na resolução de alguns problemas, mas deve ser usado com cautela. A chamada de uma rotina recursivamente implica que as variáveis locais (parâmetros incluídos) são construídas tantas vezes quantas a rotina é chamada e só são destruídas quando as correspondentes chamadas retornam. Como as chamadas recursivas se aninham umas dentro das outras, se ocorrerem muitas chamadas recursivas não só pode ser necessária muita memória para as várias versões das variáveis locais (uma versão por cada chamada aninhada), como também a execução pode tornar-se bastante lenta, pois a chamada de funções implica alguma perda de tempo nas tarefas de "arrumação da casa" do processador.

A função `factorial()`, em particular, pode ser implementada usando um ciclo, que resulta em código muito mais eficiente e claro:

```
/** Devolve o factorial do inteiro passado como argumento.
    @pre   PC ≡ 0 ≤ n.
    @post  CO ≡ factorial = n! (ou ainda factorial = ∏_{i=1}^{n} i). */
int factorial(int const n)
{
    int factorial = 1;
    for(int i = 0; i != n; ++i)
        factorial *= i + 1;

    return factorial;
}
```

[9] É possível ainda que a recursividade seja entre duas ou mais rotinas, que se chamam mutuamente.

Muito importante é também a garantia de que uma rotina recursiva termina sempre. A função factorial recursiva acima tem problemas graves quando é invocada com um argumento negativo. É que vai sendo chamada recursivamente a mesma função com valores do argumento cada vez menores (mais negativos), sem fim à vista. Podem acontecer duas coisas. Como cada chamada da função implica a construção de uma variável local (o parâmetro n) num espaço de memória reservado para a chamada pilha (*stack*), como se verá na próxima secção, esse espaço pode-se esgotar, o que leva o programa a abortar. Ou então, se por acaso houver muito, mas mesmo muito espaço disponível na pilha, as chamadas recursivas continuarão até se atingir o limite inferior dos inteiros nos argumentos. Nessa altura a chamada seguinte é feita com o menor dos inteiros menos uma unidade, que como se viu no Capítulo 2 é o maior dos inteiros. A partir daí os argumentos das chamadas recursivas começam a diminuir e, ao fim de muito, mas mesmo muito tempo, atingirão o valor 0, que terminará a sequência de chamadas recursivas, sendo devolvido um valor errado.

Os problemas não surgem apenas com argumentos negativos, na realidade. É que os valores do factorial crescem depressa, pelo que a função não pode ser invocada com argumentos demasiado grandes. No caso de os inteiros terem 32 *bits*, o limite a impor aos argumentos é que têm de ser inferiores a 13! A função deve sofrer uma actualização na pré-condição e, já agora, ser equipada com as instruções de asserção apropriadas:

```
/** Devolve o factorial do inteiro passado como argumento.
    @pre  PC ≡ 0 ≤ n ≤ 12.
    @post CO ≡ factorial = n! (ou ainda factorial = ∏_{i=1}^{n} i). */
int factorial(int const n)
{
    assert(0 <= n and n <= 12);

    if(n == 0 or n == 1)
        return 1;
    return n * factorial(n - 1);
}
```

o mesmo acontecendo com a versão não-recursiva:

```
/** Devolve o factorial do inteiro passado como argumento.
    @pre  PC ≡ 0 ≤ n ≤ 12.
    @post CO ≡ factorial = n! (ou ainda factorial = ∏_{i=1}^{n} i). */
int factorial(int const n)
{
    assert(0 <= n and n <= 12);

    int factorial = 1;
    for(int i = 0; i != n; ++i)
        factorial *= i + 1;

    return factorial;
}
```

Para se compreender profundamente o funcionamento das rotinas recursivas tem de se compreender o mecanismo de chamada ou invocação de rotinas, que se explica na próxima secção.

3.4 Mecanismo de invocação de rotinas

Quando nas secções anteriores se descreveu a chamada da função `mdc()`, referiu-se que os seus parâmetros eram construídos no início da chamada e destruídos no seu final, e que a função, ao terminar, retornava para a instrução imediatamente após a instrução de invocação. Como é que o processo de invocação funciona na prática? Apesar de ser matéria para a disciplina de Arquitectura de Computadores, far-se-á aqui uma descrição breve e simplificada do mecanismo de invocação de rotinas que será útil (embora não fundamental) para se compreender o funcionamento das rotinas recursivas.

O mecanismo de invocação de rotinas utiliza uma parte da memória do computador como se de uma pilha se tratasse, i.e., como um local onde se pode ir acumulando informação de tal forma que a última informação a ser colocada na pilha seja a primeira a ser retirada, um pouco como acontece com as pilhas de processos das repartições públicas, em os processos dos incautos podem ir envelhecendo ao longo de anos na base de uma pilha...

A pilha é utilizada para colocar todas as variáveis locais automáticas quando estas são construídas. O topo da pilha varia quando é executada uma instrução de definição de um variável automática. As variáveis definidas são colocadas (construídas) no topo da pilha e apenas são retiradas (destruídas) quando se abandona o bloco onde foram definidas. É também na pilha que se guarda o endereço da instrução para onde o fluxo de execução do programa deve retornar uma vez terminada a execução de uma rotina. No fundo, a pilha serve para o computador "saber a quantas anda".

Exemplo não-recursivo

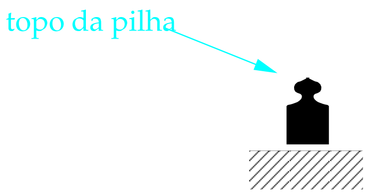
Suponha-se o seguinte programa, incluindo a função `mdc()`,

```
#include <iostream>

using namespace std;

/** Calcula e devolve o máximo divisor comum de dois inteiros passados
    como argumentos (o segundo inteiro tem de ser positivo).
    @pre  PC ≡ m = m ∧ n = n ∧ 0 < n.
    @post CO ≡ mdc = mdc(|m|, n). */
int mdc(int const m, int const n)
{
    assert(0 < n);

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

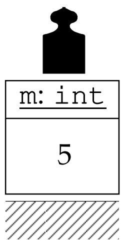
```
    while(m != 0) {
        int const auxiliar = n % m;
        n = m;
        m = auxiliar;
    }

    assert(m % k == 0 and n % k == 0);

    return n;
}

int main()
{
    int m = 5;                      // 1
                   mdc(m + 3, 6)    // 2A
    int divisor =                 ; // 2B
    cout << divisor << endl;        // 3
}
```

em que se dividiu a instrução 2 em duas "sub-instruções" 2A e 2B.

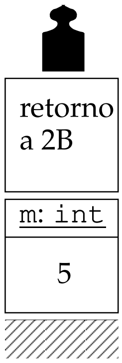
Considera-se, para simplificar, que pilha está vazia quando se começa a executar a função `main()`:



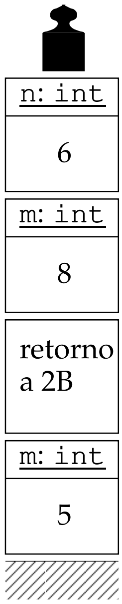
De seguida é executada a instrução 1, que constrói a variável `m`. Como essa variável é automática, é construída na pilha, que fica



Que acontece quando a instrução 2A é executada? A chamada da função `mdc()` na instrução 2A começa por guardar na pilha o endereço da instrução a executar quando a função retornar, i.e., 2B



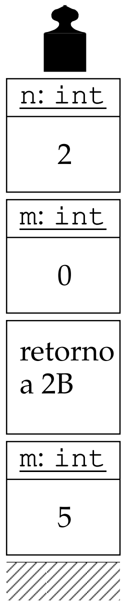
Em seguida são construídos na pilha os parâmetros da função. Cada parâmetro é inicializado com o valor do argumento respectivo:



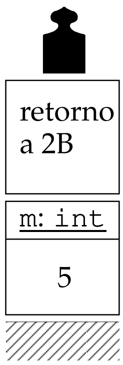
Neste momento existem na pilha duas variáveis de nome `m`: uma pertencente à função `main()` e outra à função `mdc()`. É fácil saber em cada instante quais são as variáveis automática visíveis: são todas as variáveis desde o topo da pilha até ao próximo endereço de retorno. Isto é, neste momento são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

A execução passa então para o corpo da função, onde durante o ciclo a constante local `auxiliar` é construída e destruída no topo da pilha várias vezes[10], até que o ciclo termina com valor desejado na variável `n`, i.e., 2. Assim, imediatamente antes da execução da instrução de retorno, a pilha contém:

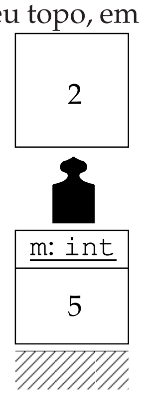
[10] Qualquer compilador minimamente inteligente evita este processo de construção e destruição repetitivas construindo a variável `auxiliar` na pilha logo no início da invocação da função.



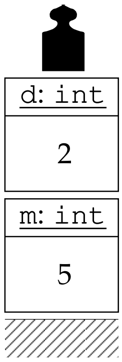
A instrução de retorno começa por calcular o valor a devolver (neste caso é o valor de `n`, i.e., 2), retira da pilha (destrói) todas as variáveis desde o topo até ao próximo endereço de retorno



e em seguida retira da pilha a instrução para onde o fluxo de execução deve ser retomado (i.e., 2B) colocando na pilha (mas para lá do seu topo, em situação periclitante...) o valor a devolver

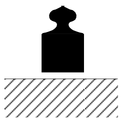


Em seguida a execução continua em 2B, que constrói a variável `divisor`, inicializando-a com o valor devolvido, colocado após o topo da pilha, que depois se deixa levar por uma "corrente de ar":



O valor de `d` é depois escrito no ecrã na instrução 3.

Finalmente atinge-se o final da função `main()`, o que leva à retirada da pilha (destruição) de todas as variáveis até à base (uma vez que não há nenhum endereço de retorno na pilha):



No final, a pilha fica exactamente como no início: vazia.

Exemplo recursivo

Suponha-se agora o seguinte exemplo, que envolve a chamada à função recursiva `factorial()`:

```
#include <iostream>

using namespace std;

/** Devolve o factorial do inteiro passado como argumento.
    @pre  PC ≡ 0 ≤ n ≤ 12.
    @post CO ≡ factorial = n! (ou ainda factorial = ∏_{i=1}^{n} i). */
int factorial(int const n)
{
    assert(0 <= n and n <= 12);

    if(n == 0 or n == 1)          // 1
        return 1;                 // 2
    return n * factorial(n - 1);  // 3
}
int main()
{
    cout << factorial(3) << endl; // 4
}
```

Mais uma vez é conveniente dividir a instrução 4 em duas “sub-instruções”

```
        factorial(3)          // 4A
cout <<              << endl; // 4B
```

uma vez que a função é invocada antes da escrita do resultado no ecrã. Da mesma forma, a instrução 3 pode ser dividida em duas "sub-instruções"

```
           factorial(n - 1)  // 3A
return n *                 ; // 3B
```

Ou seja,

```
#include <iostream>

using namespace std;

/** Devolve o factorial do inteiro passado como argumento.
    @pre  PC ≡ 0 ≤ n ≤ 12.
    @post CO ≡ factorial = n!  (ou ainda factorial = ∏_{i=1}^{n} i).  */
int factorial(int const n)
{
    assert(0 <= n and n <= 12);

    if(n == 0 or n == 1)          // 1
        return 1;                 // 2
               factorial(n - 1)  // 3A
    return n *                 ; // 3B
}
int main()
{
            factorial(3)          // 4A
    cout <<              << endl; // 4B
}
```

Que acontece ao ser executada a instrução 4A? Essa instrução contém uma chamada à função `factorial()`. Assim, tal como se viu antes, as variáveis locais da função (neste caso apenas o parâmetro constante `n`) são colocadas na pilha logo após o endereço da instrução a executar quando função retornar. Quando a execução passa para a instrução 1, já a pilha está já como indicado em (b) na Figura 3.3.

Em seguida, como a constante `n` contém 3 e portanto é diferente de 0 e de 1, é executada a instrução após o `if`, que é a instrução 3A (se tiver dúvidas acerca do funcionamento do if, consulte a Secção 4.1.1). Mas a instrução 3A consiste numa nova chamada à função, pelo que os passos acima se repetem, mas sendo agora o parâmetro inicializado com o valor do argumento, i.e., 2, e sendo o endereço de retorno 3B, resultando na pilha indicada em (c) na Figura 3.3.

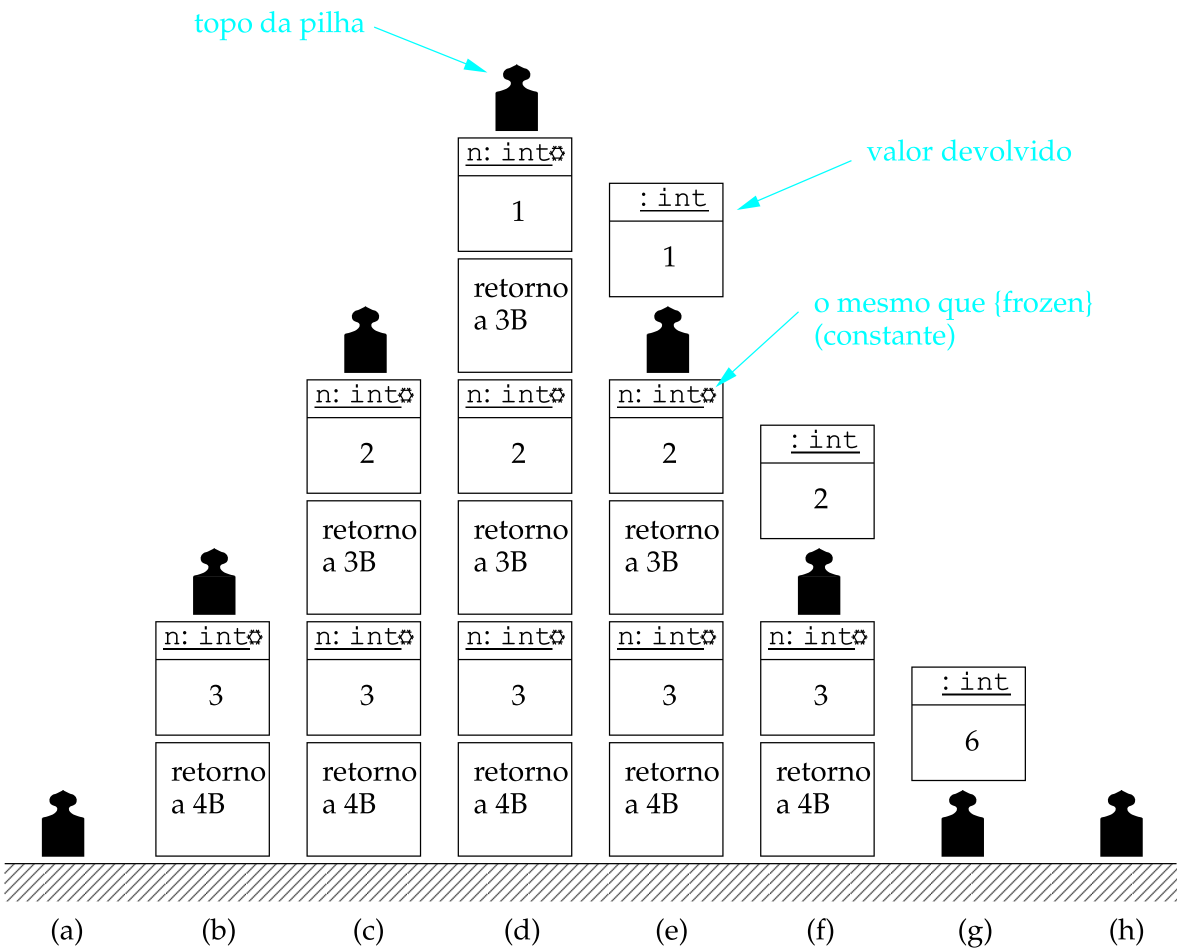


Figura 3.3: Evolução da pilha durante invocações recursivas da função `factorial()`. Admite-se que a pilha inicialmente está vazia, como indicado em (a).

Neste momento existem duas versões da constante `n` na pilha, uma por cada chamada à função que ainda não terminou (há uma chamada "principal" e outra "aninhada"). É esta repetição das variáveis e constantes locais que permite às funções recursivas funcionarem sem problemas.

A execução passa então para o início da função (instrução 1). De novo, como constante `n` da chamada em execução correntemente é 2, e portanto diferente de 0 e de 1, é executada a instrução após o `if`, que é a instrução 3A. Mas a instrução 3A consiste numa nova chamada à função, pelo que os passos acima se repetem, mas sendo agora o parâmetro inicializado com o valor do argumento, i.e., 1, e sendo o endereço de retorno 3B, resultando na pilha indicada em (d) na Figura 3.3.

A execução passa então para o início da função (instrução 1). Agora, como a constante `n` da chamada em execução correntemente é 1, é executada a instrução condicionada pelo `if`, que é a instrução 2. Mas a instrução 2 consiste numa instrução de retorno com devolução do valor 1. Assim, as variáveis e constantes locais são retiradas da pilha, o endereço de retorno (3B) é retirado da pilha, o valor de devolução 1 é colocado após o topo da pilha, e a execução continua na instrução 3B, ficando a pilha indicada em (e) na Figura 3.3.

A instrução 3B consiste numa instrução de retorno com devolução do valor `n * 1`(ou seja 2), em que `n` tem o valor 2 e o 1 é o valor de devolução da chamada anterior, que ficou após o topo da pilha. Assim, as variáveis e constantes locais são retiradas da pilha, o endereço de retorno (3B) é retirado da pilha, o valor de devolução 2 é colocado após o topo da pilha, e a execução continua na instrução 3B, ficando a pilha indicada em (f) na Figura 3.3.

A instrução 3B consiste numa instrução de retorno com devolução do valor `n * 2` (ou seja 6), em que `n` tem o valor 3 e o 2 é o valor de devolução da chamada anterior, que ficou após o topo da pilha. Assim, as variáveis locais e constantes locais são retiradas da pilha, o endereço de retorno (4B) é retirado da pilha, o valor de devolução 6 é colocado após o topo da pilha, e a execução continua na instrução 4B, ficando a pilha indicada em (g) na Figura 3.3.

A instrução 4B corresponde simplesmente a escrever no ecrã o valor devolvido pela chamada à função, ou seja, 6 (que é 3!, o factorial de 3). É de notar que, terminadas todas as chamadas à função, a pilha voltou à sua situação original (que se supôs ser vazia)indicada em (h) na Figura 3.3.

A razão pela qual as chamadas recursivas funcionam como espectável é que, em cada chamada aninhada, são criadas novas versões das variáveis e constantes locais e dos parâmetros (convenientemente inicializados) da rotina. Embora os exemplos acima se tenham baseado em funções, é evidente que o mesmo mecanismo é usado para os procedimentos, embora simplificado pois estes não devolvem qualquer valor.

3.5 Sobrecarga de nomes

Em certos casos é importante ter rotinas que fazem conceptualmente a mesma operação ou o mesmo cálculo, mas que operam com tipos diferentes de dados. Seria pois de todo o interesse que fosse permitida a definição de rotinas com nomes idênticos, distintos apenas no tipo dos seus parâmetros. De facto, a linguagem C++ apenas proíbe a definição no mesmo contexto de funções ou procedimentos com a mesma *assinatura*, i.e., não apenas com o mesmo nome, mas também com a mesma lista dos tipos dos parâmetros[11]. Assim, é de permitida a definição de múltiplas rotinas com o mesmo nome, desde que difiram no número ou tipo de parâmetros. As rotinas com o mesmo nome dizem-se *sobrecarregadas*. A invocação de rotinas sobrecarregadas faz-se como habitualmente, sendo a rotina que é de facto invocada determinada a partir do número e tipo dos argumentos usados na invocação. Por exemplo, suponha-se que estão definidas as funções:

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

int soma(int const a, int const b, int const c)
{
    return a + b + c;
}

float soma(float const a, float const b)
{
    return a + b;
}

double soma(double const a, double const b)
{
    return a + b;
}
```

Ao executar as instruções

```
int i1, i2;
float f1, f2;
double d1, d2;
i2 = soma(i1, 4);      // invoca int soma(int, int).
i2 = soma(i1, 3, i2); // invoca int soma(int, int, int).
f2 = soma(5.6f, f1);  // invoca float soma(float, float).
d2 = soma(d1, 10.0);  // invoca double soma(double, double).
```

são chamadas as funções apropriadas para cada tipo de argumentos usados. Este tipo de comportamento emula para as funções definidas pelo programador o comportamento normal dos operadores do C++. A operação `+`, por exemplo, significa soma de `int` se os operandos forem `int`, significa soma de `float` se os operandos forem `float`, etc. O exemplo mais claro talvez seja o do operador divisão (`/`). As instruções

[11]A noção de assinatura usual é um pouco mais completa que na linguagem C++, pois inclui o tipo de devolução. Na linguagem C++ o tipo de devolução não faz parte da assinatura.

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

têm como resultado no ecrã

```
0 0.5
```

porque no primeiro caso, sendo os operandos inteiros, a divisão usada é a divisão inteira. Assim, cada operador básico corresponde na realidade a vários operadores com o mesmo nome, i.e., sobrecarregados, cada um para determinado tipo dos operandos.

A assinatura de uma rotina corresponde à sequência composta pelo seu nome, pelo número de parâmetros, e pelos tipos dos parâmetros. Por exemplo, as funções `soma()` acima têm as seguintes assinaturas[12]:

- `soma,int,int`
- `soma,int,int,int`
- `soma,float,float`
- `soma,double,double`

O tipo de devolução de uma rotina não faz parte da sua assinatura, não servindo portanto para distinguir entre funções ou procedimentos sobrecarregados.

Num capítulo posterior se verá que é possível sobrecarregar os significados dos operadores básicos (como o operador +) quando aplicados a tipos definidos pelo programador, o que transforma o C++ numa linguagem que se "artilha" de uma forma muito elegante e potente.

3.6 Parâmetros com argumentos por omissão

O C++ permite a definição de rotinas em que alguns parâmetros têm argumentos por omissão. I.e., se não forem colocados os argumentos respectivos numa invocação da rotina, os parâmetros serão inicializados com os valores dos argumentos por omissão. Mas com uma restrição: os parâmetros com argumentos por omissão têm de ser os últimos da rotina.

Por exemplo, a definição

```
int soma(int const a = 0, int const b = 0,
         int const c = 0, int const d = 0)
{
    return a + b + c + d;
}
```

[12] A constância de um parâmetro (desde que não seja um referência) não afecta a assinatura, pois esta reflecte a interface da rotina, e não a sua implementação.

permite a invocação da função `soma()` com qualquer número de argumentos até 4:

```
cout << soma() << endl              // Surge 0.
cout << soma(1) << endl             // Surge 1.
cout << soma(1, 2) << endl          // Surge 3.
cout << soma(1, 2, 3) << endl       // Surge 6.
cout << soma(1, 2, 3, 4) << endl; // Surge 10.
```

Normalmente os argumentos por omissão indicam-se apenas na declaração de um rotina. Assim, se a declaração da função `soma()` fosse feita separadamente da respectiva definição, o código deveria passar a ser:

```
int soma(int const a = 0, int const b = 0,
         int const c = 0, int const d = 0);

...

int soma(int const a, int const b, int const c, int const d)
{
    return a + b + c + d;
}
```