Capítulo 2

Conceitos básicos de programação

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

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

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

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

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

2.1 Introdução

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

[1] E não só, como se verá.

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

using namespace std;

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

    // Aqui aparecem as instruções que compõem o programa.

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

A primeira linha,

```
#include <iostream>
```

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

A linha seguinte,

```
using namespace std;
```

é uma directiva de utilização do espaço nominativo `std`, e serve para se poder escrever simplesmente `cout` em vez de `std::cout`. Nem sempre o seu uso é recomendável [12, pág.171], sendo usado aqui apenas para simplificar a apresentação sem tornar os programas inválidos (muitas vezes estas duas linhas iniciais não serão incluídas nos exemplos, devendo o leitor estar atento a esse facto se resolver compilar esses exemplos). Os espaços nominativos serão abordados no Capítulo 9.

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

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

```
#include <iostream>

using namespace std;
```

```
/// Este programa calcula o máximo divisor comum de dois números.
int main()
{
    cout << "Máximo divisor comum de dois números."
         << endl;
    cout << "Introduza dois inteiros positivos: ";
    int m, n;
    cin >> m >> n; // Assume-se que m e n são positivos!

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

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

    // Como mdc(m, n) ≤ k (invariante do ciclo) e k divide m e n
    // (o ciclo terminou, não foi?), conclui-se que k = mdc(m, n)!
    cout << "O máximo divisor comum de " << m
         << " e " << n << " é " << k << '.' << endl;
}
```

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

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

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

```
        // Se o ciclo não parou, então k não divide m e n, logo,
        // k ≠ mdc(m, n) ∧ mdc(m, n) ≤ k. Ou seja, mdc(m, n) < k.
        --k; // Progresso. Afinal, o algoritmo tem de terminar...
        // Neste momento, mdc(m, n) ≤ k outra vez! É o invariante do ciclo!
    }
```

As duas partes restantes resolvem dois problemas que foram subtilmente ignorados na capítulo anterior:

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

Claramente as entradas do programa têm de vir de uma entidade exterior ao programa. Tudo depende da aplicação que se pretende dar ao programa. Possibilidades seriam as entradas serem originadas:

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

para apresentar apenas alguns exemplos.

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

2.1.1 Consola e canais

É típico os programas actuais usarem interfaces mais ou menos sofisticadas com o seu utilizador. Ao longo destas folhas, no entanto, adoptar-se-á um modelo muito simplificado (e primitivo) de interacção com o utilizador. Admite-se que os programas desenvolvidos são executados numa consola de comandos. Admite-se ainda que as entradas do programa serão feitas por um utilizador usando o teclado (ou que vêem de um ficheiro de texto no disco rígido do computador) e que as saídas serão escritas na própria consola de comandos (ou que são colocadas num ficheiro no disco rígido do computador).

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

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

Num programa em C++ o canal de saída para o ecrã[3] é designado por `cout`. O canal de entrada de dados pelo teclado é designado por `cin`. Para efectuar uma operação de escrita no ecrã usa-se o operador de inserção `<<`. Para efectuar uma operação de leitura de dados do teclado usa-se o operador de extracção `>>`.

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

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

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

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

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

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

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

Após uma operação de extracção de valores do canal de entrada, a variável para onde a extracção foi efectuada toma o valor que foi inserido no teclado pelo utilizador do programa, desde que esse valor possa ser tomado por esse tipo de variável[4].

Ao ser executada uma operação de leitura como acima, o computador interrompe a execução do programa até que seja introduzido algum valor no teclado. Todos os espaços em branco são

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

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

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

ignorados. Consideram-se espaços em branco os espaços propriamente ditos (' '), os tabuladores (que são os caracteres que se insere quando se carrega na tecla `tab` ou ⇥) e os fins-de-linha (que são caracteres especiais que assinalam o fim de uma linha de texto e que se insere quando se carrega na tecla `return` ou ↩). Para tornar evidente a presença de um espaço quando se mostra um sequência de caracteres usar-se-á o símbolo ␣. Compare-se:

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

com

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

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

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

resultam em

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

Mas se se usar o manipulador `endl`

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

o resultado é

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

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

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

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

2.1.2 Definição de variáveis

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

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

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

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

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

```
int k = 121;
```

No entanto, no exemplo dado, nenhuma das variáveis pode ser inicializada com um valor que tenha algum significado. As variáveis `m` e `n` têm de ser lidas por operações de extracção, pelo que a sua inicialização acabaria por não ter qualquer efeito prático, uma vez que os valores iniciais seriam imediatamente substituídos pelos valores extraídos do canal de entrada. À variável `k`, por outro lado, só se pode atribuir o valor desejado depois de saber qual é a menor das outras duas variáveis, pelo que a sua inicialização também seria inútil[5].

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

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

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

using namespace std;

int main()
{
    ...

   int k = min(m, n);

    ...
}
```

2.1.3 Controlo de fluxo

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

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

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

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

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

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

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

```
k = m;
```

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

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

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

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

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

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

1. O operador `or` que calcula a disjunção dos valores lógicos de duas expressões condicionais.

2. O operador `!=`, que tem valor $\mathcal{V}$ se os seus operandos forem diferentes e o valor $\mathcal{F}$ se eles forem iguais. A notação usada para a diferença deve-se à inexistência do símbolo $\neq$ na generalidade dos teclados (e tabelas de codificação).

3. O operador `%`, cujo valor é o resto da divisão inteira do primeiro operando pelo segundo. A notação usada (percentagem) deve-se à inexistência do símbolo $\div$ na generalidade do teclados.

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

2.2 Variáveis

2.2.1 Memória e inicialização

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

Os tipos das variáveis não passam, na realidade, de uma abstracção. Todas as variáveis, independentemente do seu tipo, são representadas na memória do computador por padrões de *bits*[6], os famosos "zeros e uns", colocados na zona de memória atribuída a essa variável. O tipo de uma variável indica simplesmente como um dado padrão de *bits* deve ser interpretado.

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

Cada variável tem duas características estáticas, um nome e um tipo, e uma característica dinâmica, o seu valor[7], tal como nos algoritmos. Antes de usar uma variável é necessário indicar ao compilador qual o seu nome e tipo, de modo a que a variável possa ser criada, i.e., ficar associada a uma posição de memória (ou várias posições de memória), e de modo a que a forma de interpretar os padrões de *bits* nessa posição de memória fique estabelecida. Uma instrução onde se cria uma variável com um dado nome, de um determinado tipo, e com um determinado valor inicial, denomina-se *definição*. A instrução:

```
int a = 10;
```

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

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

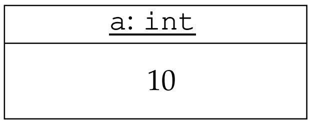

Figura 2.1: Notação gráfica para uma variável definida por `int a = 10;`. O nome e o tipo da variável estão num compartimento à parte, no topo, e têm de ser sublinhados.

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

```
int a;
```

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

[7] Mais tarde se aprenderá que existe um outro género de variáveis que não têm nome. São as variáveis dinâmicas introduzidas no Capítulo 11.

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

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

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

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

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

define três variáveis todas do tipo `int`.

2.2.2 Nomes de variáveis

Os nomes de variáveis (e em geral de todos os identificadores em C++) podem ser constituídos por letras[10] (sendo as minúsculas distinguidas das maiúsculas), dígitos decimais, e também pelo caractere '_' (sublinhado ou *underscore*), que se usa normalmente para aumentar a legibilidade do nome. O nome de uma variável não pode conter espaços nem pode começar por um dígito. Durante a tradução do programa, o compilador, antes de fazer uma análise sintáctica do programa, durante a qual verifica a gramática do programa, faz uma análise lexical, durante a qual identifica os símbolos (ou *tokens*) que constituem o texto. Esta análise lexical é "gulosa": os símbolos detectados (palavras, sinais de pontuação, etc.) são tão grandes quanto possível, pelo que `lêValor` é sempre interpretado como um único identificador (nome), e não como o identificador `lê` seguido do identificador `Valor`.

Os nomes das variáveis devem ser tão auto-explicativos quanto possível. Se uma variável guarda, por exemplo, o número de alunos numa turma, deve chamar-se `número_de_alunos_na_turma` ou então `número_de_alunos`, se o complemento "na turma" for evidente dado o contexto no programa. É claro que o nome usado para uma variável não tem qualquer importância para o compilador, mas uma escolha apropriada dos nomes pode aumentar grandemente a legibilidade dos programas pelos humanos...

2.2.3 Inicialização de variáveis

As posições de memória correspondentes às variáveis contêm sempre um padrão de *bits*: não há posições "vazias". Assim, as variáveis têm sempre um valor qualquer. Depois de uma

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

[10] A norma do C++ especifica que de facto se pode usar qualquer letra. Infelizmente a maior parte dos compiladores existentes recusa-se a aceitar caracteres acentuados. Este texto foi escrito ignorando esta restrição prática por duas razões:

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

definição, se não for feita uma inicialização explícita conforme sugerido mais atrás, a variável definida contém um valor arbitrário[11]. Uma fonte frequente de erros é o esquecimento de inicializar as variáveis definidas. Por isso, é recomendável a inicialização de todas as variáveis tão cedo quanto possível, desde que essa inicialização tenha algum significado para o resto do programa (e.g., no exemplo da Secção 1.4 essa inicialização não era possível[12]).

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

2.3 Tipos básicos

O tipo das variáveis está relacionado directamente com o conjunto de possíveis valores tomados pelas variáveis desse tipo. Para poder representar e manipular as variáveis na memória do computador, o compilador associa a cada tipo não só o número de *bits* necessário para a representação de um valor desse tipo na memória do computador, mas também a forma como os padrões de *bits* guardados em variáveis desse tipo devem ser interpretados. A linguagem C++ tem definidos *a priori* alguns tipos: os chamados tipos básicos do C++. Posteriormente estudar-se-ão duas filosofias de programação diferentes que passam por acrescentar à linguagem novos tipos mais apropriados para os problemas em causa: programação baseada em objectos (Capítulo 7) e programação orientada para objectos (Capítulo 12).

Nas tabelas seguintes são apresentados os tipos básicos existentes no C++ e a gama de valores que podem representar em Linux sobre Intel[13]. É muito importante notar que os computadores são máquinas finitas: tudo é limitado, desde a memória ao tamanho da representação dos tipos em memória. Assim, a gama de diferentes valores possíveis de guardar em variáveis de qualquer tipo é limitada. A gama de valores representável para cada tipo de dados pode variar com o processador e o sistema operativo. Por exemplo, no sistema operativo Linux em processadores Alpha, os `long int` (segunda tabela) têm 64 bits. Em geral só se pode afirmar que a gama dos `long` é sempre suficiente para abarcar qualquer `int` e a gama dos `int` é sempre suficiente para abarcar qualquer `short`, o mesmo acontecendo com os `long double` relativamente aos `double` e com os `double` relativamente aos `float`.

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

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

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

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

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

Tabela 2.1: Tipos básicos elementares.

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

Tabela 2.2: Outros tipos básicos.

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

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

2.3.1 Tipos aritméticos

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

Representação de inteiros

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

b_3	b_2	b_1	b_0

em que os b_i com $i = 0, \cdots, 3$ são *bits*, tomando portanto os valores 0 ou 1. É fácil verificar que existem apenas $2^4 = 16$ padrões diferentes de *bits* possíveis de colocar numa destas variáveis. Como associar valores inteiros a cada um desses padrões? A resposta mais óbvia é a que é usada na prática: considere-se que o valor representado é $(b_3 b_2 b_1 b_0)_2$, i.e., que os *bits* são dígitos de um número expresso na base binária (ou seja, na base 2). Por exemplo:

1	0	0	1

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

Suponha-se agora que a variável contém

[14]Os números inteiros podem-se representar de muitas formas. A representação mais evidente, corresponde a desenhar um traço por cada unidade: ||||||||||||| representa o inteiro treze ("treze" por sua vez é outra representação, por extenso). A representação romana do mesmo inteiro é XIII. A representação árabe é posicional e é a mais prática. Usam-se sempre os mesmos 10 dígitos que vão aumentando de peso da direita para a esquerda (o peso é multiplicado sucessivamente por 10) e onde o dígito zero é fundamental. A representação árabe do mesmo inteiro é 13, que significa $1 \times 10 + 3$. A representação árabe usa 10 dígitos e por isso diz-se que usa a base 10, ou que a representação é decimal. Pode-se usar a mesma representação posicional com qualquer base, desde que se forneçam os respectivos dígitos. Em geral a notação posicional tem a forma $(d_{n-1}d_{n-2}\cdots d_1 d_0)_B$ onde B é a base da representação, n é o número de dígitos usado neste número em particular, e d_i com $i = 0, \cdots, n-1$ são os sucessivos dígitos, sendo que cada um deles pertence a um conjunto com B possíveis dígitos possuindo valores de

1	1	1	1

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

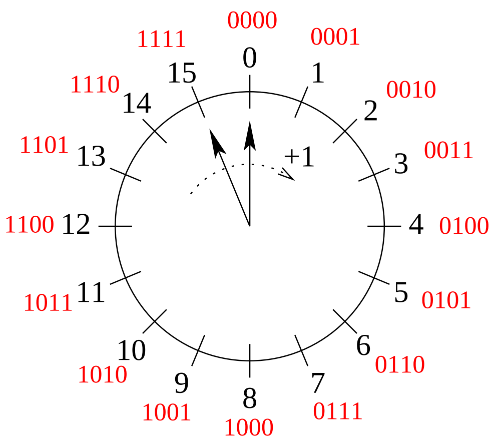

Figura 2.2: Representação de inteiros sem sinal com quatro *bits*.

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

É extremamente importante recordar as limitações dos tipos. Em particular os valores das variáveis do tipo `int` não podem crescer indefinidamente. Ao se atingir o topo do relógio volta-se a zero!

Falta verificar como representar inteiros com sinal, i.e., incluindo não só valores positivos mas também valores negativos. Suponha-se que os `int` têm, de novo num computador hipotético, apenas 4 *bits*. Admita-se uma representação semelhante à dos `unsigned int`, mas diga-se

0 a $B-1$. O número representado por $(d_{n-1}d_{n-2}\cdots d_1 d_0)_B$ pode ser calculado como

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

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

[15]Tipicamente no topo do relógio estaria 16, e não zero, mas optar-se-á pela numeração a começar em zero, que é quase sempre mais vantajosa (ver Nota 4 na página 237).

que, no lugar das 15 horas (padrão de *bits* 1111), mesmo antes de chegar de novo ao padrão 0000, o valor representado é x, ver Figura 2.3. Pelo que se disse anteriormente, somar 1 a x corresponde a rodar o ponteiro do padrão 1111 para o padrão 0000. Admitindo que o padrão 0000 representa de facto o valor 0, tem-se $x + 1 = 0$, donde se conclui ser o padrão 1111 um bom candidato para representar o valor $x = -1$! Estendendo o argumento anterior, pode-se dizer que as 14 horas correspondem à representação de -2, e assim sucessivamente, como se indica na Figura 2.4.

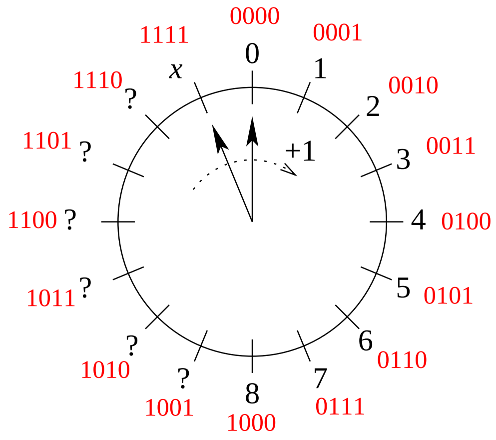

Figura 2.3: Como representar os inteiros com sinal?

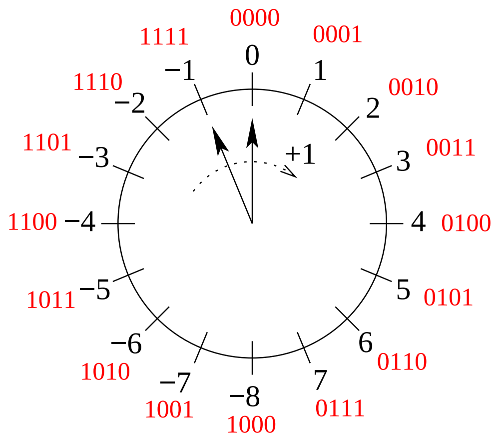

Figura 2.4: Representação de inteiros com sinal com quatro *bits*.

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

Como saber, olhando para um padrão de *bits*, qual o valor representado? Primeiro olha-se para o *bit* de sinal. Se for 0, interpreta-se o padrão de *bits* como um número binário: é esse o valor representado. Se o *bit* de sinal for 1, então o valor representado é igual ao valor binário correspondente ao padrão de *bits* subtraído de 16. Por exemplo, observando o padrão 1011, conclui-se que representa o valor negativo $(1011)_2 - 16 = 11 - 16 = -5$, como se pode confirmar na Figura 2.4.

A extensão destas ideias para o caso dos `int` com 32 bits é muito simples. Neste caso os valores representados variam de -2^{31} a $2^{31} - 1$ e a interpretação dos valores representados faz-se como indicado no parágrafo acima, só que subtraindo 2^{32}, em vez de 16, no caso dos valores negativos. Por exemplo, os padrões 00000000000000000000010000000001 e 10000000000000000000000000000111 representam os valores 1025 e -2147483641, como se pode verificar facilmente com uma máquina de calcular.

Representação de valores de vírgula flutuante

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

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

$$\begin{array}{|c|c|c|c|}\hline b_3 & b_2 & b_1 & b_0 \\ \hline\end{array}\, , \begin{array}{|c|c|c|c|}\hline b_{-1} & b_{-2} & b_{-3} & b_{-4} \\ \hline\end{array}$$

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

$$\begin{array}{|c|c|c|c|c|c|c|c|}\hline b_7 & b_6 & b_5 & b_4 & b_3 & b_2 & b_1 & b_0 \\ \hline\end{array}\, ,$$

O valor representado por um determinado padrão de *bits* $b_3 b_2 b_1 b_0 b_{-1} b_{-2} b_{-3} b_{-4}$ é dado por

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

O menor valor positivo representável nesta forma é $\frac{1}{16} = 0,0625$. Por outro lado, só se conseguem representar valores de -8 a -0,0625, o valor 0, e valores de 0,0625 a 7,9375. O número de dígitos decimais de precisão está entre 2 e 3, um antes da vírgula entre um e dois depois. Caso se utilize 32 *bits* e se desloque a vírgula 16 posições para a esquerda, o menor valor positivo representável é $\frac{1}{2^{16}} = 0,00001526$, e são representáveis valores de -32768 a -0,00001526, o valor 0, e valores de 0,00001526 a 32767,99998, aproximadamente, correspondendo a entre 9 e 10 dígitos decimais de precisão, cinco antes da vírgula e entre quatro e cinco depois.

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

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

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

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

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

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

1. atribui um *bit* (s_0) ao sinal (em que 0 significa positivo e 1 negativo),
2. atribui 23 *bits* (m_{-1} a m_{-23}) à mantissa, que são interpretados como um valor de vírgula fixa sem sinal com vírgula imediatamente antes do *bit* mais significativo, e
3. atribui oito *bits* (e_7 a e_0) ao expoente, que são interpretados como um inteiro entre 0 e 255 a que se subtrai 127, pelo que o expoente varia entre -126 e 127 (os valores 0 e 255, antes da subtracção são especiais);

ou seja,

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

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

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

Mantissa	Expoente	Valor representado
$m_{-1}\cdots m_{-23} = 0\cdots 0$	$e_7\cdots e_0 = 0\cdots 0$	± 0
$m_{-1}\cdots m_{-23} \neq 0\cdots 0$	$e_7\cdots e_0 = 0\cdots 0$	$\pm\,(0,m_{-1}\cdots m_{-23})_2 \times 2^{-126}$ (valores não-normalizados[16])
	$e_7\cdots e_0 \neq 0\cdots 0\,\wedge$ $e_7\cdots e_0 \neq 1\cdots 1$	$\pm\,(1,m_{-1}\cdots m_{-23})_2 \times 2^{(e_7\cdots e_0)_2-127}$ (valores normalizados[16])
$m_{-1}\cdots m_{-23} = 0\cdots 0$	$e_7\cdots e_0 = 1\cdots 1$	$\pm\infty$
$m_{-1}\cdots m_{-23} \neq 0\cdots 0$	$e_7\cdots e_0 = 1\cdots 1$	(valores especiais)

Quando os *bits* do expoente não são todos 0 nem todos 1 (i.e., $(e_7\cdots e_0)_2$ não é 0 nem 255), a mantissa tem um *bit* adicional à esquerda, implícito, com valor sempre 1. Nesse caso diz-se que os valores estão normalizados[16].

É fácil verificar que o menor valor positivo normalizado representável é

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

e o maior é

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

Comparem-se estes valores com os indicados para o tipo `float` na Tabela 2.1. Com esta representação conseguem-se cerca de seis dígitos decimais de precisão, menos quatro que a representação de vírgula fixa apresentada em primeiro lugar, mas uma gama bastante maior de valores representáveis.

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

No que diz respeito à programação, a utilização de valores de vírgula flutuante deve ser evitada sempre que possível: se for possível usar inteiros nos cálculos é preferível usá-los a recorrer aos tipos de vírgula flutuante, que, apesar da sua aparência inocente, reservam muitas surpresas. Em particular, é importante recordar que os valores são representados com precisão finita, o que implica arredondamentos e acumulações de erros. Outra fonte comum de erros prende-se com a conversão de valores na base decimal para os formatos de vírgula flutuante em base binária: valores inocentes como 0.4 não são representáveis exactamente (experimente escrevê-lo em base 2)! Ao estudo da forma de lidar com estes tipos de erros sem surpresas dá-se o nome de análise numérica [3].

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

2.3.2 Booleanos ou lógicos

Existe um tipo `bool` cujas variáveis guardam valores lógicos. A forma de representação usual tem oito *bits* e reserva o padrão 00000000 para representar o valor falso (`false`), todos os outros representando o valor verdadeiro (`true`). Os valores booleanos podem ser convertidos em inteiros. Nesse caso o valor `false` é convertido em 0 e o valor `true` é convertido em 1. Por exemplo,

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

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

2.3.3 Caracteres

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

A A $\mathcal{A}$ A

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

Em Linux sobre uma arquitectura Intel, as variáveis do tipo `char` são representadas em memória usando 8 *bits*, pelo que existem $2^8 = 256$ diferentes caracteres representáveis. Existem várias tabelas de codificação de caracteres diferentes, que a cada padrão de *bits* fazem corresponder um caractere diferente. De longe a mais usada neste canto da Europa é a tabela dos códigos Latin-1 (ou melhor, ISO-8859-1, ver Apêndice G), que é uma extensão da tabela ASCII (*American Standard Code for Information Interchange*) que inclui os caracteres acentuados em uso na Europa Ocidental. Existem muitas destas extensões, que podem ser usadas de acordo com os caracteres que se pretendem representar, i.e., de acordo com o alfabeto da língua mais utilizada localmente. Essas extensões são possíveis porque o código ASCII fazia uso apenas dos 7 *bits* menos significativos de um caractere (i.e., apenas 128 das possíveis combinações de zeros e uns)[17].

Não é necessário, conhecer os códigos dos caracteres de cor: para indicar o código do caractere correspondente à letra 'b' usa-se naturalmente `'b'`. Claro está que o que fica guardado numa variável não é `'b'`, é um padrão de *bits* correspondente ao inteiro 98, pelo menos se se usar

[17] Existe um outro tipo de codificação, o Unicode, que suporta todos os caracteres de todas as expressões escritas vivas ou mortas em simultâneo. Mas exige uma codificação diferente, com maior número de *bits*. O C++ possui um outro tipo, `wchar_t`, para representar caracteres com estas características.

a tabela de codificação Latin-1. Esta tradução de uma representação habitual, `'b'`, para um código específico, permite escrever programas sem quaisquer preocupações relativamente à tabela de codificação em uso.

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

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

aparece no ecrã

```
O caractere seguinte é: j
```

O que sucedeu foi que se adicionou 1 ao código do caractere 'i', de modo que a variável `caractere` passou a conter o código do caractere 'j'. Este pedaço de código só produz o efeito apresentado se, na tabela de codificação que está a ser usada, as letras sucessivas do alfabeto possuírem códigos sucessivos. Isso pode nem sempre acontecer. Por exemplo, na tabela EBCDIC *(Extended Binary Coded Decimal Interchange Code)*, já pouco usada, o caractere 'i' tem código 137 e o caractere 'j' tem código 145! Num sistema que usasse esse código, as instruções acima certamente não escreveriam a letra 'j' no ecrã.

Os `char` são interpretados como inteiros em C++. Estranhamente, se esses inteiros têm ou não sinal não é especificado pela linguagem. Quer em Linux sobre arquitecturas Intel quer no Windows NT, os `char` são interpretados como inteiros com sinal (ou seja, char≡signed `char`), de -128 a 127, devendo-se usar `unsigned char` para forçar uma interpretação como inteiros sem sinal, de 0 a 255.

O programa seguinte imprime no ecrã todos os caracteres da tabela ASCII (que só especifica os caracteres correspondentes aos códigos de 0 a 127, isto é, todos os valores positivos dos `char` em Linux e a primeira metade da tabela Latin-1)[18]:

```
#include <iostream>

using namespace std;

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

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

Quando é realizada uma operação de extracção para uma variável do tipo `char`, é lido apenas um caractere mesmo que no teclado seja inserido um código (ou mais do que um caractere). i.e., o resultado das seguintes instruções

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

seria

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

caso o utilizador inserisse 48 (após uns quantos espaços). O dígito oito foi ignorado visto que se leu apenas um caractere, e não o seu código.

2.4 Valores literais

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

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

Os valores inteiros podem ainda ser especificados em octal (base 8) ou hexadecimal (base 16). Inteiros precedidos de `0x` são considerados como representados na base hexadecimal e portanto podem incluir, para além dos 10 dígitos usuais, as letras entre A a F (com valores de 10 a 15), em maiúsculas ou em minúsculas. Inteiros precedidos de 0 simplesmente são considerados como representados na base octal e portanto podem incluir apenas dígitos entre 0 e 7. Por exemplo[19]:

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

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

Há um tipo adicional de valores literais: as cadeias de caracteres. Correspondem a sequências de caracteres colocados entre `""`. Para já não se dirá qual o tipo destes valores literais. Basta para já saber que se podem usar para escrever texto no ecrã (ver Secção 2.1.1).

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

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

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

2.5 Constantes

Nos programas em C++ também se pode definir constantes, i.e., "variáveis" que não mudam de valor durante toda a sua existência. Nas constantes o valor é uma característica estática e não dinâmica como nas variáveis. A definição de constantes faz-se colocando a palavra-chave[20] `const` após o tipo pretendido para a constante[21]. As constantes, justamente por o serem, têm obrigatoriamente de ser inicializadas no acto da definição, i.e., o seu valor estático tem de ser indicado na própria definição. Por exemplo:

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

A notação para uma constante é a indicada na Figura 2.5, que representa a constante `primeiro_primo`.

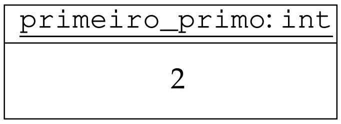

Figura 2.5: Notação usada para as constantes.

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

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

é preferível[22]

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

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

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

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

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

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

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

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

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

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

2.6 Instâncias

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

2.7 Expressões e operadores

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

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

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

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

Uma expressão é composta por valores (valores literais, valores de variáveis, de constantes, etc) e operações. Muitas vezes numa expressão existe um operador de atribuição = ou suas variantes[23] , ver Secção 2.7.5. A expressão

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

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

A expressão na última instrução de

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

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

Os operadores em C++ são de um de três tipos: unários, se tiverem apenas um operando, binários, se tiverem dois operandos, e ternários, se tiverem três operandos.

2.7.1 Operadores aritméticos

Os operadores aritméticos são

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

+ identidade (unário), e.g., `+5.5`;

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

- simétrico (unário), e.g., `-3`;

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.7.2 Operadores relacionais e de igualdade

Os operadores relacionais (todos binários) são

`>` maior,

`<` menor,

`>=` maior ou igual, e

`<=` menor ou igual,

com os significados óbvios.

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

`==` igual a, e

`!=` diferente de.

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

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

ou

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

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

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

2.7.3 Operadores lógicos

Os operadores lógicos (ou booleanos) aplicam-se operandos lógicos e são[24]

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

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

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

Por exemplo:

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

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

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

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

Estes operadores podem operar sobre operandos booleanos mas também sobre operandos aritméticos. Neste último caso, os operandos aritméticos são convertidos para valores lógicos, correspondendo o valor zero a $\mathcal{F}$ e qualquer outro valor diferente de zero a $\mathcal{V}$.

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

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

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

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

$\mathcal{F}$	falso
$\mathcal{V}$	verdadeiro
$\wedge$	conjunão
$\vee$	disjunão
$\oplus$	disjunão exclusiva
$\neg$	negaão

Figura 2.6: Tabelas de verdade das operações lógicas elementares.

2.7.4 Operadores *bit*-a-*bit*

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

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

Os operadores são

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

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

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

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

<< deslocamento para a esquerda (binário); e

>> deslocamento para a direita (binário).

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

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

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

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

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

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

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

Pois $1 = (0000000000000001)_2$ deslocado para a esquerda de quatro dígitos resulta em $1 = (0000000000010000)_2 = 16$ e $20 = (0000000000110100)_2$ deslocado para a direita de quatro posições resulta em $(0000000000000011)_2 = 3$. O deslocamento para a esquerda de n *bits* corresponde à multiplicação do inteiro por 2^n e o deslocamento de n *bits* para a direita corresponde à divisão inteira por 2^n.

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

2.7.5 Operadores de atribuição

A operação de atribuição, indicada pelo símbolo =, faz com que a variável que está à esquerda do operador (o primeiro operando) passe a tomar o valor do segundo operando. Uma consequência importante é que, ao contrário do que acontece quando os operadores que se viu até aqui são calculados, o estado do programa é afectado pelo cálculo de uma operação de atribuição: há uma variável que muda de valor. Por isso se diz que a atribuição é uma operação com efeitos laterais. Por exemplo[25]:

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

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

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

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

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

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

é que

1. a variável `i` fica com o valor 1, pois a conversão de `float` para `int` elimina a parte fraccionária (i.e., *não arredonda, trunca*), e
2. a variável `f` fica com o valor 1,0, pois é o resultado da conversão do valor 1 para `float`, sendo 1 o resultado da atribuição a `i`[26].

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

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

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

```
double const pi = 3.1415927;
pi = 4.5;  // absurdo! não faz sentido alterar o que é constante!
4.6 = 1.5; // pior ainda! que significado poderia ter semelhante instrução?
```

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

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

2.7.6 Operadores de incrementação e decrementação

As expressões da forma `i += 1` e `i -= 1`, por serem tão frequentes, merecem também uma forma especial de abreviação em C++: os operadores de incrementação e decrementação `++` e `--`. Estes dois operadores têm duas versões: a versão prefixo e a versão sufixo. Quando o objectivo é simplesmente incrementar ou decrementar uma variável, as duas versões podem ser consideradas equivalentes, embora a versão prefixo deva em geral ser preferida[27]:

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

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

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

escreve no ecrã os valores 1 e 0, enquanto

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

escreve no ecrã os valores 1 e 1.

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

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

2.7.7 Precedência e associatividade

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

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

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

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

```
210 209.9999847412109375
```

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

2.7.8 Efeitos laterais e mau comportamento

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

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

Assim:

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

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

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

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

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

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

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

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

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

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

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

2.7.9 Ordem de cálculo

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

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

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

`,` O primeiro operando é sempre calculado primeiro.

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

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

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

não se garante que `sin(x)` (seno de `x`) seja calculada em primeiro lugar e `sqrt(x)` (raiz quadrada de `x`) em último[28]. Se uma expressão não envolver operações com efeitos laterais, esta indefinição não afecta o programador. Quando a expressão tem efeitos laterais que afectam variáveis usadas noutros locais da mesma expressão, esta pode deixar de ter resultados bem definidos devido à indefinição quanto é ordem de cálculo. Nesse caso a expressão é mal comportada.

Por exemplo, depois das instruções

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

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

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

ou em

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

Este tipo de comportamento deve-se a que, como se viu, no C++ as atribuições e respectivas abreviações são operadores que têm um resultado e que portanto podem ser usados dentro de expressões envolventes. Se fossem instruções especiais, como em Pascal, era mais difícil escrever instruções mal comportadas[29]. Este tipo de operações, no entanto, fazem parte do estilo usual de programação em C++, pelo que devem ser bem percebidas as suas consequências: *uma expressão com efeitos laterais não pode ser interpretada como uma simples expressão matemática*, pois há variáveis que mudam de valor durante o cálculo! Expressões com efeitos laterais são pois de evitar, salvo nas "expressões idiomáticas" da linguagem C++. Como se verá, esta é também uma boa razão para se fazer uma distinção clara entre funções (sem efeitos laterais) e procedimentos (com efeitos laterais mas cujas invocações não podem constar numa expressão) quando se introduzir a modularização no próximo capítulo.

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

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