# Linguagens de Programação

## Uma sebenta

Editado por

Filipe Rodrigues
Raquel Bernardino

Instituto Superior de Economia e Gestão
2024

# Conteúdo

# Introdução

Saber programar é essencial no mundo em que vivemos porque a nossa civilização depende fortemente de software. A programação encontra-se em praticamente todo o lado, desde simples eletrodomésticos como máquinas de lavar até grandes objetos como navios, aviões, satélites, etc. Para programar é necessário usar uma linguagem de programação, isto é, uma linguagem codificada que possa ser entendida por computadores. Existem várias linguagens de programação tais como C++, C, C#, java, python, etc. Nesta cadeira estudaremos a linguagem C++ por ser uma das linguagens de programação mais utilizadas em todo o mundo e por estar disponível em quase todo o tipo de computadores. Com os conhecimentos adquiridos nesta cadeira poderão depois facilmente aprender outras linguagens de programação por vocês próprios.

Para escrever os programas em C++ usaremos o QT Creator. Além de permitir escrever e executar os nossos programas, o QT Creator tem ainda uma série de outras funcionalidades, nomeadamente aplicações gráficas, que não iremos explorar nesta cadeira, mas que poderão aprender sozinhos mais tarde, se assim o desejarem.

Esta sebenta contém alguma informação complementar sobre a linguagem de programação C++ que não será lecionada nem avaliada na unidade curricular de Linguagens de Programação. As secções com esta informação estão assinaladas como **Informação Complementar**.

## O básico

Programar é dizer ao computador o que tem que fazer para atingir um determinado objetivo ou para resolver um determinado problema. Para isso, é necessário escrever de forma detalhada as instruções que o computador tem que executar numa linguagem que ele consiga entender. Esse conjunto de instruções é aquilo a que chamamos *programa*. Sendo máquinas, os computadores não têm a capacidade de pensar e por isso todas as instruções devem ser escritas de forma detalhada e explicita. Para validar todas as instruções que foram escritas, o computador usa um *compilador*. A missão do compilador é verificar se o computador consegue perceber e executar todas as instruções que escrevemos. Isto significa que o compilador permite apenas identificar erros de escrita (chamados erros de *sintaxe*) no nosso código. O compilador não verifica erros de execução, isto é, não verifica se as instruções que estamos a escrever correspondem ao que queremos que, de facto, o computador faça. Para que o compilador consiga identificar o fim de cada uma das instruções que escrevemos, estas terminam geralmente com ponto e vírgula ";", havendo algumas exceções. No QT Creator, os erros de compilação aparecem a vermelho e têm obrigatoriamente que ser corrigidos para que possamos executar o programa. Além disso, o compilador lança ainda *warnings* (mostrados a amarelo) que devem também ser corrigidos, mas cuja não correção não impede a execução do programa. Como iremos perceber, alguns *warnings* podem ser ignorados pois não afetam o funcionamento do programa, no entanto, existem outros que deturpam completamente o funcionamento do programa. O compilador analisa todas as instruções que escrevemos exceto as que correspondem a *comentários* do programador e queixar-se-à sempre que alguma coisa

esteja errada. Os *comentários* são notas do programador e aparecem após o símbolo "//" ou entre os símbolos "/* ... */". Eles são completamente ignorados pelo compilador pelo que podemos escrever neles tudo o que quisermos. No QT Creator, os comentários aparecem a cor verde.

Os programas que vamos criar usam muitos comandos que foram previamente definidos por outras pessoas, tal como por exemplo a função seno ou a função raiz quadrada. A definição desses comandos envolve frequentemente muitas linhas de código e por isso encontra-se em *pacotes* específicos do C++. Estes comandos predefinidos (seno, raiz quadrada, potência, etc) podem ser usados diretamente nos programas que fazemos sem que tenhamos que saber como foram implementados originalmente. No entanto, para isso, teremos que informar primeiro o compilador onde é que esses comandos foram definidos. Para tal, usamos a instrução *#include* seguida do nome do pacote que contém os comandos que queremos usar. Por exemplo, todas as funções matemáticas estão disponíveis no pacote *cmath*, pelo que, se as quisermos usar no nosso programa, teremos que começar por escrever *#include <cmath>*. Aos poucos iremos perceber quais dos muitos pacotes existentes nos interessam. Para já, ficamos apenas com a ideia de que todas as funções matemáticas estão no pacote *cmath* e que o pacote *iostream* tem que ser incluído sempre que queiram ler/escrever informação do ecrã pois contém as definições de todos os comandos básicos de leitura e escrita que vamos usar.

O cabeçalho de um programa em C++ é designado por *Preâmbulo* e é nessa região que é feita a inclusão de todos os elementos externos que o programa precisa, como por exemplo os pacotes.

```
//Preâmbulo

int main(){
   //Escreva aqui o seu código

   /* Isto também
       é um
   comentário */

   return 0;
}
```

Quando um programa em C++ é executado, o ponto de partida é sempre a função *main* pelo que esta função tem sempre que existir no programa. Ao encontrar esta função, o programa percorre todas as linhas do código de forma sequencial de cima para baixo (a não ser que existam instruções que alterem o fluxo de execução do programa, como veremos mais adiante). A função *main*, é delimitada por duas chavetas "{ }" dentro das quais devemos escrever o nosso código. A última instrução da função *main* é "*return 0;*". Esta instrução era usada em muitos sistemas operativos para verificar se o programa tinha conseguido chegar ao fim sem qualquer problema. Assim sendo, o que está representado no código acima é o esqueleto de um qualquer programa em C++.

A função *main* apresentada acima tem apenas vários comentários no seu interior que serão completamente ignorados pelo compilador, pelo que o programa apresentado é na verdade um programa vazio pois não há qualquer instrução a ser executada.

# Capítulo 1

# Variáveis e operadores

As variáveis são os elementos base de qualquer linguagem de programação e servem essencialmente para guardar informação na memória do computador.

Na Secção 1.1 são apresentados tipos de variáveis. As Secções 1.2 e 1.3 contêm detalhes sobre a escrita e a leitura de variáveis para o ecrã, respetivamente. Na Secção 1.4 são apresentadas as variáveis constantes e, por fim, na Secção 1.5 são detalhados os vários tipos de operadores existentes.

## 1.1 Tipos de variáveis

Um objeto é uma região da memória do computador que pode conter um valor. Uma *variável* é um objeto com nome. Cada variável é caracterizada por um nome e por um tipo. Em relação ao nome, existem algumas regras que têm de ser respeitadas:

1. Só pode começar com letras (pode também começar com *underscore* mas devem evitar fazê-lo).
2. Não pode coincidir com palavras reservadas da linguagem tais como: *main*, *if*, *else*, *while*, *int*, *double*, *try*, etc. As palavras reservadas aparecem geralmente escritas a uma cor diferente no editor.
3. Não pode conter espaços nem carateres que não sejam números, letras ou o *underscore*.

Os nomes que usamos não devem ser muito longos e devem ser o mais sugestivos possível para facilitar a leitura do programa. Por exemplo, se pretendermos criar uma variável para guardar a idade de uma pessoa, o nome *idade* é talvez o mais sugestivo para essa variável. É necessário, no entanto, ter em conta que no mesmo programa não podem existir duas variáveis com o mesmo nome. Se precisarmos de duas variáveis para guardar duas idades, podemos, por exemplo, usar os nomes *idade_1* e *idade_2* ou simplesmente *idade1* e *idade2*. É importante também ter em conta que o C++ faz distinção entre letras maiúsculas e minúsculas pelo que, por exemplo, os nomes *idade* e *Idade* não correspondem à mesma variável. Nos programas que escrevemos, devemos ter o cuidado de não usar variáveis com nomes parecidos para evitar confusões.

Todas as variáveis usadas num programa têm que ser primeiro *declaradas* para que o computador crie espaço na memória para o tipo de dados que se pretende guardar. Para isso, devemos escrever o tipo da variável e depois o seu nome, ou seja,

```
int idade;  //Declaração de uma variável chamada idade do tipo int
```

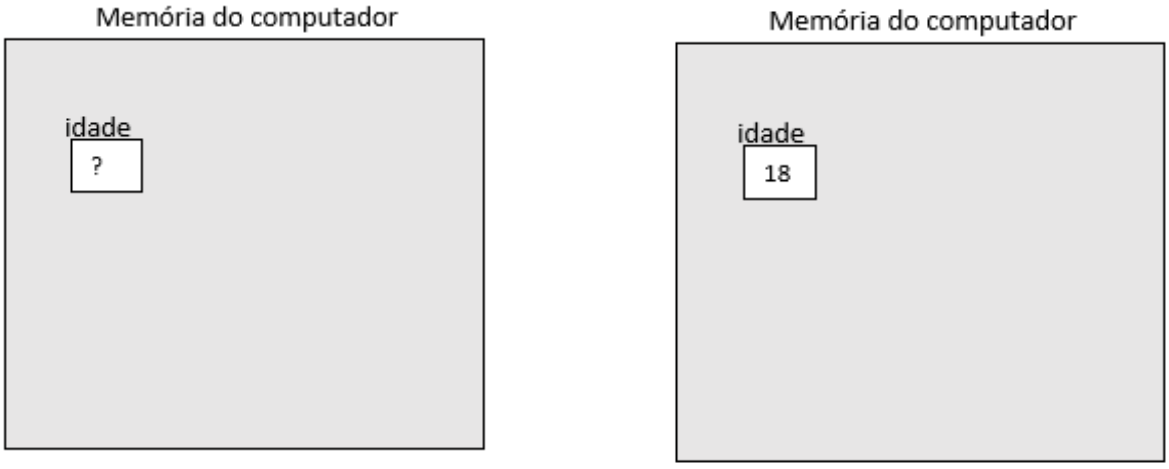
Ao declarar uma variável, é-lhe imediatamente atribuído o valor que se encontra no pedaço de memória que lhe está associado e que é um "valor lixo". Para que tal não aconteça, devemos, no momento da criação das variáveis, atribuir-lhes um valor inicial, o que é chamado de *inicialização* e é exemplificado de seguida:

```
int idade;
idade = 18;  //Inicialização da variável idade com o valor 18
```

A declaração e a inicialização de uma variável podem ser feitas na mesma instrução da seguinte forma:

```
int idade = 18; //Inicialização da variável idade com o valor 18
```

Informalmente, criar variáveis é criar caixas na memória do computador onde vão ser colocados valores do tipo que foi definido. A Figura 1.1 contém uma representação esquemática do que acontece na memória do computador quando a variável é declarada (Figura 1.1a) e de quando é inicializada (Figura 1.1b).




(a) Declaração de uma variável.

(b) Inicialização de uma variável.

Figura 1.1: Declaração *versus* inicialização de uma variável.

O tipo de uma variável faz com que esta tenha diferentes propriedades. A Tabela 1.1 contém exemplos de vários tipos de dados existentes.

Tabela 1.1: Exemplos de tipos de dados.

| | **Tipos** |
|---|---|
| Números inteiros | *short*<br>*int*<br>*long*<br>*long long int* |
| Números inteiros positivos | *size_t* |
| Números decimais | *float*<br>*double* |
| Carater | *char* |
| Texto | *string* |
| Valor lógico | *bool* |

Os primeiros quatro tipos de dados são usados para guardar valores inteiros (positivos ou negativos) e diferem entre si na gama de valores que podem guardar, isto é, no espaço de memória que ocupam. O tipo *short* permite guardar valores inteiros mais pequenos (poucos dígitos) enquanto o tipo *long long int* permite guardar valores inteiros maiores (com mais dígitos). Nos nossos programas, o tipo *int* é, em regra, o mais usado uma vez que permite representar números com uma ordem de grandeza suficientemente elevada. O tipo *size_t* representa números inteiros não negativos. Para números decimais, podemos usar os tipos *float* ou *double*. No entanto, como a precisão do tipo *double* é maior do que a do *float* - isto é, o tipo *double* permite guardar mais casas decimais - será esse o tipo de dados que usaremos. Numa variável do tipo *char* podemos guardar um qualquer caráter, isto é, uma letra (sem acento nem cedilha), um algarismo (de 0 a 9) ou um símbolo (/, +, ;, $, etc.). Uma *string* é um tipo de dados que permite guardar uma sequência de carateres, isto é, permite guardar texto. Finalmente, o tipo *bool* assume apenas os valores lógicos *true* ou *false*, sendo que *true* corresponde ao valor 1 e *false* corresponde ao valor 0. Os valores booleanos não são guardados como *true* e *false*, mas sim como inteiros com a correspondência referida anteriormente.

Todos os tipos apresentados são tipos de dados *fundamentais* ou *primitivos*, à exceção do tipo *string*. No código abaixo são apresentados exemplos de variáveis dos diferentes tipos de dados referidos anteriormente.

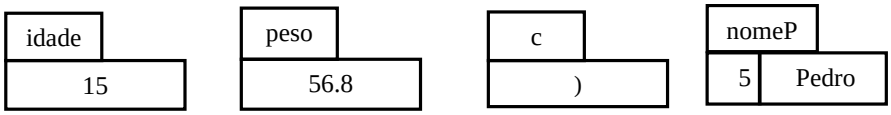
```
#include <iostream>
using namespace std;

int main(){

   int idade = 18;           //Variável inteira com o nome idade e valor 15
   double peso = 56.8;       //Variável decimal com o nome peso
   string nomeP = "Pedro";   //Variável que guarda um conjunto de carateres
   char c = ')';             //Variável que guarda um carater e tem o nome c
   bool logico = true;       //Variável booleana com o nome logico
   double altura;            //Variável com o nome altura não inicializada

   return 0;
}
```

Temos seis variáveis declaradas, estando as cinco primeiras também inicializadas. Note-se que os carateres (tipo *char*) são definidos por plicas enquanto que o texto (tipo *string*) é definido com aspas. Relembramos que não é obrigatório inicializar as variáveis quando são declaradas, mas é boa prática fazê-lo pois, quando as variáveis são declaradas mas não são inicializadas poderão assumir valores disparatados, sendo por isso *valores lixo*. Na primeira linha do programa temos o *#include <iostream>* que é necessário quando utilizamos uma variável do tipo *string*. Na segunda linha do programa temos a instrução *using namespace std;* da qual falaremos mais adiante. Como já vimos anteriormente, a instrução *int idade=18* cria uma caixa com o nome idade para guardar um número inteiro e dentro dessa caixa coloca o número 18. A lógica é a mesma para os restantes tipos de dados, exceto para as *strings*. Quando declaramos e inicializamos uma *string*, é criada uma caixa com duas divisões, sendo que na primeira é guardado o número de carateres da *string* e na segunda a *string* propriamente dita. Para o exemplo anterior temos:




Ao longo da execução de um programa, as variáveis tomam normalmente valores distintos, mas em cada instante apenas têm um valor. É importante relembrar que uma variável é como uma caixa que só pode ter um único valor dentro dela.

```
int main(){
   int x = 10;     //Variável inicializada com o valor 10
   x = 20;         //O valor da variável é alterado para 20
   //...
   x = 30;         //O valor da variável é alterado para 30
   return 0;
}
```

No código acima, a variável $x$ é inicializada com o valor 10. O seu valor é depois alterado para 20 e mais tarde para 30, sendo esse o seu valor final.

## 1.2 Escrita de variáveis - Output

Nos programas que construímos é necessário frequentemente mostrar informação ao utilizador, ou seja, escrever informação no ecrã. A isto chamamos *imprimir*. Para *imprimir* algo, usamos o comando *cout*, cujo significado é *console output*, juntamente com o operador <<.

```cpp
#include <iostream>
using namespace std;

int main(){

   int idade = 15;
   string nomeP = "Pedro";
   cout << "Output: \n";
   cout << nomeP;
   cout << endl;
   cout << idade;
   cout << "\n";

   return 0;
}
```


Para usar o comando *cout* é necessário incluir o pacote *iostream* e escrever a segunda linha do programa. A instrução $\backslash n$ é um comando de controlo que serve para mudar de linha. A instrução *endl*, significa *end of line* e serve também para mudar de linha. Ao ser executado, este programa começa por declarar e inicializar as variáveis *idade* e *nomeP*. Depois disso, encontra o primeiro *cout* e por isso escreve no ecrã a mensagem "Output: ". Ainda nessa linha, encontra o controlador $\backslash n$ e faz uma mudança de linha no que escreve no ecrã. Na linha de código seguinte, o programa vai aceder à variável (ou caixa) com o nome *nomeP* e escrever o que está lá dentro, que neste caso é "Pedro". Ao chegar à linha de código seguinte, faz apenas uma mudança de linha porque encontrou o controlador *endl*. Depois disso, o programa acede à variável com o nome *idade* e escreve o que está lá dentro. No último *cout* é apenas efetuada mais uma mudança de linha. Na prática, os cinco comandos *cout* anteriores podem (e devem) ser escritos com uma única instrução, o que permite simplificar a escrita. Assim sendo, podemos escrever apenas:

```cpp
cout << "Output: \n" << nomeP << endl << idade << "\n";
```

sendo o output exatamente o mesmo.

## 1.3 Atribuição de valores a variáveis - Input

Para atribuir valores a uma variável podemos usar simplesmente o operador `=`, por exemplo $x = 5$. A função do operador `=` é atribuir o valor que está do lado direito à variável que está do lado esquerdo. Por exemplo, a instrução $x = 5$ corresponde a fazer $x \leftarrow 5$, ou seja, a atribuir o valor 5 à variável $x$. O operador `=` é usado para atribuir valores a variáveis quando sabemos qual o valor a atribuir no momento em que o código está a ser escrito. No entanto, em muitas situações, o valor das variáveis não é previamente conhecido, sendo apenas definido posteriormente pelo utilizador. Para estes casos devemos usar o comando *cin*, cujo significado é *console input*, juntamente com o operador `>>`.

```
#include <iostream>
using namespace std;

int main(){

   int idade;
   cout << "Introduza a idade: ";
   cin >> idade;

   cout << "A idade e: " << idade << "\n";

   return 0;
}
```

Para usar o comando *cin* é também necessário incluir o pacote *iostream* e escrever a segunda linha do programa. Este programa começa por declarar uma variável do tipo *int* com o nome *idade*. Depois, escreve no ecrã o texto *"Introduza idade: "*. De seguida, o programa passa para a linha seguinte e ficará à espera até que o utilizador introduza um número inteiro. Quando o utilizador insere o número, este é guardado na variável idade. Finalmente, o programa escreve para o ecrã *"A idade e: "*, vai ver qual o valor que está na variável *idade*, escreve-o no ecrã e termina com uma mudança de linha.

Embora seja boa prática inicializar as variáveis, não é necessário fazê-lo no caso em que o seu valor é pedido ao utilizador (quase) imediatamente após a sua declaração, como no exemplo anterior. As três primeiras linhas de código dentro da função *main* no exemplo anterior são usadas sempre que queremos pedir o valor de uma variável ao utilizador. A isso chama-se *ler* a variável.

Os operadores >> e << usados, respetivamente, nas instruções *cin* e *cout* indicam movimento. No caso do *cin*, ao escrevermos *cin* >> $x$ estamos de certa forma a *enviar* o que foi escrito no ecrã (lado esquerdo) para a variável $x$ que está no lado direito. Por outro lado, ao fazermos *cout* << $x$ estamos a enviar o que está do lado direito (valor da variável $x$) para ser escrito no ecrã (lado esquerdo).

## 1.4 Variáveis constantes

Como vimos anteriormente, o valor de uma variável pode ser sucessivamente alterado ao longo da execução de um programa. No entanto, pode ser útil em algumas situações usar variáveis cujo valor não queremos alterar. Este tipo de variáveis são designadas por *constantes* e são definidas através da palavra reservada *const*. Um bom exemplo é o caso do $\pi$, que pode ser definido como *const double pi = 3.141592*. Isto permite não só que ao longo do programa usemos sempre a variável *pi* em vez de escrevermos o valor 3.141592, como também impede que o valor da variável *pi* seja alterado, isto é, não seria possível fazer uma nova atribuição do tipo *pi = 3.14*. Este exemplo é retratado no próximo excerto de código, que não é compilado pelo editor uma vez que se está a alterar o valor de uma variável definida como constante.

```
int main(){
   const double pi = 3.141592;
   pi = 3.14;   //ERRO!
   return 0;
}
```

## 1.5 Operadores

O C++ contém operadores de diferentes tipos, nomeadamente operadores aritméticos simples, operadores aritméticos compostos, operadores relacionais e operadores lógicos.

### 1.5.1 Operadores aritméticos

Os operadores aritméticos simples são os que já conhecemos da matemática e encontram-se na tabela abaixo, sendo os valores apresentados na coluna Resultado correspondentes ao caso em que $a$=*13* e $b$=*5*.

Tabela 1.2: Operadores aritméticos simples

| **Operador** | **Nome** | **Exemplo** | **Resultado** |
|---|---|---|---|
| + | Soma | $a+b$ | 18 |
| - | Subtração | $a-b$ | 8 |
| * | Multiplicação | $a*b$ | 65 |
| / | Divisão inteira ou decimal | $a/b$ | 2 ou 2.6 |
| % | Resto da divisão inteira | $a\%b$ | 3 |

O resultado do operador "/" é o resultado da divisão inteira sempre que os dois operandos forem de um tipo de dados inteiro. Se um dos operandos for do tipo decimal, o resultado do operador "/" é a divisão decimal. Para obter a divisão decimal entre duas variáveis do tipo inteiro é necessário primeiro converter uma delas para um tipo de dados decimal e só depois fazer a divisão. Este processo designa-se por *cast* e, uma das formas de o fazer, consiste em escrever o tipo de dados que se pretende obter entre parêntesis antes da variável a transformar. Isto é, sendo $a$ e $b$ variáveis do tipo *int*, a divisão decimal pode ser obtida através da instrução *(double) a/b*. Isto significa que o programa começa por converter a variável $a$ para uma variável do tipo *double* e depois faz a divisão entre um *double* e um *int*, sendo por isso o resultado um valor decimal.

Os operadores aritméticos apresentados na tabela anterior são usados entre operandos do tipo numérico. No entanto, o operador "+" pode também ser usado para *strings*, funcionando como operador de concatenação (junção).

```
#include <iostream>
using namespace std;

int main(){
    string a = "Eu tenho ";
    int c = 18;
    string b = " anos.";

    string frase = a + to_string(c) + b + " Sou Jovem!";
    cout << frase;

    return 0;
}
```

No exemplo acima, é impressa para o ecrã a variável *frase* do tipo *string* que resulta da concatenação de várias *strings* e de um inteiro. Variáveis do tipo *string* podem ser concatenadas diretamente, no entanto, no caso de tipos de dados numéricos, é necessária a utilização da função *to_string*. A finalidade da função *to_string* é converter um valor numérico para uma *string*, que depois pode ser concatenada diretamente com outras *strings*. No exemplo acima, o output impresso no ecrã - que é o valor da variável *frase* criada - é "*Eu tenho 18 anos. Sou Jovem!*".

Os operadores aritméticos compostos permitem simplificar a escrita de instruções. Por exemplo, escrever *a=a+3* é o mesmo que escrever *a+=3*. Para usar estes operadores, é necessário compreender bem o uso do operador = já explicado anteriormente. Recorde-se que este operador tem como função atribuir o valor que está do seu lado direito à variável do lado esquerdo. Assim sendo, ao escrever *a=a+3* não estamos a dizer que o lado direito é igual ao lado esquerdo, tal coisa nem faria sentido do ponto de vista matemático. O significado da expressão *a= a+3* é o mesmo que *a* $\leftarrow$ *(a+3)*. Suponhamos que o valor de *a* é 6. Quando o programa chega à instrução *a=a+3*, vai primeiro olhar para o lado direito e calcular o valor da expressão *a+3* que será 9. Depois disso, irá então atribuir o valor 9 à variável *a*. A partir daí, o valor da variável *a* é 9. Na tabela abaixo apresentam-se os operadores compostos e o resultado da variável *a* no final das operações, considerando como valores iniciais *a=6* e *b=2*.

Tabela 1.3: Operadores aritméticos compostos (considerando *a=6* e *b=2*).

| **Operador** | **Nome** | **Exemplo** | **Significado** | **Valor de *a*** |
|---|---|---|---|---|
| `+=` | Soma/atribuição | *a*`+=`*b* | *a=a+b* | 8 |
| `-=` | Subtração/atribuição | *a*`-=`*b* | *a=a-b* | 4 |
| `*=` | Multiplicação/atribuição | *a*`*=`*b* | *a=a*b* | 12 |
| `/=` | Divisão/atribuição | *a*`/=`*b* | *a=a/b* | 3 |
| `%=` | Resto/atribuição | *a*`%=`*b* | *a=a%b* | 0 |
| `++` | Incremento | *a*`++` | *a=a+1* | 7 |
| `--` | Decremento | *a*`--` | *a=a-1* | 5 |

Estes operadores são aplicados a tipos de dados numéricos, no entanto, o operador `+=` pode também ser aplicado a *strings*. Nessa situação, ele funciona como operador de concatenação+atribuição, concatenando o que estiver do lado direito ao que estiver do lado esquerdo. No exemplo abaixo, o valor final da variável *b* não é alterado, sendo ele "*BB*". O valor da variável *a* (escrito no ecrã) será "*AABB*". Ou seja, uma vez que *a+=b* é equivalente a *a=a+b*, o valor da variável *b* vai ser concatenado ao valor inicial da variável *a*, sendo o resultado guardado na variável *a*.

```
#include <iostream>
using namespace std;

int main(){
    string a = "AA";
    string b = "BB";
    a += b
    cout << a;
    return 0;
}
```

O operador de incremento $a$++ é equivalente a escrever $a$+=1 que é ainda equivalente a escrever $a$=$a$+1. Os operadores de incremento e decremento têm a particularidade de puderem ser usados como prefixo (++$a$) ou sufixo ($a$++). Quando usados de forma isolada, o seu significado é exatamente o mesmo. No entanto, em operações nas quais o resultado da operação de incremento ou de decremento é avaliado noutra expressão, os resultados podem ser diferentes. No caso do operador de incremento de prefixo (++$a$) o valor da variável é incrementado e depois é devolvido. Ou seja, a variável é incrementada antes da expressão ser avaliada e portanto é considerado na expressão o valor já incrementado. No caso do operador de incremento de sufixo ($a$++ ) o valor da variável é devolvido e depois é incrementado. Ou seja, o valor da variável é incrementado apenas após a avaliação da expressão. Vejamos o exemplo abaixo:

Tabela 1.4: Diferenças entre os operadores incremento de prefixo e de sufixo ($a$=3 e $b$=3).

| **Exemplo** | **Valor final de *a*** | **Valor final de *b*** |
|---|---|---|
| $a$=++$b$ | 4 | 4 |
| $a$=$b$++ | 3 | 4 |

Quando utilizados isoladamente, devemos dar preferência à utilização ao operador de incremento de prefixo (++$a$) pois, como veremos mais à frente, é mais eficiente que o operador de incremento de sufixo ($a$++).

### 1.5.2 Operadores relacionais

Os operadores relacionais servem para comparar duas expressões. O resultado dessa comparação é um valor do tipo *bool* que pode ser *true* (caso o resultado da comparação seja verdadeiro) ou *false* (caso contrário).

Tabela 1.5: Operadores relacionais.

| **Operador** | **Significado** |
|---|---|
| < | menor |
| > | maior |
| <= | menor ou igual |
| >= | maior ou igual |
| == | igual |
| != | diferente |

Estes operadores são fundamentais para a secção seguinte. Para já, é importante realçar o operador ==, que nada tem a ver com o operador de atribuição = usado anteriormente. O operador == verifica se o que está do seu lado direito é igual ao que está do seu lado esquerdo, devolvendo como resultado *true* ou *false*. Por exemplo, o resultado da comparação *5==8/2* é *false* pois 5 não é igual a *8/2 (=4)*. O operador = serve para atribuir o valor do lado direito à variável do lado esquerdo e não para efetuar comparações.

### 1.5.3 Operadores lógicos

Os operadores lógicos servem para negar e combinar expressões. Assim sendo, o resultado das operações com os operadores lógicos é também *true* ou *false*.

Tabela 1.6: Operadores lógicos.

| Operador | Significado |
|---|---|
| && ou *and* | conjunção (e) |
| \|\| ou *or* | disjunção (ou) |
| ! | negação |

O operador !, colocado à esquerda de uma expressão, inverte o seu valor lógico. Isto é, se a expressão é verdadeira passa a falsa e vice–versa. Como exemplo, considerem-se três variáveis inteiras $a$=*5*, $b$=*3* e $c$=*2*. Temos então

$$b > a \quad \longrightarrow \quad false$$

$$!(b > a) \quad \longrightarrow \quad true$$

$$!(b > a) \;\; \&\& \;\; c == a - b \;\; || \;\; c > b \quad \longrightarrow \quad true$$

$$(\; !(b > a) \;\; or \;\; c == a - b \;) \;\; and \;\; c > b \quad \longrightarrow \quad false$$

$$b > a \;\; || \;\; c == a - b \;\; \&\& \;\; b > c \quad \longrightarrow \quad true$$

Note-se que a última expressão é equivalente a $b > a \;\; || \;\; (\; c == a - b \;\; \&\& \;\; b > c \;)$, uma vez que a conjunção (lida como "e") tem prioridade em relação à disjunção (lida como "ou").

### 1.5.4 Operador ternário (Informação complementar)

O operador ternário ou condicional avalia uma expressão e devolve diferentes valores de acordo essa avaliação. Este operador não será lecionado nas aulas, estando aqui apenas como informação complementar. A sintaxe deste operador é a seguinte:

$$(<condição> \;?\; <resultado1> : \; resultado2>)$$

Se a <*condição*> é verdadeira então o operador vai devolver o <*resultado1*>. Caso contrário devolve o <*resultado2*>. Por exemplo, ao escrevermos

```
int x;
x = (7==5 ? 4 : 3);
```

a variável $x$ vai ficar com o valor 3, uma vez que a condição 7==5 é falsa.

# Capítulo 2

# Estruturas de controlo

As estruturas de controlo são essenciais em qualquer linguagem de programação e dividem-se em condicionais e cíclicas. As estruturas de controlo condicionais servem essencialmente para executar instruções especificas em função da satisfação ou não de determinadas condições. Isto é, servem para permitir ao programa seguir caminhos distintos. Por outro lado, as estruturas de controlo cíclicas estão associadas à repetição de instruções/processos.

As estruturas de controlo condicionais são apresentadas na Secção 2.1. Na Secção 2.2 é detalhada a importância do uso de chavetas e da indentação. Por fim, as estruturas cíclicas são apresentadas na Secção 2.3.

## 2.1 Estruturas de controlo condicionais

Serão apresentadas três estruturas condicionais diferentes, nomeadamente a estrutura *if*, a estrutura *if else* e as estruturas condicionais encadeadas.

### 2.1.1 Estrutura *if*

A estrutura de controlo condicional mais simples em programação é a estrutura *if*. A sua sintaxe geral é:

```
if ( Condição ) {
   Bloco de instruções
}
```

Um *if* carateriza-se por uma condição e um bloco de instruções. Blocos de instruções são definidos através do uso de chavetas e contêm várias instruções. A condição toma o valor *true* ou *false* podendo por isso ser uma variável booleana ou uma expressão lógica que, na maior parte dos casos, envolve os operadores relacionais e lógicos apresentados no capítulo anterior. A tradução de *if* é "se", por isso, como o próprio nome indica, o bloco de instruções do *if* será apenas executado se a condição for verdadeira.

O exemplo seguinte contém um excerto de código onde a estrutura *if* é utilizada.

```
#include <iostream>
using namespace std;

int main(){
    int a = 1;
    int b;
    cout << "Introduza o valor de b: ";
    cin >> b;

    if ( b > 10 && b % 3 == 0 ) {
        ++a;
    }

    return 0;
}
```

Neste exemplo, o programa começa por declarar duas variáveis `a` e `b`, inicializando a primeira com o valor 1. De seguida, o valor da variável `b` é pedido ao utilizador. Ao chegar à instrução *if* o programa começa por avaliar o valor lógico da condição `b>10 && b%3==0`. Se o resultado dessa avaliação for *true*, então o programa irá entrar no bloco de instruções do *if* (que neste caso contém apenas uma instrução) e irá incrementar o valor da variável `a` numa unidade. Suponhamos que o valor de `b` introduzido pelo utilizador é 20. Apesar de 20 ser maior que 10, o resto da divisão de 20 por 3 não é zero e por isso o resultado lógico da condição do *if* é *false*. Neste caso, o programa não irá executar as instruções associadas ao *if*, passando de imediato para o *return* final e, portanto, o valor final da variável `a` é 1. Suponhamos agora que o valor introduzido pelo utilizador é 15. Neste caso, como $15 > 10$ e o resto da divisão de 15 por 3 é zero, a condição do *if* é verdadeira e por isso o programa irá executar as instruções do bloco *if*. Assim sendo, o valor da variável $a$ no final do programa será 2.

Neste exemplo, é importante salientar dois aspetos. Em primeiro lugar, a utilização do operador de igualdade `==` que se justifica pelo facto de estar a ser feita uma comparação e não uma atribuição e, em segundo lugar, o significado da segunda parte da expressão lógica, isto é, $b\%3 == 0$. Verificar se o resto da divisão de $b$ por 3 é zero é o mesmo que verificar se $b$ é múltiplo de 3, pelo que será sempre esta a forma usual de definir condições do tipo "ser múltiplo de".

### 2.1.2 Estrutura *if else*

A estrutura *if else* tem dois blocos de instruções o que nos permite definir instruções caso a condição do *if* seja avaliada como falsa. Assim, o Bloco de instruções 1 será executado caso a condição do *if* seja verdadeira, enquanto o Bloco de instruções 2 será executado no caso contrário, isto é, caso a referida condição seja falsa. A sintaxe geral de um *if else* é:

```
if ( Condição ) {
   Bloco de instruções 1
} else{
   Bloco de instruções 2
}
```

Note-se que o *else* não necessita de uma condição pois, implicitamente, a condição do *else* é a negação da condição escrita no *if*. Vejamos o seguinte exemplo:

```
1. int main(){
2.     int a = 1, b = 19, c = 3;
3.
4.     if ( b % 3 == 0 or b % 2 == 0 ) {
5.         a = 10;
6.         c += a;
7.     }else{
8.          a = 20;
9.          c -= a;
10.    }
11.    return 0;
12.}
```

A condição associada ao *if* pode ser lida como "*b é múltiplo de 3 ou de 2*". Como `b`=19, a condição é falsa e por isso o programa salta imediatamente da linha 4 para a linha 7, que é a linha do *else*, executando de seguida todas as instruções nesse bloco. Assim sendo, o programa redefine o valor de `a` como sendo 20 e em seguida atualiza o valor de `c` para 3-20, ou seja, -17. No final do programa temos então `a`=20, `b`=19 e `c`=−17. Neste exemplo, a condição implícita no *else* é "*b não é múltiplo de 3 nem de 2*", isto é, `b % 3 != 0 and b % 2 != 0`.

Como comentário final é importante notar que sempre que temos um *else* este tem que estar associado a um *if*. Contudo, o inverso não é obrigatório, isto é, podemos ter um *if* sem ter qualquer *else* associado, como vimos na secção anterior.

### 2.1.3 Estruturas condicionais encadeadas

A estrutura *if else* permite ao programa seguir dois caminhos distintos em função da avaliação de uma condição. Contudo, existem situações onde há mais do que dois caminhos possíveis para o programa seguir. Para tal, podemos usar estruturas *condicionais encadeadas*. O termo *encadeado* significa que existem estruturas condicionais que estão elas próprias contidas noutras estruturas condicionais.

A sintaxe geral da forma compacta das estruturas condicionais encadeadas é apresentada do lado direito. Do lado esquerdo é apresentada a versão entendida da mesma estrutura, usando apenas várias estruturas *if else*:

```
if ( Condição 1 ) {
   Bloco de instruções 1
} else  {
    if ( Condição 2 ){
          Bloco de instruções 2
    } else {
          if ( Condição 3 ){
              Bloco de instruções 3
          } else {
              Bloco de instruções 4
          }
    }
}
```

```
if ( Condição 1 ) {
   Bloco de instruções 1
} else if ( Condição 2 ){
   Bloco de instruções 2
} else if ( Condição 3 ){
   Bloco de instruções 3
} else {
   Bloco de instruções 4
}
```

Considere-se a forma compacta das estruturas condicionais encadeadas. Caso a Condição 1 seja verdadeira, é executado o Bloco de instruções 1. Caso a Condição 2 seja verdadeira e a Condição 1 seja falsa, é executado o Bloco de instruções 2. O Bloco de instruções 3 apenas será executado se a Condição 3 for verdadeira e se as Condições 1 e 2 forem falsas. O Bloco de instruções 4 será apenas executado no caso de todas as Condições 1, 2 e 3 serem falsas. É importante realçar que nesta estrutura, um e apenas um bloco de instruções é executado, mesmo que mais do que uma condição seja verdadeira. Se duas ou mais condições forem verdadeiras, o único bloco de instruções executado é o primeira que aparecer. Por exemplo, se a Condição 2 e a Condição 3 forem ambas *true* o bloco de instruções a ser executado é o Bloco de instruções 2.

Para clarificar as diferenças entre as estruturas de controlo condicionais apresentadas até aqui, considere-se o seguinte exemplo. Suponhamos que o preço unitário de um dado produto depende da quantidade a comprar e que queremos fazer um programa que, dado o número de unidades a comprar, calcule o preço final a pagar. Se o preço unitário do produto for dado de acordo com a tabela seguinte:

| Quantidade | $<50$ | $[50, 99[$ | $[100, 150[$ | $\geq 150$ |
|---|---|---|---|---|
| Preço | 5 | 4 | 3,5 | 3,3 |

então o código abaixo, embora não use estruturas encadeadas, faz o que se pretende.

```cpp
#include <iostream>
using namespace std;

int main(){
    int qt, preco;
    cout << "Quantidade: ";
    cin >> qt;

    if ( qt < 50 ) {
        preco = 5 * qt;
    }
    if ( qt >= 50 and qt < 99 ) {
        preco = 4 * qt;
    }
    if ( qt >= 100 and qt < 150 ) {
        preco = 3.5 * qt;
    }
    if ( qt >= 150 ) {
        preco = 3.3 * qt;
    }

    cout << "Preco final: " << preco;
    return 0;
}
```

O código acima é composto por quatro estruturas *if* independentes. O mesmo código pode ser escrito usando estruturas condicionais encadeadas tal como apresentado abaixo.

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     int qt, preco;
6.     cout << "Quantidade: ";
7.     cin >> qt;
8.
9.     if ( qt < 50 ) {
10.        preco = 5 * qt;
11.    }else{
12.         if ( qt < 99 ) {
13.            preco = 4 * qt;
14.         }else{
15.              if ( qt < 150 ) {
16.                 preco = 3.5 * qt;
17.              }else{
18.                 preco = 3.3 * qt;
19.              }
20.         }
21.    }
22.
23.    cout<<"Preco final: "<<preco;
24.    return 0;
25. }
```

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     int qt, preco;
6.     cout << "Quantidade: ";
7.     cin >> qt;
8.
9.     if ( qt < 50 ) {
10.        preco = 5 * qt;
11.    }else if( qt < 99 ) {
12.        preco = 4 * qt;
13.    }else if ( qt < 150 )
14.        preco = 3.5 * qt;
15.    }else{
16.        preco = 3.3 * qt;
17.    }
18.
19.    cout<<"Preco final: "<<preco;
20.    return 0;
21. }
```

Na primeira implementação, existe uma estrutura de controlo principal que começa na linha 9 e termina na linha 21. O *else* desta estrutura (linha 11) contém no seu bloco de instruções um novo *if* que começa na linha 12 e termina na linha 20. Este novo *if*, por sua vez, inclui também um novo *if* no bloco de instruções associado ao seu *else* que começa na linha 15 e termina na 19. No entanto, é importante realçar que cada *if* só poderá ter no máximo um *else* associado (podendo esse *else* ter outros *if*s, e consequentemente outros *else*s, dentro dele). As condições dos *if*s aqui apresentados podem parecer incompletas quando comparadas com as apresentados no código anterior, mas na verdade estão corretas. Vamos analisar diferentes casos para perceber melhor como as estruturas encadeadas funcionam. Consideremos o código do lado esquerdo.

1. Suponhamos que a quantidade *qt* introduzida pelo utilizador na linha 7 é 30. Quando o programa chega à linha 9, vai verificar se *qt*<50, o que é de facto verdade. Assim sendo, o programa entrará no primeiro *if*, fará a instrução da linha 10 e passa imediatamente para a linha 21 (final do primeiro *if*). No final, teremos *preco=150*. Note-se que, uma vez que a condição do primeiro *if* é verdadeira, o programa não entra no *else* a ele associado (linhas 11-20).

2. Suponhamos que a quantidade *qt* introduzida pelo utilizador é 60. Ao chegar ao primeiro *if* (linha 9) o programa verifica que a condição *qt<50* é falsa e por isso passa imediatamente para o bloco *else* associado (linha 11). Dentro desse bloco, o programa começa por avaliar a condição do *if* da linha 12, ou seja, verifica se *qt<99*. Como a condição é verdadeira, o programa entra nesse *if*, fazendo a instrução da linha 13. De seguida, o programa passa para a linha 20 e consequentemente

para a linha 21. Note-se que o programa entrou no *if* da linha 12 pelo que não vai entrar no *else* da linha 14.

3. Suponhamos que a quantidade *qt* introduzida pelo utilizador é 200. Tal como no caso anterior, o programa vai entrar no *else* da linha 11 fazendo as instruções desse bloco. A condição do *if* da linha 12 é falsa e por isso o programa entra no *else* da linha 14. Ao chegar à linha 15, o programa analisa a condição *qt<150* cujo valor lógico é falso, e por isso o programa entra no *else* da linha 17, fazendo depois a instrução da linha 18. De seguida, o programa passa para a linha 19, depois para a linha 20 e finalmente para a linha 21 não realizando qualquer ação nestas fases.

Com base neste exemplo conseguimos perceber, por exemplo, porque é que podemos escrever simplesmente *qt<150* na linha 15 em vez de *qt<150 and qt>=100*. Isto acontece pois se o programa chegar à linha 15 é porque entrou primeiro no *else* da linha 11 (ou seja, é porque $qt \geq 50$) e no *else* da linha 14, (ou seja, é porque $qt \geq 100$).

O segundo excerto de código apresentado acima usa a forma compacta das estruturas condicionais encadeadas. Relembramos que quando usamos uma estrutura deste tipo é preciso ter em conta que o programa apenas entra num bloco de instruções: ou entra no bloco associado ao *if*, ou entra num dos blocos associados ao *else if*, ou entra no bloco do *else*. Quando o programa entrar num dos blocos - seja ele qual for - executa as instruções presentes nesse bloco e depois passa imediatamente para o final da estrutura de controlo (linha 17 neste exemplo).

Ao usar estruturas de controlo encadeadas é importante ter sempre em mente o funcionamento da estrutura *if else*, isto é, ter em conta que o programa ou executa as instruções do bloco associado ao *if* ou executa as instruções do bloco associado ao *else* e nunca ambas simultaneamente.

## 2.2 Uso de chavetas e de indentação

A utilização correta de chavetas e a indentação do código são dois aspetos muito importantes em programação. Por um lado, as chavetas servem para delimitar blocos de instruções, tal como vimos na secção anterior, e a sua errada colocação pode levar a erros de sintaxe ou de execução. Por outro lado, a indentação do código é completamente ignorada pelo compilador e por isso não afeta o funcionamento do programa. Indentar o código serve apenas para facilitar (e muito) a sua leitura de modo a tornar claro que instruções estão dentro de quais. Abaixo é apresentado o mesmo código indentado (lado esquerdo) e não indentado (lado direito). Através deste exemplo facilmente se percebe a importância da indentação uma vez que no primeiro caso conseguimos ver claramente onde é que começa e termina cada bloco de instruções.

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main(){
5.      int qt, preco;
6.      cout << "Quantidade: ";
7.      cin >> qt;
8.
9.      if ( qt < 50 ) {
10.          preco = 5 * qt;
11.      }else{
12.          if ( qt < 99 ) {
13.              preco = 4 * qt;
14.          }else{
15.              if ( qt < 150 ) {
16.                  preco = 3.5 * qt;
17.              }else{
18.                  preco = 3.3 * qt;
19.              }
20.          }
21.      }
22.
23.      cout<<"Preco final: "<<preco;
24.      return 0;
25.  }
```

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int main(){
5.  int qt, preco;
6.  cout << "Quantidade: ";
7.  cin >> qt;
8.
9.  if ( qt < 50 ) {
10. preco = 5 * qt;
11. }else{
12. if ( qt < 99 ) {
13. preco = 4 * qt;
14. }else{
15. if ( qt < 150 ) {
16. preco = 3.5 * qt;
17. }else{
18. preco = 3.3 * qt;
19. }
20. }
21. }
22.
23. cout<<"Preco final: "<<preco;
24. return 0;
25. }
```

No Qt Creator podemos indentar automaticamente o código que escrevemos carregando nas teclas *Control + A* (para selecionar tudo) e em seguida em *Control + I* (para indentar). Esta indentação automática permite perceber de que forma o compilador lê o código que escrevemos e verificar se tal coincide com a forma que de facto queremos que ele o leia.

O uso de chavetas é essencial para delimitar blocos de instruções. No entanto, quando um bloco de instruções é composto apenas por uma instrução, as chavetas podem ser omitidas. No exemplo abaixo, existe apenas uma instrução associada ao *if* da linha 4 pelo que a chaveta da linha 4 e a primeira chaveta da linha 6 podem ser omitidas. As chavetas associadas ao *if* da linha 7 não podem ser removidas porque temos mais do que uma instrução (duas neste caso) dentro do respetivo bloco. No entanto, a segunda chaveta da linha 10 e a chaveta da linha 12 podem ser removidas pois no bloco por elas delimitado apenas existe uma instrução.

As chavetas que mencionámos podem ser trivialmente removidas do código afim de o tornar mais compacto. No entanto, também a segunda chaveta da linha 6 e a chaveta da linha 13 podem ser removidas. Isto acontece porque, o primeiro bloco *else* também tem, na verdade, apenas uma instrução dentro dele, isto é, uma instrução *if else* que, apesar de ocupar várias linhas, é vista como uma única instrução. Este tipo de situações pode causar alguma confusão numa fase inicial da programação pelo que se recomenda que se mantenham as chavetas nestes casos. Mais uma vez é importante relembrar que no QT Creator podemos indentar automaticamente o nosso código e assim verificar facilmente erros relacionados com colocações/omissões de chavetas bem como verificar quais as instruções que estão dentro de quais.

```
1. int main(){
2.     int a = 1, b = 19, c = 3;
3.
4.     if ( b % 3 == 0  ) {
5.         c += a;
6.     }else{
7.         if(c>2){
8.             a = 20;
9.             c -= a;
10.         }else{
11.             a = 20;
12.         }
13.     }
14.     return 0;
18. }
```

```
1. int main(){
2.     int a = 1, b = 19, c = 3;
3.
4.     if ( b % 3 == 0  )
5.         c += a;
6.     else
7.         if(c>2){
8.             a = 20;
9.             c -= a;
10.         }else
11.             a = 20;
12.
13.
14.     return 0;
18. }
```

O próximo exemplo reforça a importância da utilização de chavetas e da indentação e ilustra uma propriedade da estrutura *if else* que ainda não foi mostrada anteriormente.

```
1. int main(){
2. int a = 1, b = 19, c = 3;
3. if (b % 3 == 0)
4. if (c>2)
5. a = 30;
6. else
7. a = 20;
8. return 0;
9. }
10.
11.
```

```
1. int main(){
2.     int a = 1, b = 19, c = 3;
3.     if (b % 3 == 0) {
4.         if (c>2){
5.             a = 30;
6.         }
7.     }else{
8.         a = 20;
9.     }
10.    return 0;
11. }
```

O primeiro excerto de código não contém chavetas nem está indentado. Além disso, contém dois *if*s e apenas um *else*, sendo por isso difícil perceber a qual *if* este está associado, originando ambiguidade. Nestes casos ambíguos, o *else* é emparelhado com o último *if* que se encontra no mesmo bloco de instruções, isto é, o *if* da linha 4. O segundo excerto de código já contém chavetas e está indentado, tornando-se assim claro que neste caso o *else* está associado ao primeiro *if*, que começa na linha 3, estando o segundo *if* (linha 4) contido no seu bloco de instruções.

## 2.3 Estruturas de controlo cíclicas

As estruturas cíclicas permitem a execução de um conjunto de instruções de forma repetitiva enquanto uma determinada condição for satisfeita. A linguagem de programação C++ dispõe de três estruturas de controlo cíclicas: *while*, *do-while* e *for*. Como referido anteriormente, um programa em C++ começa

sempre por executar a função *main*, percorrendo depois cada linha sequencialmente de cima para baixo, a não ser que existam instruções em contrário. As instruções cíclicas são as primeiras instruções que estudamos que alteram a sequência de execução de um programa.

### 2.3.1 Estrutura *while*

A estrutura *while* (que em português significa "enquanto") é a mais simples das três estruturas cíclicas disponíveis em C++ e tem uma estrutura semelhante à da estrutura *if*. A sintaxe geral de um ciclo *while* é:

```
while (Condição){
     Bloco de instruções
}
```

Quando uma estrutura *while* é executada, o programa começa por verificar se a `Condição` é verdadeira. Em caso afirmativo, o Bloco de instruções é executado. Ao contrário do que acontece na estrutura *if*, ser executado o Bloco de instruções, o programa volta ao topo da estrutura *while* e, avalia novamente a `Condição`. Caso a `Condição` continue a ser avaliada como *true*, o Bloco de instruções é executado novamente. Este processo é repetido até a `Condição` ser avaliada como *false*. Quando isto acontece, o programa abandona o ciclo, prosseguindo a sua execução para a linha de código imediatamente abaixo da estrutura *while*. Assim sendo, um ciclo *while* pode ser lido como: "Enquanto a condição for verdadeira, executa o bloco de instruções".

Considere-se o seguinte exemplo onde é utilizada uma estrutura *while* para imprimir no ecrã os números de 1 a 10:

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     int i = 1;
6.     while( i <= 10 ){
7.         cout << i << " ";
8.         ++i;
9.     }
10.
11.    return 0;
12. }
```

Vejamos agora ver em detalhe o que o programa está a fazer. Primeiramente, a variável `i` é inicializada a 1, pois esse é o primeiro número que vamos imprimir. De seguida, como a condição `i = 1 <= 10` é verdadeira, o bloco de instruções do *while* (linhas 7-8) é executado. A primeira instrução do bloco imprime para o ecrã o valor 1, que é o valor da variável `i` neste momento, e um espaço. A segunda instrução incrementa o valor da variável `i` para 2. Agora, o programa volta à linha 6 e a condição é avaliada outra vez. Como `2 <= 10` é verdade, o bloco de instruções é executado novamente sendo impresso no ecrã o valor 2 e um espaço. O ciclo é executado repetidamente até a variável `i` ter valor 11 e nesta altura a condição `11 <= 10` será falsa e o bloco de instruções associado ao *while* não é executado, passando o programa para a linha 10.

No exemplo anterior, para implementarmos um ciclo *while* precisámos de:

1. definir e inicializar uma variável de controlo de ciclo (variável *i*);
2. ter uma condição ou critério de paragem (`i <= 10`);
3. atualizar a variável de controlo de ciclo (`++i)`.

Estes três componentes estão sempre, de alguma forma, presentes num ciclo *while* e a sua não inclusão pode originar erros de código.

A não inicialização da variável de controlo de ciclo com um valor apropriado pode fazer com que o bloco de instruções associado ao ciclo *while* nunca seja executado. Se no exemplo anterior na linha 5 tivéssemos, por exemplo, `int i = 15;` a condição do *while* era avaliada como falsa e o programa saltava diretamente para a linha 10. A variável de controlo de ciclo deve ser declarada e inicializada fora da estrutura *while*. As variáveis que são declaradas dentro de blocos de instruções, em particular do bloco *while*, não existem fora desses blocos. A isto se chama o *âmbito* da variável, isto é, o local do programa onde a variável é reconhecida pelo programa.

Quando a variável de controlo de ciclo não é atualizada podemos ter um *ciclo infinito* em que a condição do *while* permanece sempre verdadeira. Significa isto que o programa executará indefinidamente as instruções do bloco de instruções, nunca conseguindo sair do ciclo e terminando, eventualmente, com um *crash* do sistema. Se no exemplo anterior não tivéssemos a instrução da linha 8 (`++i;`), a variável `i` teria sempre o valor 1 pelo que a condição `i<=10` era sempre avaliada como verdadeira. Estaríamos por isso na presença de um ciclo infinito.

### 2.3.2 Estrutura *do-while*

A sintaxe geral *do-while* é:

```
do{
   Bloco de instruções
}while (Condição);
```

O termo *do* significa "faz"e o termo *while* significa "enquanto". Assim sendo, um ciclo *do-while* pode ser interpretado como: "Fazer o que está no bloco de instruções enquanto a condição considerada for verdadeira". Ao encontrar um ciclo *do-while*, o programa começa por executar imediatamente o bloco de instruções nele contido sem verificar qualquer condição (contrariamente ao que acontece com a estrutura *while*). Após executar o Bloco de instruções, o programa avalia o valor lógico da `Condição`. Caso a condição tenha valor lógico verdadeiro, o programa volta a executar novamente todas as instruções do bloco de instruções do ciclo e a avaliar a `Condição`. Este processo é repetido enquanto a condição do *while* permanecer verdadeira, pelo que poderão ser efetuadas várias iterações até que tal aconteça. Assim que no momento da avaliação da condição do *while* esta seja falsa, o programa abandona imediatamente o ciclo prosseguindo para a linha de código imediatamente abaixo dele.

A principal diferença entre a estrutura *do-while* e a estrutura *while* é que na primeira iteração do ciclo *do-while* o bloco de instruções é sempre executado dado que a condição apenas é avaliada a seguir. Assim sendo, o bloco de instruções da estrutura *do-while* é sempre executado pelo menos uma vez enquanto o que o bloco de instruções da estrutura *while* pode nunca ser executado.

Vejamos os seguintes dois exemplos.

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     int n = 3;
6.
7.     int i = 1;
8.     do{
9.         cout << i << " ";
10.        ++i;
11.    }while( i <= n );
12.
13.    return 0;
14. }
```

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     int n;
6.     int conta = 0;
7.
8.     do{
9.         cout << "Valor: ";
10.        cin >> n;
11.        if ( n > 0)
12.            ++conta;
13.    }while( n > 0 );
14.
15.    cout << "Total: " << conta;
16.    return 0;
17. }
```

Ao ser executado, o primeiro programa começa por declarar e inicializar a variável inteira `n` com o valor 3. Ao chegar à linha 7, o programa declara e inicializa uma nova variável `i` com o valor 1. Esta variável é usada na condição do ciclo *do-while* e será em função do seu valor que o ciclo irá continuar a ser executado ou será interrompido. Assim sendo, a variável `i` neste programa é a variável de controlo do ciclo. Após a linha 7, o programa passa para a linha 8 e de seguida para a linha 9 onde vai escrever no ecrã o valor da variável `i` (que é 1) e um espaço em branco. De seguida, passa para a linha 10 onde vai aumentar o valor de `i` numa unidade, isto é, `i=2`. O programa passa depois para a linha 11 e verifica que a condição associada ao *while*, isto é, `i <= n`, é satisfeita. Assim sendo, o programa volta à linha 8 para executar novamente todas as instruções do bloco *do-while*. Ao chegar à linha 9, o programa volta a escrever o novo valor da variável `i` (que agora é 2) e um espaço em branco, incrementando depois o valor da variável `i` na linha 10. A condição associada ao *while* volta novamente a ser avaliada e o resultado dessa avaliação continua a ser `true` pois $i = 3 <= 3 = n$. Assim sendo, o programa volta novamente a executar o bloco de instruções do *do-while*, isto é, volta à linha 8 e imediatamente à linha 9 onde vai imprimir para o ecrã o valor 3. De seguida, na linha 10, volta a incrementar o valor da variável $i$ numa unidade, passando este a ser 4. De seguida, a condição do *while* é avaliada e, uma vez que ela agora é falsa (pois `i=4`), o programa sai do ciclo, passando para a linha 12 e em seguida para a linha 13, terminando aí o programa. Como todas as instruções referidas anteriormente são executadas muito rapidamente, o utilizador apenas verá no ecrã o resultado final obtido, que neste caso é: "1 2 3 ". Através desta análise, percebemos que o objetivo do primeiro programa é, na verdade, escrever os $n$ primeiros números naturais.

Para reforçar a importância da instrução `++i` presente na linha 9, experimentemos executar o programa sem ela. A variável `i` é inicializada com o valor 1 na linha 7. Se a linha 10 não existir no programa, o valor da variável `i` nunca será alterado. Isto significa que a condição `i<=n` será sempre verdadeira e por isso o programa executará *indefinidamente* o bloco de instruções do *do-while*, isto é, o programa escreverá indefinidamente o valor de $i$ (que é 1) no ecrã, entrando por isso num ciclo infinito.

Vejamos agora o segundo programa. O objetivo deste programa é pedir sucessivamente números inteiros positivos ao utilizador até que este insira um número inteiro negativo ou nulo. No final do programa, é apresentada uma mensagem com o número total de números positivos introduzidos pelo

utilizador. Em cada iteração do ciclo, o programa pede ao utilizador que insira um valor (linhas 9 e 10). Caso o valor introduzido seja positivo (linha 11), o programa incrementa a variável `conta` numa unidade (linha 12). Esta variável é usada para contar os números positivos introduzidos. Após executar o bloco de instruções, o programa avalia a condição do *while*, isto é, verifica se o último número introduzido pelo utilizador é positivo. Em caso afirmativo, o programa volta a executar todas as instruções do bloco de instruções. Caso contrário, o programa sai do ciclo, passando imediatamente para a linha 15 e escrevendo a mensagem final.

Existem neste programa alguns aspetos que é importante realçar. Em primeiro lugar, a inicialização obrigatória da variável `conta` feita na linha 6. Uma vez que na linha 12 a variável `conta` está a ser incrementada, isto é, ela passa a assumir o seu valor anterior mais uma unidade, é essencial que esta variável tenha de facto um valor inicial bem definido. Neste caso, esse valor é zero pois inicialmente (quando o programa está na linha 6) ainda não foi inserido qualquer número positivo. O segundo aspeto a realçar é a utilização do `if` na linha 11. Será esse *if* mesmo necessário? Uma vez que o objetivo do programa é contar quantos números positivos foram introduzidos pelo utilizador, a utilização deste *if* é fundamental. Caso este *if* não existisse, quando o utilizador introduzisse um número negativo ou nulo para parar o ciclo, ele seria também contado pela variável `conta`, uma vez que o incremento da variável `conta` é feito antes da condição do *while* ser avaliada. Uma última observação é o facto da variável `conta` ter que ser declarada fora do ciclo, sendo por isso o seu âmbito a função *main*. Se a variável fosse declarada dentro do ciclo, além do programa não dar o resultado pretendido, não seria possível aceder-lhe na linha 15 para imprimir o seu valor.

### 2.3.3 Estrutura *for*

A última estrutura de controlo que vamos apresentar é a estrutura *for*. Qualquer ciclo *for* pode ser convertido num ciclo *while* (ou *do-while*) e vice-versa. A sintaxe geral de um *for* é:

```
for (Inicialização; Condição; Incremento){
     Bloco de instruções
}
```

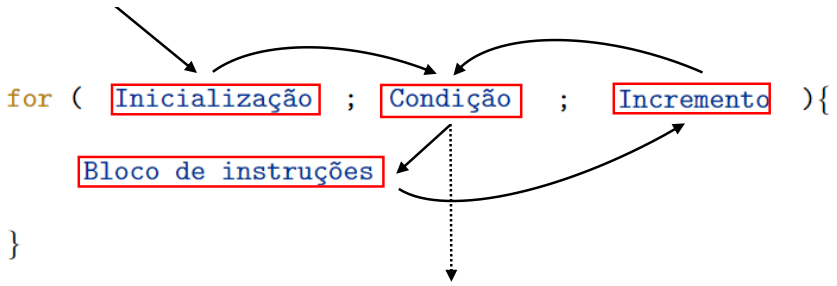
O ciclo *for* agrega a declaração e inicialização da variável de controlo do ciclo, a condição de execução do ciclo e o incremento da variável de controlo num só lugar.

Vejamos o seguinte exemplo em que é apresentado o programa para imprimir no ecrã os números de 1 a 10 utilizando um ciclo *while* e um ciclo *for*:

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     int i = 1;
6.      while( i <= 10 ){
7.         cout << i << " ";
8.         ++i;
9.     }
10.
11.    return 0;
12. }
```

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.      for(int i = 1; i <= 10; ++i){
6.         cout << i << " ";
7.     }
8.
9.    return 0;
10. }
```

Como se pode ver neste exemplo, as linhas 5, 6 e 8 do primeiro programa foram de certa forma juntas numa só linha (linha 5) no segundo programa. Vejamos então qual o esquema de fluxo seguido por um ciclo *for*.




Ao encontrar um ciclo *for*, o programa começa por inicializar a variável de controlo do ciclo e, em seguida, avaliar a expressão lógica que define a condição. Enquanto a referida condição for satisfeita, o programa, por esta ordem, executa o bloco de instruções do *for*, incrementa a variável de controlo do ciclo e avalia novamente a condição. Este processo é sucessivamente repetido até que a condição seja avaliada como falsa.

Foram apresentadas três estruturas cíclicas diferentes, que podem ser convertidas umas nas outras. No entanto, em algumas situações, a utilização de um ciclo é mais recomendada do que a de outros. Ciclos *while* e *do-while* são geralmente usados quando não se sabe à partida quantas iterações irão efetuar. Por exemplo, no programa em que se está sucessivamente a pedir valores positivos ao utilizador, não se sabe à partida quantos valores serão introduzidos, pelo que neste caso faz mais sentido a utilização de um ciclo *while* ou *do-while*. Nos casos em que se sabe exatamente quantas iterações serão efetuadas, é preferível a utilização do ciclo *for*, pois toda a informação do ciclo (variável de controlo, condição e incrementação) está agregada na sua primeira linha. Por exemplo, se se quiser efetuar a soma dos 30 primeiros números naturais, já se sabe à partida que será necessário efetuar 30 iterações sendo por isso aconselhada a utilização de um ciclo *for*.

### 2.3.4 Ciclos encadeados

Muitas vezes, a utilização de um único ciclo é insuficiente para programar determinados algoritmos, havendo a necessidade de usar *ciclos encadeados*. Isto é, usar ciclos em que eles próprios têm no seu bloco de instruções outros ciclos. Vejamos o seguinte exemplo:

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.    int n = 5;
6.
7.    for(int i = 1; i < n; ++i)
8.       for(int j = i + 1; j < n; ++j)
9.          cout<<"("<<i<<","<<j<<") ";
10.
11.   return 0;
12. }
```

```
i = 1
    j = 2
        Escreve no ecrã "(1,2)  "
    j = 3
        Escreve no ecrã "(1,3)  "
    j = 4
        Escreve no ecrã "(1,4)  "
    j = 5 (Fim do segundo ciclo)
i = 2
    j = 3
        Escreve no ecrã "(2,3)  "
    j = 4
        Escreve no ecrã "(2,4)  "
    j = 5 (Fim do segundo ciclo)
i = 3
    j = 4
        Escreve no ecrã "(3,4)  "
    j = 5 (Fim do segundo ciclo)
i = 4
    j = 5 (Fim do segundo ciclo)
i = 5 (Fim do primeiro ciclo)
```

Este programa inclui um ciclo *for* principal (que começa na linha 7 e termina na linha 11) que inclui no seu bloco de instruções um segundo ciclo *for* (que começa na linha 8 e termina na linha 10). A variável de controlo do primeiro ciclo é a variável `i` e a do segundo ciclo é a variável `j`. O processo de execução do programa é então o que é apresentado no lado direito.

Ao chegar à linha 7, o programa declara e inicializa a variável `i` com o valor 1. De seguida, verifica se a condição `i<n` é satisfeita, o que neste caso é verdade pois $1 < 5$. Assim sendo, o programa entra no bloco de instruções do primeiro *for*, que é um novo ciclo *for*, e executa-o até ao fim. Neste segundo ciclo, o programa declara e inicializa a variável `j` com o valor `i+1` que neste caso é 2. De seguida, é avaliada a condição `j<n` que, neste caso, é verdadeira. Desta forma, o programa executa o bloco de instruções do segundo *for*, isto é escreve no ecrã "(1,2)  ". O passo seguinte é o incremento da variável `j`, isto é, `++j`, ficando esta variável com o valor 3. Uma vez que 3 ainda é menor que `n`, o programa volta a executar o bloco de instruções do segundo *for*, isto é, escreve no ecrã "(1,3)  ". Posto isto, a variável `j` é novamente incrementada, ficando com o valor 4 que é ainda menor que $n$. Isto significa que o bloco de instruções do segundo *for* é novamente executado e por isso o programa escreve no ecrã "(1,4)  ". O valor de `j` é novamente incrementado, passando a ser 5. Uma vez que 5 já não é menor que $n$, o segundo ciclo termina e o programa volta ao primeiro ciclo. Assim sendo, a variável `i` é incrementada passando a ter valor 2. Uma vez que 2 é menor que $n$, o programa volta a executar o segundo *for* inicializando a variável `j` com o valor de `i+1` que é 3. O processo continua até que a condição do primeiro *for* não seja satisfeita. O output final do programa é então "(1,2)  (1,3)  (1,4)  (2,3)  (2,4)  (3,4) ".

Através deste exemplo conseguimos perceber que, no caso de ciclos encadeados, o ciclo interior será executado cada vez que uma iteração do ciclo exterior for efetuada. É também importante referir que a inicialização da variável de controlo do segundo ciclo depende da variável de controlo do primeiro ciclo. Contudo, o inverso não seria possível pois o âmbito da variável `j` é entre as linhas 8 e 10.

### 2.3.5 As instruções *break* e *continue*

Independentemente da estrutura usada para implementar um ciclo (*do-while*, *while* ou *for*) esse ciclo terminará apenas quando a respetiva condição deixar de ser verificada. No entanto, em algumas situações, pode ser útil terminar um ciclo antes que tal aconteça. Para isso, poderemos usar a instrução *break* dentro do bloco de instruções do ciclo. Ao encontrar esta instrução, o programa abandona imediatamente o ciclo. Contudo, em ciclos encadeados, a instrução *break* permite apenas terminar um dos ciclos (dependendo do local onde está colocada) e não todos. Vejamos o seguinte exemplo:

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     int n = 5;
6.
7.     for(int i = 1; i < n; ++i) {
8.        for(int j = i + 1; j < n; ++j) {
9.            cout<<"("<<i<<","<<j<<") ";
10.               if ( j % i == 0 )
11.                    break;
12.       }
13.    }
14.
15.    return 0;
16. }
```

```
i = 1
    j = 2
        Escreve no ecrã "(1,2) "
        break (pois 2%1==0)
i = 2
    j = 3
        Escreve no ecrã "(2,3) "
    j = 4
        Escreve no ecrã "(2,4) "
        break (pois 4%2==0)
i = 3
    j = 4
        Escreve no ecrã "(3,4) "
    j = 5 (Fim do segundo ciclo)
i = 4
    j = 5 (Fim do segundo ciclo)
i = 5 (Fim do primeiro ciclo)
```

Este programa difere do anterior pelo facto de conter a instrução *break*, que apenas afeta o ciclo interior (o ciclo do `j`). O uso desta instrução neste programa faz com que o segundo ciclo possa terminar por duas razões: (i) quando $j \geq n$; ou (ii) quando `j` é múltiplo de `i`. O esquema de execução do programa é apresentado do lado direito.

A instrução *continue* permite parar uma iteração de um ciclo (*while*, *do-while* ou *for*) sem terminar o ciclo por completo. Mais precisamente, a instrução *continue* faz com que o programa "salte" da linha onde essa instrução está para o fim do ciclo em que está inserida. Vejamos o seguinte exemplo:

```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     for(int i = 1; i <= 9; ++i) {
6.        if(i % 4 == 0) {
7.             continue;
8.        }
9.       cout << i << endl;
10.    }
11.    return 0;
12. }
```

```
1
2
3
5
6
7
9
```

O programa apresentado do lado esquerdo imprime todos os números de 1 a 9 que não sejam divisíveis por 4. Assim, o ciclo *for* percorre todos os números de 1 e 9 e, quando estes forem divisíveis por 4, "salta" para o fim do ciclo *for*, não executando a instrução que imprime os números no ecrã (linha 9). Consideremos que a variável $i$ tem valor 3. Neste caso, a condição do *if* é avaliada como falsa e o programa não entra no *if*. Desta forma, a próxima instrução a ser executada é a da linha 9, sendo o valor 3 impresso no ecrã. De seguida, a variável de controlo de ciclo $i$ é incrementada para 4 sendo a condição do *if* avaliada como verdadeira e a instrução *continue* executada, o que faz com que o programa "salte" da linha 7 para a linha 10, onde termina a iteração do ciclo *for*, não imprimindo o valor 4 no ecrã. O resultado da execução do programa é o apresentado do lado direito.

É preciso ter cuidado ao usar a instrução *continue* em ciclos *while* e *do-while* uma vez que, nestes ciclos, o valor da variável de controlo do ciclo é atualizado dentro do corpo do ciclo e, como o uso da instrução *continue* faz com que o programa "salte" instruções, a atualização da variável de controlo de ciclo pode não ser efetuada, resultando num ciclo infinito. Considere-se o seguinte programa:

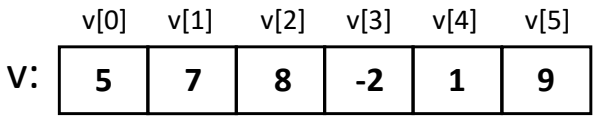
```
1. #include <iostream>
2. using namespace std;
3.
4. int main(){
5.     int i = 1;
6.     while(i <= 9) {
7.        if(i == 5) {
8.           continue;
9.        }
10.       cout << i << endl;
11.       ++i;
12.    }
13.
14.    return 0;
15. }
```

O programa anterior deveria imprimir no ecrã os números inteiros de 1 a 9, contudo apenas imprime os números de 1 a 4 entrando depois num ciclo infinito. Quando a variável de controlo $i$ toma o valor 5, a condição do *if* da linha 7 é avaliada como verdadeira e a instrução *continue* da linha 8 é executada fazendo, com que o programa "salte" para a linha 12. Significa isto que, a variável de controlo de ciclo $i$ nunca mais é incrementada, originando assim um ciclo infinito.

# Capítulo 3

# Variáveis indexadas - Vetores

Um *vetor* é um tipo de dados não primitivo que permite armazenar uma sequência de variáveis do mesmo tipo, por exemplo, várias variáveis do tipo *int*. Cada uma das variáveis é um *elemento* do vetor e é identificada por um número inteiro não negativo designado por *índice*. O índice do primeiro elemento é zero, pelo que um vetor cujo último índice seja $n$ terá $n+1$ elementos. A figura abaixo representa um vetor de inteiros `v` com 6 elementos (índices de 0 a 5).




Os elementos de um vetor não têm nome e são identificados pelo seu índice. Por exemplo, a variável que contém o valor 5, e que ocupa a primeira posição do vetor `v`, é identificada como `v[0]` enquanto que a variável que contém o valor -2 é identificada como `v[3]`. Intuitivamente, um vetor pode ser visto como um armário composto por várias gavetas onde cada gaveta contém um e um só elemento.

## 3.1 Declaração de vetores

A utilização de vetores em C++ requer a inclusão do pacote *vector* [1], pelo que será necessário escrever a instrução: `#include<vector>`, sempre que quisermos usar vetores. Existem duas formas principais de declarar um vetor: com dimensão e sem dimensão.

### 3.1.1 Declaração de vetor com dimensão

Esta forma de declaração é usada quando sabemos quantos elementos terá o vetor a criar, sendo feita de uma das seguintes formas:

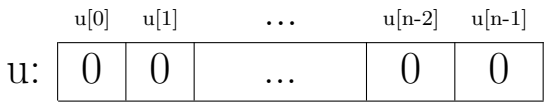
```
vector<Tipo_de_dados>  nome_vetor(n);
                    ou
vector<Tipo_de_dados>  nome_vetor(n, x);
```

Ambas as instruções criam uma vetor com o nome nome_vetor com `n` posições (desde 0 até `n-1`) para armazenar elementos do tipo Tipo_de_dados. A diferença entre elas é que a primeira inicializa todos

[1] Ver `https://cplusplus.com/reference/vector/vector/` para mais informações sobre o pacote *vector*.

os seus elementos com o valor *default* do tipo Tipo_de_dados do vetor e a segunda inicializa todos os elementos do vetor com o valor `x`.

Considerando que o Tipo_de_dados é `int` e que o nome_vetor é `u`, a figura que se segue representa o que a primeira instrução faz.




Tal como já foi dito, o vetor `u` tem $n$ elementos mas cada um dos elementos do vetor terá o valor *default* do tipo `int`, que é o valor 0. Para construir o vetor apresentado anteriormente $v = (5, 7, 8, -2, 1, 9)$ é necessário atribuir valores específicos a cada um dos elementos do vetor, o que é feito no seguinte excerto de código:

```
1.   #include <vector>
2.   using namespace std;
3.
4.   int main(){
5.       vector<int> v(6);   //Declaração
6.
7.       v[0] = 5;     //ou v.at(0) = 5;
8.       v[1] = 7;     //ou v.at(1) = 7;
9.       v[2] = 8;     //ou v.at(2) = 8;
10.      v[3] = -2;    //ou v.at(3) = -2;
11.      v[4] = 1;     //ou v.at(4) = 1;
12.      v[5] = 9;     //ou v.at(5) = 9;
13.
14.      return 0;
15.  }
```

### 3.1.2 Declaração de vetor sem dimensão

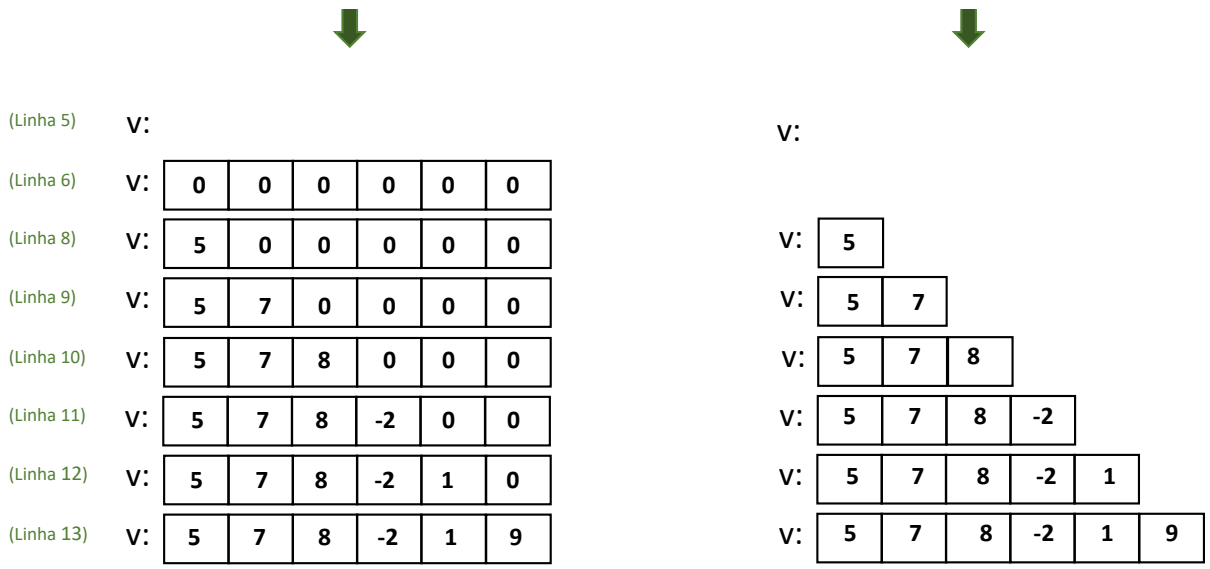
O C++ permite também a criação de vetores sem que seja especificada a sua dimensão no momento da criação. Para tal, pode ser usada a seguinte instrução:

```
vector<Tipo_de_dados>  nome_vetor;
```

No entanto, há que ter em conta que esta instrução por si só é inútil uma vez que apenas declara o vetor, isto é, cria um vetor sem posições. Assim sendo, caso se pretenda armazenar elementos no vetor, teremos primeiro que criar as posições necessárias para os colocar. Estas posições podem ser criadas todas de uma só vez fazendo um redimensionamento do vetor através da instrução `resize` ou podem ser criadas uma por uma usando a instrução `push_back`, conforme mostrado a seguir.

```
1. #include <vector>
2. using namespace std;
3.
4. int main(){
5.     vector<int> v; //Declaração
6.     v.resize(6);   //Redimensionar
7.
8.     v[0] = 5;      //Preenchimento
9.     v[1] = 7;
10.    v[2] = 8;
11.    v[3] = -2;
12.    v[4] = 1;
13.    v[5] = 9;
14.
15.    return 0;
17. }
```

```
1. #include <vector>
2. using namespace std;
3.
4. int main(){
5.     vector<int> v;
6.
7.    //Cria nova posição e preenche-a
8.     v.push_back(5);
9.     v.push_back(7);
10.    v.push_back(8);
11.    v.push_back(-2);
12.    v.push_back(1);
13.    v.push_back(9);
14.
15.    return 0;
17. }
```




No programa à esquerda é criado um vetor sem posições (linha 5) e logo depois é redimensionado (linha 6), passando a ter 6 posições. Neste redimensionamento, é atribuído automaticamente o valor *default* do tipo de dados do vetor (zero no caso dos tipos numéricos) a todos os elementos do vetor. Uma vez criadas as posições, estas são então preenchidas nas linhas 8-13. No programa à direita, o vetor é também declarado sem posições. No entanto, cada vez que se pretende adicionar um elemento ao vetor, é primeiro criada a posição para esse elemento, sendo depois preenchida com o elemento em causa. Tudo isto é feito internamente pela instrução `push_back`.

Em termos de eficiência computacional, o primeiro excerto de código com a instrução `resize` é mais eficiente que o segundo onde é usada a instrução `push_back`. Assim sendo, de entre estas duas instruções devemos priveligiar a escolha da primeira sempre que possível.

A dimensão de um vetor pode ser alterada várias vezes no decorrer do programa através do método `resize` verificando-se o seguinte: i) se a nova dimensão do vetor for superior à anterior, todos os

elementos do vetor são preservados e são criadas as posições em falta, sendo que os elementos nessas posições assumirão o valor *default* do tipo de dados do vetor (0 no caso de tipos numéricos); ii) se a nova dimensão do vetor for inferior à dimensão atual, então o vetor é simplesmente "cortado", sendo as últimas posições eliminadas. Importa ainda salientar que o método `resize` pode ainda ser chamado com dois argumentos, isto é,

```
v.resize(n,x);
```

neste caso, `n` será a nova dimensão do vetor e `x` será o valor de todos os elementos colocados nas novas posições criadas. Por exemplo, se tivermos $v = (1, 2, 3)$, a instrução `v.resize(5,10)` altera o vetor para $v = (1, 2, 3, 10, 10)$.

Tal como acontece com outros tipos de dados, é possível inicializar um vetor no momento da sua declaração. No entanto, tal inicialização terá que ser feita através de uma lista de elementos. Assim, no exemplo que estamos a considerar bastaria escrever:

```
1. #include <vector>
2. using namespace std;
3.
4.   int main(){
5.      vector<int> v = {5, 7, 8, -2, 1, 9};
6.      return 0;
7.   }
```

Esta seria a forma mais simples de criar o vetor pretendido, no entanto, este tipo de criação nem sempre é possível uma vez que frequentemente os valores a colocar no vetor (bem como a dimensão do vetor) não são conhecidos quando o vetor é declarado.

## 3.2 Método `.at()` *vs* operador [ ]

Para aceder/preencher uma posição de um vetor, pode ser usado tanto o operador [ ] como o método `.at()`. A principal diferença entre eles é o facto do método `.at()` fazer a validação da posição do vetor que se pretende aceder, isto é, verifica se essa posição existe no vetor. Vejamos o seguinte exemplo:

```
1. #include <vector>
2. using namespace std;
3.
4.  int main(){
5.      vector<int> v(2);
6.      v[0] = 5;  //ou v.at(0) = 5;
7.      v[1] = 7;  //ou v.at(1) = 7;
8.      v[2] = 8;      //ERRO  (o programa pode não terminar)
9.      v.at(2) = 8;   //ERRO  (o programa termina imediatamente)
10.     //...
11.     return 0;
12. }
```

O vetor `v` declarado na linha 5 tem apenas duas posições (posição 0 e posição 1). Assim sendo, o acesso a estas posições pode ser feito através do operador [ ] ou do método `.at()` (linhas 6 e 7). Quando tentamos aceder a uma posição do vetor que não existe (posição 2, por exemplo) usando o operador [ ], o programa não nos informa que tal posição não existe e acede a uma qualquer localização da memória do computador, devolvendo-nos o valor lixo aí existente. Dependendo do contexto, o programa pode terminar imediatamente sem apresentar qualquer mensagem de erro ou pode continuar a ser executado ficando o erro "camuflado". Neste último caso, o erro acaba por ser propagado ao longo do programa sem que nos apercebamos dele. Ao utilizarmos o método `.at()` para tentar aceder a uma posição do vetor que não existe, o programa terminará imediatamente e apresentará a mensagem de erro `out_of_range`.

Mas então, porque não usar sempre o método `.at()` uma vez que é mais seguro? A principal razão é a eficiência. Como o método `.at()` faz sempre a validação da posição a que se está a tentar aceder, o seu esforço computacional é maior, o que poderá ter grande impacto na eficiência computacional do programa. Além disso, o operador [ ] é mais simples de escrever. Assim sendo, cada opção tem as suas vantagens e desvantagens, pelo que ambas as formas podem ser usadas nesta cadeira.

## 3.3 Manipulação de vetores

Um vetor `v` pode ser visto como um conjunto de variáveis indexadas `v[i]` em que `i` é a posição da variável no vetor. Significa isto que cada uma dessas variáveis pode ser manipulada através dos operadores/métodos definidos para o seu tipo, como por exemplo os `cin`, `cout`, `+`, `==` para objetos do tipo *string*. No entanto, é necessário ter sempre presente que estes operadores não estão definidos para vetores. Isto é, sendo `v` e `u` dois objetos do tipo `vector` não é possível fazer, por exemplo, `v+u`. Significa isto que, para já, um vetor será sempre manipulado posição a posição e não como um todo como a seguir se explica.

### 3.3.1 Preenchimento de vetores

Já vimos anteriormente como preencher um vetor com valores específicos, conhecidos à priori. Suponhamos agora que pretendemos criar um vetor `v` de inteiros com dimensão `3`, sendo os seus valores pedidos ao utilizador. Sabemos já que a instrução `cin` permite pedir valores ao utilizador, mas esta instrução não está definida para vetores, pelo que não é possível fazer algo como "`cin>>v`". No entanto, é possível fazer "`cin>>v[0]`", " `cin>>v[1]`" e "`cin>>v[2]`" uma vez que `v[·]` é uma variável do tipo `int`. Como estamos a repetir várias vezes o procedimento de pedir valores ao utilizador, podemos usar uma estrutura de controlo cíclica para preencher o vetor. Além disso, dado que sabemos exatamente quantos valores vão ser pedidos, devemos utilizar o *for*. Nos excertos de código abaixo é feita a criação e preenchimento do vetor `v` com valores pedidos ao utilizador com e sem a utilização de um ciclo `for`.

```
1. #include <iostream>
2. #include <vector>
3. using namespace std;
4.
5. int main(){
6.    vector<int> v(3);
7.
8.    //cin >> v; //ERRO
9.    cin >> v[0]; //ou cin>>v.at(0);
10.   cin >> v[1];
11.   cin >> v[2];
12.
13.   return 0;
14. }
```

```
1. #include <iostream>
2. #include <vector>
3. using namespace std;
4.
5. int main(){
6.    vector<int> v(3);
7.
8.    for(int i = 0; i < v.size(); ++i){
9.       cin >> v[i]; //ou cin>>v.at(i);
10.   }
11.
12.
13.   return 0;
14. }
```

O código da direita permite preencher o vetor de forma automática uma vez que percorre sucessivamente cada posição do vetor (desde 0 até 2) pedindo um valor ao utilizador para essa posição. O método *size()* devolve a dimensão do vetor (neste caso 3). A manipulação de vetores requer frequentemente a utilização de um ciclo *for* como o que se apresenta na linha 8 do código da direita para percorrer todas as posições do vetor (desde 0 até *size()-1*).

Quando a dimensão do vetor não é conhecida, aconselha-se a utilização de um ciclo *while* para preenchê-lo. Suponhamos que se pretende pedir sucessivamente valores numéricos ao utilizador até ser introduzido um valor não numérico. Neste caso, não sabemos à partida quantos valores numéricos o utilizador vai introduzir, pelo que devemos definir um vetor sem especificar a sua dimensão e usar a instrução `push_back` para criar uma nova posição no vetor cada vez que for inserido um valor numérico. Este procedimento é efetuado no código abaixo.

```
1.  #include <iostream>
2.  #include <vector>
3.  using namespace std;
4.
5.  int main(){
6.      vector<double> v;
7.
8.      double x;
9.      while(cin >> x){ //Enquanto forem lidos valores numéricos
10.        v.push_back(x);
11.     }
12.
13.     return 0;
14. }
```

Cada vez que o ciclo *while* começa, a instrução `cin>>x` irá tentar atribuir à variável `x` (que é do tipo *double*) o valor inserido pelo utilizador. Caso esse valor seja numérico, a atribuição é bem sucedida e por isso o programa entra no ciclo *while*, cria uma nova posição no vetor e coloca nela o valor da variável

`x`. Caso o utilizador insira um valor não numérico, esse valor não consegue ser atribuído à variável `x` e a instrução `cin>>x` devolve `false`, fazendo com que o ciclo *while* termine imediatamente.

### 3.3.2 Impressão de vetores

Tal como o preenchimento de um vetor, também a sua impressão deve ser feita posição a posição uma vez que o método *cout* não está definido para vetores. Assim sendo, como o vetor que queremos imprimir tem dimensão conhecida (uma vez que já está criado), a forma mais comum de o imprimir é utilizando um ciclo *for* conforme mostrado no excerto de código abaixo.

```
1.   #include <iostream>
2.   #include <vector>
3.   using namespace std;
4.
5.  int main(){
6.      vector<int> v = {5, 7, 8, -2, 1, 9};
7.
8.      cout << "(";
9.      for(int i = 0; i < v.size(); ++i){
10.         if( i < v.size() - 1)
11.           cout << v[i] << ", ";  //Para todos os elementos que não o último
12.         else
13.           cout << v[i] << ")";   //Para o último elemento
14.     }
15.     return 0;
16.  }
```

Note-se que o ciclo *for* tem exatamente a mesma estrutura do que aquele que é usado para preencher o vetor, dado que é necessário percorrer todas as posições do vetor. O *if* presente no interior do ciclo *for* tem como objetivo distinguir a impressão do último elemento da dos restantes. Isto porque, após a escrita do último elemento do vetor, deve ser escrito um parêntesis e não uma virgula como acontece para os restantes elementos. O output do programa será então

*(5, 7, 8, -2, 1, 9).*

### 3.3.3 Ordenação de vetores

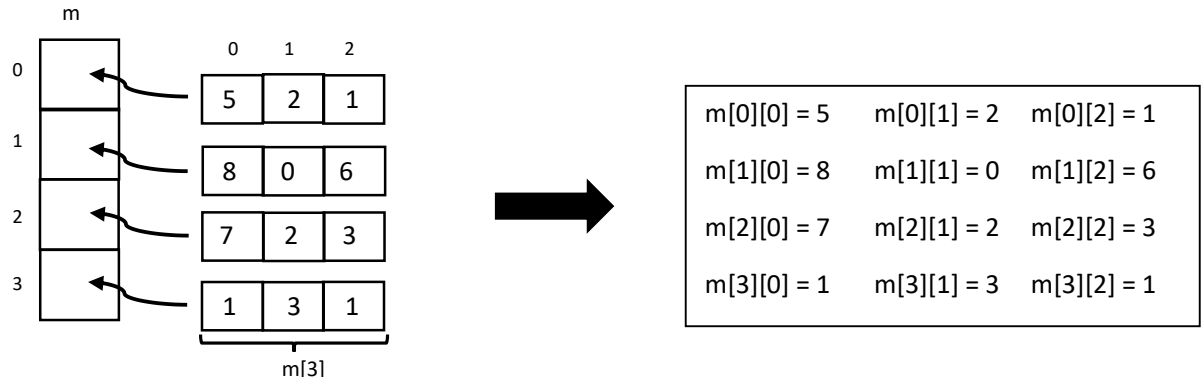
A ordenação de vetores é essencial para simplificar tarefas que habitualmente realizamos com vetores, tais como a pesquisa de elementos ou a obtenção de estatísticas descritivas. Existem vários algoritmos de ordenação, como o *Bubble Sort*, o *Insertion Sort*, o *Sequencial Sort*, etc. O C++ dispõe já de um método de ordenação, o método *sort*, que iremos usar nesta cadeira sempre que precisarmos de ordenar vetores. Este método pertence ao pacote *algorithm*, pelo que a sua utilização requer a inclusão da instrução `#include<algorithm>` no preâmbulo. No código abaixo é exemplificada a ordenação de um vetor por ordem crescente e decrescente.

```
1.  #include <vector>
2.  #include <algorithm>
3.  using namespace std;
4.
5.  int main(){
6.      vector<int> v = {5, 7, 8, -2, 1, 7};
8.
9.      //Ordenar vetor v por ordem crescente
10.        sort( v.begin(), v.end() );
11.
12.     //Ordenar vetor v por ordem decrescente
13.        sort( v.begin(), v.end(), greater <>() );
14.
15.     return 0;
16.  }
```

Na linha 6 temos, por exemplo, `v[0] = 5`. A partir da linha 10, o vetor `v` passa a estar ordenado por ordem crescente, pelo que, `v[0] = -2`. A partir da linha 13, o vetor passa a estar ordenado por ordem decrescente, pelo que `v[0] = 9`.

## 3.4 Vetores de vetores - Matrizes

Como vimos no início deste capítulo, um objeto do tipo `vetor` armazena variáveis de um determinado tipo. Em particular, pode armazenar variáveis também do tipo `vetor`, o que origina um vetor de vetores. Esta estrutura de dados é a forma mais natural de representar uma matriz em C++. Esquematicamente, uma matriz pode ser representada da seguinte forma:




O vetor `m` é composto por 4 elementos, sendo que cada elemento é um novo vetor de dimensão 3. O vetor `m` pode então ser visto como uma matriz de dimensão $4 \times 3$, isto é, uma matriz com 4 linhas e 3 colunas. Cada elemento da matriz é identificado por `m[i][j]`, sendo $i$ o índice no vetor principal `m` (linha) e $j$ o índice no vetor secundário `m[i]` (coluna). Sendo uma matriz um vetor de vetores, é necessário definir as dimensões de todos os vetores envolvidos antes de preencher a matriz. A criação da matriz do exemplo acima pode ser feita da seguinte forma:

```
1. #include <vector>
2. using namespace std;
3. int main(){
4.     //Opção 1: Declarar e preencher a matriz
5.     vector<vector<int>> m = { {5, 2, 1}, {8, 0, 6}, {7, 2, 3}, {1, 3, 1} };
6.
7.     //Opção 2: Criar matriz sem dimensões e depois redimensionar
8.     vector<vector<int>> m; //ou vector<vector<int>> m(4); e eliminar a linha 9
9.     m.resize(4); //Definir número de linhas (dimensão do vetor principal)
10.
11.    //Definir número de colunas (dimensão de cada vetor secundário)
12.    for(int i = 0; i < m.size(); ++i)
13.          m[i].resize(3);
14.
15.    //Preencher matriz
16.    m[0][0] = 5;
17.    //...
18.    m[3][2] = 1;
19.
20.    return 0;
21. }
```

A primeira opção de criação da matriz é claramente a forma mais simples de o fazer. No entanto, esta opção é apenas possível no caso em que quer as dimensões quer os elementos da matriz são conhecidos no momento da sua criação, o que frequentemente não acontece. Quando as dimensões da matriz e os seus elementos não são conhecidos no momento da criação da matriz - o que acontece por exemplo se essa informação for pedida ao utilizador durante a execução do programa - teremos de usar a segunda opção. É importante também realçar que o número de linhas da matriz pode ser definido aquando da sua declaração, conforme é descrito no comentário da linha 8. No entanto, não é possível usar algo semelhante para definir também o número de colunas, isto é, não é possível escrever `vector<vector<int>> m(4,3)`.

No exemplo apresentado o vetor de vetores foi usado para representar uma matriz, e para que tal aconteça, os vetores secundários devem ter todos a mesma dimensão. Contudo, seria possível definir um vetor de vetores onde os vetores secundários teriam dimensões diferentes.

Tal como no caso dos vetores simples, também as matrizes são manipuladas entrada a entrada e não como um todo. Significa isto que para manipular uma matriz é necessário percorrer todas as suas entradas, isto é, todas as suas linhas e colunas. A forma mais simples de o fazer é utilizar dois ciclos *for* encadeados, sendo que o primeiro irá percorrer "as linhas" da matriz e o segundo "as colunas". O código abaixo mostra como se pode imprimir uma matriz em C++.

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main(){
    vector<vector<int>> m = { {5, 2, 1}, {8, 0, 6}, {7, 2, 3}, {1, 3, 1} };

    for(int i = 0; i < m.size(); ++i){           //Linha i
        for(int j = 0; j < m[i].size(); ++j){    //Coluna j
           cout << m[i][j] << " ";
        }
        cout << endl;   //Muda de linha após escrever uma linha completa
     }

    return 0;
}
```

Note-se que usamos o método `cout` aplicado a cada entrada da matriz e não à matriz em si. Fazer algo como "`cout << m`" não é possível pois o comando `cout` não está definido para objetos do tipo `vector<vector<int>>`. O preenchimento de uma matriz é feito de modo semelhante, isto é, através da utilização de dois ciclos *for*, pelo que não será aqui apresentado.

# Capítulo 4

# Funções

O termo *função* remete-nos intuitivamente para a área da matemática onde uma função é caracterizada por um domínio, um contradomínio e uma expressão analítica. Por exemplo, a função $f$

$$\begin{aligned} f: \ \mathbb{Z}\times\mathbb{Z} &\longrightarrow \mathbb{R} \\ (x,y) &\longrightarrow f(x,y)=\frac{x}{y^2+1} \end{aligned}$$

recebe dois argumentos inteiros ($x$ e $y$) e devolve uma valor real que é o resultado da divisão de $x$ por $y^2+1$. Em programação é importante distinguir três conceitos associados a uma função: *declaração*, *definição* e *chamada*. A *declaração* de uma função consiste em indicar o seu nome, o tipo dos seus argumentos (tipo de dados que recebe - domínio) e o tipo de retorno (tipo de dados que devolve - contradomínio). A *definição* de uma função consiste em explicitar o que é que a função faz (expressão analítica). Por fim, a *chamada* da função consiste em executar a função para valores concretos dos seus argumentos. No exemplo em causa temos:

$$\begin{aligned} f: \ \mathbb{Z}\times\mathbb{Z} &\longrightarrow \mathbb{R} && \text{(declaração)} \\ (x,y) &\longrightarrow f(x,y)=\frac{x}{y^2+1} && \text{(definição)} \\ f(2,3),\ f(5^2,-8)\ \ldots & && \text{(chamadas)} \end{aligned}$$

Em C++, para este mesmo exemplo teríamos:

```
//declaração
     double f(int, int);

//declaração e definição
     double f(int x, int y){
          return x / (y * y + 1);
     }

//chamadas
     cout << f(2,3);            //chamada 1
     double z1 = 5 * f(1,7);    //chamada 2
     int a = 6, b = 1;
     double z2 = f(a,b);        //chamada 3
```

No primeiro bloco de código, é feita apenas a declaração da função. Isto é, é indicado que a função $f$ recebe dois argumentos do tipo *int* e devolve um resultado do tipo *double*. No segundo bloco de código, é feita simultaneamente a declaração e a definição da função. A definição da função corresponde ao bloco de código que aparece dentro das chavetas {...}. Finalmente, no último bloco de código, são feitas três *chamadas* da função. Note-se que nestes casos, tal como em matemática, apenas necessitamos de colocar os valores nos argumentos da função sem indicar o seu tipo uma vez que tal já ficou explícito na declaração/definição da função.

Neste caso, a função $f$ devolve um valor real que tanto pode ser diretamente impresso no ecrã (chamada 1), como usado para definir o valor de uma variável (chamadas 2 e 3). Na primeira chamada da função, o primeiro argumento `x` assumirá o valor 2 e o segundo argumento `y` o valor 3. Na terceira chamada da função, `x` assumirá o valor da variável `a` (que é 6), `y` o valor da variável `b` que é 1 e o valor devolvido pela função será armazenado na variável `z2`.

É importante notar que o nome dos argumentos de uma função é apenas usado internamente na função, sendo por isso independente do nome das variáveis usadas na chamada dessa função. Além disso, uma vez que esta função foi declarada com 2 argumentos do tipo *int*, ela terá sempre que ser chamada com 2 argumentos do tipo *int*, pelo que, por exemplo, $f(1)$, $f(1,8,5)$, $f(\text{`}j\text{'},5)$ e $f$ não são chamadas válidas para esta função.

## 4.1 Sintaxe geral de uma função

A estrutura geral da declaração e definição de uma função em C++ é a seguinte:

```
Tipo_retorno nome(Tipo_a1 nome_a1, ..., Tipo_an nome_an){
   //...
   return ... ; //Se "Tipo" for diferente de "void"
}
```

nesta declaração e definição:

- `nome` é o nome dado à função;
- `Tipo_retorno` é o tipo de dados devolvido pela função. Caso a função não devolva qualquer resultado o tipo de retorno será *void*;
- `nome_v1`, ..., `nome_an` são os nomes dos argumentos da função;
- `Tipo_a1`, ..., `Tipo_an` são os tipos de dados dos argumentos da função.

Quando a instrução *return* é executada, o programa sai da função imediatamente e o valor de retorno da função é devolvido/retornado (através de uma cópia) para o programa onde esta foi chamada. Assim sendo, esta instrução não precisa de ser usada em funções do tipo *void*, uma vez que estas nada devolvem. As funções do tipo *void* são frequentemente usadas para imprimir algo no ecrã, não devolvendo por isso qualquer resultado que possa ser usado no programa onde a função foi chamada, contrariamente ao que acontecia com a função $f$ apresentada anteriormente. De seguida é apresentado um exemplo da declaração e definição de duas funções, uma com o tipo de retorno *int* e outra com o tipo de retorno *void*. No entanto, é importante chamar a atenção para o facto de que as funções não têm necessariamente de ter argumentos, como é o caso da função *main* que sempre temos usado.

**Exemplo 1**

Suponhamos que se pretende implementar uma função que devolva o máximo entre dois números inteiros e outra que escreva dois números inteiros por ordem crescente. Ambas as funções recebem os mesmos argumentos: dois números inteiros que internamente serão denotados por `n1` e `n2`. Uma vez que a primeira (linhas 4-9) tem como objetivo calcular o máximo entre dois números inteiros, o seu tipo de retorno é também um número inteiro. Por outro lado, a segunda função (linhas 11-16) apenas escreve no ecrã os dois números inteiros recebidos por ordem crescente e por isso não devolve qualquer resultado para o programa onde for chamada. Assim sendo, esta função é do tipo *void* e por isso não existe qualquer instrução *return* no seu interior.

```
1.  #include <iostream>
2.  using namespace std;
3.
4.  int maximo(int n1, int n2){
5.      int max = n1;
6.      if( max < n2 )
7.          max = n2;
8.      return max;
9.   }
10.
11.  void ordem(int n1, int n2){
12.     if( n1 < n2 )
13.        cout << n1 << " <= " << n2;
14.     else
15.        cout << n2 << " <= " << n1;
16.  }
17.
18.  int main(){
19.     int a;
20.     int b;
21.     cout << "Introduza a e b: ";
22.     cin >> a >> b;
23.
24.     //Chamadas  das funções
25.     int x = maximo(5,7);
26.     int y = maximo(a,b) - 6;
27.     cout << "O maximo entre 2 e 8 é " <<  maximo(2,8) << endl;
28.     int z = maximo( maximo(7,8) , maximo(1,6) );
29
30.     cout << "Ordem: ";
31.     ordem(7,5);
32.     cout << "\n Ordem: ";
33.     ordem( maximo(1,8) , 3 );
34.     return 0;
35.  }
```

Dado que um programa em C++ inicia sempre a sua execução pela função *main*, todas as restantes funções terão de ser declaradas antes dela, por isso é que as funções `maximo` e `ordem` estão declaradas/-definidas nas linhas 4-16. Como referido anteriormente, ao *chamar* uma função, é obrigatório escrever o seu nome e todos os seus argumentos (sem especificar o seu tipo). Ao chamar a função `maximo` na linha 25, indicamos que o valor dos seus argumentos `n1` e `n2` (linha 4) é respetivamente 5 e 7. Assim sendo, quando o programa chega à linha 25, irá saltar para a linha 4 e executará as instruções da função `maximo` considerando `n1=5` e `n2=7`. Ao chegar à linha 8, o programa devolve o valor da variável `max` (que será 7) para o local onde a função foi chamada, isto é, para a linha 25, sendo por isso o valor da variável `x` igual a 7.

Ao chamar a função `maximo` na linha 26, os valores das variáveis `a` e `b` (anteriormente pedidos ao utilizador) são passados como argumento à função, definindo por isso os valores de `n1` e `n2`. A execução da função `maximo` termina com o retorno do máximo entre os valores de `a` e `b` para a linha 26. A esse máximo, é subtraído o valor 6, sendo o resultado final guardado na variável `y`.

O valor devolvido por uma função não tem necessariamente que ser guardado numa variável como nas linhas 25 e 26. Uma vez que a função `maximo` devolve um `int`, ele pode ser diretamente impresso no ecrã, como é feito na linha 27.

Uma função pode ainda ser chamada com argumentos que são eles próprios funções desde que os tipos de retorno das funções interiores estejam de acordo com o tipo dos argumentos das funções exteriores, tal como acontece na linha 28. A chamada das funções `maximo(7,8)` e `maximo(1,6)` resulta, respetivamente, nos valores 8 e 6, que são do tipo *int* pois é esse o tipo de retorno da função `maximo`. Estes valores serão então os argumentos da função exterior, pelo que a linha 28 é equivalente a `int z = maximo(8,6);`. Assim sendo, o valor da variável `z` será 8.

Contrariamente à função *maximo*, a função *ordem* não devolve qualquer resultado para o programa principal, apenas escreve informação no ecrã, sendo por isso uma função do tipo *void*. Assim sendo, esta função terá de ser chamada de forma isolada (linhas 31 e 33), não podendo ser usada nem para definir valores de variáveis nem dentro de um *cout*. A função *ordem* recebe dois argumentos do tipo *int* e por isso pode ser chamada como na linha 33, uma vez que a chamada da função `maximo(1,8)` tem como resultado um valor do tipo *int* que será o primeiro argumento da função *ordem*.

No código abaixo são apresentados vários exemplos de chamadas incorretas das duas funções anteriores.

```
int maximo(int n1, int n2){...}
void ordem(int n1, int n2){...}

//Chamadas incorretas das funções
    int w = maximo;                     //Faltam argumentos
    int a = 9, b = 4;
    int r = maximo(int a , int b);      //Não colocar os tipos
    int s = maximo(c, d);               //Variáveis c e d não declaradas
    int t = maximo("A", 3);             //Argumento não do tipo int
    int y = maximo(n1, n2);             //Argumento não definidos
    int u = maximo(5);                  //Faltam argumentos
    cout << "O maximo  é " <<  maximo;  //Faltam argumentos
    int h = ordem(2,8);                 //Função ordem não devolve um int
    cout << "Ordem: " <<  ordem(7,5);   //Função ordem nada devolve
    ordem;                              //Faltam argumentos
   cout << maximo(ordem(1,6), 5);      //Função ordem não devolve um int
```

**Exemplo 2**

Uma função pode ter como argumento qualquer tipo de dados, em particular pode ter argumentos do tipo `vector`. No programa abaixo (lado esquerdo) está declarada e definida uma função *print* que recebe como argumento um vetor de inteiros e devolve esse vetor escrito como *string*.

```
1. #include <iostream>
2. #include <vector>
3. using namespace std;
4.
5. string print(vector<int> x){
6.    string s = "(";
7.    for(int i = 0; i<x.size(); ++i){
8.       if( i < x.size() - 1)
9.          s+=to_string(x[i]) + ", ";
10.       else
11.          s+=to_string(x[i]) + ")";
12.   }
13.   return s;
14. }
15.
16. int main(){
17.    vector<int> v = {2, 3, 1, 7};
18.    vector<int> u = {3, 5, 1, 1};
19.    vector<int> w(4);
20.
21.    for(int i = 0; i<v.size(); ++i)
22.       w[i] = v[i] + u[i];
23.
24.    cout << print(v) << "+";
25.    cout << print(u);
26.    cout << "=" << print(w);
27.
28.    return 0;
29. }
```

```
1. #include <iostream>
2. #include <vector>
3. using namespace std;
4.
5. int main(){
6.    vector<int> v = {2, 3, 1, 7};
7.    vector<int> u = {3, 5, 1, 1};
8.    vector<int> w(4);
9.
10.   for(int i = 0; i<v.size(); ++i)
11.      w[i] = v[i] + u[i];
12.
13.   string s1 = "(";
14.   for(int i = 0; i<v.size(); ++i){
15.      if( i < v.size() - 1)
16.         s1+=to_string(v[i]) + ", ";
17.      else
18.         s1+=to_string(v[i]) + ")";
19.   }
20.
21.   string s2 = "(";
22.   for(int i = 0; i<u.size(); ++i){
23.      if( i < u.size() - 1)
24.         s2+=to_string(u[i]) + ", ";
25.      else
26.         s2+=to_string(u[i]) + ")";
27.   }
28.
29.   string s3 = "(";
30.   for(int i = 0; i<w.size(); ++i){
31.      if( i < w.size() - 1)
32.         s3+=to_string(w[i]) + ", ";
33.      else
34.         s3+=to_string(w[i]) + ")";
35.   }
36.
37.   cout<< s1 << "+" << s2 << "=" << s3;
38.   return 0;
39. }
```

Na função *print*, o vetor recebido é sempre designado por `x`. Significa isto, que cada vez que a função *print* for chamada com um determinado vetor (`u`, `v` ou `w` como nas linhas 24, 25 e 26), será criada uma cópia desse vetor (designada por `x`) que será o argumento da função.

Este exemplo tem como objetivo mostrar duas das grandes utilidades das funções: a não repetição de código e a simplificação do programa onde a função é chamada (função *main*). Ambos os excertos de código acima produzem o mesmo output, isto é,

$$(2, 3, 1, 7) + (3, 5, 1, 1) = (5, 8, 2, 8).$$

No entanto, o código da direita não utiliza funções e por isso, cada vez que se pretende imprimir um vetor é necessário replicar as linhas 13-19 para o vetor em causa. Note-se que o processo efetuado nas linhas 21-27 e 29-35 é o "mesmo" das linhas 13-19, só que para vetores diferentes. Além de originar um código mais extenso, a repetição de excertos de código também é mais propícia a erros. Definir uma função para imprimir um (qualquer) vetor `x`, como é feito no código da esquerda, permite que sempre que se pretenda imprimir um vetor de inteiros (independentemente da sua dimensão) apenas seja necessário chamar a função para esse vetor, tal como é feito nas linhas 24, 25 e 26 do código à direita. Note-se que a função *main* do lado esquerdo é muito mais simples de ler do que a do lado direito.

## 4.2 Vantagens das funções

As funções são extremamente úteis em programação. Como ilustrado no exemplo anterior, as funções evitam repetições de código uma vez que são implementadas de forma bastante genérica, podendo depois ser chamadas várias vezes com argumentos diferentes.

Outra das grandes vantagens das funções é o facto de permitirem a modularidade do código. Isto é, com as funções é possível dividir o código em pedaços mais pequenos que são mais fáceis de organizar, testar e usar. Desta forma, as funções facilitam a divisão de tarefas em programas que envolvem várias pessoas, uma vez que são estruturas completamente independentes.

Uma vez definidas, as funções podem ser usadas por várias pessoas em diversos programas. Do ponto de vista do utilizador, apenas é necessário saber como é que uma determinada função foi declarada e não como foi definida, isto é, saber o seu nome, argumentos e tipo de retorno. Note-se que, sem que talvez tenhamos dado conta disso, já usamos várias funções que não sabemos como foram definidas. Alguns exemplos dessas funções são as funções *size*, *resize* e *at* para objetos do tipo *vector* e a função de conversão de valores numéricos para *string* (*to_string*).

## 4.3 Passagem por valor, por referência e por referência constante

Os argumentos de uma função podem ser passados *por valor*, *por referência* ou *por referência constante*, sendo a forma de o fazer a seguinte:

```
Tipo nome(Tipo_argumento argumento){...}       //Passagem por valor

Tipo nome(Tipo_argumento& argumento){...}       //Passagem por referência

Tipo nome(const Tipo_argumento& argumento){...} //Passagem por referência constante
```

Na passagem *por valor*, é passado como argumento à função uma cópia do valor do argumento usado no momento da chamada da função. Significa isto que todas as alterações que ocorram dentro da função serão aplicas a essa cópia e não à variável passada como argumento.

Na passagem *por referência*, o que é passado como argumento à função não é uma cópia do valor da variável mas sim o endereço de memória onde a variável está guardada. Assim sendo, a função consegue "ver" e "alterar" diretamente o valor dessa variável. Significa isto, que qualquer alteração que ocorra dentro da função será aplicada à variável que foi passada como argumento na chamada da função.

A passagem *por referência constante* é semelhante à passagem *por referência* no sentido em que o que chega como argumento à função é também o endereço de memória da variável. No entanto, ao usar uma referência constante, a função apenas consegue "ver"a variável não conseguindo fazer alterações.

Vejamos o seguinte exemplo:

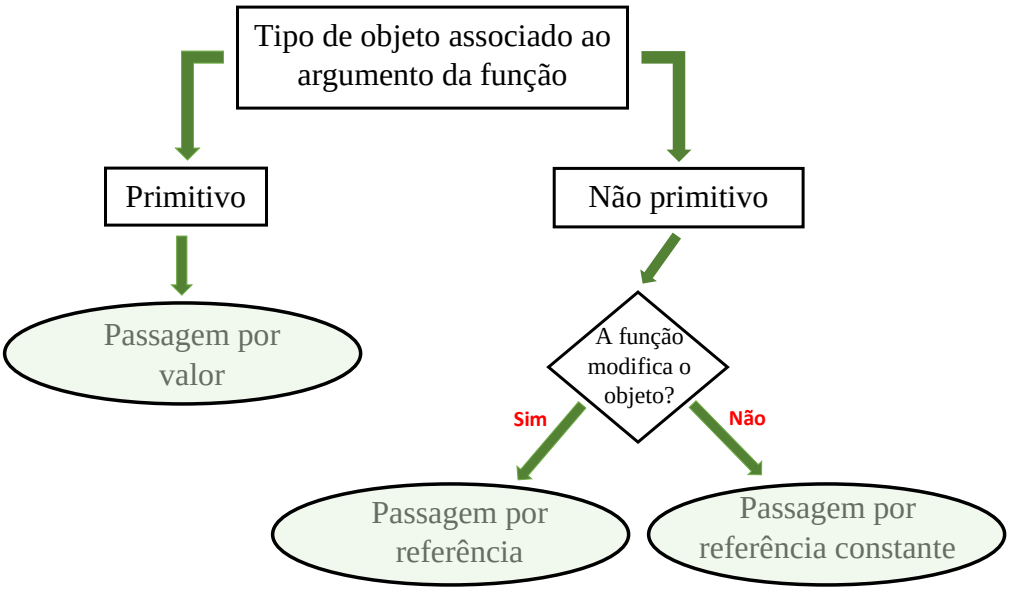
```
1.  #include <iostream>
2.  using namespace std;
3.
4.  void f(int a, int& b, const int& c){
5.     a += 10 + c;
6.     b += 10 + c;
7.     //c += 10;    ERRO!
8.     cout << a << " " << b << " " << c;  //a=12, b=12, c=1
9.  }
10.
11. int main(){
12.    int x = 1;
13.    int y = 1;
14.    int z = 1;
15.    f(x, y, z);
16.    cout << x << " " << y << " " << z;  //x=1, y=12, z=1
17.
18.    return 0;
19. }
```

A função *f* é chamada na linha 15 com os argumentos `x`, `y` e `z`. Esta função recebe três argumentos, sendo o primeiro passado *por valor*, o segundo *por referência* e o terceiro *por referência constante*. Assim sendo, na chamada da função na linha 15 é passado como argumento o valor da variável `x` (que é 1), o endereço de memória da variável `y` e o endereço de memória da variável `z`. Significa isto que `a=1`, `b` é exatamente a variável `y` e `c` é exatamente a variável `z`. Assim sendo:

- alterar a variável `a` dentro da função não altera a variável `x`, uma vez que `a` é uma cópia de `x` e não a variável `x`. Note-se que no final da execução da função temos `a=12` e `x=1`.
- alterar a variável `b` é o mesmo que alterar a variável `y` e por isso no final da execução da função temos `b=y=12`.
- `c` é uma referência constante para a variável `z`, pelo que o valor de `c` não pode ser alterado pela função (linha 7) e por isso temos `c=z=1`.

## Que tipo de passagem usar?

A utilização de cada tipo de passagem depende dos objetivos do programador. No entanto, existem algumas boas práticas a este respeito relacionadas principalmente com o tamanho dos objetos associados aos argumentos da função e à necessidade de modificação ou não desses argumentos. Tipos de dados primitivos como *int*, *double* e *char* estão associados a objetos considerados pequenos, enquanto que tipos de dados não primitivos como vetores e *strings* estão associados a objetos grandes. O esquema abaixo traduz as boas práticas gerais associadas aos tipos de passagem de argumentos.




O tempo e o esforço computacional requerido para criar cópias de objetos de tipos primitivos é desprezável e por isso estes tipos de objetos são geralmente passados *por valor* às funções. O mesmo não acontece com objetos de tipos não primitivos onde a criação de cópias pode ser demorada. Assim sendo, estes tipos de objetos devem ser passados para as funções *por referência* ou *por referência constante*. A escolha entre estes dois tipos de passagem depende do objetivo da função a implementar. Quando a função em causa apenas necessita de ter acesso ao objeto sem precisar de o modificar, deve ser usada uma referência constante. Contudo, se for necessário modificar o objeto, então terá que ser usada uma referência não constante.

Por exemplo, suponhamos que se pretende implementar uma função que receba um vetor e o imprima. A função em causa apenas necessita de aceder aos elementos do vetor e não de os alterar. Assim sendo, esta função deve receber uma referência constante para um vetor.

O uso de referências constantes é também uma segurança para o programador uma vez que garante que um determinado objeto não é alterado dentro de uma função.

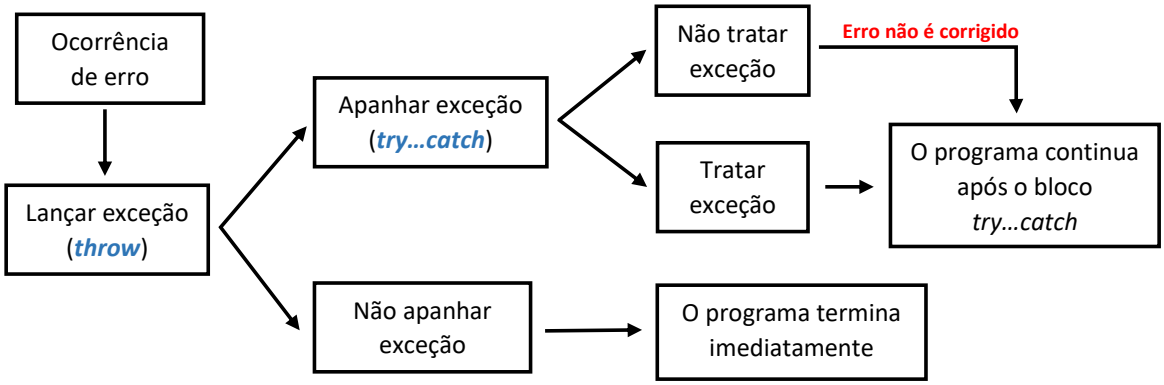
Como já sabemos, uma função pode apenas retornar um objeto. Contudo, as referências não constantes podem ser usadas como forma "artificial" de retornar um ou mais objetos. Note-se que no exemplo da secção anterior embora a função *f* sendo do tipo *void* não devolva qualquer valor, o novo valor da variável `b` (que é 12) é "devolvido" para a função *main*, devido ao facto de ter sido usada uma referência não constante.

# Capítulo 5

# Tratamento de erros

Para ser robusto, um programa deve ter a capacidade de lidar eficazmente com todos os tipos de erros que possam ocorrer durante a sua execução. Existem várias formas de lidar com erros sendo que a escolha da mais adequada depende do erro em causa. Os erros comprometem a execução do programa e, por isso, devem ser identificados. Um programa deve ter mecanismos para corrigir os erros identificados de modo a que possa continuar a sua execução. A não correção dos erros pode levar à interrupção imediata do programa ou a propagação de erros que comprometam o funcionamento do programa.

Um dos mecanismos mais comum no tratamento de erros é o uso de *exceções*. Exceções são situações anómalas que ocorrem durante a execução de um programa. Assim sendo, um programa deve estar preparados para sinalizar todas as exceções que possam ocorrer. Sinalizar uma exceção significa identificar um problema que pode ocorrer durante a execução do programa e informar o sistema da sua existência - a isto chama-se *lançar* uma exceção. Para lançar uma exceção, usamos a instrução *throw*. O lançamento de uma exceção faz com que o programa termine imediatamente, a não ser que seja usado um mecanismo que permita *apanhar* (e eventualmente tratar) a exceção lançada. Para apanhar uma exceção lançada para o sistema, usamos um bloco *try...catch*. A instrução *try* procura por exceções lançadas no seu bloco de instruções. A instrução *catch* permite definir ações para lidar com a exceção lançada e assim continuar a execução do programa. Estas ações podem ser meramente informativas, isto é, podem apenas informar o utilizador da existência do erro sem o corrigir, ou podem de facto corrigir o erro existente. O fluxograma abaixo resume o que acontece ao programa em função dos mecanismos usados para apanhar/tratar uma exceção.



A sintaxe geral das estruturas *throw* e *try...catch* são as seguintes:

```
//Bloco_de_instruções_1

if(condição_de_erro)
    throw  Exceção_a_lançar;

//Bloco_de_instruções_2

//Bloco_de_instruções_3
```

```
try{

    //Bloco_de_instruções_1

    if(condição_de_erro)
        throw  Exceção_a_lançar;

    //Bloco_de_instruções_2

}catch( Exceção_a_apanhar ){
    //Lidar com a exceção
}

//Bloco_de_instruções_3
```

O lançamento de exceções é geralmente feito dentro de instruções condicionais, uma vez que apenas acontece se determinada condição de erro se verificar. No excerto de código da esquerda, o programa começa por executar o `Bloco_de_instruções_1`. Se a `condição_de_erro` se verificar, é lançada uma exceção e o programa termina imediatamente. Caso contrário, o programa executará os dois blocos de instruções seguintes.

Para uma exceção ser apanhada pelo bloco *catch*, esta terá que ser lançada dentro do bloco *try* associado ao bloco *catch*. No bloco *catch*, é especificado o que deve ser feito no caso da `Exceção_a_apanhar` ter sido lançada dentro do bloco *try*. No excerto de código da direita, o programa entra diretamente no bloco *try* e começa por executar o `Bloco_de_instruções_1`. De seguida, verifica se a `condição_de_erro` é verdadeira e em caso afirmativo lança uma exceção. Após lançar a exceção, o programa salta o `Bloco_de_instruções_2` passando imediatamente para o bloco *catch*. Ao chegar ao bloco *catch*, o programa verifica se a exceção que foi lançada é do mesmo tipo da `Exceção_a_apanhar`. Caso não seja, o programa termina imediatamente. Caso contrário, são executadas as instruções do bloco *catch* para lidar com a exceção e o programa continua a sua execução passando para o `Bloco_de_intruções_3`.

No código da direita, se após entrar no bloco *try* e executar o `Bloco_de_instruções_1` não se verificar a `condição_de_erro`, o que significa ausência de erro, o programa executará o `Bloco_de_instruções_2` e em seguida o `Bloco_de_instruções_3`, ignorando o bloco *catch*.

A uma única instrução *try* podem estar associados vários blocos *catch*, um para cada exceção que se pretenda apanhar. Se uma exceção for lançada no bloco *try* e o bloco *catch* não estiver preparado para lidar com ela, a exceção não será apanhada, fazendo com que o programa termine imediatamente. Para evitar que tal aconteça, o último bloco *catch* associado ao *try* deve ser um bloco geral que permita apanhar todas as exceções que não foram apanhadas pelos blocos *catch* anteriores. Para apanhar qualquer exceção, devemos usar reticências (...) no argumento do *catch*. No exemplo abaixo é apresentado um bloco *try* com três blocos *catch* associados, sendo o primeiro e o segundo para duas exceções específicas (`Exceção_1` e `Exceção_2`) e o último um bloco geral para qualquer outra exceção que não estas. Note-se que quando a instrução *throw* é executada dentro do bloco *try*, o programa salta imediatamente para o bloco *catch* e, como apenas uma exceção foi lançada, apenas um dos blocos *catch* (`Exceção_1`, `Exceção_2` ou `...`) será executado.

```
try{

    //...

}catch( Exceção_1 ){
      //Lidar_com_exceção_1
}catch( Exceção_2 ){
      //Lidar_com_exceção_2
}catch( ... ){
      //Lidar_com_exceções_restantes
}
```

As exceções são objetos de um determinado tipo (`int`, `string`, entre outros). Contudo, para se conseguirem identificar claramente as exceções lançadas e lidar com elas de modo diferente, iremos definir uma *classe* para cada exceção. As classes apenas serão introduzidas em detalhe no Capítulo 7. Assim sendo, basta para já ter a noção de que uma classe é um tipo de dados não primitivo criado pelo utilizador. De seguida será explicado como podemos utilizar uma classe definida por nós para lançar uma exceção e como podemos utilizar classes já existentes na biblioteca *standard* para o fazer.

## 5.1 Classes vazias

A utilização de classes vazias para lidar com exceções é particularmente útil quando são lançadas exceções diferentes ao longo do programa e as queremos tratar separadamente. Quando este método é usado, devemos começar por declarar uma "classe vazia" para cada tipo de exceção que possa ocorrer. Estas declarações devem ser feitas antes das funções onde as exceções vão ser lançadas. No exemplo que se segue, as declarações são feitas antes da função *main*, que é a função que lançará as exceções. Neste exemplo, são criadas duas classes vazias, cada uma associada a uma condição de erro diferente. O bloco *try* apresentado neste exemplo tem dois blocos *catch* associados para lidar com duas exceções diferentes. É de referir ainda a utilização de referências constantes nos argumentos da instrução *catch*, o que se justifica por estarmos a lidar com objetos de tipos de dados não primitivos (classes).

```
//Preâmbulo
class Nome_exceção_1{};
class Nome_exceção_2{};

int main{

    //...

    try{
        if(condição_de_erro_1)
            throw  Nome_exceção_1();

        //...

        if(condição_de_erro_2)
            throw  Nome_exceção_2();

        //...

    }catch( const Nome_exceção_1& ){
            //Lidar com a exceção_1
    }catch( const Nome_exceção_2& ){
            //Lidar com a exceção_2
    }catch( ... ){
            //Lidar com outras exceções
    }

    //...
    return 0;
}
```

No exemplo abaixo são pedidos ao utilizador os valores das variáveis $n$ e $m$, e é criada uma nova variável *result* cujo valor é a divisão inteira de $n$ por $m$. Caso algum dos valores de $n$ ou de $m$ não seja lido corretamente, por exemplo, se tiver sido introduzido um valor não numérico, é lançada uma exceção do tipo `Leitura_Incorreta` e a divisão entre $n$ e $m$ não é efetuada uma vez que o programa passa diretamente para o primeiro bloco *catch*. Nesse bloco, a exceção é tratada atribuindo o valor 1 à variável *result* e o programa continua a ser executado passando para o último *cout* onde é impresso o valor 2 no ecrã. Note-se que a verificação do sucesso da leitura de um valor de uma determinada variável é feita através da instrução `if(!cin)`, que significa "se não foi lido o tipo de dados correto no *cin* anterior", uma vez que a instrução `cin` assume valor `false` caso a leitura falhe.

Caso a leitura dos valores de $n$ e $m$ seja feita com sucesso mas o valor de $m$ seja zero, é lançada uma exceção do tipo `Valor_Nulo` e o programa não faz a divisão entre $n$ e $m$. Neste caso, o programa passa imediatamente para o último bloco *catch* onde apenas apresenta uma mensagem de erro. Este é por isso um exemplo em que lidamos com a exceção (apresentando uma mensagem de erro) mas em que não a tratamos, ou seja, o programa continua a ser executado mas o erro é ignorando. Neste caso, a variável *result* continuará com o seu valor inicial (zero) e o último *cout* escreve no ecrã o valor 1.

Note-se que quando tratámos a exceção `Leitura_Incorreta` atribuímos o valor 1 à variável `result`, sendo este um valor arbitrário. A questão que se põe é: qual é o valor que devemos atribuir à divisão inteira quando um dos operandos é um valor não numérico? A resposta a esta questão não é clara, sendo esta a razão principal pela qual normalmente não tratamos as exceções e apenas lidamos com elas produzindo uma mensagem de erro.

```
//Preâmbulo
class Valor_Nulo{};
class Leitura_Incorreta{};

int main(){
  int n, m, result = 0;

  try{
      cout << "Valor de n: ";
      cin >> n;
      if(!cin)
          throw  Leitura_Incorreta();

      cout << "Valor de m: ";
      cin >> m;
      if(!cin)
          throw  Leitura_Incorreta();

      if(m == 0)
          throw  Valor_Nulo();

      result = n/m;

  }catch( const Leitura_Incorreta& ){
            result = 1;
  }catch( const Valor_Nulo& ){
            cout << "Atencao! o valor de m e zero...";
            cout << "... mas o programa vai continuar a executar";
  }

  cout << result + 1;
  return 0;
}
```

## 5.2 Classes da biblioteca *standard*

No C++ existem várias classes predefinidas para o tratamento de erros [1]. Dentre elas, salientamos a classe `runtime_error` e a classe `out_of_range` que iremos explorar de seguida.

### 5.2.1 Classe `runtime_error`

A classe `runtime_error` da biblioteca *standard* é uma das que mais iremos usar no tratamento de erros. Esta classe tem como argumento um objeto do tipo *string* onde se pode colocar uma mensagem de erro apropriada. Assim sendo, esta é a opção mais simples para fazer o tratamento de erros quando se pretende apenas escrever mensagens de erro específicas para cada tipo de erro.

Qualquer exceção to tipo `runtime_error` que seja lançada mas não seja apanhada causa a interrupção imediata do programa, como acontecia anteriormente com a instrução *throw*. Assim, caso queiramos lidar com a exceção, deve ser usado um bloco *try...catch*. A estrutura geral de um bloco *try...catch* para lidar com exceções do tipo `runtime_error` é a seguinte:

```
try{
    //...
    if(condição_de_erro_1)
        throw  runtime_error("Mensagem_de_erro_1");
    //...
    if(condição_de_erro_2)
        throw  runtime_error("Mensagem_de_erro_2");
    //...
 }catch( const runtime_error& e){
        cout << e.what();
 }
```

Em função do erro em causa, é passada como argumento à classe `runtime_error` uma mensagem específica. Quando uma exceção do tipo `runtime_error` é lançada, é depois apanhada pelo *catch*. Dentro do bloco *catch* é usado o método *.what()*, que devolve a mensagem que foi passada como argumento no lançamento da exceção. Essa mensagem é então escrita no ecrã. O método *.what()* tem que estar associado a um objeto do tipo `runtime_error`. Neste caso, esse objeto foi guardado na variável chamada `e`. No código abaixo é exemplificado o uso da classe `runtime_error` no tratamento de erros do programa apresentado na secção anterior.

[1] Mais informações sobre estas classes em https://cplusplus.com/reference/exception/exception/.

```
int n, m, result = 0;

try{
    cout << "Valor de n: ";
    cin >> n;
    if(!cin)
        throw  runtime_error("Leitura incorreta do n");

    cout << "Valor de m: ";
    cin >> m;
    if(!cin)
        throw  runtime_error("Leitura incorreta do m");

    if(m == 0)
        throw  runtime_error("O valor de m e zero");

    result = n/m;

}catch( const runtime_error& x){
    cout << x.what();
}
```

Como este exemplo é muito semelhante ao anterior, vamos apenas analisar as principais diferenças. Comecemos por reparar nas mensagens de erro existentes nos vários `runtime_error`. Caso o valor de `n` seja um valor não numérico, a mensagem de erro é "Leitura incorreta de n". Caso seja o valor de `m` que é não numérico, temos a mensagem de erro "Leitura incorreta de m". Note-se que aqui, contrariamente ao exemplo anterior, conseguimos saber pela mensagem de erro qual dos valores introduzidos é o não numérico. Por fim, caso o valor de `m` seja 0, a mensagem de erro é "O valor de m e zero". Apenas temos um bloco *catch*, onde a exceção será guardada na variável `x` e a mensagem de erro será impressa para o ecrã com o método *.what()*.

### 5.2.2 Classe `out_of_range`

A classe `out_of_range` é utilizada no tratamento de erros relacionados com acessos a posições inexistentes. Dos tipos de dados que conhecemos, os únicos que têm posições a eles associadas são o tipo `string` e o tipo `vector`. A análise seguinte é válida para os tipos de dados referidos.

Como vimos anteriormente, para aceder a elementos de vetores podemos usar tanto o operador [ ] como o método *.at()*, sendo a principal diferença entre eles o facto do método *.at()* fazer a validação da posição a que estamos a tentar aceder. Ao tentar aceder a uma posição do vetor que não exista através do método *.at()*, é lançada uma exceção do tipo `out_of_range`, que pode ou não ser apanhada através do uso de um bloco *try...catch*.

Vejamos o seguinte exemplo:

```
vector<int> v(2);

try{
    v.at(0) = 7;
    v.at(1) = 2;
    v.at(2) = 5; // É lançada uma exceção

}catch( const out_of_range& e){
    cout << "Erro: Posicao inexistente";
    // ou cout << e.what();
}
```

Neste exemplo é criado um vetor de dimensão 2 (com posições 0 e 1). Ao tentar aceder à posição 2, que não existe, é lançada uma exceção do tipo `out_of_range` que é apanhada pelo *catch*. Para lidar com a exceção, pode ser impressa uma mensagem de erro personalizada (como no primeiro caso) ou utilizado o método *.what()* que devolve informação sobre qual a dimensão do vetor e qual a posição a que se está a tentar aceder. Por fim, é importante ressalvar que não é necessário verificar a condição de erro através de um *if* porque isto já é feito na implementação do método *.at()*.

# Capítulo 6

# Separação de um projeto em ficheiros

À medida que um programa se torna maior, é pertinente dividir o código em diferentes ficheiros, cada um contendo partes independentes das outras, de forma a tornar o código mais *modular*. Os ficheiros criados podem ser compilados individualmente, o que permite adicionar novas funcionalidades ao programa sem ser necessário compila-lo todo novamente. Assim sendo, novos erros que possam surgir serão mais facilmente detetados, uma vez que estarão, provavelmente, associados às novas funcionalidades do programa, estando assim circunscritos a um ficheiro.

Uma das grandes vantagens da modularidade é também o facto de permitir que cada um dos diferentes ficheiros criados possa ser usado em vários programas, evitando assim repetir implementações de processos. Por exemplo, os pacotes da biblioteca *standard* estão implementados num único módulo, que não podemos alterar mas que podemos consultar. Assim, cada vez que queiramos usar uma funcionalidade já existente na biblioteca *standard*, basta fazer a sua inclusão no preâmbulo de cada programa e chamar diretamente os métodos lá existentes. Conforme explicado mais adiante, nós também podemos definir módulos, que podem ser partilhados por vários programas.

Para separar um projeto em ficheiros, devemos começar por criar um projeto e adicionar-lhe dois ficheiros: um ficheiro *cabeçalho* (ou *header* ou *.h*) e um ficheiro *corpo* (ou *.cpp*). Para adicionar o ficheiro cabeçalho, clicamos com o botão direito do rato no projeto criado, selecionamos *Add new*, selecionamos depois *C/C++ Header File* e finalmente definimos o nome do módulo. Para adicionar o ficheiro corpo, repetimos o mesmo processo, selecionando *C/C++ Source File* e atribuindo o mesmo nome que foi usado para o ficheiro cabeçalho. Este processo é ilustrado na Figura 6.1.

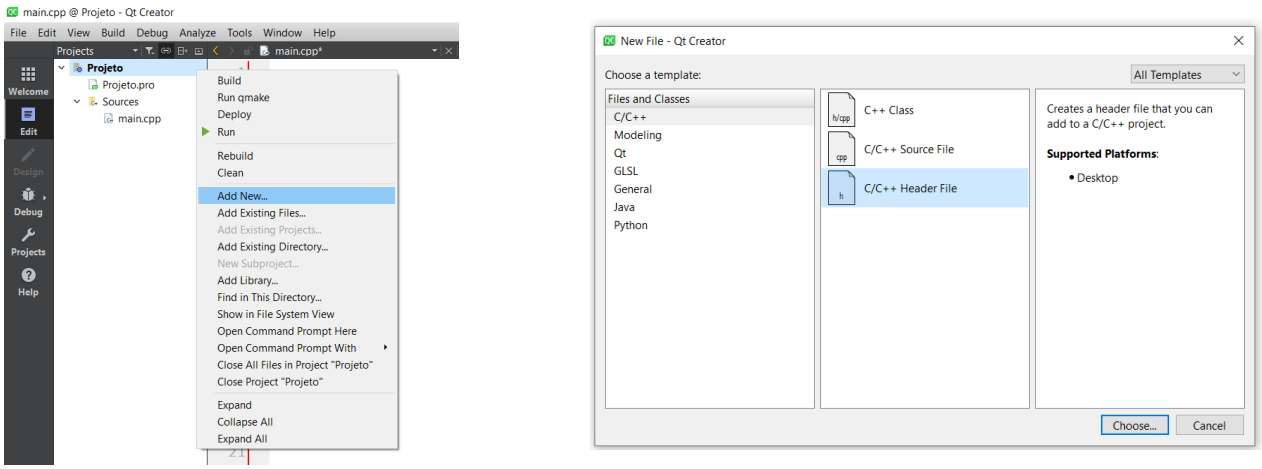



Figura 6.1: Como adicionar um ficheiro cabeçalho e um ficheiro corpo a um projeto.

Após a criação do ficheiro cabeçalho e do ficheiro corpo, a estrutura do projeto será a que se apresenta na Figura 6.2. Neste caso, o módulo criado tem o nome *Vetores*. O ficheiro cabeçalho tem a estrutura apresentada na figura, sendo que o nosso código é escrito no local indicado na imagem. As instruções `#ifndef VETORES_H` e `#define VETORES_H` servem para definir o conteúdo do ficheiro cabeçalho caso ainda não tenha sido definido. Caso já tenha sido definido (através da instrução `#include "Vetores.h"`), o conteúdo deste ficheiro é ignorado. O ficheiro corpo é criado vazio.

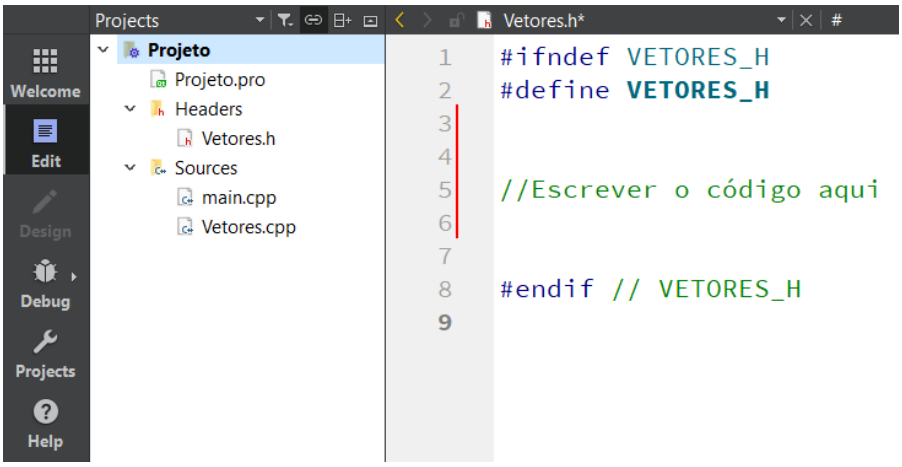



Figura 6.2: Estrutura do projeto com ficheiros cabeçalho e corpo.

O ficheiro cabeçalho deve conter apenas as declarações dos métodos, o que se justifica por duas principais razões: reduzir o tempo de compilação, e permitir que o utilizador identifique facilmente quais as componentes do módulo. Assim, o ficheiro cabeçalho pode ser visto como um "índice" de um livro. O "conteúdo" do livro, que no nosso caso corresponde ao conjunto das definições dos métodos, encontra-se no ficheiro corpo. Vejamos o exemplo seguinte.

Ficheiro Cabeçalho

```
#ifndef VETORES_H
#define VETORES_H

#include <iostream>
#include <vector>
using namespace std;

void print( const vector<int>& );

#endif // Vetores_H
```

Ficheiro Corpo

```
#include "Vetores.h"

void print( const vector<int>& v){
    for(int i = 0; i<v.size(); ++i){
        if(i == 0)
            cout << "(" << v[i];
        else if(i == v.size() - 1)
            cout << ", " << v[i] << ")";
        else
            cout << ", " << v[i];
    }
}
```

Programa Principal

```
#include "Vetores.h"

int main(){
    vector<int> u = {1, 2, 3};
    print(u);
    return 0;
}
```

Neste exemplo, é criado um módulo (*Vetores*) que contém, no ficheiro cabeçalho, a declaração de uma função *print* com tipo de retorno *void* que recebe como argumento uma referência constante para um vetor. A definição desta função é feita no ficheiro corpo. A instrução `#include "Vetores.h"` é necessária no topo deste ficheiro para estabelecer ligação entre os ficheiros cabeçalho e corpo. Uma vez definido o módulo, este pode ser usado em qualquer programa, bastando para isso a sua inclusão no programa através da instrução `#include "Vetores.h"`. Note-se que não é necessário incluir no programa principal os pacotes já incluídos no ficheiro cabeçalho, pois ao incluirmos o ficheiro cabeçalho estamos automaticamente a incluir todos os pacotes nele incluídos. Depois da inclusão do módulo, todos os métodos que lhe pertencem podem ser chamados diretamente, tal como é exemplificado com a instrução `print(u)`. A modularidade torna o programa principal muito mais compacto e legível, conforme foi ilustrado neste exemplo.

## 6.1 Espaços de nomes

Programas mais complexos requerem muitas vezes a inclusão de vários módulos, que podem ser criados por pessoas diferentes de forma independente. Isto pode fazer com que existam elementos declarados com o mesmo nome em módulos diferentes, por exemplo, duas funções com o mesmo nome. Ao fazer *include* de vários ficheiros cabeçalho no programa principal, corremos o risco de haver declarações repetidas, o que não é aceite pelo compilador por existir um conflito de nomes. Uma das formas de evitar este problema é através do uso de espaços de nomes - *namespaces*.

Um *namespace* é um âmbito com nome no qual podem ser declarados vários elementos. Quando usamos *namespaces*, o acesso aos seus elementos é feito indicando explicitamente a que *namespace* eles pertencem. Assim, mesmo que haja dois elementos com o mesmo nome em módulos diferentes, o acesso a cada um deles será feito de forma diferente, evitando assim conflitos de nomes.

Consideremos os seguintes módulos `M1` e `M2` onde é declarada uma função *print*. A primeira função pertence ao *namespace* `X` e a segunda ao *namespace* `Y`. Desta forma, para definir estas funções nos ficheiros corpo respetivos, é necessário especificar a que *namespace* pertencem usando as instruções `X::` e `Y::` antes do seu nome.

Ficheiro Cabeçalho M1

```
#ifndef M1_H
#define M1_H

#include <iostream>
#include <vector>
using namespace std;

namespace X{
     void print(const vector<int>&);
}

#endif // M1_H
```

Ficheiro Cabeçalho M2

```
#ifndef M2_H
#define M2_H

#include <iostream>
#include <vector>
using namespace std;

namespace Y{
     void print(const vector<int>&);
}

#endif // M2_H
```

Ficheiro Corpo M1

```
#include "M1.h"

void X::print(const vector<int>& v){
     // ...
}
```

Ficheiro Corpo M2

```
#include "M2.h"

void Y::print(const vector<int>& v){
     // ...
}
```

Programa Principal

```
#include "M1.h"
#include "M2.h"

int main(){
    vector<int> u = {1, 2, 3};
    print(u); // ERRO!
    X::print(u); // OK!
    Y::print(u); // OK!
    return 0;
}
```

No programa principal, onde são incluídos os dois ficheiros cabeçalho, podemos então chamar ambas as funções especificando a que *namespace* pertencem. Desta forma, fica inequivocamente identificada a função que queremos usar.

Indicar sempre o *namespace* a que pertence determinado elemento torna o programa mais extenso e

difícil de ler, mas é fundamental caso existam elementos com o mesmo nome pertencentes a *namespaces* diferentes. No entanto, quando esta questão não se coloca, podemos simplificar a escrita através da instrução *using*. Adicionar ao preâmbulo a instrução `using namespace X;` permite aceder aos elementos do *namespace* `X` diretamente sem ter que usar `X::` em cada um deles, uma vez que já indicámos que estamos a usar os elementos do *namespace* `X`. Note-se que isto é o que fazemos com os elementos do *namespace* `std` quando usamos a instrução `using namespace std;`. Deste *namespace* fazem parte, por exemplo, os elementos *cout*, *cin*, *string* e *vector*. Assim, para escrever algo como

```
vector<string> v;
// ...
cout << v[0];
```

sem usar a instrução `using namespace std;` teríamos de escrever

```
std::vector<std::string> v;
// ...
std::cout << v[0];
```

o que torna claramente o programa mais difícil de ler. A utilização da instrução `std::` tem apenas a vantagem de identificar claramente a que *namespace* os elementos pertencem.

No mesmo ficheiro cabeçalho podemos definir vários *namespaces*. Além disso, podemos ainda definir *namespaces* dentro de outros *namespaces*, conforme ilustrado no exemplo abaixo.

Ficheiro Cabeçalho M1

```
#ifndef M1_H
#define M1_H

#include <iostream>
#include <vector>      using namespace std;

namespace X{
    void print1(const vector<int>&);

     namespace Z{
        void print2(const vector<int>&);
     }
}

namespace W{
     void print3(const vector<int>&);
}

#endif // M1_H
```

Programa Principal

```
#include "M1.h"

int main(){
    vector<int> u = {1, 2, 3};
    X::print1(u);
    X::Z::print2(u);
    W::print3(u);
    return 0;
}
```

Note-se que as funções `print1`, `print2` e `print3` poderiam ter o mesmo nome por estarem associadas a *namespaces* diferentes. Além disso, todas estas funções poderiam estar definidas no mesmo ficheiro

corpo, sendo a identificação de cada uma deles feita de forma semelhante à que é feita no programa principal, isto é, usando as instruções `X::`, `X::Z::` e `W::`.

## 6.2 Redefinição de tipos de dados - *typedef*

A instrução *typedef* permite definir um nome alternativo (pseudónimo), em geral mais simples, para um determinado tipo de dados. Esse pseudónimo pode depois ser usado em todo o código em vez do tipo de dados original. A instrução *typedef* tem a seguinte sintaxe geral:

```
typedef tipo_de_Dados nome_alternativo;
```

No exemplo abaixo, apresentamos um exemplo de um ficheiro cabeçalho que contém a declaração de duas funções para manipular matrizes. A primeira (do tipo *void*) imprime uma matriz, enquanto que a segunda devolve uma matriz que é a soma das duas matrizes que recebe como argumento. No primeiro código, não é usada a instrução *typedef*, pelo que é necessário escrever `vector<vector<int>>` sempre que nos referirmos a esse tipo de dados. No segundo código, é usada a instrução *typedef* para criar o pseudónimo `matriz` para o tipo de dados `vector<vector<int>>`. O segundo excerto de código é claramente menos extenso e mais legível que o primeiro, sendo esta a principal vantagem do uso da instrução *typedef*.

Ficheiro Cabeçalho M1 - Sem typedef

```
#ifndef M1_H
#define M1_H

#include <iostream>
#include <vector>
using namespace std;

void print(const vector<vector<int>>&);
vector<vector<int>> soma(const vector<vector<int>>&; const vector<vector<int>>&);

#endif // M1_H
```

Ficheiro Cabeçalho M1 - Com typedef

```
#ifndef M1_H
#define M1_H

#include <iostream>
#include <vector>
using namespace std;
typedef vector<vector<int>> matriz;  // Definir "vector<vector<int>>" como "matriz"

void print(const matriz&);
matriz soma(const matriz&; const matriz&);

#endif // M1_H
```

# Capítulo 7

# Classes

Como sabemos, existem em C++ vários tipos de dados primitivos como *int*, *double*, *char*, etc. Devido à sua simplicidade, estes tipos de dados não permitem representar *objetos* com que frequentemente nos deparamos, tais como vetores, frações, números complexos, carros, livros, etc. Estes objetos têm: (i) atributos (por exemplo, matrícula, cor e número de portas, no caso de um carro); e (ii) funcionalidades (arrancar, travar, etc.), que podem ser representadas em C++ através de *classes*. Uma classe é um novo tipo de dados definido pelo utilizador para representar e manipular objetos que não são possíveis de representar e manipular através de tipos de dados primitivos. As classes tornam mais claro qual é o objeto ao qual estamos a aplicar a sua funcionalidade, sendo assim a base de qualquer linguagem de programação orientada a objetos. As classes não vêm substituir o que aprendemos até agora, vêm dar-nos uma ferramenta adicional para lidar com a complexidade do código, permitindo que o código seja escrito de forma mais modular.

Um exemplo de uma classe que tão bem já conhecemos é a classe *vector*. Esta classe foi criada para representar vetores e contém por isso métodos para aceder às suas propriedades, como é o caso do método *size()*, e funcionalidades para os manipular, como o método *push_back()*.

Suponhamos que queremos fazer um programa que lide com números complexos. No C++ não existe um tipo de dados que permita representar um número complexo, pelo que não é possível declarar um objeto desse tipo, isto é, fazer algo como

```
Complexo z;
```

No entanto, o utilizador pode criar uma classe `Complexo` para representar e manipular números complexos, o que tornaria possível a escrita anterior.

Uma classe deve ser declarada num ficheiro cabeçalho e definida num ficheiro corpo, de forma a ser reutilizada facilmente. Assim, para criar uma classe, clicamos com o botão direito do rato em cima do projeto e de seguida selecionamos *Add New*, tal como ilustrado na Figura 6.1 apresentada no capítulo anterior. Depois disso, selecionamos a opção *C++ Class* (ver Figura 6.1) e damos um nome à classe. Os excertos de código abaixo ilustram a estrutura dos ficheiros cabeçalho e corpo criados para a classe `Complexo`.

Ficheiro Cabeçalho

```
1.  #ifndef COMPLEXO_H
2.  #define COMPLEXO_H
3.
4.  class Complexo{
5.
6.  public:
7.      Complexo();
8.  };
9.
10. #endif // COMPLEXO_H
```

Ficheiro Corpo

```
1.  #include "complexo.h"
2.
3.  Complexo::Complexo(){
4.
5.  }
```

Note-se que a declaração da classe contém um ";"no final (ver linha 8 do ficheiro cabeçalho) e que, ao criar uma *Class C++*, o ficheiro corpo começa com o *include* do ficheiro cabeçalho.

Qualquer novo tipo de dados criado pelo utilizador tem por base outros tipos de dados. Um número complexo tem a forma $a+bi$, sendo $a$ a parte real e $b$ a parte imaginária. Assim, um número complexo pode ser representado por duas variáveis do tipo *double* que correspondem a essas partes real e imaginária. Numa classe, as variáveis usadas para representar um objeto são os *atributos* da classe, que podem ter qualquer tipo, incluindo outras classes.

Na classe podemos ter também vários *métodos* que permitam manipular o objeto da classe. Além disso, uma classe tem sempre pelo menos um *construtor*, cujo propósito é inicializar os seus atributos quando um novo objeto é criado. O construtor tem o mesmo nome da classe, podendo ou não ter argumentos. Assim, um construtor pode ser visto como uma "função" sem tipo de retorno. Os atributos, construtor(es) e métodos de uma classe são designados por *membros* da classe.

Numa classe podemos ter membros públicos (*public*) e privados (*private*). Os membros públicos podem ser acedidos dentro e fora da classe, enquanto que os membros privados apenas podem ser acedidos dentro dela, isto é, em métodos da classe. A declaração de um membro da classe como público ou privado depende da sua finalidade. No entanto, por serem os elementos estruturais da classe, os atributos devem ser privados para impedir que sejam diretamente modificados pelo utilizador. Para visualizar e modificar os atributos privados será então necessário criar métodos públicos, como é ilustrado no código abaixo.

Ficheiro Cabeçalho

```
#ifndef COMPLEXO_H
#define COMPLEXO_H

  //Colocar os includes necessários

  class Complexo
  {
  private:
     //Atributos
      double Real;
      double Im;

  public:
     //Construtores
      Complexo();
      Complexo(double , double);

     //Métodos
      void AlterarReal(double);
      void AlterarIm(double);
      double VerReal() const;
      double VerIm() const;

  };

#endif // COMPLEXO_H
```

Ficheiro Corpo

```
#include "complexo.h"

// Construtor por omissão
Complexo::Complexo(){
      Real = 0;
      Im = 0;
}

// Outro construtor
Complexo::Complexo(double a, double b){
      Real = a;
      Im = b;
}

 //Métodos
 void Complexo::AlterarReal(double x){
      Real = x;
 }

 void Complexo::AlterarIm(double x){
       Im = x;
 }

 double Complexo::VerReal() const{
       return Real;
 }

 double Complexo::VerIm() const{
      return Im;
 }
```

A classe `Complexo` contém como atributos privados duas variáveis do tipo *double*, uma com o nome `Real` e outra com o nome `Im` que guardam, respetivamente, a parte real e a parte imaginária do número complexo. Note-se que os atributos de uma classe aparecem no QT Creator a cor vermelha. Sendo os atributos privados, a classe dispõe de dois métodos públicos que permitem alterar cada um desses atributos, os métodos `AlterarReal` e `AlterarIm`. Existem também dois outros métodos públicos (`VerReal` e `VerIm`) que permitem aceder ao valor dos atributos da classe. É importante referir que estes dois métodos, tais como todos os que não alterem os atributos da classe, devem ser definidos como constantes. Para definir um método como constante, escrevemos a palavra *const* a seguir aos seus argumentos.

Nesta classe estão presentes dois construtores, o *construtor por omissão*, que não tem argumentos, e um outro, que recebe dois argumentos do tipo *double*. O primeiro construtor não recebe argumentos e, por isso, foi programado para inicializar os atributos da classe com o seu valor *default*, o valor zero. O segundo construtor recebe dois argumentos e utiliza-os para inicializar os atributos da classe. Os dois

construtores referidos podem ser alternativamente implementados através da listagem dos atributos da classe da seguinte forma:

Ficheiro Corpo

```
// Construtor por omissão
Complexo::Complexo(): Real(0), Im(0){ }

// Outro construtor
Complexo::Complexo(double a, double b): Real(a), Im(b){ }
```

sendo o significado de, por exemplo, `Real(a)` semelhante a `Real=a`. O propósito dos construtores é inicializar os atributos do objeto aquando da sua criação. Como pudemos ver no exemplo, uma classe pode ter vários construtores, desde que tenham argumentos diferentes.

Vejamos agora como é que esta classe pode ser usada no programa principal, na função *main*.

Programa Principal

```
1.  #include "complexo.h"
2.
3.  int main(){
4.     Complexo z1;
5.     Complexo z2(3, 2);
6.
7.      z1.Real = 1;   //ERRO
8.      z1.Im = 5;     //ERRO
9.      z1.AlterarReal(7);
10.     z1.AlterarIm(0);
11.
12.     //Imprimir z2
13.        cout << z2.VerReal() << "+" <<  z2.VerIm() << "i";
14.
15.     return 0;
16.  }
```

Para usar uma classe num programa é necessário fazer o *include* do seu ficheiro cabeçalho. Na linha 4, é criado um objeto `z1` do tipo `Complexo`. Neste momento, é implicitamente chamado o construtor por omissão, pelo que a parte real e a parte imaginária de `z1` serão inicializadas com o valor zero. Na linha 5, é criado um novo objeto `z2` do tipo `Complexo`. No entanto, neste caso, uma vez que são recebidos dois argumentos, é implicitamente chamado o segundo construtor, sendo por isso `z2=3+2i`;

Para aceder aos métodos públicos de uma classe usamos um ponto (".") depois do nome do objeto da classe, seguido do nome do método, tal como é feito nas linhas 9 e 10. Na linha 9, é chamado o método `AlterarReal` com o argumento 7, que permite alterar a parte real do complexo `z1` para 7. Uma vez que os atributos da classe são privados, não é possível aceder-lhes fora da classe, tal como é ilustrado nas linhas 7 e 8. Na linha 13 é impresso no ecrã o número complexo `z2` na forma $a+bi$, sendo por isso usados os métodos `VerReal()` e `VerIm()` para aceder aos seus atributos.

É importante ressalvar que os métodos de uma classe estão sempre associados a um objeto dessa classe, pelo que nunca poderão ser chamados sem serem aplicados a um objeto da classe. Por exemplo,

a única forma de utilizar o método `VerReal` é escrevendo `x.VerReal()`, onde `x` é um qualquer objeto do tipo `Complexo`. Assim, escrever algo como `.VerReal()` ou `VerReal()` fora da classe não é possível.

Uma classe pode conter métodos para manipular o tipo de objeto que representa. No caso da classe `Complexo`, faz sentido ter, por exemplo, um método público que permita imprimir um número complexo na forma $a+bi$. Nesta classe podemos ter ainda incluir métodos públicos para calcular o módulo de um número complexo, para verificar se um número complexo é um imaginário puro, entre outros. Estes métodos podem ter como argumentos e tipo de retorno objetos da própria classe. No código abaixo é ilustrada a inclusão de quatro funções (`Imprime`, `ImPuro`, `Simetrico` e `Soma`) na classe `Complexo`.

Ficheiro Cabeçalho

```
#ifndef COMPLEXO_H
#define COMPLEXO_H

//Colocar os includes necessários

class Complexo{
private:
  //Atributos
  double Real;
  double Im;

public:
  //Construtores
  Complexo();
  Complexo(double , double);

  //Métodos
  void AlterarReal(double);
  void AlterarIm(double);
  double VerReal() const;
  double VerIm() const;
  void Imprime() const;
  bool ImPuro() const;
  Complexo Simetrico() const;
  Complexo Soma(const Complexo&) const;
};




#endif // COMPLEXO_H
```

Ficheiro Corpo

```
#include "complexo.h"

// ...
// Restantes definições
// ...

void Complexo::Imprime() const{
    cout << Real;
    if ( Im >= 0 )
        cout << " + " << Im << "i";
    else
        cout << Im << "i";
}

bool Complexo::ImPuro() const{
    if (Real == 0 and Im != 0)
        return true;
    else
        return false;
}

Complexo Complexo::Simetrico() const{
    Complexo z;
    z.Real = -1*real;
    z.Im = -1*Im;
    //ou
    Complexo z(-1*Real, -1*Im);

    return z;
}

Complexo Complexo::Soma(const Complexo& z) const{
    Complexo zSoma(Real + z.Real, Im + z.Im);
    return zSoma;
}
```

A função `Imprime`, tal como o nome indica, imprime o objeto do tipo `Complexo` na forma "a+bi". O método `ImPuro` verifica se o número complexo é um imaginário puro. A função `Simetrico` calcula um novo complexo que é o simétrico do objeto da classe. Por fim, o método `Soma` devolve um número complexo que é a soma do objeto da classe com outro `Complexo`.

Estas funções não alteram os atributos da classe e, por isso, são definidas como constantes. A função `Simetrico` devolve um objeto `z` do tipo `Complexo` que é o simétrico do objeto que lhe deu origem. Nesta função, o `Complexo z` é criado usando o construtor da classe e é depois devolvido. A função `Soma` tem como argumento uma referência constante para um objeto do tipo `Complexo`, por se tratar de um tipo de dados não primitivo que não é alterado pela função. Esta função cria um novo objeto do tipo `Complexo`, que resulta da soma do objeto da classe com o `Complexo z`, e devolve-o.

No programa principal abaixo é exemplificada a utilização dos métodos `Imprime`, `Simetrico` e `Soma`. Inicialmente, é usado o construtor para criar o `Complexo z1` que é impresso no ecrã como *3+2i*. De seguida, é criado um novo `Complexo z2` que é o simétrico do `Complexo z1`. Este novo `Complexo` é depois impresso na forma *-3-2i*. Por fim, é criado um novo objeto `z3` do tipo `Complexo`, que resulta da soma de `z1` com `z2` e que é impresso no ecrã.

Programa Principal

```
#include "complexo.h"

int main(){
   Complexo z1(3, 2);
   z1.Imprime();
   Complexo z2 = z1.Simetrico();
   z2.Imprime();
   Complexo z3 = z1.Soma(z2);
   z3.Imprime();
   return 0;
}
```

Como a soma é comutativa, obteríamos o mesmo resultado no programa anterior fazendo `Complexo z3 = z2.Soma(z1);`. Uma função `Soma` mais intuitiva receberia dois números complexos e devolveria a sua soma, podendo ser chamada da seguinte forma `Complexo z3 = Soma(z1, z2);`. Isto é possível, criando funções globais, isto é, funções declaradas nos ficheiros da classe que não são membros da classe.

Ficheiro Cabeçalho

```
#ifndef COMPLEXO_H
#define COMPLEXO_H

//Colocar os includes necessários

class Complexo{
  //Classe complexo definida anteriormente sem a função soma
};

Complexo Soma(const Complexo&, const Complexo&);

#endif // COMPLEXO_H
```

Ficheiro Corpo

```
#include "complexo.h"

// ...
// Definições dos membros da classe
// ...

Complexo Soma(const Complexo& z1, const Complexo& z2){
    Complexo zSoma(z1.VerReal()+z2.VerReal(), z1.VerIm()+z2.VerIm());
    return zSoma;
}
```

Dado que a função `Soma` não é um membro da classe, a sua declaração é feita fora da classe, não sendo por isso necessário incluir o identificador "`Complexo::`"antes do seu nome no ficheiro corpo. Por esta mesma razão, o acesso aos atributos da classe (definidos como privados) tem que ser feito usando as funções *Ver*. A chamada da função `Soma` no programa principal é feita da seguinte forma:

```
Complexo z3 = Soma(z1, z2);.
```

Para terminar, considere-se uma nova classe cujo propósito é representar uma pessoa. Esta classe tem como atributos o nome da pessoa e a sua idade. Para além do construtor por omissão, a classe `Pessoa` tem um construtor com atributos e funções para aceder aos seus atributos.

Ficheiro Cabeçalho

```
#ifndef PESSOA_H
#define PESSOA_H

//Colocar os includes necessários

class Pessoa{
private:
    string Nome;
    int Idade;

public:
    //Construtores
    Pessoa();
    Pessoa(const string&, int);

    //Métodos
    const string& VerNome() const;
    int VerIdade() const;
};

#endif // PESSOA_H
```

Ficheiro Corpo

```
#include "pessoa.h"

Pessoa::Pessoa(): Nome(" "), Idade(-1){ }

Pessoa::Pessoa(const string& nome, int id):
   Nome(nome), Idade(id){ }

const string& Pessoa::VerNome() const{
    return Nome;
}

int Pessoa::VerIdade() const{
    return Idade;
}
```

Na análise deste exemplo vamos apenas focar as principais diferenças em relação aos exemplos anteriores. Comecemos por reparar na definição do método `VerNome`, mais especificamente no seu tipo de retorno. Este método devolve uma referência constante para uma *string*, uma vez que a função deve devolver o atributo `Nome` e não uma cópia dele (que seria o que aconteceria se o tipo de retorno fosse apenas *string* em vez da referência constante). Assim, os atributos de tipos não primitivos devem ser retornados como referências constantes, de forma a evitar cópias desnecessárias. Note-se que o atributo `Idade` sendo do tipo *int* (que é um tipo de dados primitivo) não necessita de ser devolvido na função `VerIdade` através de referências.

Para sumarizar, no contexto das classes, a palavra reservada *const* pode aparecer em três situações distintas:

i. nos argumentos dos métodos, sendo a sua função semelhante à que vimos no Capítulo 4;

ii. associada aos métodos, sendo o seu propósito indicar que o método não irá alterar os atributos da classe;

iii. no tipo de retorno de métodos que devolvam atributos da classe cujo tipo não seja um tipo de dados primitivo, de forma a retornar esses atributos sem que possam ser alterados e evitando a criação de cópias.

# Capítulo 8

# Sobrecarga de operadores

A maioria dos operadores que vimos no Capítulo 1 apenas estão definidos para tipos de dados primitivos. De facto, já usámos os operadores "+", "=", "==" e "<<" para, por exemplo, variáveis do tipo *int*. Como uma classe é um novo tipo de dados criado pelo utilizador, os operadores usuais não estão definidos para objetos dessas classes. Contudo, é possível definir "versões" dos operadores que permitam que eles funcionem quando aplicados a objetos de uma determinada classe. A isso chamamos sobrecarga de operadores.

Tomemos como exemplo a classe `Complexo` criada no capítulo anterior que inclui o método `Simetrico`. A utilização deste método fora da classe é feita com a seguinte instrução:

```
Complexo w = z.Simetrico();
```

sendo `z` um objeto do tipo `Complexo` e `w` o seu simétrico. Ora, o operador simétrico "-" já existe para tipos de dados numéricos, mas não para o tipo de dados `Complexo`, embora possa ser sobrecarregado para tal. Após fazer a sobrecarga do operador, será possível escrever

```
Complexo w = -z;
```

tornando assim o código mais legível e intuitivo. O mesmo acontece, por exemplo, para o método `Imprime`, que também integra a classe `Complexo`. A alternativa direta a este método é o operador de output `<<` que, após ser sobrecarregado, permite imprimir um objeto `Complexo` utilizando a segunda forma de escrita apresentada abaixo.

```
// Usando o método Imprime
    z.Imprime();

// Usando o operador <<
    cout << z;
```

Existem vários operadores que podem ser sobrecarregados em C++, sendo alguns deles apresentados na tabela seguinte.

| **Operadores Unários** | | **Operadores Binários** | |
|---|---|---|---|
| Incremento | `++` | Aritméticos Simples | `+, -, *, /,%` |
| Decremento | `--` | Aritméticos Compostos | `+=, -=, *=, /=, %=` |
| Simétrico | `-` | Relacionais | `<, >, <=, >=, !=, ==` |
| Negação | `!` | Escrita e Leitura | `<<, >>` |
| | | Acesso e Parêntesis | `[], ()` |

Os operadores podem ser unários ou binários. Os operadores unários têm um único operando, que será um objeto da classe. Já os operadores binários atuam sobre dois operandos, sendo pelo menos um deles um objeto da classe. Vejamos então como fazer a sobrecarga de alguns destes operadores tomando como exemplo a classe `Complexo`. É importante referir que, também no caso dos operadores, as declarações devem ser feitas no ficheiro cabeçalho e as definições no ficheiro corpo. No código abaixo é ilustrado um possível ficheiro cabeçalho da classe `Complexo` onde são declarados vários operadores.

Ficheiro Cabeçalho

```
//Colocar instruções para definir o ficheiro cabeçalho e includes necessários
class Complexo {
private:
    double Real;
    double Im;

public:
    Complexo( );
    Complexo(double , double);

    void AlterarReal(double);
    void AlterarIm(double);
    double VerReal( ) const;
    double VerIm( ) const;

    //Operador simétrico: para escrever -z
      Complexo operator-( ) const;
    //Operador soma/atribuição: para escrever z1+=z2
      Complexo& operator+=( const Complexo& );
    //Operador incremento (de prefixo): para escrever ++z
      Complexo& operator++( );
    //Operador incremento (de sufixo): para escrever z++
      Complexo operator++( int );
    //Operador parêntesis: para escrever z(a)
       double operator()( double a) const;
    //Operador negação: para escrever !z
       bool operator!( ) const;
};

 //Operador soma: para escrever z1+z2
    Complexo operator+( const Complexo& , const Complexo& );
 //Operador relacional: para escrever z1==z2
    bool operator==( const Complexo& , const Complexo& );
 //Operador de escrita: para escrever cout << z
    ostream& operator<<( ostream& , const Complexo& );
 //Operador de leitura: para escrever cin >> z
    istream& operator>>( istream& , Complexo& );
```

Note-se que alguns operadores são membros da classe e outros não. Existem operadores que podem ser definidos tanto fora como dentro da classe, sendo a forma como são declarados e definidos dependente da opção escolhida. Contudo, para não tornar este capítulo demasiado moroso, adotaremos a seguinte lógica:

- operadores aritméticos simples, relacionais, de escrita e de leitura serão definidos fora da classe;
- operadores aritméticos compostos, de incremento/decremento, simétrico, de negação, de acesso e parênteses serão definidos dentro da classe.

Vejamos então como definir cada um destes operadores no ficheiro corpo da classe.

## Operador simétrico

O simétrico de um número complexo é um número complexo e por isso o operador simétrico tem como tipo de retorno um objeto do tipo `Complexo`. Este operador está definido dentro da classe, pelo que é necessário usar o identificador `Complexo::`. Ao escrever `z1=-z2`, sendo `z1` e `z2` objetos do tipo `Complexo`, estamos a aplicar o operador "-" a `z2`. O resultado de `-z2` é guardado em `z1`, mas o valor de `z2` não é alterado. Assim sendo, ao implementar o operador simétrico, devemos criar um novo `Complexo` cuja parte real e imaginária são o simétrico da parte real e imaginária, respetivamente, do objeto ao qual foi aplicado o operador.

Ficheiro Corpo

```
Complexo Complexo::operator-( ) const{
    Complexo novo;
    novo.Real = -1*Real;
    novo.Im = -1*Im;
    return novo;

    //ou
    Complexo novo = Complexo( -1*Real, -1*Im );
    return novo;

    //ou simplesmente...
    return Complexo( -1*Real, -1*Im );
}
```

## Operador de soma/atribuição

Contrariamente ao operador simétrico, o operador aritmético composto da soma/atribuição requer dois operandos, sendo que um deles é alterado e o outro se mantém. Por exemplo, ao escrevermos `a+=b` estamos a adicionar `b` a `a` e, por isso, apenas o valor de `a` é alterado. Assim, este operador recebe como argumento uma referência constante para o objeto que não será alterado (operando `b`) e devolve uma referência (não constante) para o objeto que foi alterado (objeto `a`). A instrução `return *this;` significa "devolve uma referência para este", sendo "este" o objeto sobre o qual o operador foi aplicado, o objeto `a`.

Ficheiro Corpo

```
Complexo& Complexo::operator+=(const Complexo& b ){
     Real += b.VerReal();
     Im += b.VerIm();
     return *this;
}
```

A declaração e definição dos restantes operadores aritméticos compostos é semelhante à que foi aqui apresentada para o operador `+=` e por isso será omitida da sebenta. Note-se, no entanto, que o resto da divisão não é uma operação válida para números complexos uma vez que os seus atributos são do tipo *double*, pelo que a sobrecarga do operador `%=` não faz sentido para este tipo de dados.

## Operadores de incremento

Como vimos anteriormente, o operador de incremento pode ser usado como prefixo ou sufixo. Quando usado como prefixo (`++a`), o objeto é primeiro incrementado e depois devolvido. Quando usado como sufixo (`a++`), é primeiro feita uma cópia do objeto e apenas depois o objeto é incrementado, sendo por fim devolvida a cópia do objeto (que não foi incrementada). Conforme ilustrado no código abaixo, o que distingue a declaração dos operadores de incremento é o argumento *int* no operador sufixo. Note-se que o argumento *int* apenas serve para distinguir qual o operador que queremos sobrecarregar e não para indicar que o operador de sufixo necessita de um argumento do tipo *int*.

Contrariamente ao que acontece com outros tipos de dados, o significado dos operadores de incremento para objetos do tipo `Complexo` não é claro e poderá nem fazer sentido. Contudo, para explicar como deve ser feita a declaração e definição destes operadores, consideramos que eles aumentam em uma unidade a parte real e a parte imaginária do `Complexo`.

Ficheiro Corpo

```
//Operador de incremento prefixo (++a)
    Complexo& Complexo::operator++( ){
         ++Real;
         ++Im;
         return *this;
    }

//Operador de incremento de sufixo (a++)
    Complexo Complexo::operator++( int ){
         Complexo aux(Real,Im);
          ++Real;
          ++Im;
         return aux;
    }
```

A implementação dos operadores de decremento é semelhante e por isso não será apresentada na sebenta.

## Operadores aritméticos simples

Os operadores aritméticos efetuam uma operação aritmética entre dois objetos da classe, que recebem como argumento, e devolvem um novo objeto da classe. Uma vez que estes operadores não alteram os objetos que recebem como argumento, tais objetos devem ser passados por referência constante. É importante ainda notar que, sendo estes operadores definidos fora da classe, o identificador `Complexo::` deixa de ser necessário. No código abaixo, é definido o operador soma, sendo a definição dos restantes operadores aritméticos simples semelhante.

Ficheiro Corpo

```
Complexo operator+( const Complexo& z1, const Complexo& z2 ){
     double novo_real = z1.VerReal() + z2.VerReal();
     double novo_im = z1.VerIm() + z2.VerIm();
     Complexo aux( novo_real, novo_im );
     return aux;

      // ou simplesmente...
      return Complexo( z1.VerReal()+z2.VerReal(), z1.VerIm()+z2.VerIm() );
}
```

## Operadores relacionais

Os operadores relacionais permitem comparar dois objetos da classe e devolvem um resultado do tipo *bool*. Tal como os operadores aritméticos simples, também os relacionais não alteram os objetos sobre os quais operam. Assim, esses objetos devem ser passados por referência constante. No código abaixo é definido o operador `==` para objetos do tipo `Complexo`. A definição dos restantes operadores relacionais seria feita de forma semelhante. No entanto, é importante referir que os operadores `<`, `>`, `<=` e `>=` não têm um significado claro para objetos do tipo `Complexo`.

Ficheiro Corpo

```
bool operator==( const Complexo& z1, const Complexo& z2 ){
     if ( z1.VerReal()==z2.VerReal() && z1.VerIm()==z2.VerIm() )
         return true;
     else
         return false;
}
```

Para avaliar se dois objetos do tipo `Complexo` são diferentes, não podemos usar a instrução `z1 != z2`, pois não sobrecarregámos o operador `!=`. Contudo, como o resultado desse operador é a negação do resultado do operador `==`, poderíamos usar o operador `==` para fazer essa verificação da seguinte forma: `!(z1 == z2)`.

## Operador de escrita

O operador de escrita (`<<`) é um operador binário que recebe dois argumentos, nomeadamente uma referência para um objeto `ostream` e uma referência constante para um objeto da classe, e devolve uma

referência para um objeto `ostream`. O `ostream` é uma classe da biblioteca *standard* e significa *output stream*. Esta classe permite escrever e formatar sequências de carateres. Note-se que a definição do operador de escrita é bastante semelhante à definição da função `Imprime`, sendo a única diferença a utilização do objeto do tipo `ostream` em vez do tradicional *cout*.

Ficheiro Corpo

```
ostream& operator<<( ostream& output, const Complexo& z){
     if ( z.VerIm() >= 0)
          output << z.VerReal() << "+" << z.VerIm() << "i";
     else
          output << z.VerReal() << z.VerIm() << "i";

    return output;
}
```

Tendo em conta o que aprendemos sobre funções, a questão que se coloca é: porque é que o operador `<<` não é *void* dado que recebe um argumento como referência? A resposta é simples, o operador `<<` devolve uma referência para que possamos encadear instruções, isto é, para podermos fazer, por exemplo, `cout << z << endl;`, sendo `z` um objeto do tipo `Complexo`. Se o tipo de retorno do operador fosse *void* ficaríamos com `void << endl;`, sendo isto algo que o computador não sabe interpretar. Ao devolvermos a referência do objeto *ostream* ficamos com `cout << endl;`, algo que o computador já conhece.

## Operador de leitura

O operador de leitura (`>>`) é também um operador binário que recebe dois argumentos, nomeadamente uma referência para um objeto `istream` e uma referência para um objeto da classe, e devolve uma referência para um objeto `istream`. O `istream` é uma classe da biblioteca *standard* e significa *input stream*. Esta classe permite ler sequências de carateres. No código abaixo, é definido o operador de leitura para objetos do tipo `Complexo`, assumindo que esses objetos são introduzidos pelo utilizador na forma $a \pm bi$. De acordo com esse formato, o utilizador deve começar por inserir a parte real do número complexo, que é guardada na variável `a`. De seguida, introduz um carater que se espera que seja ou o sinal `+` ou o sinal `-`, e que fica armazenado em `c1`. Por fim, é inserida a parte imaginária do número complexo (armazenada na variável `b`) e um caracter, que se espera que seja '$i$' e que é armazenado em `c2`.

Ficheiro Corpo

```
istream& operator>>( istream& input, Complexo& z){
      char c1, c2;
      double a, b;
      input >> a >> c1 >> b >> c2;
      if( c1 == '-')
          b = -b;

     if( !input || (c1!='-' && c1!='+') || c2!='i' )
         //lançar exceção

      z.AlterarReal(a);
      z.AlterarIm(b);
      return input;
}
```

A declaração do operador de leitura >> é bastante semelhante à do operador de escrita <<, as únicas diferenças são a utilização da classe *istream* em vez da *ostream* e o objeto `Complexo` ser passado como referência em vez de referência constante. Foquemos-nos na parte da passagem por referência. Isto acontece porque o operador de leitura vai modificar o objeto do tipo `Complexo`, uma vez que altera os valores da sua parte real e imaginária para os valores introduzidos pelo utilizador.

## Operador Parêntesis

O operador parêntesis `()` pode receber um qualquer número de argumentos e retornar um qualquer tipo de dados. Esta sua flexibilidade permite que seja usado em muitas situações, sendo duas delas ilustradas no exemplo abaixo para o classe `Complexo` considerada anteriormente.

Ficheiro Corpo

```
double Complexo::operator()( double a) const{
      return Real*a + Im;
}

Complexo Complexo::operator()( double a, double b ) const{
      return Complexo( Real*a , Im*b );
}
```

No primeiro exemplo, que corresponde ao apresentado no ficheiro cabeçalho, o operador `()` recebe apenas um argumento do tipo *double* e devolve o resultado obtido através da soma da multiplicação da sua parte real pelo valor recebido com a parte imaginária do objeto do tipo `Complexo`, sendo por isso o tipo de retorno *double*. No segundo caso, o operador `()` é usado para calcular um novo objeto do tipo `Complexo` que terá como partes real e imaginária os valores do objeto original multiplicados por constantes reais. Assim, recebe dois *doubles* que são multiplicados pelas partes real e imaginária do `Complexo` original, dando origem a um novo `Complexo`.

## Operador de negação

O operador de negação `!` não recebe qualquer argumento e tem como tipo de retorno o tipo *bool*. No exemplo abaixo, é definido operador de negação da classe `Complexo`.

Ficheiro Corpo

```
bool Complexo::operator!( ) const{
      if (Real == 0 and Im == 0 )
            return true;
      else
            return false;
}
```

No exemplo apresentado, o operador de negação é usado para indicar se um `Complexo` é ou não nulo, ou seja, se tem as suas partes real e imaginária iguais a zero.

Após a sobrecarga dos operadores anteriores, a sua utilização pode ser feita como é ilustrado abaixo.

Programa Principal

```
#include "complexo.h"

int main(){
   Complexo z1(3,2), z2;
   cout << "z2: ";
   cin >> z2;                                    //Operador de leitura

   Complexo z3 = z1 + z2;                        //Operador soma
   cout << z1 << " + " << z2 << " = " << z3;     //Operador de escrita
   ++z1;                                         //Operador de incremento
   Complexo z4 = -z1;                            //Operador simétrico
   z4 += z2;                                     //Operador soma e atribuição

   if(z1 == z2)                                  //Operador de igualdade
        cout << "Sao iguais";
   else
        cout << "Sao diferentes";

    double a = z1(2);                             //Operador parêntesis exemplo 1
    Complexo z5 = z2(2, 3);                       //Operador parêntesis exemplo 2

    if ( !z1 )                                    //Operador negação
        cout << "Complexo nulo";
    else
        cout << "Complexo nao nulo";

   return 0;
}
```

## Operador de acesso

Habitualmente, usamos o operador `[]` para aceder a elementos de vetores e, por isso, a sobrecarga deste operador é particularmente útil quando na classe existe um atributo do tipo *vector*. Para ilustrar a sobrecarga do operador de acesso, consideremos então uma classe fictícia `VetorLP` que tem como atributo privado um vetor de *strings* e cujo ficheiro cabeçalho é mostrado de seguida.

Ficheiro Cabeçalho

```
#ifndef VETORLP_H
#define VETORLP_H
//Colocar os includes necessários

  class VetorLP {
  private:
      vector<string> V;

  public:
      //Construtor e outros membros públicos

      //Operador acesso - versão não constante
        string& operator[]( int );

      //Operador acesso - versão constante
        const string& operator[]( int ) const;
  };

#endif // VETORLP_H
```

Tal como já foi referido, a sobrecarga do operador `[]` é feita dentro da classe. Nessa sobrecarga, são usualmente consideradas duas suas versões: a versão constante e a versão não constante. Comecemos por analisar a versão não constante. Esta versão recebe como argumento um valor inteiro, que corresponde a uma posição do vetor, e devolve uma referência para o elemento do vetor que está nessa posição. Assim sendo, esta versão do operador permite a alteração dos elementos do vetor, sendo a sua definição feita como indicado abaixo.

```
//versão não constante
string& VetorLP::operator[]( int i ){
   return V[i];
}
```

É importante reforçar que o uso de referência no tipo de retorno desta versão do operador não está relacionado com o facto de estarmos a retornar uma *string* (tipo não primitivo), mas sim com o facto de querermos que o objeto devolvido possa ser alterado. Assim sendo, esta versão do operador devolve sempre uma referência, mesmo que o tipo de retorno seja um tipo de dados primitivo.

O operador `[]`, após sobrecarregado, é muitas vezes usado na definição de outros métodos/operadores. Alguns desses métodos recebem como argumento objetos constantes, isto é, referências constantes para

objetos da classe. Objetos constantes só podem ser manipulados por métodos/operadores também constantes e, por isso, tais objetos não podem ser manipulados pela versão não constante do operador `[]`. Para contornar esta situação, é necessário implementar a versão constante do operador `[]`. Essa versão - apresentada abaixo - devolve uma referência constante para um elemento do vetor, pelo que não permite a alteração desse elemento.

```
//versão constante
const string& VetorLP::operator[]( int i ) const{
   return V[i];
}
```

Por fim, é importante referir que não é necessário fazer na implementação das versões do operador `[]` qualquer validação do argumento `i`, uma vez que esse operador não faz essa validação, tal como vimos no caso dos vetores. Note-se que, na implementação de um operador, devemos sempre manter as suas propriedades originais.

### Operador de acesso para matrizes

Como já vimos, uma matriz é um vetor de vetores, isto é, um vetor em que cada um dos seus elementos é um vetor. Por exemplo, a matriz

$$m = \begin{bmatrix} 5 & 2 & 1 \\ 8 & 0 & 6 \\ 7 & 2 & 3 \\ 1 & 0 & 1 \end{bmatrix}$$

pode ser definida em C++ como

```
vector<vector<int>> m = { {5, 2, 1}, {8, 0, 6}, {7, 2, 3}, {1, 0, 1} }
```

Significa isto que `m` é na verdade um vetor "principal" com quatro elementos (`m[0] = {5, 2, 1}`, `m[1] = {8, 0, 6}`, `m[2] = {7, 2, 3}` e `m[3] = {1, 0, 1}`) sendo cada um desses elementos um vetor "secundário" de dimensão 3. A sobrecarga do operador `[]` para um objeto de uma classe que tenha como atributo uma matriz apenas permite aceder ao vetor principal dessa matriz, pelo que o operador terá como tipo de retorno um vetor de elementos com o mesmo tipo que o vetor secundário, tal como é ilustrado no próximo exemplo.

Nos ficheiros cabeçalho e corpo do exemplo, é criada uma classe `XPTO` que tem como atributo uma matriz de *doubles* e é definido o operador de acesso `[]` na sua versão constante e não constante. A definição desse operador permite aceder diretamente ao atributo `Matriz` de um objeto do tipo `XPTO`, tal como é ilustrado na definição do operador de escrita. Note-se que `x` é um objeto do tipo `XPTO` e não um vetor ou uma matriz, pelo que, sem a sobrecarga do operador `[]`, não seria possível escrever algo como `x[i]`. Contudo, após a definição do operador de acesso, passa a ser possível escrever `x[i]` e, consequentemente, `x[i][j]`. Ao escrever `x[i][j]`, é primeiramente chamado o operador `[]` definido na classe, o qual devolve um vetor de *doubles*, isto é, devolve `x[i]`. Sendo `x[i]` um vetor de *doubles*, o acesso aos seus elementos pode ser feito diretamente usando o operador de acesso `[]` já predefinido na biblioteca *standard* para vetores e que permite então obter o elemento que está na posição $j$ do vetor `x[i]`, isto é, `x[i][j]`. Assim sendo, não é necessária a sua implementação.

Ficheiro Cabeçalho

```
#ifndef XPTO_H
#define XPTO_H
//Colocar os includes necessários

  class XPTO {
  private:
      vector<vector<double>> Matriz;

  public:
      //Construtor e outros membros públicos

      //Operador acesso - versão não constante
        vector<double>& operator[]( int );

      //Operador acesso - versão constante
        const vector<double>&  operator[]( int ) const;
  };

  ostream& operator<<( ostream& , const XPTO& );

#endif // XPTO_H
```

Ficheiro Corpo

```
//Operador acesso - versão não constante
    vector<double>& XPTO::operator[]( int i ){
        return Matriz[i];
    }

//Operador acesso - versão constante
    const vector<double>& XPTO::operator[]( int i ) const{
        return Matriz[i];
    }

//Operador de escrita
    ostream& operator<<( ostream& output, const XPTO& x){
        for(int i = 0; i < x.N_linhas(); ++i){   //Definir método N_linhas()
            for(int j = 0; j < x[i].size(); ++j){
                output << x[i][j] << " ";
            }
            output << endl;
        }
        return output;
    }
```

# Capítulo 9

# Herança e polimorfismo

A *herança* é a capacidade de criar novas classes (classes derivadas ou filhas) a partir de classes já existentes (classes base ou mãe), sendo os membros das classes mãe *herdados* pelas classes derivadas. Numa classe mãe podemos ter membros públicos (*public*), privados (*private*) ou protegidos (*protected*). Membros protegidos, tal como membros privados, não podem ser acedidos fora da classe onde foram definidos. A diferença entre membros privados e protegidos apenas existe no contexto da herança: as classes derivadas têm acesso aos membros protegidos da classe mãe, mas não aos seus membros privados.

A indicação de que uma classe (classe derivada) herda de outra classe (classe mãe) é feita da seguinte forma:

Ficheiro Cabeçalho

```
#ifndef CLASSE_DERIVADA_H
#define CLASSE_DERIVADA_H

class  Classe_Derivada: tipo_de_acesso Classe_Mae {
    //...
};

#endif // CLASSE_DERIVADA_H
```

O `tipo_de_acesso` define o acesso aos membros da classe mãe e pode ser *public*, *private* ou *protected*. Membros privados da classe mãe nunca podem ser acedidos pelas classes derivadas, independentemente do tipo de acesso usado. Assim sendo, a diferença entre os três tipos de acesso diz apenas respeito aos membros públicos e protegidos da classe mãe:

i) *private*: a classe derivada herda todos os membros públicos e protegidos da classe mãe, mas esses membros são definidos como privados na classe derivada.

ii) *protected*: a classe derivada herda todos os membros públicos e protegidos da classe mãe, mas esses membros são definidos como protegidos na classe derivada.

iii) *public*: a classe derivada herda todos os membros públicos e protegidos da classe mãe e o seu tipo de acesso não é alterado. Ou seja, membros públicos da classe mãe são também públicos na classe derivada e membros protegidos da classe mãe são também protegidos na classe derivada.

Tomemos como exemplo uma classe `Poligono` para representar e manipular um polígono. Como existem caraterísticas comuns a todos os polígonos, podemos implementar uma classe mãe que represente

essas caraterísticas. Por exemplo, todos os polígonos podem ser definidos através de um vetor que contenha o comprimento de cada um dos seus lados, pelo que esse vetor pode ser o único atributo da classe. O cálculo do perímetro de um polígono corresponde à soma de todos os seus lados, sendo também independente do polígono em causa. O mesmo não acontece com a área, cujo cálculo depende do tipo de polígono. A classe mãe `Poligono` poderia então ser implementada como mostrado abaixo.

Ficheiro Cabeçalho

```
#ifndef POLIGONO_H
#define POLIGONO_H
//Colocar os includes necessários

class Poligono{

protected:
    //atributos
    vector<double> Lados;

public:
    //Construtor
    Poligono(const vector<double>& );

    //Métodos
    double Perimetro() const;
    double Area() const;
};

#endif // POLIGONO_H
```

Ficheiro Corpo

```
#include "poligono.h"

Poligono::Poligono(const vector<double>& v){
   Lados = v;
}

double Poligono::Perimetro() const{
    double p = 0;
    for( int i = 0; i<Lados.size(); ++i){
        p += Lados[i];
    }
    return p;
}

double Poligono::Area() const{
    throw runtime_error("ERRO!");
}
```

Suponhamos que queremos criar duas novas classes para representar triângulos e quadrados. Quer o triângulo quer o quadrado são polígonos e, por isso, partilham as caraterísticas da classe `Poligono`. Estas caraterísticas podem ser *herdadas* da classe mãe `Poligono` em vez de voltarem a ser definidas nas classes `Triangulo` e `Quadrado`, evitando assim repetições de código. Além dos membros herdados da classe mãe, as classes derivadas podem ainda incluir outros membros específicos.

Vejamos então a declaração e a definição destas duas classes. Para simplificar, apresentamos apenas o ficheiro cabeçalho das classes, onde incluímos as definições (que deveríamos incluir no ficheiro corpo). Note-se que, o cálculo da área está bem definido para um qualquer quadrado e triângulo, pelo que as respetivas classes podem conter um método para o fazer.

Ficheiro Cabeçalho

```
#ifndef QUADRADO_H
#define QUADRADO_H
//Colocar os includes necessários

class Quadrado: public Poligono{

public:
    //Construtor
    Quadrado(double x): Poligono(vector<double>(4,x)){ }

    //Métodos
    double Area() const{ return Lados[0]*Lados[0]; }
};

#endif // QUADRADO_H
```

Ficheiro Cabeçalho

```
#ifndef TRIANGULO_H
#define TRIANGULO_H
//Colocar os includes necessários

class Triangulo: public Poligono{

private:
    double Base;
    double Altura;

public:
    //Construtor
    Triangulo(double a, double b, double c): Poligono({a,b,c}){
        Base = a;
        double s = (a+b+c)/2; //fórmula de Heron
        Altura = 2*sqrt(s*(s-a)*(s-b)*(s-c))/Base;
    }

    //Métodos
    double Area() const{ return Base*Altura/2; }
};

#endif // TRIANGULO_H
```

Ambas as classes herdam publicamente da classe `Poligono` e têm definido um método específico para calcular a área. A classe `Quadrado` apresentada, não inclui atributos extra. Nesta classe, existe um construtor que recebe como argumento a medida do lado do quadrado. O construtor de uma classe

derivada é sempre definido através do construtor da classe mãe. Ora, o construtor da classe mãe recebe como argumento um vetor, pelo que é necessário criar um vetor com quatro posições sendo o valor de cada uma delas igual ao lado do quadrado ($x$). A classe `Triangulo` tem dois atributos específicos além dos atributos gerais herdados da classe `Poligono`. Assim, qualquer objeto do tipo `Triângulo` terá três atributos: a base e a altura definidas na classe `Triângulo` e o vetor com as medidas dos lados herdado da classe `Poligono`. O construtor da classe `Triangulo` recebe as três medidas dos lados do triângulo e com elas preenche os seus atributos específicos (base e altura) e o vetor *Lados* através do construtor da classe `Poligono`.

Apesar do método *Perimetro* não estar implicitamente declarado nestas duas classes, é herdado da classe `Poligono`, pelo que pode ser usado pelos objetos do tipo `Quadrado` e do tipo `Triangulo`, conforme ilustrado no exemplo abaixo

```
//includes necessários

int main(){
    Poligono P( {3, 1, 3, 5, 7} );
    cout << "Perimetro: " << P.Perimetro();

    Quadrado Q(3);
    cout << "\nPerimetro: " << Q.Perimetro();
    cout << "\nArea: " << Q.Area();

    Triangulo T(3, 4, 5);
    cout << "\nPerimetro: " << T.Perimetro();
    cout << "\nArea: " << T.Area();

    return 0;
}
```

sendo obtido o output esperado, isto é,

```
PerimetroP: 19
PerimetroQ: 12
AreaQ: 9
PerimetroT: 12
AreaT: 6
```

Consideremos agora o programa abaixo. Neste programa é criada uma função global $f$ que recebe como argumento uma referência constante para um `Poligono`. Ora, um `Quadrado` e um `Triangulo`, sendo classes derivadas da classe `Poligono`, são também do tipo `Poligono` e, por isso, podem também ser argumentos da função $f$. A chamada do método *Perimetro*, por ser definido na classe `Poligono`, não levanta qualquer problema, independentemente da função $f$ receber um `Poligono`, um `Quadrado` ou um `Triangulo`. Contudo, o mesmo não acontece com o método *Area*. Apesar das classes `Quadrado` e `Triangulo` terem o seu próprio método *Area*, ao passar um objeto de um desses tipos para a função $f$, o método *Area* chamado será sempre o da classe *Poligono*, sendo por isso lançada uma exceção, que é o que o método *Area* da classe `Poligono` faz.

```
//includes necessários

void f(const Poligono& P){
     cout << "\nPerimetro: " << P.Perimetro();
     cout << "\nArea: " << P.Area();
}

int main(){
    Quadrado Q(3);
    f(Q);
    Triangulo T(3, 4, 5);
    f(T);
}
```

O que se pretende seria que para objetos do tipo `Quadrado` e `Triangulo` fosse chamado o método *Area* definido nas respetivas classes. Para que isso aconteça, o método *Area* na classe `Poligono` deve ser declarado como um método *virtual* ou *puramente virtual*. Isto é, como um método que será redefinido nas classes derivadas. Ao chamar um método virtual para um objeto da classe derivada através da classe mãe, será usado o método da classe derivada (se existir) em vez do da classe mãe. A declaração e definição do método virtual ou puramente virtual *Area* no ficheiro cabeçalho da classe `Poligono` pode ser feita como apresentado abaixo, sendo que nenhuma alteração precisa de ser feita nas classes derivadas.

```
//Declaração como método virtual
    virtual double Area() const { };

//Declaração como método puramente virtual
    virtual double Area() const = 0;
```

A principal diferença entre métodos virtuais e puramente virtuais é a possibilidade de criar objetos da classe em que são definidos. Caso o método *Area* seja declarado como um método puramente virtual, deixa de ser possível criar objetos do tipo `Poligono`, sendo apenas possível a criação de objetos do tipo das classes derivadas. Isto é, se a função *Area* for puramente virtual temos:

```
Poligono P({1, 4, 7, 3})  //ERRO!
Quadrado Q(3)             //OK
Triangulo T(3, 4, 5)      //OK
```

O mesmo não acontece com um método que seja apenas virtual. Neste caso, a criação de objetos do tipo `Poligono` é também possível. Além disso, um método puramente virtual tem que estar implementado em todas as classes derivadas enquanto que um método virtual não.

A definição do método *Area* como virtual ou puramente virtual, permite que a função $f$ anterior funcione corretamente, isto é, que o `Poligono` recebido como argumento seja visto como um `Quadrado` quando a função é chamada com um `Quadrado` e que esse `Poligono` seja visto como um `Triangulo` quando a função é chamada com um `Triangulo`. A esta capacidade de um objeto se comportar como se fosse de outro tipo chama-se *polimorfismo*.

# Capítulo 10

# Escrita e leitura de ficheiros

A utilização de ficheiros é essencial para importar e exportar grandes quantidades de informação de um programa. O pacote `fstream` da biblioteca *standard* do C++ - cujo significado é *file stream* - contém métodos para manipular ficheiros, pelo que é necessária a sua inclusão no preâmbulo através da instrução

```
#include <fstream>
```

Este pacote contém as classes `ofstream` e `ifstream` que permitem, respetivamente, a escrita e a leitura de ficheiros, sendo o seu significado *output file stream* e *input file stream*, respetivamente.

## 10.1 Escrita de ficheiros

Para escrever um ficheiro, é necessário criar um objeto da classe `ofstream`, que cria um canal para enviar informação para um ficheiro. Para tal, devemos usar a instrução:

```
ofstream nome(Caminho, modo_de_abertura);
```

ou as instruções:

```
ofstream nome;
nome.open(Caminho, modo_de_abertura);
```

onde `nome` é o nome do objeto `ofstream` que queremos associar ao ficheiro, `Caminho` é a localização do ficheiro onde se pretende escrever, incluindo o seu nome, e `modo_de_abertura` é a opção que indica o que queremos fazer com o conteúdo já existente no ficheiro (caso exista). Para o `modo_de_abertura` existem duas opções: `ios_base::out` e `ios_base::app`. Ambas as opções criam um ficheiro caso ainda não exista, no entanto, no caso do ficheiro já existir, a primeira opção apaga o seu conteúdo, enquanto que com a segunda o conteúdo do ficheiro é preservado, sendo a nova informação adicionada no final do ficheiro. Se o modo de abertura não for especificado, é assumido o modo `ios_base::out`.

O `Caminho` é uma *string* que indica o caminho completo[1] até ao ficheiro e que inclui o seu nome.

[1]Para aceder à localização de um ficheiro em Windows, devemos ir à pasta onde está o ficheiro, carregar no ficheiro com o botão direito do rato e selecionar *Propriedades*. De seguida basta copiar o caminho completo que lá aparece substituindo cada barra de separação ("\") por duas barras ("\\"). Em Mac, é usado como separador a barra invertida ("/") em vez das duas barras.

Quando no `Caminho` apenas é indicado o nome do ficheiro, é assumida a localização *default*, que é a pasta *build* do projeto.

Após criar o objeto *ofstream* `nome`, devemos verificar se o ficheiro foi aberto com sucesso. Para tal, podemos usar o método *is_open* (`nome.is_open()`) ou o operador de negação ( `!nome`). Tendo a garantia de que o ficheiro está aberto, podemos então escrever no ficheiro usando a variável `nome`. O processo de escrita no ficheiro é semelhante ao da escrita para o ecrã, mas em vez de usar a instrução *cout* usamos a variável `nome`. Vejamos o exemplo abaixo.

```
#include <iostream>
#include <fstream>
using namespace std;

int main(){
    ofstream file("C:\\Users\\A\\Novo.txt", ios_base::out);

    if(!file)   //ou  if( !file.is_open() )
         throw runtime_error("ERRO: Nao foi possivel abir o ficheiro.");

    file << "Texto";

    file.close();
    return 0;
}
```

Neste exemplo é criado um objeto `ofstream` com o nome `file` para enviar informação para o ficheiro com o nome *Novo.txt* e cuja localização é *"C:\Users\A"*. O modo de abertura é `ios_base::out`, pelo que todo o conteúdo que existir no ficheiro - se existir - é apagado. De seguida, verificamos se o ficheiro foi aberto com sucesso e caso não tenha sido lançamos uma exceção. Caso não seja lançada a exceção - o que significa que o ficheiro foi aberto com sucesso - é então escrita no ficheiro a palavra *"Texto"*. Por fim, o canal de escrita para o ficheiro é fechado através do método *close()*.

## 10.2 Leitura de ficheiros

Para ler um ficheiro, é necessário criar um objeto da classe `ifstream`, que cria um canal de informação para ler de um ficheiro. Para tal, devemos usar a instrução:

```
ifstream nome(Caminho)
```

onde, tal como anteriormente, `nome` é o nome do objeto `ifstream` e `Caminho` é a localização do ficheiro que se pretende ler (incluindo o seu nome). Após criar o objeto *ifstream* `nome`, devemos também verificar se o ficheiro foi aberto com sucesso, tal como no caso da escrita de ficheiros. A leitura do conteúdo do ficheiro é semelhante à leitura de informação do ecrã, mas usamos a variável `nome`, em vez da instrução *cin*. Vejamos o exemplo abaixo.

```cpp
#include <iostream>
#include <fstream>
using namespace std;

int main(){
    ifstream file;
    file.open("Dados.txt");
    //ou simplesmente: ifstream file("Dados.txt");

    if(!file)   //ou  if( !file.is_open() )
         throw runtime_error("ERRO: Nao foi possivel abir o ficheiro.");

    string s;
    file >> s;

    file.close();
    return 0;
}
```

Neste exemplo é criado um objeto `ifstream` com o nome `file` que irá ler informação do ficheiro com o nome *Dados.txt* localizado na pasta *build* do projeto, dado que não foi especificada uma localização. De seguida, verificamos se o ficheiro foi aberto com sucesso e, em caso afirmativo, é lida a primeira sequência de carateres do ficheiro, que é guardada na variável `s`. Por fim, o canal de informação para ler o ficheiro é fechado através do método *close()*. Note-se que apenas a primeira sequência de carateres do ficheiro é lida uma vez que a leitura de uma *string* termina após ser encontrado um espaço ou uma mudança de linha. Por exemplo, se o conteúdo do ficheiro *Dados.txt* for

```
Dados.txt
Amanha vai chover muito.
Hoje nao chove.
```

apenas a palavra *Amanha* é lida e guardada na variável `s`. Para ler uma linha completa, ou seja, ler até encontrar uma mudança de linha, podemos usar a função *getline* que tem dois ou três argumentos. No caso de três argumentos, a sua sintaxe é a seguinte:

```cpp
getline(InputStream, guardaInformacao, delimitador)
```

onde `InputStream` é um objeto do tipo `ifstream` (por exemplo, o *cin*) que é usado para indicar o canal de onde será lida a informação (consola, ficheiro, etc.). O segundo argumento é a variável *string* (chamada `guardaInformacao`) onde queremos guardar a informação lida. O terceiro argumento é o caráter delimitador do tipo *char* que indica até onde deve ser lida a *string*. Se o caráter especificado não existir na linha que se pretende ler, a leitura continua para as linhas seguintes até que o caráter delimitador seja encontrado ou, no caso de ficheiros, até que se chegue ao final do ficheiro. Na sintaxe da função *getline* com dois argumentos, é removido o último argumento (caráter delimitador), que por defeito se assume ser a mudança de linha. Vejamos o exemplo abaixo:

```cpp
int main(){
    ifstream file("Dados.txt");
    if(!file)
        throw runtime_error("ERRO: Nao foi possivel abir o ficheiro.");

    string s1;
    getline(file, s1, 'i');

    string s2;
    getline(file, s2);

    file.close();

    string s3, s4;
    getline(cin, s3);
    getline(cin, s4, '/');
    return 0;
}
```

O conteúdo da *string* `s1` é lido do ficheiro *Dados.txt*. Assim, o primeiro *getline* irá ler a primeira linha do ficheiro até encontrar o caráter `'i'`, ou seja, lê *"Amanha va"*, sendo este o conteúdo de `s1`. O segundo *getline* também irá ler do ficheiro *Dados.txt*. A leitura do segundo *getline* começa onde terminou a leitura anterior, isto é, após o primeiro caráter `'i'` na primeira linha. Uma vez que neste caso não é especificado o caráter delimitador, a leitura terminará no final da primeira linha, pelo que teremos `s2 =` *" chover muito."*.

Nos dois últimos *getlines* é usado o *cin*, pelo que o conteúdo das *strings* `s3` e `s4` será lido do ecrã. A leitura da *string* `s3` termina quando for inserida uma mudança de linha. De seguida, começa a leitura da *string* `s4`, onde será armazenada toda a informação introduzida antes do caráter `'/'`.

Para ler toda a informação de um ficheiro podemos usar um ciclo *while* semelhante ao que usamos para pedir sucessivamente valores ao utilizador. A instrução `while(getline(file, s))` pode ser lida como "enquanto houver linhas no ficheiro para ler".

```cpp
//ler linhas sucessivamente

int main(){
    ifstream file("Dados.txt");

    string s;
    while( getline(file, s) ){
        //...
    }

    file.close();
    return 0;
}
```

```cpp
//ler palavras sucessivamente

int main(){
    ifstream file("Dados.txt");

    string s;
    while( file >> s ){
        //...
    }

    file.close();
    return 0;
}
```

## 10.3 Instruções `clear()` e `ignore()`

As instruções `clear()` e `ignore()` servem para manipular canais de leitura, sendo muito importantes para garantir o bom funcionamento do programa. O *cin* é um canal de leitura do ecrã de onde é extraída informação e tem dois estados possíveis: *"com erro"* e *"sem erro"*. Um dos motivos que pode levar a que o canal *cin* fique com erro é a leitura de um tipo de dados diferente do da variável que o irá armazenar. Uma vez com erro, o canal de leitura não voltará ao estado *"sem erro"* enquanto o programa não for executado novamente ou enquanto o canal não for "limpo", o que impossibilitará a utilização do canal no resto da execução do programa. A instrução `cin.clear()` tem como finalidade "limpar" o canal, restaurando o seu estado *sem erro* e permitindo assim que este continue a ser usado no decorrer do programa. Consideremos o exemplo abaixo.

```
int n1;
int n2;
string s;

cout << "Primeiro numero: ";
cin >> n1;



cout << "Segundo numero: ";
cin >> n2;

cout << "Texto: ";
cin >> s;

cout << n1 << " "<< n2 << " " << s;
```

```
int n1;
int n2;
string s;

cout << "Primeiro numero: ";
cin >> n1;
cin.clear();
cin.ignore(10000, '\n');

cout << "Segundo numero: ";
cin >> n2;

cout << "Texto: ";
cin >> s;

cout << n1 << " " << n2 << " " << s;
```

Consideremos primeiro o código do lado esquerdo. Suponhamos que aquando da leitura da variável `n1` (numérica) o utilizador introduz um caráter não numérico. Neste caso, a variável `n1` ficará com um *valor lixo* e o canal de leitura ficará *com erro*, não sendo por isso efetuada a leitura da segunda e terceira variáveis (que ficarão também com um *valor lixo*).

No código da direita foram adicionadas as instruções `cin.clear()` e `cin.ignore(10000, '\n')` ao programa. Comecemos por analisar o que acontece se apenas tivermos a instrução `cin.clear()`. Ao introduzir um valor não numérico, por exemplo `'g'`, aquando da leitura de `n1`, o canal do *cin* passará a estar com erro. Ao chegar à instrução `cin.clear()`, o programa altera o estado do canal de leitura *cin* para *sem erro*, pelo que pode ser lida novamente informação a partir desse canal. Ora, a informação que neste momento existe no canal é o caráter que foi introduzido aquando da leitura de `n1` e uma mudança de linha acrescentada automaticamente no momento em que se carregou na tecla *enter* após inserir `g` na consola, isto é, "g\n". Uma vez que existe ainda essa informação no canal pois não foi lida para nenhuma variável, o utilizador não terá possibilidade de introduzir novos valores no canal. Significa isto que o programa irá tentar associar o conteúdo já existente no canal à variável `n2`, causando novamente o problema anterior. Note-se que no caso da leitura da variável `s` (do tipo *string*) ser feita primeiro do que a leitura da variável `n2` não iria existir qualquer problema com o canal de leitura pois a informação nele contida seria associada à variável `s`, isto é, teríamos `s=''g''` e o utilizador teria a possibilidade de

introduzir um novo valor para `n2`.

Podemos então concluir que alterar apenas o estado do canal de leitura para *sem erro* pode não ser suficiente para resolver problemas de leitura uma vez que a informação existente no canal não é removida até que seja guardada numa variável. Para apagar todo o conteúdo existente no canal de leitura, devemos usar a instrução `cin.ignore(10000, '\n')`. Esta instrução tem como objetivo apagar todos os carateres (até ao máximo de 10000) que foram introduzidos antes de ter sido efetuada uma mudança de linha (causada por carregar na tecla *enter*).

No código do lado direito, quando o utilizador introduz erradamente um caráter não numérico na leitura da variável `n1`, o canal *cin* fica *com erro* sendo depois o seu estado alterado para *sem erro* pela instrução `cin.clear()`. De seguida, a instrução `cin.ignore(10000,'\n')` apaga todo o conteúdo existente no canal de leitura. Assim sendo, uma vez que o canal de leitura não está em erro e não tem conteúdo, é pedido um novo valor ao utilizador para a variável `n2` e depois para a variável `s`, caso não tenha havido erro na leitura de `n2`.

Uma outra situação em que utilização da instrução `cin.ignore()` pode ser necessária é em programas onde sejam usadas conjuntamente as instruções `cin >>` e `getline(cin, ...)`. Consideremos os seguintes exemplos:

```
int n1;
string s;

cout << "Nome: ";
getline(cin, s);

cout << "Numero: ";
cin >> n1;

cout << n1 << " " << s;
```

```
int n1;
string s;

cout << "Numero: ";
cin >> n1;

cin.clear();
cin.ignore(10000, '\n');

cout << "Nome: ";
getline(cin, s);

cout << n1 << " " << s;
```

Como já sabemos, a leitura do `getline` termina quando for encontrada uma mudança de linha que, no caso deste exemplo, será implicitamente inserida ao carregar na tecla *enter*, sendo essa mudança de linha também lida mas ignorada. Assim sendo, no código do lado esquerdo, após a leitura da *string* s, o canal do *cin* estará sem erro e vazio, pelo que o utilizador pode introduzir nova informação no canal aquando da leitura da variável `n1`. Significa isto que aqui não serão necessárias as instruções `cin.clear()` ou `cin.ignore()`.

No código do lado direito, é primeiramente lida a variável `n1` e, aquando dessa leitura, a mudança de linha `"\n"` incluída automaticamente através da tecla *enter* ficará armazenada no canal. Ao ser colocado um *getline* logo de seguida, a informação existente no canal (`"\n"`) é lida pelo *getline*, pelo que teremos `s="\n"`. A utilização da instrução `cin.ignore(10000, '\n')` permite apagar todo o conteúdo do canal incluindo a mudança de linha lá existente. Assim sendo, o utilizador terá a possibilidade de voltar a introduzir informação que será guardada na variável `s`. É importante notar que neste exemplo o `cin.clear()` apenas é necessário para prevenir o caso em que o utilizador coloca um valor não numérico aquando da leitura do `n1`, pelo que não tem qualquer efeito se tal não acontecer.

Por fim, é importante reforçar que as instruções `.clear()` e `.ignore()` podem ser usadas exatamente da mesma forma para outros canais de leitura que não o *cin*. Tais canais são, por exemplo, canais de leitura para ficheiros ou para *strings* (*string streams*, que serão introduzidas na secção seguinte).

## 10.4 *String streams*

Um *stream* é um "canal" onde pode ser inserida informação ou de onde pode ser extraída informação. Como vimos nas secções anteriores, um objeto do tipo *ofstream* é um canal para inserir informação num ficheiro enquanto um objeto do tipo *ifstream* é um canal para extrair informação de um ficheiro. Existem também canais para inserir e extrair informação do ecrã, que são o *cout* e o *cin*, respetivamente. Nesta secção veremos como inserir e extrair informação de *strings*.

Para usar uma *string* como um canal de leitura ou de escrita devemos utilizar objetos do tipo `istringstream` e `ostringstream`, respetivamente, estando ambos disponíveis no pacote `sstream` (*string stream*) da biblioteca *standard*. Assim, é necessária a inclusão deste pacote no preâmbulo através da instrução

```
#include <sstream>
```

Para criar um canal de leitura para uma *string*, usamos um objeto do tipo `istringstream`. Depois disso, será então possível usar o operador `>>` para extrair informação da *string*. Vejamos o exemplo abaixo.

```
string s = "Amanha terei 30 anos";
istringstream iss(s);

string s1, s2, s3;
int n;

iss >> s1 >> s2 >> n >> s3;
```

Neste exemplo é criado um objeto do tipo `istringstream` com o nome `iss`, que é um canal de leitura para a *string* `s`. Assim, será possível extrair cada elemento da *string* `s` e armazená-lo numa variável do tipo adequado, o que é ilustrado pela variável `n` do tipo *int*. Nessa variável ficará armazenado o valor inteiro 30. Note-se que a leitura de uma *string* sem a utilização do *getline* termina assim que for encontrado um espaço ou uma mudança de linha.

Um objeto do tipo `ostringstream` cria um canal de escrita para uma *string* ao qual podemos adicionar informação facilmente com o operador `<<`. A grande vantagem da utilização de `ostringstreams` é o facto de se tornar possível concatenar facilmente objetos de diferentes tipos numa única *string*. Recorde-se que, até aqui, para concatenar uma variável de um tipo numérico numa *string* era necessário utilizar a função `to_string`. Após construir o canal de informação para a *string* pretendida através do objeto `ostringstream`, é necessário extrair a *string* criada, sendo para isso usado o método `.str()`. Vejamos o exemplo abaixo.

```
string s = "A Maria pesa ";
double x = 60.25;

ostringstream oss;
oss << s << x << "kg e mede " << 1.7 << 'm';

string nova = oss.str();
cout << nova;
```

Neste exemplo, é criado um objeto do tipo `ostringstream` com o nome `oss` que será usado para concatenar informação, sendo essa informação resultante de diferentes tipos de dados, nomeadamente *strings*, variáveis numéricas e carateres. O conteúdo da *string* criada é depois devolvido e guardado numa nova variável do tipo *string* (chamada `nova`) através do método `str()`, sendo essa variável escrita no ecrã.

# Bibliografia

[1] Stroustrup, B. (2014). *Programming: principles and practice using C++*. Pearson Education.
[2] Stroustrup, B. (2018). *A Tour of C++*. Addison-Wesley Professional.
[3] `https://www.learncpp.com/`