Apontamentos de Programação em C/C++

Paulo Baltarejo e Jorge Santos

Instituto Superior de Engenharia do Porto
Departamento de Engenharia Informática

Versão *Draft* – Março de 2006

Aviso de licença de utilização: Este documento pode ser utilizado livremente para fins não comerciais, é permitido aos seus utilizadores, copiar, distribuir e exibir publicamente os seus conteúdos, desde que sejam ressalvados os direitos de autor do mesmo, nomeadamente, deverá ser sempre incluída esta página em todas as cópias.
Paulo Baltarejo e Jorge Santos, 2006

Índice

1 Programação em C/C++ 1

1.1 História da linguagem C/C++ . . . 1
1.2 Estrutura de um programa em C++ . . . 1
1.3 Criação de um programa em C++ . . . 3
1.4 Modelo de compilação da linguagem C++ . . . 4
1.5 Variáveis, Tipos de dados e Constantes . . . 4
1.5.1 Variáveis . . . 5
1.5.2 Tipos de dados . . . 6
1.5.3 Declaração de variáveis . . . 6
1.5.4 Constantes . . . 8
1.5.4.1 Literais . . . 8
1.5.4.2 Definidas . . . 9
1.5.4.3 Declaradas . . . 10
1.6 Estruturas de controlo . . . 11
1.6.1 Instruções de Sequência . . . 11
1.6.1.1 Operadores aritméticos . . . 12
1.6.1.2 Operadores relacionais e lógicos . . . 13
1.6.2 Exercícios Resolvidos . . . 16
1.6.2.1 Distância euclidiana entre dois pontos . . . 16
1.6.2.2 Determinar perímetro e área de circunferência . . . 17
1.6.3 Exercícios Propostos . . . 17
1.6.3.1 Calcular índice de massa corpórea (IMC) . . . 17
1.6.3.2 Converter horas, minutos e segundos . . . 18
1.6.3.3 Teorema de Pitágoras . . . 18
1.6.3.4 Converter temperaturas . . . 18
1.6.4 Instruções de Decisão . . . 18
1.6.4.1 Decisão binária . . . 18
1.6.4.2 Decisão múltipla . . . 19
1.6.5 Prioridade dos operadores . . . 21
1.6.6 Exercícios Resolvidos . . . 22
1.6.6.1 Distância euclidiana entre dois pontos . . . 22
1.6.6.2 Classificar em função da média . . . 23
1.6.6.3 Determinar o máximo de 3 valores . . . 23
1.6.6.4 Determinar triângulo válido . . . 24
1.6.7 Exercícios Propostos . . . 25
1.6.7.1 Classificar triângulo . . . 25
1.6.7.2 Divisão . . . 25
1.6.7.3 Resolver equação da forma $ax^2 + bx + c = 0$ . . . 25

1.6.7.4 Converter entre escalas de temperaturas 25
1.6.7.5 Calcular índice de massa corpórea (IMC) 26
1.6.7.6 Determinar ano bissexto 26
1.6.7.7 Parque de estacionamento 26
1.6.8 Instruções de Repetição 27
1.6.8.1 Instrução `do-while` 27
1.6.8.2 Instrução `while` 28
1.6.8.3 Instrução `for` 28
1.6.9 Exercícios Resolvidos 30
1.6.9.1 Calcular somatório entre dois limites 30
1.6.9.2 Calcular factorial de um número 31
1.6.9.3 Determinar se um número é primo 32
1.6.9.4 Determinar número e idade da pessoa mais nova de um grupo 32
1.6.9.5 Determinar o aluno melhor classificado e a média das notas de uma turma 33
1.6.10 Exercícios Propostos 34
1.6.10.1 Divisão através de subtracções sucessivas 34
1.6.10.2 Determinar o máximo e mínimo de uma série 35
1.6.10.3 Determinar quantidade de números primos 35
1.6.10.4 Determinar se um número é perfeito 35
1.6.10.5 Calcular potência por multiplicações sucessivas . . . 35
1.6.10.6 Maior número ímpar de uma sequência de valores . . 35
1.6.10.7 Algarismos de um número 35
1.6.10.8 Apresentação gráfica de temperaturas 35
1.6.10.9 Soma dos algarismo de um número 36
1.6.10.10 Jogo de adivinhar o número 36
1.6.10.11 Capicua de um número 36
1.6.10.12 Conversão de base numérica 36
1.7 Funções 36
1.7.1 Âmbito da variáveis – global e local 38
1.7.2 Passagem de argumentos 41
1.7.2.1 Passagem por valor 41
1.7.2.2 Passagem por referência 42
1.7.2.3 Valores por omissão nos argumentos 43
1.7.3 Protótipos de funções 44
1.7.4 Estrutura de um programa em C++ 45
1.7.5 Exercícios resolvidos 46
1.7.5.1 Função que devolve o maior algarismo de um número 46
1.7.5.2 Função que indica se um número é perfeito 46
1.7.6 Exercícios propostos 47
1.7.6.1 Função média de dois números 47
1.7.6.2 Função lei de Ohm 47
1.7.6.3 Função somatório 48
1.7.6.4 Funções para codificar e descodificar números 48
1.7.6.5 Números primos 48
1.8 Vectores 48
1.8.1 Definição de vectores 49

1.8.2 Atribuição dos valores iniciais 49
1.8.3 Acesso aos elementos de um vector 50
1.8.4 Exercícios resolvidos 51
1.8.4.1 Funções manipulando vectores 51
1.8.5 Exercícios propostos 52
1.8.5.1 Determinar desvio padrão de uma série 52
1.8.5.2 Prova de atletismo 52
1.8.5.3 Suavização 53
1.9 Vectores multi–dimensionais 53
1.9.1 Exercícios resolvidos 55
1.9.1.1 Funções manipulação de matrizes 55
1.9.2 Exercícios propostos 58
1.9.2.1 Máximo local 58
1.9.2.2 Determinar se uma matriz é simétrica 58
1.10 Vectores como parâmetros 58
1.11 *Strings* 59
1.11.1 Iniciação de *strings* 60
1.11.2 Funções para manipulação de *strings* 61
1.11.3 Conversão de *strings* para outros tipos 62
1.11.4 Exercícios resolvidos 63
1.11.4.1 Programa para manipulação de *strings* e caracteres . 63
1.11.5 Exercícios propostos 68
1.11.5.1 Função que determine o número de ocorrências . . . 68
1.11.5.2 Função que verifique se uma *string* é inversa de outra 68
1.11.5.3 Função que conta as palavras de uma *string* 68
1.11.5.4 Função que formate uma *string* 68
1.12 Ponteiros 68
1.12.1 Operador endereço `&` 68
1.12.2 Operador de referência `*` 69
1.12.3 Ponteiros e vectores 72
1.12.4 Ponteiros para ponteiros 72
1.12.5 Ponteiros do tipo `void` 73
1.13 Tipos de dados não nativos 74
1.13.1 Estruturas de dados – instrução `struct` 74
1.13.2 Definição de tipos – instrução `typedef` 79
1.13.3 União – instrução `union` 79
1.13.4 Enumeradores – instrução `enum` 81
1.13.5 Exercícios resolvidos 83
1.13.5.1 Ponto e recta 83
1.13.5.2 Gestão de clientes de uma discoteca 85
1.13.6 Exercícios propostos 86
1.13.6.1 Empresa de construção civil 86
1.13.6.2 Empresa de construção civil 86
1.14 Programas de grandes dimensões 90
1.14.1 Divisão em módulos 91

Lista de Figuras

1.1 Ciclo de vida de um programa em C 3
1.2 Modelo de compilação 4
1.3 Execução de uma função 38
1.4 Arquitectura do modelo de memória 40
1.5 Execução da pilha 41
1.6 Passagem de variáveis por valor 42
1.7 Passagem de variáveis por referência 43
1.8 Estrutura de um programa em C/C++ 45
1.9 Ilustração da lei de Ohm 48
1.10 Representação gráfica do vector `vec` 49
1.11 Representação gráfica de uma matriz 53
1.12 Representação de uma *string* 60
1.13 Representação da memória do computador 69
1.14 Ponteiro 70
1.15 Ponteiro para ponteiro 73
1.16 Representação de uma união 80
1.17 Tipos de dados 83
1.18 Representação gráfica de um módulo 91
1.19 Inclusão de um ficheiro 92
1.20 Inclusão de um ficheiro (directivas de pré-processamento) 93

Lista de Tabelas

1.1 Palavras reservadas 6
1.2 Tipos de dados 7
1.3 Caracteres especiais 9
1.4 Exemplo operadores compostos 12
1.5 Operadores aritméticos 13
1.6 Operadores aritméticos compostos 13
1.7 Operadores de incremento (modo prefixo e sufixo) 14
1.8 Operadores relacionais 14
1.9 Exemplos de operadores relacionais 14
1.10 Operadores lógicos 15
1.11 Exemplos de operadores lógicos 15
1.12 Tabelas de verdade: conjunção, disjunção e negação 15
1.13 Operadores de manipulação de bits 16
1.14 Tabela de verdade dos operadores de manipulação bits 16
1.15 Relação de precedência dos operadores 22
1.16 Índice de massa corpórea 26

Resumo

Estes apontamentos têm como objectivo principal apoiar os alunos que pretendam aprender programação de computadores utilizando a linguagem C++, em particular aqueles que frequentam a disciplina de *Introdução à Informática* do Ano 0 leccionada no Instituto Superior de Engenharia do Porto (ISEP).

A estrutura destes apontamentos foi definida de acordo com a abordagem de *aprender-por-exemplo*, pelo que, os conceitos são apenas introduzidos de acordo com a necessidade de explicar a resolução de um determinado programa. De forma a suportar esta abordagem é apresentado um grande número de exercícios resolvidos.

Porto, Janeiro de 2006
Jorge Santos e Paulo Baltarejo

Capítulo 1

Programação em C/C++

1.1 História da linguagem C/C++

O C é uma linguagem de programação imperativa (procedimental) típica. Foi desenvolvida em 1970 por Dennis Ritchie para utilização no sistema operativo Unix. Esta linguagem é particularmente apreciada pela eficiência e é a mais utilizada na escrita de software para sistemas operativos e embora menos, no desenvolvimento de aplicações. A sua utilização também é comum no ensino, apesar de não ser a linguagem inicial para iniciados.

De acordo com Ritchie, o desenvolvimento inicial da linguagem C aconteceu nos laboratórios da AT&T entre 1969 e 1973. O nome "C"foi escolhido porque algumas das suas características derivavam de uma linguagem já existente chamada "B".

Em 1973, a linguagem C tornou-se suficientemente poderosa para suportar a escrita de grande parte do *kernel* do Unix que tinha sido previamente escrito em código *assembly*.

Em 1978, Ritchie e Brian Kernighan publicaram a primeira edição do livro "The C Programming Language"[Kernighan e Ritchie, 1988] que durante muitos anos funcionou como a especificação informal da linguagem (K&R C). Posteriormente a segunda versão versou sobre a especificação ANSI C. A especificação K&R C é considerado o conjunto mínimo obrigatório que um compilador da linguagem deve implementar.

O C++ [Stroustrup, 2000], pronunciado "cê mais mais", é uma linguagem de programação genérica que suporta tanto os paradigmas da programação estruturada (procedimental) como o orientado ao objecto. A partir de 1990, o C++ tornou-se uma das linguagens de programação mais populares.

Bjarne Stroustrup desenvolveu o C++ (originalmente designado "C com classes") nos laboratórios da Bell em 1983, como um melhoramento da linguagem C. Este melhoramentos incluíram: adição de classes, funções virtuais, sobrecarga de operadores, múltipla herança, templates e tratamento de excepções.

O standard do C++ foi ratificado em 1998 como ISO/IEC 14882:1998, sendo que a versão actual, de 2003, é o ISO/IEC 14882:2003.

1.2 Estrutura de um programa em C++

Como primeira abordagem à linguagem C++, considere-se o programa 1.1.

Listing 1.1: Programa clássico - Bom dia mundo

```
1 // o meu primeiro programa em C++
2 #include<iostream.h>
3 int main()
4 {
5    cout<<"Bom dia Mundo!";
6    return 0;
7 }
```

Este programa depois compilado e executado produziria o seguinte resultado:

```
Bom dia Mundo!
```

Apesar do resultado simples, o programa contém um conjunto de elementos que merecem um análise detalhada.

- `//O meu primeiro programa em C++`

  Isto é uma linha de comentário. Todas as linhas começadas por duas barras `//` são consideradas comentários e não tem qualquer efeito na compilação/execução do programa. Servem para inserir explicações e observações no código do programa;

- `#include <iostream.h>`

  As linhas que começam pelo carácter cardinal (`#`) são directivas de pré-compilação. Estas não são linhas de código executável mas apenas indicações para o compilador. Neste caso `#include <iostream.h>` indica ao pré–processador do compilador que inclua os cabeçalhos existentes no ficheiro `iostream.h` relativos a funções utilizadas na entrada e saída de dados de um programa;

- `int main()`

  Esta linha corresponde ao início da declaração da função `main`. A função `main` é o ponto por onde todos os programas em C++ começam a sua execução. `main` é seguido de parêntesis (`(`, `)`) porque é uma função. O conteúdo da função `main` que está imediatamente a seguir à declaração formal, está contido entre chavetas (`{ }`) conforme o exemplo;

- `cout << "Bom dia Mundo!";`

  Esta instrução coloca no ecrã a frase "Olá Mundo!". `cout` (*console output*) é um objecto normalmente associado ecrã. `cout` está declarado no ficheiro de cabeçalhos (*header file*) `iostream.h`, portanto para poder usar o `cout` é necessário incluir o ficheiro `iostream.h`. Note–se que esta frase termina com ponto vírgula (`;`). Este carácter significa o fim da instrução e tem obrigatoriamente que ser incluído depois de qualquer instrução;

- `return 0;`

  A instrução `return` provoca o fim da execução da função `main` e (devolve) retorna o que está a seguir, neste caso o zero (0).

1.3 Criação de um programa em C++

O desenvolvimento de programas em linguagem C++, tal como na maioria das linguagens compiladas, é um processo que compreende quatro fases: escrita, compilação, *"linking"* e execução (ver figura 1.1).

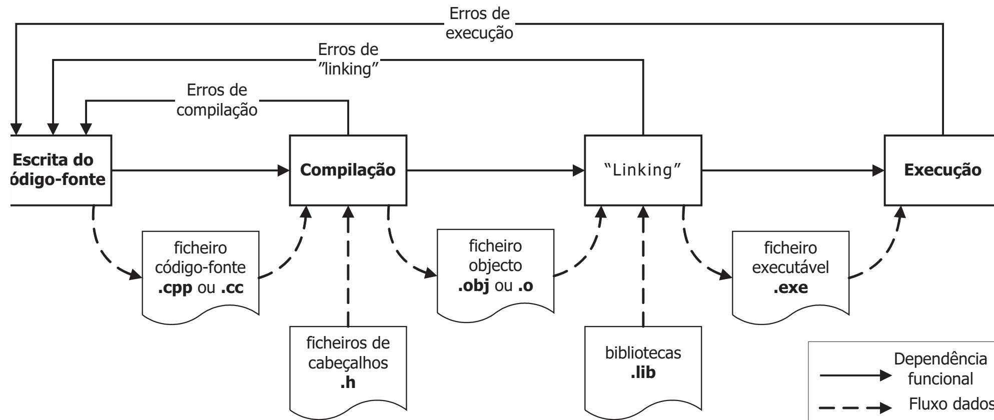


Figura 1.1: Ciclo de vida de um programa em C

Normalmente, os ambiente de desenvolvimento integrado de aplicações (IDE[1]) incluem um conjunto de ferramentas que suportam as referidas fases do desenvolvimento de um programa/aplicação.

- **Escrita do código–fonte** – A primeira fase do processo é criação/edição de um (ou mais) ficheiro de texto contendo o código–fonte. Isto pode ser realizado com recurso a um editor de texto. O referido ficheiro tem que ter a extensão **".cpp"** ou **".cc"**. O conteúdo do programa tem que obedecer rigorosamente à sintaxe da linguagem.

- **Compilação** – A segunda fase, a da compilação, é realizada com recurso a um compilador específico para linguagem, neste caso o de C++. Nesta fase se existirem erros de sintaxe, o compilador detecta-os e reportará a sua localização (tipicamente número de linha) e uma breve descrição do erro. Note-se que os erros de lógica não são detectados nesta fase. Se o programa não tiver erros de sintaxe o compilador produzirá o código executável, isto é, um programa pronto a ser executado. Nesta fase são incluídos os ficheiros de cabeçalhos nos quais são declaradas as funções que serem posteriormente incluídas no fase de *linking*.

- ***"Linking"*** – A terceira fase, a da *linking*, é feita com recurso a um programa especial, designado *linker* que se responsabiliza por transformar os programa objecto (independente da plataforma de *hardware*) numa aplicação executável na plataforma em *hardware* específica. Nesta fase, as declarações das funções (cabeçalhos) são substituídas por código executável, propriamente dito. Caso ocorram erros nesta fase, o processo retorna à primeira fase.

- **Execução** – A quarta e última fase, execução, só poderá ocorrer no caso das fases terminarem com sucesso.

[1]Do anglo-saxónico *Integrated Environment Development*

1.4 Modelo de compilação da linguagem C++

A figura 1.2 apresenta o modelo de compilação da linguagem C++.

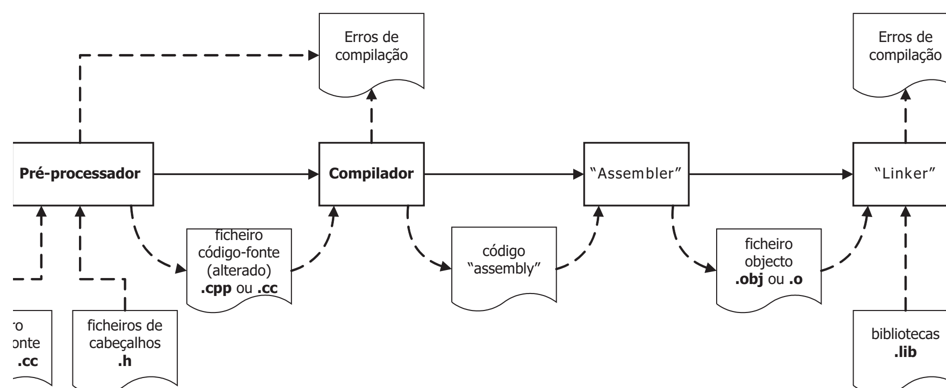


Figura 1.2: Modelo de compilação

No modelo de compilação da linguagem C++ são quatro os intervenientes principais: pré-processador, compilador, *assembler* e *linker*.

- **Pré–processador** – O pré-processador executa as directivas existentes no código fonte. As directivas de pré-processamento começam pelo carácter `#` e de entre as tarefas mais comuns compreendidas na pré–compilação, destacam-se:
  - `#include` – Inclusão de ficheiros, tipicamente ficheiros de cabeçalhos;
  - `#define` – definição de constantes (ver secção 1.5.4.2);
  - Remoção de linhas de comentários.
- **Compilador** – Alguns compiladores traduzem o código fonte (texto) recebido do pré–processamento para linguagem assembly (também texto). Enquanto que outros geram directamente o código objecto (instruções do processador em código binário);
- ***Assembler*** – O assembler traduz código em linguagem assembly (texto) para código binário. Pode estar integrado no compilador. O código gerado pelo assembler geralmente tem a extensão **".o"** (nos sistemas operativos Unix e Linux) e **".obj"** (nos sistema operativos Windows);
- ***Linker*** – Se o programa fonte tem referências a elementos definidos nas bibliotecas padrão ou outras, o *Linker* é o responsável por adicionar o código desses elementos de forma a criar o ficheiro executável. A referência a variáveis globais externas é resolvida pelo *Linker*.

1.5 Variáveis, Tipos de dados e Constantes

O programa `OlaMundo` (ver listagem 1.1) apesar de estruturalmente completo, tem apenas como funcionalidade, escrever na consola de saída, tipicamente o ecrã, a frase *"Ola Mundo!"*.

1.5.1 Variáveis

Na escrita de programas mais sofisticados, regra geral, será necessária: a introdução de dados (eventualmente introduzidos pelo utilizador) e guardar informação relativa a cálculos. É neste contexto que será introduzido o conceito de variável.

Suponha-se que é pedido a uma determinada pessoa para memorizar o número 5 e depois o número 2. A seguir, é-lhe pedido que adicione a quantidade 1 ao primeiro número memorizado. Depois é-lhe pedido o resultado da subtracção do primeiro com o segundo. Após a realização correcta destas contas, a pessoa em questão diria que o resultado seria 4. Este processo também pode ser executado pelo computador. Para tal é necessário descrever este processo através da linguagem de programação (ver programa da listagem 1.2).

Listing 1.2: Somar dois números

```
1  #include<iostream.h>
2  int main()
3  {
4      int a,b,result;
5      a=5;
6      b=2;
7      a=a+1;
8      result=a-b;
9      cout<<"Resultado: "<<result<<endl;
10
11     return 0;
12 }
```

Da mesma forma que esta pessoa guardou temporariamente os valores em memória o computador também tem capacidade para isso, para tal foi necessário definir algumas variáveis.

Um variável é uma porção de memória que permite armazenar um determinado valor. Cada variável tem um identificador e o tipo de dados. O identificador permite referir de forma única uma variável enquanto que tipo de dados é útil para a optimização do código pois diferentes tipos de dados requerem quantidade distintas de memória. Por exemplo, no código do programa da listagem 1.2, os identificadores utilizados: `a`, `b` e `result` são do tipo inteiro (`int`).

Na escolha de identificadores aquando da definição das variáveis, o programador deve obedecer a conjunto de regras para que estes sejam válidos, a saber:

- Ser uma sequência de uma ou mais letras, dígitos e/ou carácter "underscore"(_), iniciada obrigatoriamente por letra ou "underscore";
- Os espaços não podem fazer parte de um identificador;
- Embora o comprimento de um identificador não seja limitado, alguns compiladores só identificam os primeiros 32 caracteres, sendo o resto ignorado;
- Não podem ser iguais a qualquer palavra reservada da linguagem C++. A tabela 1.1 apresenta as palavras reservadas segundo a norma ANSI-C++.

Um outro aspecto a ter em consideração na definição de um identificador, é que a linguagem C++ é sensível à capitalização, isto é, faz a distinção entre letras

Tabela 1.1: Palavras reservadas

asm, auto, bool, break, case, catch, char, class, const, const_cast, continue, default, delete, do, double, dynamic_cast, else, enum, explicit, extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace, new, operator, private, protected, public, register, reinterpret_cast, return, short, signed, sizeof, static, static_cast, struct, switch, template, this, throw, true, try, typedef, typeid, typename, union, unsigned, using, virtual, void, volatile, wchar_t

maiúsculas e minúsculas, por exemplo os identificadores: `result`,`Result` e `RESULT` são distintos para o compilador.

1.5.2 Tipos de dados

Aquando da declaração de uma variável, o compilador da linguagem C++ necessita, por uma questão de optimização de código, de saber qual o tipo da referida variável de forma a reservar em memória apenas o espaço estritamente necessário. Por exemplo, para armazenar uma letra não é necessário tanto espaço como para armazenar um número real.

Note–se que a memória do computador esta organizada em bytes, sendo que um byte é a unidade mínima para atribuição de memória. Com um byte é possível armazenar:

- Um número sem sinal entre 0 e 255;
- Um número com sinal entre -128 a 127;
- Um carácter simples.

No entanto, em muitas situações será necessário armazenar valores superiores à capacidade de um byte, para este efeito existem outros tipos de dados. A tabela 1.2 apresenta os tipos dados fundamentais em C++, espaço necessário (em bytes) e a gama de valores respectivos.

Note-se que efectivamente em C++ os caracteres são armazenados como um valor numérico, de acordo com a norma ASCII[2].

1.5.3 Declaração de variáveis

A utilização de uma variável num programa C++ requer a sua declaração prévia. Considere-se os exemplos constantes no excerto de código da listagem 1.3.

Listing 1.3: Declaração de variáveis

```
1 {
2     // Declaração de variáveis
3     int a;
```

[2] *American Standard Code for Information Interchange.*

Tabela 1.2: Tipos de dados

Nome	Tam.	Descrição	Gama de valores
char	1	Caracter ou inteiro de 8 bits de comprimento	c/sinal: -128 a 127 e s/sinal: 0 a 255
short	2	Inteiro de 16 bits de comprimento	c/sinal: -32768 a 32767 e s/sinal: 0 a 65635
long	4	Inteiro de 32 bits de comprimento	c/sinal: -2147483648 a 2147483647 e s/sinal: 0 a 42949967295
int	*		ver short e long
float	4	Número real	3.4e±38
double	8	Número real, virgula flutuante, dupla precisão	1.7e±308
long double	10	Número real longo, virgula flutuante, dupla precisão	1.2e±4932
bool	1	Valores lógicos - booleanos	true e false
wchar_t	2	Caracter extendido, necessário para a representação de caracteres internacionais	Caracteres (incluindo internacionais)

```
4      long int saldo;
5      float media;
6      double raiz;
7      bool sentinela;
8      char tecla;
9      ...
10
11     // Declaração de variáveis
12     // mais do que uma variável por linha
13     int x,y,z;
14     float area,perimetro;
15     ...
16
17     // Declaração de variáveis
18     // mais do que uma variável por linha
19     // e com iniciações
20     bool ePrimo=false;
21     int nDiv=0,produto=1;
22     long double numero=1.0,factorial=1.0;
23     char _1letra='a',_2letra='b';
24 }
```

No excerto são apresentados três blocos que pretendem ilustrar:

- No primeiro bloco, a declaração de uma variável por linha de código;
- No segundo bloco, múltiplas declarações por linha;
- No terceiro bloco, para além da declaração múltipla por linha, a iniciação de variáveis.

Conforme se verifica no excerto, no caso de se declarar mais do que uma variável por instrução, estas têm estar separadas por virgula.

Quando uma variável é declarada, o seu valor é indeterminado por omissão, isto é, não se sabe qual é o valor. Por vezes é útil iniciar uma variável com um determinado valor aquando da sua declaração. Para tal acrescenta-se o sinal de igual e o valor com que se quer inicializar a variável.

Formalmente a sintaxe subjacente à declaração/iniciação de uma ou mais variáveis é:
`<tipo-de-dados> <id1>=<Valor inicial1>, <id2>=<Valor inicial2>,...;`
sendo que os parêntesis rectos representam o caracter opcional, neste caso, as iniciações de variáveis são opcionais.

1.5.4 Constantes

Uma constante é qualquer identificador que tem um valor fixo, portanto que não pode ser alterada durante a execução do programa. Num programa em C++ podem coexitir três tipos de constantes: literais, definidas e declaradas.

1.5.4.1 Literais

Um literal é uma qualquer identificador com o valor fixo e podem ser divididos em: números inteiros, números reais, caracteres e cadeias de caracteres[3]

- **Números inteiros** - são constantes numéricas que identificam números na base decimal. Note-se que para expressar uma constante numérica não é necessário escrever aspas(") ou qualquer outro caracter especial.

  ```
  1776
  707
  -203
  ```

  Quando é escrito `1776` num programa está-se a referir ao valor numérico `1776` (mil setecentos e setenta e seis).

  Além da base decimal é possível utilizar a notação octal ou hexadecimal. Para a notação octal precede-se o numero do caracter `0` (zero) e para hexadecimal precede-se o número dos caracteres `0x`(zero e 'x').

  De seguida são apresentadas várias formas de representar o mesmo valor, neste caso o `75` (setenta e cinco).

Valor	Notação
75	decimal
0113	octal
0x4b	hexadecimal

- **Numeros Reais**

  Expressam números com décimas e/ou expoente. Podem incluir um ponto decimal, o caracter `e` (que expressa "10 elevado a X", onde X é um valor inteiro) ou ambos.

[3]Do anglo-saxónico *String*.

Notacão	Valor
3.14159	3.14159
6.022e23	$6.022 * 10^{23}$
1.6e-19	$1.6 * 10^{-19}$
1.41421356237309504	$\sqrt{2}$
3.0	3.0

Estas são cinco formas válidas de representar números reais em C++. Respectivamente, o valor de π, o número de Avogadro, a carga eléctrica de um electrão (extremamente pequena), raiz de dois e o numero três expresso como decimal;

- **Caracteres e *strings***

  Também existem constantes não numéricas, como a seguir se exemplifica:

```
'z'
'p'
"Olá Mundo!"
"Como estás?"
```

  As primeiras duas expressões representam caracteres simples enquanto as outras duas representam *strings* (cadeia de caracteres). Para os caracteres simples é usada a plica (') e para as *strings* são usados as aspas (").

  Existem caracteres especiais que não podem ser expressos sem ser no código fonte do programa, como por exemplo, nova linha ('\n') ou tabulação ('\t'). Todos são precedidos pela barra "\". A tabela 1.3 seguinte apresenta a lista dos caracteres especiais.

Tabela 1.3: Caracteres especiais

Caracter	Significado
\n	nova linha
\r	cursor para 1ªcoluna
\t	tabulação
\b	*backspace*
\'	plica simples
\"	aspas simples

1.5.4.2 Definidas

Para além dos literais existem constantes que podem ser declaradas pelo programador, podendo ser definidas através de uma directiva de pré-processamento `#define` ou através da palavra–chave `const` (ver secção 1.5.4.3).

A seguir apresenta-se a forma como definir uma constante através da directiva de pré–processamento `#define` e cuja a sintaxe é a seguinte:

```
#define <identificador> <expressão>
```

Como já foi referido anteriormente o pré–processador única e simplesmente actua sobre o código (texto) e neste caso substituí todas as ocorrências no código da constante pelo seu valor. Por esta razão estas constantes são designadas de macros. **Nota: Um erro muito frequente é a colocação do ponto e virgula (;) na declaração da macro.**

O excerto de código 1.4 apresenta um exemplo da utilização de uma macro. A utilização de macros tem duas vantagens:

- **Legibilidade do código** – por exemplo é mais fácil associarmos a mnemónica PI ao π do que o valor 3.14159265;
- **Facilidade de manutenção do código** – o programador evita a repetição da declaração das constantes, tornando mais fácil alterar o seu valor, pois basta alterar a referida definição da macro, não é necessário procurar em todo o código o valor que se pretende alterar.

Listing 1.4: Utilização de macros

```
1 #include<iostream.h>
2
3 #define PI          3.14159265
4 #define NEWLINE     '\n'
5
6 int main()
7 {
8     int raio=3;
9     double areaCirc;
10
11    areaCirc = raio*raio*PI;
12    cout<<"Raio=" << raio << " Area=";
13    cout<< areaCirc << NEWLINE;
14
15    raio=5;
16    areaCirc = raio*raio*PI;
17    cout<<"Raio=" << raio << " Area=";
18    cout<< areaCirc << NEWLINE;
19
20    return 0;
21 }
```

1.5.4.3 Declaradas

Através da palavra–chave `const` é possível declarar constantes de um determinado tipo de dados, conforme a seguir se exemplifica:

```
const int width = 100;
const char tab = \t;
const minhaConstante=12440;
```

No caso do tipo de dados não ser declarado, no exemplo anterior este caso surge na última linha, o compilador assume que é do tipo inteiro.

- **Operador** `sizeof()` – Este operador recebe um parâmetro, que pode ser uma variável ou um tipo de dados e devolve o espaço em memória (em bytes) ocupado por essa variável ou tipo de dados, conforme exemplo seguinte, no qual a variável `a` tomará o valor 1.

  ```
  int a=sizeof(char);
  ```

- **Conversão de tipos de dados (instrução *cast*)** – A conversão de dados consiste em converter dados de um tipo noutro tipo de dados. Por exemplo:

  ```
  int i;
  float f = 3.14;
  i = (int) f;
  ```

  Este código converte o numero `3.14` para o inteiro `3`. Outra forma de o poder fazer seria com a seguinte instrução:

  ```
  i=int(f);
  ```

  Apesar do resultado desta conversão resultou numa perda de informação, em algumas situações tal esta situação pode ser desejável.

  Nota: O que é convertido é o valor da variável f não a variável f. Esta é do tipo `float` e continuará a ser.

1.6 Estruturas de controlo

De acordo com o paradigma da programação estruturada qualquer programa pode ser descrito utilizando exclusivamente as três estruturas básicas de controlo: **instruções de sequência**, **instruções de decisão** e **instruções de repetição**.

1.6.1 Instruções de Sequência

As instruções de sequência são instruções atómicas (simples) permitem a leitura/escrita de dados, bem como o cálculo e atribuição de valores. Todas as instruções são executadas pela ordem apresentada no programa.

O programa da listagem 1.5 codifica em C++ um algoritmo cujo objectivo é cambiar euros em dólares considerando a taxa de conversão 1.17.

Listing 1.5: Cambiar euro para dólar

```
1  #include<iostream.h>
2  const double taxa=1.17;
3  int main()
4  {
5    double valorEuro,valorDolar;
6    cout<<"Introduza o valor em euros=";
7    cin>>valorEuro;
8    valorDolar=valorEuro*taxa;
9    cout<<"Valor em dolar = "<<valorDolar<<endl;
10
11   return 0;
12 }
```

- cin >> valorEuro; – Esta instrução fica aguardar que o utilizador digite (via teclado) o valor em euros. O valor digitado é extraído para a variável valorEuro através do operador de extracção >>. O cin (*console input*) é um objecto que está normalmente associado ao teclado. O operador de extracção extraí do objecto que representa o teclado na aplicação para uma variável da aplicação;

- #include<iostream.h> – A directiva de pré-processador promove a inclusão a definição dos dois objectos associados quer a entrada de dados (cin) quer à saída de dados (cout);

- valorDolar=valorEuro*taxa; – Esta linha de código realiza duas operações, a operação de multiplicação valorEuro*taxa; e a operação de atribuição do cálculo à variável valorDolar;

- cout << "Valor em dolar = "<< valorDolar << endl; – Esta linha de código coloca no ecrã da nossa aplicação o texto "Valor em dolar = "concatenado com o conteúdo da variável valorDolar seguido do manipulador de saída de dados endl que coloca o caracter '\n' (nova linha) e obriga o objecto cout a escrever no ecrã.

1.6.1.1 Operadores aritméticos

O operador de atribuição serve para atribuir o resultado de uma expressão a uma variável. Conforme exemplo seguinte:
a=5;

A linha de código anterior atribuí o valor 5 à variável a. A parte esquerda do operador = é conhecida como lvalue (*left value*) e a parte direita como rvalue (lado direito). rvalue é uma expressão (*e.g.*, uma variável, uma constante, uma operação ou qualquer combinação destes elementos) enquanto que o lvalue é o nome de uma variável;

A operação de atribuição é realizada da direita para a esquerda e nunca o inverso. De seguida são apresentados alguns exemplos (ver tabela 1.4):

Tabela 1.4: Exemplo operadores compostos

Exemplo	**Resultado**

a=10;
b=4;
c=a;
	O valor da variável c será 10

a=10;
b=4;
c=5;
a=b*(c + a);
	O valor da variável a será 60 (4*(10+5))

b=c=a=0;	O valor das variáveis a, b e c será 0

A linguagem C++ define os seguintes operadores aritméticos simples (ver tabela 1.5). Na exemplificação dos operadores considerem-se as variáveis a e b com valores, 13 e 5, respectivamente.

Tabela 1.5: Operadores aritméticos

Operador	**Nome**	**Exemplo**	**Resultado**
`+`	soma	`a+b`	18
`-`	subtracção	`a-b`	8
`*`	multiplicação	`a*b`	65
`/`	divisão	`a/b`	2.6
`%`	resto da divisão inteira	`a%b`	5

O C++ fornece igualmente os seguintes operadores aritméticos compostos (ver tabela 1.6). Na exemplificação dos operadores considerem-se as variáveis `a` e `b` com valores, 13 e 5, respectivamente.

Tabela 1.6: Operadores aritméticos compostos

Operador	**Nome**	**Exemplo**	**Significado**
`+=`	soma/atribuição	`a+=b`	`a=a+b`
`-=`	subtração/atribuição	`a-=b`	`a=a-b`
`*=`	multiplicação/atribuição	`a*=b`	`a=a*b`
`/=`	divisão/atribuição	`a/=b`	`a=a/b`
`%=`	resto divisão inteira/atribuição	`a%=b`	`a=a%b`
`++`	incremento	`a++`	`a=a+1`
−−	decremento	`b`−−	`b=b-1`

Note-se que os operadores aritméticos compostos da linguagem de programação C++ permitem modificar o valor de uma variável com um operador básico.

Os operadores incremento (`++`) e decremento (−−) só podem ser usados com variáveis. Estes incrementam e decrementam o valor da variável em uma unidade. Portanto, (`++`) e (−−) são equivalentes a `+=1` e a `-=1`, respectivamente.

Uma característica destes operadores é que podem ser usado como prefixo (pré–incremento ou pré–decremento) ou como sufixo (pos–incremento ou pos-decremento). Para tal, o operador tem de ser escrito antes da variável (`++a`) ou depois da (`a++;`), prefixo e sufixo, respectivamente. Embora quando usadas em expressões simples tipo (`++a;`) ou (`a++;`) tenham o mesmo significado. Em operações na quais o resultado da operação de incremento ou de decremento é avaliada noutra expressão, podem ser diferentes. No caso do operador de incremento usado como prefixo (`++a;`) o valor é incrementado e depois a expressão é avaliada e portanto o valor incrementado é considerado na expressão. No caso operador de incremento ser usado como sufixo (`a++;`), o valor da variável é incrementado após a avaliação da expressão.

A seguir apresentam-se dois exemplos para explicitar melhor as diferenças entre os dois modos:

1.6.1.2 Operadores relacionais e lógicos

Nesta secção são apresentados os operadores relacionais e lógicos utilizados na linguagem C++, bem como exemplos da sua utilização.

Tabela 1.7: Operadores de incremento (modo prefixo e sufixo)

Exemplo	**Resultado**
`b=3;` `a=++b;`	O valor final de `a` será 4 e o de `b` também
`b=3;` `a=b++;`	O valor final de `a` será 3 e o de `b` 4

Operadores relacionais

A tabela 1.8 apresenta os operadores relacionais da linguagem C++. Os operadores relacionais avaliam a comparação entre duas expressões. O resultado dessa comparação é um valor do tipo `bool` que pode ser `true` ou `false`, obviamente de acordo com o resultado da comparação.

Tabela 1.8: Operadores relacionais

Símbolo	**Significado**
$<$	menor que
$>$	maior que
$\leq$	menor ou igual que
$\geq$	maior ou igual que
$==$	igual
$!=$	diferente

Na tabela 1.9 apresenta-se alguns exemplos da utilização dos operadores relacionais nos quais se consideram os valores `a=5; b=6; c=7;`.

Tabela 1.9: Exemplos de operadores relacionais

Exemplo	**Resultado**
`(7==5)`	falso
`(a!=b)`	verdade
`(a<=7)`	verdade
`((a*b)>=c)`	verdade

Operadores lógicos

A tabela 1.10 apresenta os operadores lógicos da linguagem C++

O resultado das operações com os operadores lógicos também é verdade ou falso. O operador !, colocado à esquerda da expressão, inverte o valor lógico da mesma. Isto é, se a expressão é verdadeira passa a falsa e vice–versa. A tabela 1.11 apresenta alguns exemplos da utilização dos operadores lógicos.

De seguida são apresentadas as tabelas de verdades das operações lógicas: conjunção, disjunção e negação (tabela 1.12).

Tabela 1.10: Operadores lógicos

<table>
<tr><th>Símbolo</th><th>Significado</th></tr>
<tr><td>&&</td><td>conjunção</td></tr>
<tr><td>||</td><td>disjunção</td></tr>
<tr><td>!</td><td>negação</td></tr>
</table>

Tabela 1.11: Exemplos de operadores lógicos

<table>
<tr><th>Exemplo</th><th>Resultado</th></tr>
<tr><td>((5==5) & & (3>6))</td><td>falso</td></tr>
<tr><td>((5==5) || (3>6))</td><td>verdade</td></tr>
<tr><td>!(5==5)</td><td>falso</td></tr>
<tr><td>!verdade</td><td>falso</td></tr>
</table>

Operador ternário

O operador ternário ou condicional avalia uma expressão e devolve diferentes valores de acordo com a avaliação da expressão. A sintaxe deste operador é a seguinte:
<condição> ? <resultado1> : <resultado2>

Se a <condição> é verdade então o operador vai devolver o <resultado1>. Caso contrário devolve o <resultado2>. Por exemplo:

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

A variável x vai ficar com o valor 3, uma vez que 7 é diferente de 5.

```
bool x;
x=(5>3 ? true : false);
```

Neste caso é atribuída à variável x o valor verdade.

Operadores manipulação bits

Os operadores de manipulação de bits aplicam-se apenas a expressões numéricas inteiras.

O operador ~ é um operador unário e complementa (os bits 1s passam a 0s e vice–versa) todos os bits da variável que estiver colocada ao seu lado direito. Os operadores de deslocamento executam os deslocamento do operando colocado à sua

Tabela 1.12: Tabelas de verdade: conjunção, disjunção e negação

<table>
<tr><th>a</th><th>b</th><th>a && b</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td></tr>
<tr><td>1</td><td>0</td><td>0</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>

<table>
<tr><th>a</th><th>b</th><th>a || b</th></tr>
<tr><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td></tr>
</table>

a	!a
0	1
1	0

Tabela 1.13: Operadores de manipulação de bits

Símbolo	Significado
~	Complemento (not)
&	Conjunção (and)
|	Disjunção inclusiva (or)
^	Disjunção exclusiva (xor)
>>	Deslocamento à direita (shift)
<<	Deslocamento à esquerda (shift)

Tabela 1.14: Tabela de verdade dos operadores de manipulação bits

a	b	a & b	a | b	a ^ b
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	1	0

esquerda, um número de posições indicado pelo operando da direita. Os operadores `&, |` e `^` são binários e caracterizam-se pelas seguintes relações (ver tabela 1.14):

No deslocamento para a esquerda (`<<`) as posições que ficam livres são ocupadas com bits a `0`. No deslocamento para a direita (`>>`) as posições livres são ocupadas com bits em `0`, se a quantidade deslocada for sem sinal(`unsigned`), ou com bits bits idênticos ao mais significativo, se as quantidades deslocadas possuírem sinal. Por exemplo, `x<<2`, desloca a representação binária do valor contido em `x`, duas posições (bits) para a esquerda. Se `x` contiver o valor binário `00000010` (`2` em decimal) então `x<<2` faz com que `x` passe a conter o valor `00001000` (`8` em decimal).

1.6.2 Exercícios Resolvidos

Nesta secção são apresentados alguns problemas e respectivas soluções com o objectivo de ilustrar a utilização de instruções sequenciais.

1.6.2.1 Distância euclidiana entre dois pontos

O programa da listagem 1.6 permite realizar o cálculo da distância euclidiana entre dois pontos, sendo que cada ponto é definido pelas coordenadas (x,y). A distância pode ser calculada de acordo com a fórmula 1.6.1.

$$\text{distância} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{1.6.1}$$

Listing 1.6: Distância euclidiana entre dois pontos

```
1 #include<iostream.h>
2 #include<math.h>
3
4 int main()
5 {
6   int x1,y1,x2,y2;
```

```
7    double distancia;
8    cout<<"Coordenadas ponto1(x/y): ";
9    cin>>x1>>y1;
10   cout<<"Coordenadas ponto2(x/y): ";
11   cin>>x2>>y2;
12
13   distancia=sqrt(pow((x2-x1),2)+pow((y2-y1),2));
14
15   cout<<"Distancia="<<distancia<<endl;
16   return 0;
17 }
```

1.6.2.2 Determinar perímetro e área de circunferência

O programa da listagem 1.7 permite determinar o perímetro e área de uma circunferência, a partir do valor do raio.

Listing 1.7: Área e perímetro de um circunferência

```
1 #include <iostream.h>
2 #include <math.h>
3 const double pi=3.1415;
4 int main ()
5 {
6     double area, perimetro;
7     int raio;
8
9     cout<<"Introduza o valor do raio: ";
10    cin>>raio;
11    area=pi*pow(raio,2);
12    perimetro=2 * pi * raio;
13    cout<<"Area: "<<area<<endl;
14    cout<<"Perimetro: "<<perimetro<<endl;
15
16    return 0;
17 }
```

1.6.3 Exercícios Propostos

Nesta secção são propostos alguns problemas com vista à aplicação conjugada de instruções sequenciais.

1.6.3.1 Calcular índice de massa corpórea (IMC)

O índice de massa corpórea (IMC) de um indivíduo é obtido dividindo-se o seu peso (em Kg) por sua altura (em m) ao quadrado. Assim, por exemplo, uma pessoa de 1,67m e pesando 55kg tem IMC igual a 20,14, já que:

$$IMC = \frac{peso}{altura^2} = \frac{55kg}{1,67m * 1,67m} = 20,14$$

Escreva um programa que solicite ao utilizador o fornecimento do seu peso em kg e de sua altura em m e a partir deles calcule o índice de massa corpórea do utilizador.

1.6.3.2 Converter horas, minutos e segundos

Escreva um programa que a partir de um determinado número de segundos calcula o número de horas, minutos e segundos correspondentes. Conforme o seguinte exemplo:

$$8053s = 2h + 14m + 13s$$

1.6.3.3 Teorema de Pitágoras

Escreva um programa para determinar a hipotenusa de um triângulo rectângulo, dados os catetos.

1.6.3.4 Converter temperaturas

Escreva um programa que a partir de uma temperatura expressa em graus Fahrenheit (tempF), calcule a temperatura expressa em graus Celsius (tempC). A conversão pode ser realizada de acordo com a fórmula 1.6.2.

$$tempF = 32 + \frac{9 * tempC}{5} \tag{1.6.2}$$

1.6.4 Instruções de Decisão

As instruções de decisão, ou selecção, permitem a selecção em alternância de um ou outro conjunto de acções após a avaliação lógica de uma condição.

1.6.4.1 Decisão binária

A decisão binária permite bifurcar a execução de um programa em dois fluxos distintos, para tal é utilizada instrução `if`. Esta instrução pode ser utilizada de duas formas: `if` e `if-else`.

No primeiro caso, se a condição for verdadeira é executado o `bloco-instruções` caso contrário nada acontece:

```
if (<condição>)
{
    <bloco-instruções>
}
```

No caso do `<bloco-instruções>` ser constituído por uma só instruções não é necessário delimitar essa instrução por chavetas (`{` e `}`).

A listagem 1.8 apresenta um programa codificado em C++, cujo objectivo é escrever no ecrã que o aluno foi aprovado no caso da nota ser superior a `9.5` valores.

Listing 1.8: Estrutura de decisão – `if`

```
1  #include<iostream.h>
2  int main()
3  {
4     double nota;
5     cout<<"Introduza nota: ";
6     cin>>nota;
7     if(nota>=9.5)
8         cout<<"O aluno foi aprovado";
9
10    return 0;
11 }
```

No segundo caso, se a condição for verdadeira é executado o `bloco-instruções 1` senão é executado o `bloco-instruções 2`:

```
if (<condição>)
{
    <bloco-instruções 1>
}
else
{
    <bloco-instruções 2>
}
```

Considere-se o programa em C++ presente na listagem 1.9. Neste programa são lidas as medidas dos lados de uma figura rectangular, sendo que no caso particular de os dois lados serem iguais estamos na presença de um quadrado. Em qualquer um dos casos é apresentada a mensagem correspondente assim como o valor da área.

Listing 1.9: Exemplo de utilização da instrução `if-else`

```
1  #include<iostream.h>
2  int main()
3  {
4    int lado1, lado2, area;
5    cout<<"Introduza medidas dos lados: ";
6    cin>>lado1>>lado2;
7    area=lado1*lado2;
8    if(lado1==lado2)
9      cout<<"Area do quadrado= "<<area;
10   else
11     cout<<"Area do rectangulo= "<<area;
12
13   return 0;
14 }
```

1.6.4.2 Decisão múltipla

A instrução de de decisão múltipla é um caso particular de instruções `if-else` encadeadas.

```
if (<condição 1>)
{
    <bloco-instruções 1>
}
else
{
    if(<condição 2>)
    {
        <bloco-instruções 2>
    }
    else
    {
        if(<condição 3>)
        {
            <bloco-instruções 3>
        }
        else
            ...
    }
}
```

Considere-se o programa da listagem 1.10 em que o objectivo é determinar qual o maior de três números.

Listing 1.10: Determinar o maior três números

```
1 #include<iostream.h>
2
3 int main()
4 {
5   int num1, num2, num3, maximo;
6     cout<<"Introduza numero1,numero2 e numero3: ";
7   cin>>num1>>num2>>num3;
8
9   if(num1>=num2)
10  {
11    if(num1>=num3)
12      maximo=num1;
13  }
14  else //se num1<num2
15  {
16    if(num2>=num3)
17      maximo=num2;
18    else
19      maximo=num3;
20  }
21  cout<<"O numero maior e: "<<maximo;
22
23    return 0;
24 }
```

A instrução `switch` proporciona uma forma especial de tomada de decisões múltiplas. Ela permite examinar os diversos valores de uma expressão compatível com números inteiros e seleccionar o resultado adequado.

```
switch(expressão)
{
   case constante 1:<bloco-instruções 1>
   break;
   case constante 2:<bloco-instruções 2>
   break;
   ....
   default:<bloco-instruções N>
}
```

Considere uma máquina que permite apenas três operações, ligar, desligar e furar. O programa da listagem 1.11 permite modelar o funcionamento da respectiva máquina. Sendo que aquando da digitação das letras: 'L', 'D' e 'F', são apresentados, respectivamente, as mensagens: *Ligar*, *Desligar* e *Furar*. No caso da letra digitada ser outra é apresentada mensagem de erro.

Listing 1.11: Exemplo da instrução `switch`

```
1 #include<iostream.h>
2
3 int main()
4 {
5   char letra;
6   cout<<"Introduza letra (L/D/F):";
7     cin>>letra;
8   switch(letra)
9   {
10    case 'L':
11      cout<<"Ligar";
12      break;
13    case 'D':
14      cout<<"Desligar";
15      break;
16    case 'F':
17      cout<<"Furar";
18      break;
19    default:
20      cout<<"Operacao invalida";
21  }
22
23    return 0;
24 }
```

1.6.5 Prioridade dos operadores

Na avaliação de expressões são complexas, com vários operandos e operadores, é fundamental saber qual a ordem pela qual a expressão será avaliada. A tabela

1.15 apresenta a relação dos principais operadores da linguagem C++. Note-se que alguns dos operadores referidos na tabela serão posteriormente apresentados.

Tabela 1.15: Relação de precedência dos operadores

Prior.	Operador	Descrição	Sentido
1	(,), [,], ->, ., sizeof		⟶
2	++, - -	incremento/decremento	⟵
	~	complemento para um	⟵
	!	negação	⟵
	& , *	referência/ponteiro	⟵
	(tipo de dados)	conversão de tipos de dados	⟵
	+, -	sinal	⟵
3	*, /, %	operadores aritméticos	⟶
4	+, -	operadores aritméticos	⟶
5	< <, > >	operador de deslocamento (*bit* a *bit*)	⟶
6	<, <=, >, >=	operadores relacionais	⟶
7	==, !=	operadores relacionais	⟶
8	&, ^, |	operadores de manipulação de *bits*	⟶
9	&&, ||	operadores lógicos	⟶
10	?, :	condicional	⟵
11	=, +=, -=, *=, /=, %=, > >=, < <=, &=, ^=, |=	atribuição	⟵
12	,	vírgula, separador	⟶

No entanto pode-se resolver algumas dúvidas em relação à precedência e sequência de avaliação com a utilização de parêntesis((,)).

1.6.6 Exercícios Resolvidos

Nesta secção são apresentados alguns problemas e respectivas soluções com o objectivo de ilustrar a utilização de instruções de decisão.

1.6.6.1 Distância euclidiana entre dois pontos

O programa da listagem 1.12 permite realizar o cálculo da distância euclidiana entre dois pontos, sendo que cada ponto é definido pelas coordenadas (x,y). no cálculo da distância pode ser utilizada a fórmula 1.6.3.

$$\text{distância} = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \tag{1.6.3}$$

Caso os pontos sejam coincidentes mostra mensagem *"Pontos Coincidentes"*.

Listing 1.12: Distância euclidiana entre dois pontos

```
1 #include<iostream.h>
2 #include<math.h>
3
```

```
4 int main()
5 {
6   int x1,y1,x2,y2;
7   double distancia;
8   cout<<"Coordenadas ponto1(x/y): ";
9   cin>>x1>>y1;
10  cout<<"Coordenadas ponto2(x/y): ";
11  cin>>x2>>y2;
12
13  distancia=sqrt(pow((x2-x1),2)+pow((y2-y1),2));
14
15  if ((int)distancia==0)
16    cout<<"Os pontos sao coincidentes"<<endl;
17  else
18    cout<<"Distancia="<<distancia<<endl;
19  return 0;
20 }
```

1.6.6.2 Classificar em função da média

O programa da listagem 1.13 permite ler as notas de um aluno às disciplinas de Matemática, Português, Inglês e Geografia e calcular a média. Em função da média mostra uma mensagem com o conteúdo *"Aprovado"* ou *"Reprovado"*. Consideram-se notas positivas as notas iguais ou superiores a 9,5.

Listing 1.13: Calcular a média

```
1 #include<iostream.h>
2
3 int main()
4 {
5   int mat,por,ing,geo;
6   double media;
7   cout<<"Introduza as notas(mat/por/ing/geo): ";
8   cin>>mat>>por>>ing>>geo;
9   media=(double)(mat+por+ing+geo)/4;
10
11  if (media>=9.5)
12    cout<<"Aprovado"<<endl;
13  else
14    cout<<"Reprovado"<<endl;
15  return 0;
16 }
```

1.6.6.3 Determinar o máximo de 3 valores

O programa da listagem 1.14 permite determinar o maior de três números.

Listing 1.14: Máximo de três valores

```
1 #include<iostream.h>
2
3 int main()
```

```
4 {
5     int A,B,C,maximo;
6     cout<<"Introduza numero1, numero2, numero3: ";
7     cin>>A>>B>>C;
8     if(A>=B)
9     {
10        if(A>=C)
11            maximo=A;
12    }
13    else
14    {
15        if(B>=C)
16            maximo=B;
17        else
18            maximo=C;
19    }
20    cout<<"O numero maior e: "<<maximo;
21    return 0;
22 }
```

Sugestão: Baseando-se nas soluções propostas escreva um programa que permita a determinação do máximo entre 5 números. Qual é a solução mais elegante?

1.6.6.4 Determinar triângulo válido

O programa da listagem 1.15 permite ler três pontos geométricos e determinar se estes formam um triângulo. Pode ser utilizada a fórmula da distância entre dois pontos para calcular as medidas dos lados do triângulo. Note-se que um triângulo só é válido se a medida de cada um dos seus lados é menor que a soma dos lados restantes.

Listing 1.15: Triângulo válido

```
1 #include<iostream.h>
2 #include<math.h>
3
4 int main()
5 {
6   int x1,y1,x2,y2,x3,y3;
7   double lado1,lado2,lado3;
8   cout<<"Coordenadas ponto1(x/y): ";
9   cin>>x1>>y1;
10  cout<<"Coordenadas ponto2(x/y): ";
11  cin>>x2>>y2;
12  cout<<"Coordenadas ponto3(x/y): ";
13  cin>>x3>>y3;
14
15
16  lado1=sqrt(pow((x2-x1),2)+pow((y2-y1),2));
17  lado2=sqrt(pow((x3-x1),2)+pow((y3-y1),2));
18  lado3=sqrt(pow((x3-x2),2)+pow((y3-y2),2));
19
20  if(lado1<(lado2+lado3)
```

```
21      && lado2<(lado1+lado3)
22      && lado3<(lado1+lado2))
23        cout<<"triangulo valido"<<endl;
24    else
25        cout<<"Os pontos não forman um trianguo"<<endl;
26    return 0;
27 }
```

1.6.7 Exercícios Propostos

Nesta secção são propostos alguns problemas com vista à aplicação de instruções de decisão.

1.6.7.1 Classificar triângulo

Classificar um triângulo quanto aos lados, sendo que um triângulo com todos lados iguais é designado *Equilátero*, com todos os lados diferentes entre si é designado *Escaleno* e caso tenha apenas dois lados iguais entre si, designa-se *Isósceles*.

1.6.7.2 Divisão

Escreva um programa que dados dois valores, divide o primeiro pelo segundo. Note que não é possível fazer a divisão por zero, neste caso deve ser apresentada a mensagem adequada.

1.6.7.3 Resolver equação da forma $ax^2 + bx + c = 0$

Calcular as raízes de uma equação na forma $ax^2 + bx + c = 0$. Note que os valores a, b e c podem ser zero, podendo dar origem a equações sem solução ou equações de primeiro grau. Considere as fórmulas 1.6.4 e 1.6.5 na resolução do problema.

$$\mathsf{binómio} = b^2 - 4ac \qquad (1.6.4)$$

$$\mathsf{x} = \frac{-b \mp \sqrt{\mathsf{binómio}}}{2a} \qquad (1.6.5)$$

1.6.7.4 Converter entre escalas de temperaturas

Escrever um programa que faça conversões entre as três escalas de temperaturas, Kelvin, Celsius e Fahrenheit, com base em três valores de entrada: a temperatura e escala actual e escala pretendida. Conforme o seguinte exemplo:

As entradas 38, 'C' e 'K', significam que o utilizador pretende converter a temperatura 38 Celsius para Kelvin. Considere as fórmulas 1.6.6 e 1.6.7 na resolução do programa.

$$\mathsf{tempF} = 32 + \frac{9 * \mathsf{tempC}}{5} \qquad (1.6.6)$$

$$\mathsf{tempC} = \mathsf{tempK} + 273 \qquad (1.6.7)$$

Sugestão: Tentar a resolução com as estruturas `se-então-senão` e alternativamente utilizar a estrutura de múltipla decisão.

1.6.7.5 Calcular índice de massa corpórea (IMC)

O índice de massa corpórea (IMC) de um indivíduo é obtido dividindo-se o seu peso (em Kg) por sua altura (em m) ao quadrado. Assim, por exemplo, uma pessoa de 1,67 m e pesando 55 Kg tem IMC igual a 20,14, já que:

$$IMC = \frac{peso}{altura^2} = \frac{55kg}{1,67m * 1,67m} = 20,14$$

IMC	**Interpretação**
Até 18,5 (inclusive)	Abaixo do peso normal
De 18,5 a 25 (inclusive)	Peso normal
De 25 a 30 (inclusive)	Acima do peso normal
Acima de 30	Obesidade

Tabela 1.16: Índice de massa corpórea

Considerando a tabela 1.16, escreva um programa que leia o peso em kg e a altura em m de uma determinada pessoa de forma a calcular o índice de massa corpórea do mesmo e de seguida, estabeleça as comparações necessárias entre o IMC calculado e os valores da tabela 1.16 e escreva uma das frases, conforme for o caso:

- *Você está abaixo do peso normal*;
- *O seu peso está na faixa de normalidade*;
- *Você está acima do peso normal*;
- *Você precisa de perder algum peso*.

1.6.7.6 Determinar ano bissexto

Um ano é bissexto se é divisível por 4, excepto se, além de ser divisível por 4, for também divisível por 100. Então ele só é bissexto se também for divisível por 400. Escrever um programa que leia o valor de um ano e escreva se o ano é ou não bissexto.

1.6.7.7 Parque de estacionamento

Considere um parque de estacionamento que pratica os preços seguintes:

- 1ª hora: 2 €;
- 2ª hora: 1,5 €;
- a partir da 2ª hora: 1 €/hora.

O tempo de permanência no parque é contabilizado em horas e minutos. Por exemplo, se uma viatura permanecer 2 horas e 30 minutos no parque, pagará 2 € (1ª hora) + 1,5 € (2ª hora) + 0,5 € (30 minutos a 1 €/hora) = 4 €.

Elabore um programa que, lido o tempo que determinada viatura permaneceu estacionada no parque, diga a quantia que deve ser paga.

1.6.8 Instruções de Repetição

As instruções de repetição, ou ciclos, permitem a execução, de forma repetitiva, de um conjunto de instruções. Esta execução depende do valor lógico de uma condição que é testada em cada iteração para decidir se a execução do ciclo continua ou termina.

A linguagem de programação C++ compreende três estruturas distintas de controlo do tipo cíclico: `do-while`, `while` e `for`.

1.6.8.1 Instrução `do-while`

O ciclo `do-while` é um ciclo condicional que executa a iteração enquanto uma condição for verdadeira. Esta condição é avaliada no fim. A seguir apresenta-se a sintaxe da estrutura `do-while`. O controlo do ciclo processa-se da seguinte forma. O `<bloco-instruções>` é sempre executado pelo menos uma vez de pois é avaliada a `<condição>`. Caso o resultado da avaliação seja verdade o ciclo continua até que o resultado da avaliação seja falso. Portanto quando o resultado da avaliação da condição for falso o ciclo termina.

```
do
{
    <bloco-instruções>
}
while (<condição>);
```

Considere-se o seguinte exemplo em cuja utilização da estrutura `do-while` permite garantir que o valor da nota introduzida está situado entre `0` e `20`.

O programa da listagem 1.16 apresenta uma solução para o problema anterior utilizando a estrutura `do-while`. Neste programa o ciclo é executado uma vez e caso o valor introduzido pelo utilizador (armazenado na variável nota) seja inferior a `0` ou superior a `20` a ciclo continua até que o valor da nota seja válido, isto é, entre `0` e `20`.

Listing 1.16: Exemplo da instrução `do-while`

```
1  #include<iostream.h>
2
3  int main()
4  {
5    int nota;
6    do
7    {
8      cout<<"Introduzir nota entre 0-20: ";
9      cin>>nota;
10   }
11   while(nota<0 || nota>20);
12
13   return 0;
14 }
```

1.6.8.2 Instrução `while`

O ciclo `while` é também um ciclo condicional que realiza a iteração enquanto uma determinada condição for verdadeira. A condição é avaliada no inicio da iteração. Por isso, o ciclo while pode não realizar nenhuma iteração. A seguir apresenta–se a sintaxe da estrutura `while`. O controlo do ciclo processa–se da seguinte forma. A avaliação da condição é feita no inicio e caso o resultado da seja verdade então o `<bloco-instruções>` é executado. No caso do resultado ser falso então o ciclo termina.

```
while (<condição>)
{
    <bloco-instruções>
}
```

Considere-se o seguinte exemplo em cuja utilização da estrutura `while` permite calcular e escrever a tabuada de um número. A seguir apresenta–se uma possível codificação (1.17) em C++ do problema anterior. Neste exemplo o ciclo `while` vai fazer `10` iterações, uma vez que a variável `i` é iniciada a `1`, em cada iteração esta é incrementada em um valor e o ciclo só termina quando a variável `i` for `>10`.

Listing 1.17: Exemplo da instrução `while`

```
1  #include<iostream.h>
2
3  int main()
4  {
5    int numero, resultado, i;
6    cout<<"Introduza o numero: ";
7    cin>>numero;
8    i=1;
9    while(i<=10)
10   {
11     resultado=numero * i;
12     cout<<numero<<" * "<<i<<" = "<<resultado<<endl;;
13     i++;
14   }
15
16   return 0;
17 }
```

1.6.8.3 Instrução `for`

O ciclo `for` é bastante versátil porque aceita tanto iterações fixas como condicionais. Esta instrução é constituída por três elementos (todos opcionais). O primeiro elemento é a iniciação das variáveis de controlo das iterações. A iniciação só acontece na primeira iteração. O segundo elemento é a condição de controlo das iterações. Uma iteração só acontece se a condição for verdadeira. A condição é sempre avaliada. O terceiro elemento é usado para actualizar as variáveis de controlo do ciclo. Este elemento não é executado na primeira iteração, só nas subsequentes. Um aspecto muito importante é que a avaliação acontece sempre depois dos outros elementos.

Na primeira iteração, as variáveis são iniciadas e só depois é avaliada a condição. Nas iterações subsequentes as variáveis são actualizadas e só depois é feita a avaliação. Obviamente, a iteração só acontece se a condição for verdadeira. Embora não seja muito frequente, pode acontecer que nenhuma iteração seja efectuada.

```
for(<iniciação>;<condição>;<actualização>)
{
    <bloco-instruções>
}
```

Considere–se o seguinte exemplo em cuja utilização da instrução `for` permite calcular a soma os 100 primeiros números inteiros.

Neste exemplo é introduzido um conceito importante para a programação, o conceito de acumulador. A variável `soma` em cada iteração é adicionada do valor da variável `i`, permitindo que no final:

$$soma = 1 + 2 + 3 + 4 + 5 + ... + 100 = 5050$$

Por outro lado, a instrução `i++;` faz com que a variável `i` tome todos os valores inteiros de 1 a 100.

Listing 1.18: Exemplo da instrução `for`

```
1  #include<iostream.h>
2
3  int main()
4  {
5    int soma=0,i;
6    for(i=1;i<=100;i++)
7      soma+=i;
8    cout<<soma;
9    return 0;
10 }
```

Instrução `break`

A instrução `break` permite sair de um ciclo mesmo que a condição ainda seja verdadeira. O código 1.19 apresenta um exemplo de um ciclo que é interrompido pela instrução `break`. O ciclo começa com a variável `n =10` e enquanto `n` for `>0` vai executar o bloco de instruções. De cada iteração a variável `n` é decrementada numa unidade. O bloco de instruções tem uma estrutura de decisão que quando o `n` for igual a `3` vai escrever uma mensagem no ecrã e executar a instrução `break`. O que significa que o ciclo vai terminar quando a variável `n` é `3`, portanto maior que `0`.

Listing 1.19: Exemplo da instrução `break`

```
1  #include <iostream.h>
2
3  int main ()
4  {
5    int n;
6    for (n=10; n>0; n--)
7    {
8      cout << n << ", ";
```

```
9      if (n==3)
10     {
11       cout << "terminou o ciclo!";
12       break;
13     }
14   }
15   return 0;
16 }
```

Instrução continue

A instrução `continue` faz com que o ciclo passa para a iteração seguinte e as instruções que estão a seguir não sejam executadas. A listagem seguinte apresenta um exemplo com a instrução `continue`. Neste exemplo todos os números `10` a `1` são escritos no ecrã com excepção do `5`. Isto porque, quando `n` é igual a `5` a instrução `continue` é executada e por conseguinte a linha de código `cout << n << ", ";` não é executada.

Listing 1.20: Exemplo da instrução `continue`

```
1 #include <iostream.h>
2
3 int main ()
4 {
5   int n;
6   for (n=10; n>0; n--)
7   {
8     if (n==5)
9       continue;
10    cout << n << ", ";
11  }
12  return 0;
13 }
```

1.6.9 Exercícios Resolvidos

Nesta secção são apresentados alguns problemas e respectivas soluções com o objectivo de ilustrar a utilização de instruções cíclicas. Nas soluções são exploradas situações com utilização simples dos ciclos e/ou imbricados.

1.6.9.1 Calcular somatório entre dois limites

O programa da listagem 1.21 permite calcular a somatório dos números existentes num intervalo definido por limites inferior e superior. Note que o utilizador pode introduzir os limites na ordem que entender, desta forma os intervalos [5-10] e [10-5] são igualmente válidos.

Listing 1.21: Calcular somatório entre dois limites

```
1 #include<iostream.h>
2
3 int main()
```

```
4 {
5    int lim1,lim2, min,max, soma,i;
6    cout<<"Introduza numero1: ";
7    cin>>lim1;
8    cout<<"Introduza numero2: ";
9    cin>>lim2;
10   if(lim2>lim1)
11   {
12      min=lim1;
13      max=lim2;
14   }
15   else
16   {
17      min=lim2;
18      max=lim1;
19   }
20   soma=0;
21
22   for(i=min;i<=max;i++)
23      soma+=i;
24
25   cout<<"Somatorio["<<min<<","<<max<<"]: "<<soma;
26
27   return 0;
28 }
```

1.6.9.2 Calcular factorial de um número

O programa da listagem 1.22 permite calcular o factorial de um número sabendo que:

$$\texttt{factorial}(n) = \begin{cases} n = 0 & \rightarrow & 1 \\ n \geq 1 & \rightarrow & n * \texttt{factorial}(n-1) \end{cases}$$

Exemplo: `factorial`(5)=5*4*3*2*1=120

Listing 1.22: Calcular factorial de um número

```
1 #include<iostream.h>
2
3 int main()
4 {
5    int factorial,i,numero;
6    cout<<"Introduza numero: ";
7    cin>>numero;
8    factorial=1;
9    for(i=1;i<=numero;i++)
10      factorial*=i;
11
12   cout<<"Factorial("<<numero<<"): "<<factorial;
13
14   return 0;
15 }
```

1.6.9.3 Determinar se um número é primo

Um número é primo se for apenas divisível por si próprio e pela unidade, por exemplo: 11 é número primo (visto que é apenas divisível por 11 e por 1), enquanto que 21 não é primo, pois tem os seguintes divisores: 1,3,7 e 21. O programa da listagem 1.23 permite determinar se um número é ou não primo.

Listing 1.23: Determinar se um número é primo

```
#include<iostream.h>

int main()
{
  int i,numero;
  bool primo=true;
  cout<<"Introduza numero: ";
  cin>>numero;
  i=2;
  while(primo==true && i<=(numero/2))
  {
    if(numero%i==0)
      primo=false;
    i++;
  }
  if (!primo)
    cout<<"O numero "<<numero<<" nao e primo"<<endl;
  else
    cout<<"O numero "<<numero<<" e primo"<<endl;

  return 0;
}
```

1.6.9.4 Determinar número e idade da pessoa mais nova de um grupo

O programa da listagem 1.24 permite ler o número e a idade de uma série de pessoas. Este programa deve terminar quando for introduzido o número da pessoa = `0`. No final deve ser mostrado o número e idade da pessoa mais nova.

Listing 1.24: Pessoa mais nova

```
#include<iostream.h>

int main()
{
  int numero,idade,menorNumero=0,menorIdade=0;
  bool primeiro=true;
  do
  {
    cout<<"Introduza numero: ";
    cin>>numero;
    if(numero>0)
    {
      cout<<"Introduza a idade: ";
```

```
14          cin>>idade;
15          if(!primeiro)
16          {
17             if(idade<menorIdade)
18             {
19                menorNumero=numero;
20                menorIdade=idade;
21             }
22          }
23          else
24          {
25             menorNumero=numero;
26             menorIdade=idade;
27             primeiro=false;
28          }
29       }
30    }
31    while(numero!=0);
32
33    if (primeiro)
34       cout<<"Nao foram inseridos elementos"<<endl;
35    else
36    {
37       cout<<"O numero e idade da pessoa mais nova: ";
38       cout<<menorNumero<<","<<menorIdade<<endl;
39    }
40    return 0;
41 }
```

1.6.9.5 Determinar o aluno melhor classificado e a média das notas de uma turma

O programa da listagem 1.25 permite ler as notas de português obtidas pelos elementos de uma turma. Este programa termina quando for introduzido o numero do aluno `0`. No final deve ser mostrado o número do aluno melhor classificado e a média de notas de turma.

Listing 1.25: Melhor aluno e média das notas

```
1 #include<iostream.h>
2
3 int main()
4 {
5    int numero, nota, melhorNumero=0, melhorNota=0;
6    int somaNotas=0, numAlunos=0;
7    bool primeiro=true;
8    double media=0.0;
9    do{
10      cout<<"Introduza numero: ";
11      cin>>numero;
12      if(numero>0)
13      {
```

```
14          numAlunos++;
15          do
16          {
17             cout<<"Introduza a nota: ";
18             cin>>nota;
19          }
20          while(nota<0 || nota>20);
21
22          somaNotas+=nota;
23
24          if(!primeiro)
25          {
26             if(nota>melhorNota)
27             {
28                melhorNumero=numero;
29                melhorNota=nota;
30             }
31          }
32          else
33          {
34             melhorNumero=numero;
35             melhorNota=nota;
36             primeiro=false;
37          }
38       }
39    }
40    while(numero!=0);
41    if(numAlunos>0)
42    {
43       media=(double)somaNotas/numAlunos;
44       cout<<"Numero do melhor aluno: "<<melhorNumero<<endl;
45       cout<<"Nota do melhor aluno: "<<melhorNota<<endl;
46       cout<<"Media das notas: "<<media<<endl;
47    }
48    else
49       cout<<"Nao foram inseridos notas"<<endl;
50
51    return 0;
52 }
```

1.6.10 Exercícios Propostos

Nesta secção são propostos alguns problemas com vista à aplicação dos diferentes tipos de instruções anteriormente introduzidas com particular ênfase na instruções cíclicas.

1.6.10.1 Divisão através de subtracções sucessivas

O resultado da divisão inteira de um número inteiro por outro número inteiro pode sempre ser obtido utilizando–se apenas o operador de subtracção. Assim, se quiser-

mos calcular (7/2), basta subtrair o dividendo (2) ao divisor (7), sucessivamente, até que o resultado seja menor do que o dividendo.

O número de subtracções realizadas corresponde ao quociente inteiro, conforme o exemplo seguinte:

$$\begin{aligned} 7-2 &= 5 \\ 5-2 &= 3 \\ 3-2 &= 1 \end{aligned}$$

Descrever um programa para o cálculo da divisão de um inteiro pelo outro. Note que se o dividendo for zero, esta é uma operação matematicamente indefinida.

1.6.10.2 Determinar o máximo e mínimo de uma série

Ler 100 valores e determinar os valores máximo e mínimo da série.

1.6.10.3 Determinar quantidade de números primos

Determinar quantos são os números primos existentes entre os valores 1 e 1000 (excluindo os limites do intervalo).

1.6.10.4 Determinar se um número é perfeito

Um número n é perfeito se a soma dos divisores inteiros de n (excepto o próprio n) é igual ao valor de n. Por exemplo, o número 28 tem os seguintes divisores: 1, 2, 4, 7, 14, cuja soma é exactamente 28. (Os seguintes números são perfeitos: 6, 28, 496, 8128.)

Escreva um programa que verifique se um número é perfeito.

1.6.10.5 Calcular potência por multiplicações sucessivas

Escrever um programa que permita calcular uma potência do tipo $\mathsf{base}^{\mathsf{expoente}}$ através de multiplicações sucessivas. Por exemplo: $2^4 = 2 * 2 * 2 * 2$. Considere as diferentes situações relacionadas com os valores da base e/ou expoente iguais a zero.

1.6.10.6 Maior número ímpar de uma sequência de valores

Escreva um programa que lê uma sequência de números inteiros terminada pelo número zero e calcule o maior ímpar e a sua posição na sequência de valores.

1.6.10.7 Algarismos de um número

Escreva um programa para extrair os algarismos que compõem um número e os visualize individualmente.

1.6.10.8 Apresentação gráfica de temperaturas

Escreva um programa que leia a temperatura de N cidades portuguesas e que represente a temperatura de cada uma delas com uma barra de asteriscos (*), em que cada asterisco representa um intervalo de 2°C. De acordo com os exemplos seguintes:

Porto	11	*****
Lisboa	16	********
Faro	20	**********
Chaves	8	****

1.6.10.9 Soma dos algarismo de um número

Escreva um programa que calcule a soma dos algarismos que compõem um número. Por exemplo: 7258 = 7+2+5+8 = 22

1.6.10.10 Jogo de adivinhar o número

Escrever um programa para o o jogo de adivinhar um número. Este jogo consiste no seguinte: o programa sorteia um número e o jogador deve tentar adivinhar o número sorteado. Para isso o programa deve indicar se o palpite do jogador foi maior, menor ou se acertou no número sorteado. Caso o jogador acerte deve visualizado no ecrã o número de tentativas utilizadas.

1.6.10.11 Capicua de um número

Escreva um programa que leia um número inteiro positivo e verifique se se trata de uma capicua, isto é, uma sequência de dígitos cuja leitura é a mesma nos dois sentidos (exemplo:32523). Sugestão: Inverter a ordem dos dígitos e verificar se o número obtido coincide com o original. Por exemplo, 327 invertido é ((7*10)+2)*10+3=723.

1.6.10.12 Conversão de base numérica

Elaborar um programa para converter um número escrito em binário para o correspondente na base decimal. A conversão faz–se de acordo com o exemplo seguinte:

$$
\begin{aligned}
10110011_{(2)} &= \\
&= \mathbf{1} * 2^7 + \mathbf{0} * 2^6 + \mathbf{1} * 2^5 + \mathbf{1} * 2^4 + \mathbf{0} * 2^3 + \mathbf{0} * 2^2 + \mathbf{1} * 2^1 + \mathbf{1} * 2^0 \\
&= 128 + 0 + 32 + 0 + 16 + 0 + 0 + 2 + 1 \\
&= 179_{(10)}
\end{aligned}
$$

Note que os expoentes das potências na fórmula de conversão correspondem, respectivamente, à posição ocupada por cada algarismo no número em binário. Sendo que o algarismo mais à direita corresponde à posição zero.

1.7 Funções

As funções permitem estruturar o código de uma forma mais modular, aproveitando as potencialidades da programação estruturada que a linguagem C++ oferece.

Uma função é um bloco de instruções que é executado quando é chamada em alguma parte do programa. A sintaxe de uma função é a seguinte:

```
<tipo-de-dados>  <id-da-função>( <argumento 1>,<argumento 2>,...)
 {
```

```
    <bloco-de-instruções>
}
```

na qual:

- `<tipo-de-dados>` – este é o tipo de dados devolvido pela função;
- `<id-da-função>` – este é o identificador pela qual a função é conhecida. As regras para definição de um identificador são as mesmas que para as variáveis;
- `<argumento 1>, <argumento 2>,...` – estes são os argumentos da função (o número de argumentos é variável, pode ser zero). Cada argumento consiste num tipo de dados seguido do identificador pelo qual esse argumento vai ser identificado dentro da função (por exemplo `int x`). Um argumento é como uma declaração de uma variável dentro da função. Os argumentos permitem passar parâmetros para dentro de uma função quando esta é invocada. Os diferentes parâmetros tem que ser separados pelo operador vírgula (`','`);
- `<bloco de instruções>` – é o corpo da função e tem que estar sempre delimitado por chavetas(`{}`). É constituído por zero ou mais instruções.

A listagem 1.26 apresenta um exemplo de uma função. Para examinar este código interessa lembrar que um programa em C++ começa a sua execução pela função `main`.

Listing 1.26: Exemplo da utilização de funções

```
1  #include <iostream.h>
2
3  int soma (int n1, int n2)
4  {
5      int r;
6      r=n1+n2;
7
8      return r;
9  }
10
11 int main ()
12 {
13     int a,b,z;
14     a=5;
15     b=3;
16     z = soma (a,b);
17     cout << "Resultado: " << z;
18
19     return 0;
20 }
```

Na função `main` é declarada uma variável do tipo `int` com o identificador `z`. A seguir é chamada a função `soma`. Os parâmetros tem uma correspondência clara com os argumentos. Na função `main` é invocada a função `soma` passando–lhe dois valores `5` e o `3` que na função `soma` correspondem às variáveis `n1` e `n2`. Quando a função `soma` é invocada o fluxo de execução sai função `main` e passa para a função

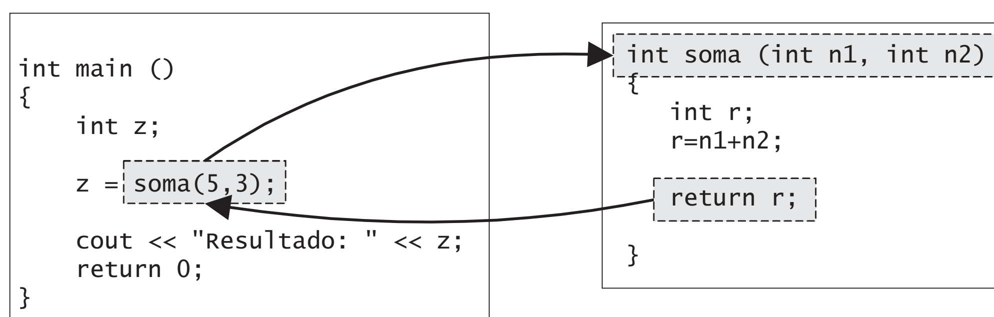


Figura 1.3: Execução de uma função

soma (ver figura 1.3). Os valores são copiados para as variáveis `int n1` e `int n2` da função `soma` .

Na função `soma` é definida a variável `r` do tipo `int` à qual é atribuído o resultado da operação `a` mais `b`. Neste caso como a variável `a` tem o valor `5` e a variável `b` tem o valor `3` o resultado é `8`. Portanto a variável `r` tem o valor `8`. A seguinte linha
`return r;`
termina a função `soma` e retorna o controlo do fluxo de execução à função `main`, que recomeça onde foi interrompida pela chamada à função `soma`. Além disso, a instrução `return` retorna o valor de `r`. Este valor retornado pela função `soma` é atribuído à variável `z`;

Uma função não tem obrigatoriamente que ter argumentos e pode não retornar nada. Para os casos em que a função não retorna nenhum valor o tipo da função é `void`, neste caso a função é também designada de procedimento. A seguir (1.27) apresenta–se uma função que não tem argumentos nem retorna qualquer valor.

Listing 1.27: Exemplo de uma função `void`

```
1 #include <iostream.h>
2
3 void imprime (void)
4 {
5   cout<<"Isto e uma funcao!";
6 }
7
8 int main ()
9 {
10   imprime ();
11   return 0;
12 }
```

Apesar da função imprime não receber nenhum parâmetros é necessário colocar os parêntesis ((,)).

1.7.1 Âmbito da variáveis – global e local

Em relação ao âmbito de uma variável, a linguagem de programação C++ permite dois tipos de variáveis, as variáveis globais e as variáveis locais. Uma variável global é definida fora de qualquer função, inclusive da função `main`. As variáveis locais

são definidas dentro de uma função. Quando uma variável é definida como global significa que está disponível em qualquer parte do programa. Estar disponível significa que em qualquer ponto do programa é possível manipular (ler e escrever) esta variável. Pelo contrário uma variável local só está disponível dentro da função onde está definida. No exemplo da listagem 1.28 é definida uma variável global do tipo int e com o identificador `a` e iniciada com o valor 0.

Listing 1.28: Variáveis locais e globais

```
1 #include <iostream.h>
2 //variavel global
3 int a=0;
4
5 void f1 (int d)
6 {
7 //variavel local
8     int x;
9 //leitura da variavel a
10    x=a+3;
11 //escrever na variavel a
12    a=d+x;
13 }
14 int main ()
15 {
16    //variavel local
17    int z=5;
18    cout<<"Valor de a= "<<a<<endl;
19    f1 (z);
20    cout<<"Valor de a= "<<a<<endl;
21    return 0;
22 }
```

Na função `main` é definida uma variável local (`int z=5;`).
A instrução `cout<<"Valor de a=«<a<<endl;` imprime no ecrã a seguinte frase:

```
Valor de a= 0
```

Na qual `0` refere o valor da variável `a`. A variável `a` não está definida na função `main` logo para que se possa aceder tem que ser uma variável global. Na chamada à função `f1` na função `main` é passado o valor da variável local `z`, que é `5`. Portanto o parâmetro `d` (a variável `d` na função `f1`) é iniciado com o valor `5`. Além da variável `int d` na função `f1` é definida outra variável (`x` do tipo `int`). À variável `x` da função `f1` é atribuído o valor da variável `a`. Esta variável não está definida na função `f1`, mas como é global pode ser acedida para leitura e também para escrita. O que acontece na instrução `a=d+x;` que altera o valor da variável `a` para `5`. Assim que a função `f1` terminar é executada a instrução `cout<<"Valor de a= «<a<<endl;` que imprime no ecrã:

```
Valor de a=5
```

Uma variável global "vive"enquanto o programa estiver em execução. Uma variável local "vive"enquanto a função onde está definida estiver a ser executada.

Sempre que a função for invocada as variáveis locais são criadas e sempre que a função termina estas deixam de existir. Sempre que um programa é executado (passa a processo do Sistema Operativo) é–lhe associado uma quantidade de memória que é dividida em três grandes partes. Uma parte para o código do programa, outra para os dados e uma outra designada de *stack*. A parte do código do programa é onde são colocadas as instruções do programa, obviamente instruções máquina. A parte dos dados pode ser dividida em duas áreas: A área das variáveis globais e estáticas e a *heap*. A *heap* é uma zona de memória reservada à alocação dinâmica de memória em tempo de execução. A *stack* opera segundo o principio último a entrar primeiro a sair (LIFO[4]) e serve para armazenar os argumentos e variáveis passados ao programa. E sempre que uma função é chamada é criada uma *stack frame*. A *stack frame* serve para passar argumentos para a função, guardar o endereço do local onde foi chamada, definir as variáveis locais à função e passar o valor de retorno. Quando a função termina a correspondente stack frame é destruída. A figura 1.4 ilustra a forma como a memória associada a um programa está dividida.

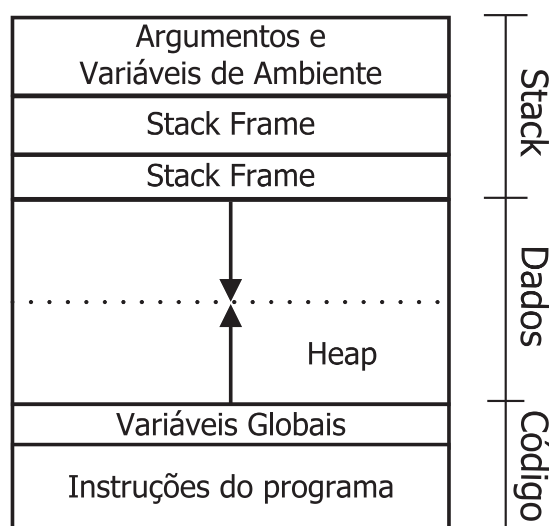


Figura 1.4: Arquitectura do modelo de memória

A figura 1.5 apresenta a evolução da *stack* para o programa que é apresentado na listagem 1.29.

O facto de as variáveis terem o mesmo nome não significa que exista alguma relação entre elas. Não existe nenhuma relação entre as variáveis das diferentes funções nem nas variáveis da mesma função em invocações diferentes.

Listing 1.29: Exemplo de chamadas a funções

```
1 #include <iostream.h>
2
3 void h(int a)
4 {
5   int x;
6   cout<<"Funcao h"<<endl;
7 }
```

[4]do anglo-saxónico *Last In First Out*

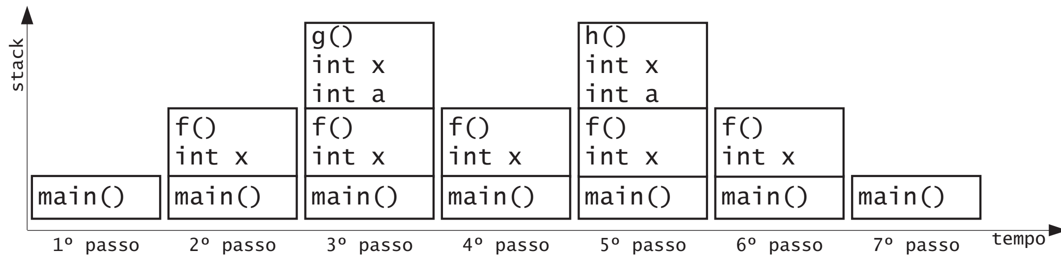


Figura 1.5: Execução da pilha

```
8
9  void g(int a)
10 {
11   int x;
12   cout<<"Funcao g"<<endl;
13 }
14
15 void f()
16 {
17   int x;
18   cout<<"Funcao f"<<endl;
19   g(x);
20   h(x);
21 }
22
23 int main ()
24 {
25   f();
26   return 0;
27 }
```

1.7.2 Passagem de argumentos

Na linguagem C++ os parâmetros podem ser passados para uma função por valor ou por referência.

1.7.2.1 Passagem por valor

Até agora em todas as funções implementadas, a passagem de parâmetros utilizada foi por valor. Isto significa que quando uma função é chamada com parâmetros, o que lhe é passado é o valor da variável (ou o literal). Por exemplo a função `soma(int n1, int n2)` do programa da listagem 1.30 recebe os valores `5` e `3` e não as variáveis `x` e `y`(figura 1.6).

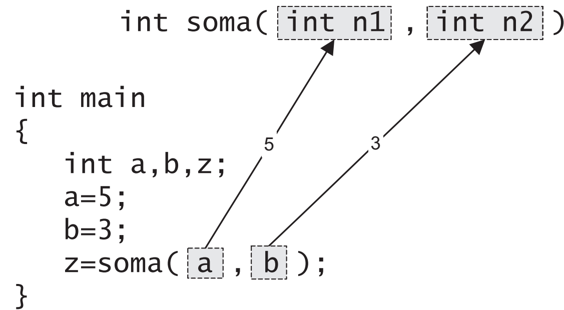


Figura 1.6: Passagem de variáveis por valor

Daí que, qualquer alteração à variáveis `n1` e `n2` na função `soma` não tem qualquer significado nas variáveis `a` e `b` na função `main`.

Listing 1.30: Variáveis locais e globais

```
#include <iostream.h>

int soma (int n1, int n2)
{
    int r;
    r=n1+n2;

    return r;
}

int main ()
{
    int a,b,z;
    a=5;
    b=3;
    z = soma (a,b);
    cout << "Resultado: " << z;

    return 0;
}
```

1.7.2.2 Passagem por referência

Existem situações em que é necessário alterar o valor de uma variável dentro de uma função que foi definida noutra função. Para este caso a variável tem que ser passada por referência. O programa da listagem 1.31 apresenta a função `divisaoInteira` que tem quatro argumentos, dois passados por valor (`int dividendo` e `int divisor`) e dois por referência (`int &quociente` e `int &resto`). A sintaxe da linguagem C++ obriga a que os argumentos passados por referência são identificados por um & entre o tipo de dados e o identificador da variável.

Listing 1.31: Passagem por referência

```
#include<iostream.h>

```

```
3 void divisaoInteira(int dividendo,int divisor,int &quociente,int &resto)
4 {
5     quociente=0;
6     while(dividendo>=divisor)
7     {
8         dividendo=dividendo-divisor;
9         quociente=quociente+1;
10    }
11    resto=dividendo;
12 }
13
14 int main()
15 {
16    int D=23, d=5, q, r;
17
18    divisaoInteira(D,d,q,r);
19
20    cout<<"O quoc. da div. inteira. de "<<D<<" por "<<d<<" e "<<q<<endl;
21    cout<<"O res. da div. inteira de "<<D<<" por "<<d<<" e "<<r<<endl;
22
23    return 0;
24 }
```

Desta forma as variáveis `q` e `r` definidas na função `main` que dentro da função `divisaoInteira` são `quociente` e `resto`, respectivamente. Isto significa que qualquer alteração dentro da função `divisaoInteira` nas variáveis `quociente` ou `resto` altera o valor das variáveis `q` e `r` da função `main` (listagem 1.7).

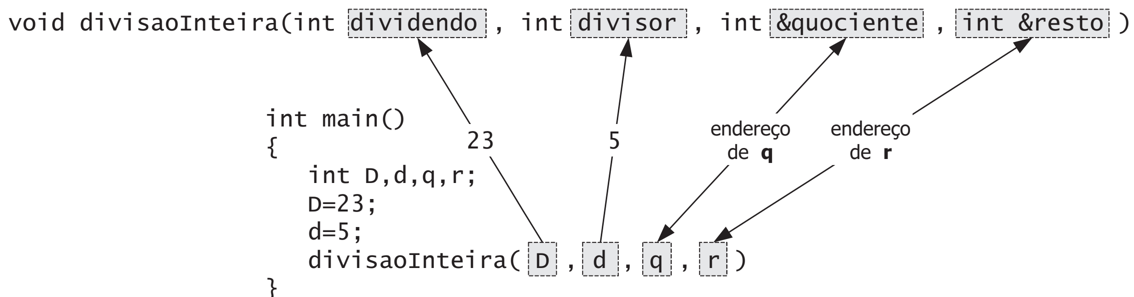


Figura 1.7: Passagem de variáveis por referência

Esta é uma forma de fazer uma função retornar mais do que um valor.

1.7.2.3 Valores por omissão nos argumentos

Quando se declara uma função pode-se especificar valores para cada um dos parâmetros. Para tal basta atribuir valores com o operador = na declaração da função. O programa da listagem 1.32 apresenta a função `divide`, cujo o segundo argumento tem um valor por omissão. Significa isto que a função pode ser chamada de duas formas, com um parâmetro ou com dois. Com um serve para iniciar o argumento `a`, sendo que o `b` está iniciado por omissão. Com dois inicia os dois argumentos.

Listing 1.32: Passagem por referência

```
#include <iostream.h>

int divide(int a, int b=2)
{
    int r;
    r=a/b;
    return r;
}
int main ()
{
    cout<<divide(12)<<endl;
    cout<<divide(20,4)<<endl;
    return 0;
}
```

Deste modo a saída deste programa é `6` e `5`. A chamada `divide(12)` tem como retorno o `6`, resultado da operação 12/6. A chamada (20,4) tem como resultado 5 (20/4).

1.7.3 Protótipos de funções

Nos programas apresentados anteriormente as funções aparecem definidas antes da primeira chamada a essa função. A estrutura usada foi então definir as funções antes da função `main`, portanto esta aparece no fim do ficheiro de código. Mas caso, as funções aparecessem depois da função `main` os programas apresentados anteriormente dariam erro de compilação. No exemplo apresentado no programa da listagem 1.33 o compilador indicaria que não conhecia o identificador da `"divide"`. Isto porque no processo de compilação é feita uma analise sequencial e neste caso quando o compilador encontra o identificador `"divide"` não sabe o que é que ele é.

Listing 1.33: Erro de compilação

```
#include <iostream.h>

int main ()
{
    int x;
    x=divide(20,5);
    return 0;
}

int divide(int a, int b=2)
{
    int r;
    r=a/b;
    return r;
}
```

A solução para este problema consiste em definir os protótipos das funções antes da sua chamada. No protótipo da função é necessário referir qual o tipo ou tipos de dados dos argumentos (respeitando a ordem) e qual o tipo de dados de retorno além do identificador da função. A forma para definir um protótipo é a seguinte:

```
<tipo-de-dados> <id-da-função> (<tipo-de-dados>,<tipo-de-dados>,...);
```

Listing 1.34: Protótipo de função (exemplo)

```
1 #include <iostream.h>
2 //protótipo da função
3 int divide(int , int);
4
5 int main ()
6 {
7     cout<<divide(20,4)<<endl;
8     return 0;
9 }
10
11 int divide(int a, int b)
12 {
13     int r;
14     r=a/b;
15     return r;
16 }
```

Embora não seja necessário colocar o identificador dos argumentos a sua utilização é aconselhável. Efectivamente, em alguns casos poder–se–ia contornar a definição de protótipos, inserindo a definição das função antes da primeira chamada, portanto antes da função `main`. No entanto, esta prática é desaconselhada, pois é ineficaz no caso de chamadas circulares entre funções.

1.7.4 Estrutura de um programa em C++

Um programa em C/C++ deve obedecer a estrutura apresentada na figura 1.8.

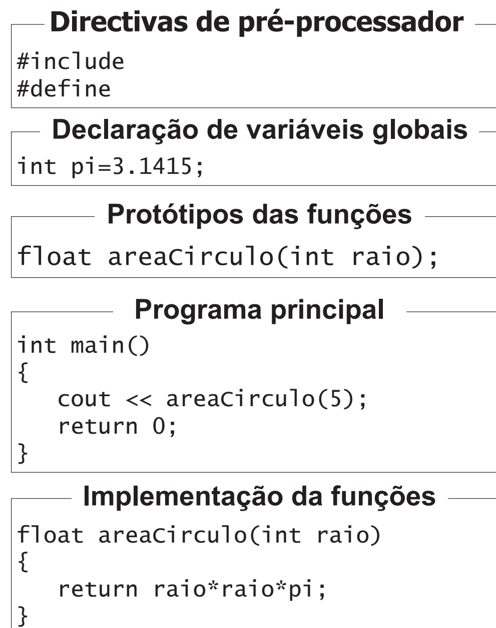


Figura 1.8: Estrutura de um programa em C/C++

No inicio do ficheiro são colocadas as directivas do pré–processador (*e.g.*, `#include`, `#define`). Caso se utilizem variáveis globais, estas devem ser definidas logo a seguir à directivas de pré-processador. Depois deve-se colocar o protótipo de todas as funções implementadas. A seguir á declaração dos protótipos é implementada a função `main` e depois a implementação das funções anteriormente declaradas nos protótipos.

1.7.5 Exercícios resolvidos

Nesta secção são apresentados alguns problemas e respectivas soluções com o objectivo de ilustrar a utilização de funções na produção de programas modulares.

1.7.5.1 Função que devolve o maior algarismo de um número

O programa da listagem 1.35 apresenta uma função que recebe um número inteiro e devolve o maior algarismo contido nesse número.

Listing 1.35: `maior(n)` que devolve o maior algarismo de um número

```
1 #include<iostream.h>
2
3 int maior(int n)
4 {
5   int alg,max=n%10;
6   n/=10;
7   while(n!=0)
8   {
9     alg=n%10;
10    if(alg>max)
11      max=alg;
12    n=(n-alg)/10;
13  }
14  return max;
15 }
16
17 int main()
18 {
19   int numero;
20   cout<<"Introduza a numero= ";
21   cin>>numero;
22
23   cout<<"O maior digito de "<<numero<<" e ";
24   cout<<maior(numero)<<endl;
25
26   return 0;
27 }
```

1.7.5.2 Função que indica se um número é perfeito

Um número `n` é perfeito se a soma dos divisores inteiros de `n` (excepto o próprio `n`) é igual ao valor de `n`. Por exemplo, o número 28 tem os seguintes divisores: 1, 2, 4,

7, 14, cuja soma é exactamente 28. (Os seguintes números são perfeitos: 6, 28, 496, 8128.)

A listagem do programa da listagem 1.36 apresenta uma função que recebe um número inteiro e devolva os valores booleanos `true` ou `false` se o número é ou não perfeito, respectivamente.

Listing 1.36: `perfeito(N)` que indica se um número é perfeito

```
#include<iostream.h>

bool perfeito(int n)
{
  int soma=0,x;
  for(x=1;x<=(n/2);x++)
  {
    if(n%x==0)
      soma+=x;
  }
  if(soma==n)
    return true;
  return false;
}

int main()
{
  int numero;
  cout<<"Introduza a numero= ";
  cin>>numero;

  if(perfeito(numero))
    cout<<"O numero "<<numero<<" e perfeito"<<endl;
  else
    cout<<"O numero "<<numero<<" nao e perfeito"<<endl;

  return 0;
}
```

1.7.6 Exercícios propostos

Nesta secção são propostos alguns problemas relacionados com a utilização de procedimentos e funções na escritas de programas modulares.

1.7.6.1 Função média de dois números

Escreva uma função que, dados dois números reais, retorna a média deles arredondada para um inteiro, e devolve os números por ordem crescente. Faça um programa que permita testar a função anterior.

1.7.6.2 Função lei de Ohm

A lei de Ohm é uma relação entre a corrente (I), a tensão (V) e a resistência (R), de acordo com o circuito eléctrico representado na figura 1.9.

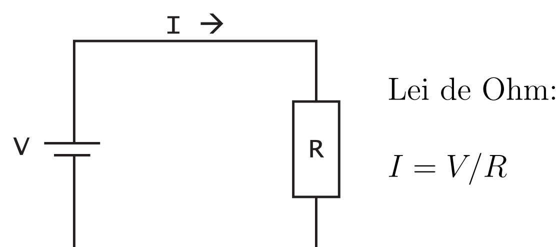


Figura 1.9: Ilustração da lei de Ohm

a) Escreva uma função que recebe os valores de V e R como parâmetros, e calcule a corrente I.

b) Escreva um programa que permita testar a função anterior.

1.7.6.3 Função somatório

Calcular o somatório $\sum_{i=1}^{n} \frac{2^i}{\sqrt{i}}$

Sugestão: crie uma função para determinar cada termo i da série.

1.7.6.4 Funções para codificar e descodificar números

Uma empresa pretende enviar cifrada uma sequência de inteiros decimais de 4 dígitos (`DigDigDigDig`). A cifra consiste em: substituir cada dígito `Dig` por `(Dig+8)%10` (*i.e.*, adiciona 8 e calcula o resto da divisão do resultado por 10); depois troca o terceiro dígito com o primeiro e troca o quarto dígito com o segundo.

a) Escreva uma função que receba um inteiro decimal de 4 dígitos e o devolva cifrado.

b) Escreva uma função que receba um inteiro cifrado e o decifre para o valor original.

c) Escreva uma função que apresente um «menu» com 2 opções, cifrar e decifrar número, peça ao utilizador para escolher uma das opções, e retorne a opção escolhida.

d) Faça um programa que permita testar as funções anteriores.

1.7.6.5 Números primos

Escreva um procedimento que imprima os números primos existentes entre dois números. Na resolução deste problema deve ser utilizada uma função que determina se um número é primo.

1.8 Vectores

Nesta secção é descrita a forma de definir e manipular vectores.

1.8.1 Definição de vectores

Os vectores são conjuntos de elementos (variáveis) do mesmo tipo de dados, colocados consecutivamente na memória. Podem ser acedidos individualmente ou referenciados por indexação. Isto significa que, por exemplo, para guardar vinte valores do tipo `int` não é necessário declarar vinte variáveis. Para tal é possível declarar um vector com vinte posições do tipo `int` com um único identificador. A seguir é apresentada a sintaxe para declarar um vector:

```
<tipo-de-dados> <id-do-vector> [<num-de-elem>]
```

na qual:

- <tipo-de-dados> – indica qual o tipo de dados (*e.g.*, : `int`, `float`, `char`) de cada elemento do vector;
- <id-do-vector> – este é o identificador do vector;
- <num-de-elem> – entre parêntesis rectos ([,]) especifica o número de elementos do vector. O número de elementos tem que ser um valor constante, isto é, não pode ser uma variável, porque os vectores são conjuntos estáticos de memória de um determinado tamanho e o compilador tem que ser capaz de determinar exactamente a quantidade de memória necessaria para o vector antes de qualquer instrução ser considerada.

Por exemplo um vector para armazenar vinte números inteiros (do tipo de dados `int`) pode ser definido da seguinte forma:

```
int vec[20];
```

A figura 1.10 representa graficamente um vector com 20 elementos. Onde cada rectângulo representa um elemento do vector, que neste caso são elementos do tipo `int`. Os elementos estão numerados de `0` a `19`, uma vez que o primeiro elemento de um vector é sempre a posição `0`, independentemente do números de elementos do vector.

Índice	0	1	2	3	4	...	19
Valor	12	8	9	17	15	...	11

Figura 1.10: Representação gráfica do vector `vec`

1.8.2 Atribuição dos valores iniciais

Quando se declara um vector é possível, como acontece com qualquer outra variável, iniciar os seus elementos. Para tal deve-se colocar os elementos separados por virgulas (`','`) dentro de chavetas (`{}`). Como por exemplo:

```
int vec[5]={1,3,2,55,67};
```

O número de elementos deve ser igual ao número de posições do vector. No entanto, é possível não definir o número de elementos do vector, sendo o número de posições definido em função dos elementos que estão dentro das chavetas. A

definição apresentada a seguir é equivalente à definição anterior. Ambos os vectores tem 5 posições.

`int vec[]={1,3,2,55,67};`

1.8.3 Acesso aos elementos de um vector

Em qualquer ponto do programa no qual um vector esteja definido é possível aceder a qualquer elemento do vector, quer para leitura quer para escrita como se fosse uma variável comum. A sintaxe a utilizar é a seguinte:

`<identificador-do-vector> [índice]`

Por exemplo a seguinte instrução permite armazenar o valor `75` na terceira posição do vector `vec`.

`vec[2]=75;`

enquanto a seguinte instrução permite atribuir o valor do terceiro elemento do `vec` a uma variável `a` do tipo `int`.

`a=vec[2];`

Portanto, para todos os propósitos a expressão `vec[2]` é como uma qualquer variável do tipo `int`.

Um aspecto muito importante em C++ está relacionado com a dimensão dos vectores. Quando se define um vector indica–se a sua dimensão, ou seja o número de elementos do vector. Sendo que, a primeira posição do vector tem o índice `0`, o que faz com que a última seja o numero de elementos menos `1`. No caso de ser referenciada uma posição para além da dimensão do vector o compilador de C++ não indicará nenhum erro de compilação. Apenas em tempo de execução o erro será detectado, através da ocorrência de resultados inesperados ou mesmo "crash"do programa. Cabe ao programador, controlar a dimensão do vector.

Nos vectores, os parêntesis rectos (`[`,`]`) são utilizados com dois propósitos que importa distinguir. Primeiro, são usados na definição de vectores para se colocar a dimensão do vector. Segundo, são usados para referenciar os elementos do vector. Na listagem do programa da listagem 1.37 é apresentada as duas formas de utilização dos parêntesis rectos (`[`,`]`).

Listing 1.37: Exemplo da utilização dos parêntesis rectos `[]` nos vectores

```
1  #include<iostream.h>
2  #define MAX 5
3
4  int main()
5  {
6    int vec[MAX]={23,45,32,78,98},i,soma=0;
7    for(i=0;i<MAX;i++)
8      soma+=vec[i];
9    cout<<"Somatorio: "<<soma<<endl;
10
11   return 0;
12 }
```

1.8.4 Exercícios resolvidos

1.8.4.1 Funções manipulando vectores

Faça um programa que inclua:

a) Uma função que faça a leitura de 10 valores (inteiros), guardando–os num vector;

b) Uma função que retorne a diferença entre o maior e o menor valor do vector;

c) Uma função que devolva o número de valores pares e ímpares do vector;

O procedimento apresentado na listagem 1.38 permite realizar a leitura do vector. Note–se que tanto o próprio vector como a respectiva dimensão são passados para o procedimento como argumentos.

Listing 1.38: Leitura (vectores)

```
void leitura( int vec[], int dim)
{
    int i;
    for(i=0;i<dim;i++)
    {
        cout <<"vec[" << i << "]: ";
        cin >> vec[i];
    }
}
```

A função presente na listagem 1.39 permite contabilizar a quantidade de números pares existentes no vector. A função recebe próprio vector e a respectiva dimensão como parâmetros e retorna a quantidade de pares.

Listing 1.39: Conta o número de números pares

```
int contaPares( int vec[], int dim)
{
    int i,npar=0;

    for(i=0;i<dim;i++)
  {
        if(vec[i]%2==0)
            npar++;
    }
    return npar;
}
```

A função apresentada na listagem 1.40, recebe o próprio vector e a respectiva dimensão como parâmetros e retorna a diferença entre os valores máximo e mínimo existentes no vector.

Listing 1.40: Determina a maior diferença entre os elementos de um vector

```
int maiorDiferenca( int vec[],int dim )
{
    int i,  maximo,minimo;
    maximo=minimo=vec[0];
```

```
5      for(  i=1;i<dim;i++)
6    {
7           if(vec[i]>maximo)
8               maximo=vec[i];
9           else
10              if(vec[i]<minimo)
11                  minimo=vec[i];
12     }
13     return  (maximo-minimo);
14 }
```

No seguinte extracto (listagem 1.41) é definido o vector e evocadas as funções e procedimento anteriormente definidos.

Listing 1.41: Função `main` e protótipos das funções

```
1 #include<iostream.h>
2 #define  MAX     10
3
4 void leitura( int vec[], int dim);
5 int contaPares( int vec[], int dim);
6 int maiorDiferenca( int vec[],int dim);
7
8 int main()
9 {
10     int vector[MAX];
11
12     leitura(vector,MAX);
13     cout<<contaPares(vector,MAX)<<endl;
14     cout<<maiorDiferenca(vector,MAX)<<endl;
15
16     return 0;
17 }
```

1.8.5 Exercícios propostos

1.8.5.1 Determinar desvio padrão de uma série

Escreva um programa modular que permita determinar o desvio padrão de um série de números de acordo com a formula 1.8.1. Considere a definição de funções e procedimento para os diversos sub–problemas.

$$\texttt{desvioPadrao} = \sqrt{\frac{\sum_{i=1}^{n}(x_i - \texttt{media})}{n-1}} \tag{1.8.1}$$

1.8.5.2 Prova de atletismo

Faça a leitura das pontuações que 5 juízes de uma determinada prova atribuíram a um atleta (valores compreendidos entre 0 e 9 inclusive). Determine e apresente com formato adequado, os seguintes valores:

- média obtida pelo atleta;
- a pior e a melhor pontuação;
- a percentagem de pontuações iguais ou superiores a 8 valores;
- supondo que a 1ª nota foi atribuída pelo juiz nº1 e assim sucessivamente determine os números dos juízes que atribuíram a melhor nota do atleta.

1.8.5.3 Suavização

Em qualquer experiência existe um certo erro associado aos valores obtidos. Uma técnica conhecida como suavização pode ser utilizada para reduzir o efeito desse erro na análise dos resultados. Escreva então um programa que permita ler os dados para um vector de N reais e implemente uma função que produza uma suavização sobre esses dados. A suavização consiste em substituir o valor actual de uma posição pela média do valor da posição anterior, da posterior e dele próprio. Assumindo que o identificador do vector é `v`, então `v[i]=(v[i-1]+v[i]+v[i+1])/3`, (excepto o primeiro e o último). O primeiro elemento do vector é suavizado com base na média entre os dois primeiros valores e o último elemento é suavizado com base na média entre os dois últimos.

1.9 Vectores multi–dimensionais

Um vector multidimensional pode ser descrito como um vector de vectores. Por exemplo, um vector bidimensional (matriz) pode ser visto como uma tabela bidimensional em que todos os elementos são do mesmo tipo dados. A figura 1.11 faz a representação gráfica de uma matriz.

	0	1	2	...	799
0	4	56	11	...	6
1	12	8	1	...	5
...	...	...	...	...	...
639	5	83	9	...	4

Figura 1.11: Representação gráfica de uma matriz

A variável `mat` representa um vector bidimensional de 3 linhas e 5 colunas. A forma de o definir em C++ é:

```
int mat[3][5];
```

A instrução `mat[1][3]` referencia o elemento da segunda linha e da quarta coluna.

Importa relembrar que os indices começam sempre em `0`. Os vectores multi–dimensionais não estão limitados a duas dimensões. Podem ter as dimensões que o

programador achar necessário. O exemplo seguinte define um vector de três dimensões.

`int multiVec[100][200][50];`

Há contudo um aspecto a ter em consideração na definição de vectores multi–dimensionais, a memória necessária na definição destes. Pois o número de elementos que o vector `multiVec` aloca é obtido pela multiplicação das dimensões e não pela soma, como erroneamente se poderia pensar. Portanto, este vector aloca 100*200*50=1000000 elementos. Sendo que a memória necessária para armazenar 1000000 elementos do tipo `int` é cerca de 4Mbytes (1000000*4bytes).

Note–se que os vectores multi–dimensionais não são mais do que uma abstracção, uma vez que é possível obter os mesmos resultados com um vector simples. Por exemplo:

`int mat[3][5];` é equivalente a `int mat[15];` (3*5=15)

A única diferença consiste no facto do compilador guardar a profundidade de cada dimensão imaginária. A seguir apresenta-se duas listagens de programas, na quais, numa é usado um vector multidimensional (listagem 1.42) e na outra um vector simples (listagem 1.43). Apesar de aparentemente guardarem a informação em estruturas de dados diferentes, na prática são idênticos.

Listing 1.42: Vector multidimensional

```
1  #define COLUNAS 5
2  #define LINHAS 3
3
4  int main ()
5  {
6     int mat [LINHAS][COLUNAS];
7     int n,m;
8     for (n=0;n<LINHAS;n++)
9        for (m=0;m<COLUNAS;m++)
10       {
11          mat[n][m]=(n+1)*(m+1);
12       }
13    return 0;
14 }
```

Listing 1.43: Vector pseudo-multidimensional

```
1  #define COLUNAS 5
2  #define LINHAS 3
3
4  int main ()
5  {
6     int mat [LINHAS * COLUNAS];
7     int n,m;
8     for (n=0;n<LINHAS;n++)
9        for (m=0;m<COLUNAS;m++)
10       {
11          mat[n * COLUNAS + m]=(n+1)*(m+1);
12       }
13    return 0;
14 }
```

É boa prática de programação utilizar macros para especificar a dimensão de um vectores tanto uni como multi-dimensionais. A utilização de macros permite diminuir o esforço de programação quando se pretende alterar as suas dimensões. Isto porque basta alterar as macros e desta forma altera as dimensões dos vectores assim como do código usado para manipular estes vectores. Por exemplo para alterar o numero de linhas de 3 para 4 bastava alterar a macro de
`#define LINHAS 3`
para
`#define LINHAS 4`

1.9.1 Exercícios resolvidos

1.9.1.1 Funções manipulação de matrizes

Escreva um programa que defina uma matriz quadrada de dimensão máxima 4 e implemente as seguintes funções:

1. Uma função que faça a leitura dos elementos da matriz, guardando os valores em memória RAM. O protótipo deve ser `void leitura( int matriz[][MAX_C] );`
2. Uma função que devolva a média da diagonal principal, com o seguinte protótipo (`float mediaDiagonal( int matriz[][MAX_C] )`) ;
3. Uma função que devolva um vector com os elementos cujo valor seja superior à média da diagonal principal, com o seguinte protótipo (`int superiorMedia( int matriz[][MAX_C], int vector[] )`);
4. Uma função que devolva uma matriz com o número de ocorrências de cada elemento, com o seguinte protótipo (`int ocorrencias( int matriz[][MAX_C], int matriz_ocorr[][2])`).

Listing 1.44: Matrizes

```
1  #include<iostream.h>
2  #include<stdlib.h>
3
4  #define MAX    50
5  #define MAX_L 4
6  #define MAX_C 4
7
8  int menu();
9  void leitura( int matriz[][MAX_C] );
10 float mediaDiagonal( int matriz[][MAX_C] );
11 int superiorMedia( int matriz[][MAX_C], int vector[] );
12 int ocorrencias( int matriz[][MAX_C], int matriz_ocorr[][2]);
13 void inc_ocorr( int valor, int m[][2], int *n );
14
15 int main()
16 {
17   int matriz[MAX_L][MAX_C];
18   int vector[MAX_L*MAX_C];
19   int matriz_ocorr[MAX_L*MAX_C][2];
```

```
20    int n_elem;
21    int i,op;
22    do{
23      op=menu();
24      switch(op)
25      {
26        case 1:
27          cout << "Introduza valores para a matriz ";
28          cout<< MAX_L << "x" << MAX_C<< endl;
29          leitura( matriz );
30          break;
31        case 2:
32          cout << "Media da diagonal: ";
33          cout << mediaDiagonal( matriz ) << endl;
34          break;
35        case 3:
36          cout << "Elementos que sao superiores a media da diagonal: ";
37          n_elem = superiorMedia( matriz, vector );
38          for( i=0; i<n_elem; i++ )
39            cout << vector[i] << " ";
40          cout << endl;
41
42          break;
43        case 4:
44          cout << "Ocorrencias: " << endl;
45          n_elem = ocorrencias( matriz, matriz_ocorr );
46          for( i=0; i<n_elem; i++ )
47          {
48            cout << matriz_ocorr[i][0] << ": ";
49            cout<< matriz_ocorr[i][1] << " ocorrencia(s)" << endl;
50          }
51          break;
52      }
53    }while(op!=0);
54    return 0;
55  }
56
57  int menu()
58  {
59    int op;
60    char buffer[MAX];
61    do{
62      cout<<"Menu\n";
63      cout<<"1 - Ler matriz\n";
64      cout<<"2 - Media da diagonal principal\n";
65      cout<<"3 - Elementos superiores a media\n";
66      cout<<"4 - Numero de ocorrencias\n";
67      cout<<"0 - Sair\n";
68      cout<<"\nDigite a opcao: ";
69      cin.getline(buffer,MAX);
70      op=atoi(buffer);
```

```
71    }while(op<0 || op>4);
72    return op;
73 }
74
75 void leitura( int mat[][MAX_C] )
76 {
77    int i, j;
78    char buffer[MAX];
79    for( i=0; i<MAX_L; i++ )
80       for( j=0; j<MAX_C; j++ )
81       {
82          cout << "  mat[" << i << "," << j << "]: ";
83          cin.getline(buffer,MAX);
84          mat[i][j]=atoi(buffer);
85       }
86 }
87
88 float mediaDiagonal( int mat[][MAX_C] )
89 {
90    int i, soma=0;
91
92    for( i=0; i<MAX_L; i++ )
93       soma += mat[i][i];
94
95    return (float) soma / MAX_L;
96 }
97
98 int superiorMedia( int mat[][MAX_C], int vec[] )
99 {
100   int i, j, n=0;
101   float media;
102
103   media = mediaDiagonal( mat );
104
105   for( i=0; i<MAX_L; i++ )
106      for( j=0; j<MAX_C; j++ )
107         if( mat[i][j] > media )
108            vec[n++] = mat[i][j];
109
110   return n;
111 }
112
113 int ocorrencias( int mat[][MAX_C], int mat1[][2])
114 {
115   int i, j, n=0;
116
117   for( i=0; i<MAX_L; i++ )
118      for( j=0; j<MAX_C; j++ )
119         inc_ocorr(mat[i][j], mat1, &n);
120
121   return n;
```

```
122 }
123
124 void inc_ocorr( int valor, int m[][2], int *n )
125 {
126   int i=0;
127   bool inc = false;
128
129   do
130   {
131     if( m[i][0] == valor )
132     {
133       m[i][1]++;
134       inc = true;
135     }
136     i++;
137   }
138   while(!inc && i<*n);
139
140   if( !inc )
141   {
142     m[*n][0] = valor;
143     m[(*n)++][1] = 1;
144   }
145 }
```

1.9.2 Exercícios propostos

1.9.2.1 Máximo local

Um elemento de uma matriz é considerado um máximo local se for superior a todos os seus vizinhos. Escreva um programa que dada uma matriz forneça todos os máximos locais e as respectivas posições, considerando que os elementos da periferia da matriz não podem ser máximos locais.

1.9.2.2 Determinar se uma matriz é simétrica

Escreva um programa que dada uma matriz quadrada de dimensão `n` determine se ela é ou não simétrica. Uma matriz `A` diz–se simétrica se $a_{ij} = a_{ji}$ com $1 \leq i, j \leq n$
.

1.10 Vectores como parâmetros

A programação modular que é possível através da utilização de funções obriga em que em certos casos é necessário passar vectores (uni e multi–dimensionais) para as funções como parâmetro. Na linguagem C++ não é possível passar por valor um vector, isto é, um vector é sempre passado por referência (através do seu endereço). Para passar um vector por parâmetro a única coisa que é necessário fazer é colocar na declaração da função especificar que a função tem um argumento que é um vector. Por exemplo a seguir apresenta-se o protótipo de uma função que recebe um vector por parâmetro.

`void funcao(int v[])`

Neste exemplo os elementos do vector são do tipo `int` e vector dentro da função é identificado pelo identificador `v`. Além do tipo e do identificador é necessário colocar os parêntesis rectos (`[]`), sendo que é opcional colocar a dimensão, isto no caso de ser um vector unidimensional.

O exemplo presente na listagem 1.45 ilustra a vantagem da utilização da passagem de vectores a uma função. A função `void printvector (int v[], int len)` imprime no ecrã os `len` elementos do vector `v`. Importa realçar que a mesma função é usada para imprimir o conteúdo do vector `vector1` e do `vector2`. Na chamada è função `printvector` na função `main` não é especificado a dimensão do vector, basta colocar o nome do vector.

Listing 1.45: Vector como argumento de funções

```
1  #include <iostream.h>
2
3  void printvector (int v[], int len)
4  {
5    for (int n=0; n<len; n++)
6      cout << v[n] << " ";
7    cout << "\n";
8  }
9
10 int main ()
11 {
12   int vector1[] = {5, 10, 15};
13   int vector2[] = {2, 4, 6, 8, 10};
14   printvector (vector1,3);
15   printvector (vector2,5);
16   return 0;
17 }
```

Quando se pretende passar vectores multi–dimensionais como parâmetro a uma função é obrigatório especificar o numero de elementos de cada dimensão com excepção da primeira. A seguir apresenta o formato:

`<tipo-de-dados> <id-do-vec> [][dim][dim][dim]...`

Por exemplo, se se pretende que uma função receba um vector tridimensional é obrigatório especificar pelo menos a 2ª e 3ª dimensão.

`void funcao(int m[][3][4])`

Isto é obrigatório porque o compilador precisa de ser capaz de determinar o número de elementos das 2ª e 3ª dimensões.

1.11 *Strings*

As *strings* são vectores de elementos do tipo `char` e permitem representar palavras, frases e textos. De seguida apresenta–se a definição de uma *string* com capacidade para 20 caracteres.

`char str[20];`

Está convencionado que uma *string* é terminada pelo carácter `'\0'`, assim, na prática a *string* acima definida tem capacidade para 19 caracteres + 1. Portanto no

dimensionamento de um *strings* é necessário ter em atenção que é preciso uma posição para o caracter `'\0'`. Por exemplo, caso se pretendesse definir uma *string* para armazenar a matricula de uma viatura e partindo do principio que uma matrícula é composta por 6 caracteres alfanuméricos (4 dígitos + 2 letras) essa *string* teria de ter tamanho 7.

`char matricula[7];`

Em C++ existe uma biblioteca que contém um conjunto de funções que permitem manipular as *strings*. A maioria destas funções partem do princípio que na última posição "válida"está o caracter especial que é o `null` ou `'\0'`. A figura 1.12 apresenta a representação de uma *string* iniciada.

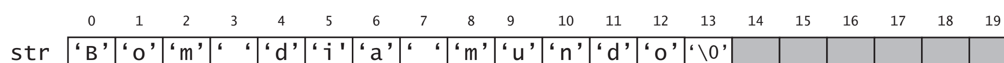


Figura 1.12: Representação de uma *string*

Para poder usar estas funções é necessário fazer o inclusão do ficheiro de cabeçalho (`string.h`).

1.11.1 Iniciação de *strings*

Tal como os vectores de dados numéricos as *strings* também podem ser iniciadas aquando da sua definição. A seguir apresenta-se duas formas para iniciar uma *string* na declaração

```
char str[]={'B','o','m',' ','d','i','a',' ','m','u','n','d','o','\0'};
char str[]="Bom dia mundo";
```

O compilador além de as iniciar também define o seu tamanho. No caso da primeira definição é necessário colocar o caracter `'\0'`, explicitamente. No segundo caso o compilador acrescenta o esse caracter à *string*.

Da mesma forma que os vectores de dados numéricos as *strings* também podem ser manipuladas por indexação.

```
str[0]='B';
str[1]='o';
```

Outro método para atribuir dados a uma *strings* é através do objecto `cin`, nomeadamente do método `getline` cujo protótipo é:

`cin.getline (char buffer[], int length, char delimiter = '\n');`

na qual

- `buffer` – este é o endereço da *string*;
- `length` – este designa o número máximo de caracteres que a *string* pode conter;
- `delimiter` – é o caracter delimitador usado para determinar o fim da cadeia de caracteres inserida pelo utilizador, que por omissão é o o caracter nova linha (`'\n'`).

1.11.2 Funções para manipulação de *strings*

A biblioteca `cstring` define um conjunto de funções para manipulação de strings. Todas estas funções esperam que *strings* estejam terminadas com o carácter `'\0'`. A seguir apresenta-se algumas das funções de biblioteca `cstring` assim como uma breve descrição.

`char *strcat(char *string1, const char *string2);` – Concatena o conteúdo de `string2` a `string1`, que deve ter espaço suficiente para isso. Retorna `string1`.

`char *strchr(const char *string, int c);` – Procura a primeira ocorrência de `c` (0-255) no vector de caracteres apontado por `string`. Retorna um ponteiro para a primeira ocorrência, ou `NULL` se não existir.

`int strcmp(const char *string1, const char *string2);` – Compara as duas cadeias `string1` e `string2`. Retorna: `<0` se `string1` for menor do que `string2`; `=0` se `string1` for igual a `string2`; e `>0` se `string1` for maior do que `string2`.

`char *strcpy(char *string1, const char *string2);` – Copia o conteúdo de `string2` para `string1`, que deve ter espaço suficiente para isso. Retorna `string1`.

`size_t strcspn(const char *string1, const char *string2);` – Retorna o índice do primeiro carácter de `string1` que pertence ao conjunto de caracteres definido em `string2`. Se não existir nenhum retorna o comprimento de `string1`.

`size_t strlen(const char *string);` – Retorna o comprimento de `string`.

`char *strncat(char *string1, const char *string2, size_t count);` – Idêntica a `strcat()`, mas concatenando no máximo `count` caracteres.

`int strncmp(const char *string1, const char *string2, size_t count);` – Idêntica a `strncmp()`, mas comparando no máximo `count` caracteres.

`char *strncpy(char *string1, const char *string2, size_t count);` – Idêntica a strcpy(), mas copiando no máximo `count` caracteres.

`char *strnset(char *string, int c, size_t count);` – Idêntica a `strset()`, mas iniciando no máximo `count` caracteres.

`char *strrchr(const char *string, int c);` – Procura a última ocorrência de `c` (0-255) no vector de caracteres apontado por `string`. Retorna um ponteiro para a última ocorrência, ou `NULL` se não existir.

`char *strnset(char *string, int c);` – Inicia todos os caracteres de `string` com o valor `c` (0-255). O caracter terminador (`'\0'`) não é afectado.

`char *strtok(char *string1, char *string2);` – permite decompor a `string1` em vários vectores de caracteres. A decomposição baseia-se no principio de que a `string1` é formada por uma sequência de palavras separadas por um padrão (constituído por um ou mais caracteres) referenciado por `string2`. Caso não exista o padrão referenciado por `string2` a função `strtok` retorna `NULL`.

O exemplo 1.46 apresenta algumas funções de manipulação de *strings*.

Listing 1.46: Exemplo da utilização de funções da biblioteca `cstring`

```
1 #include<iostream.h>
2 #include<string.h>
3
4 #define SIZE 50
5
6 int main()
```

```
7 {
8   char string1[SIZE]="Bom dia mundo!";
9   char string2[SIZE];
10  int n;
11
12  strcpy(string2,string1);
13
14  if(strcmp(string2,string1)==0)
15    cout<<"As strings sao iguais"<<endl;
16
17  n=strlen(string1);
18  cout<<"A string1 tem"<<n<<" caracteres"<<endl;
19
20  return 0;
21 }
```

1.11.3 Conversão de *strings* para outros tipos

Dado que uma *string* pode conter representações de outros tipo de dados, como números, em algumas situações pode ser útil transformar uma *string* num dado numérico. Por exemplo, uma *string* qualquer pode conter o seguinte número `"1977"`, mas em termos de codificação esta não é mais nem menos do que uma sequência de 5 caracteres, portanto, não sendo possível realizar operações aritméticas. Para tal é necessário, converter esta *string* num tipo de dado numérico, como por exemplo: `int`, `long`, `float`. Na biblioteca `cstdlib` (é necessário a inclusão do ficheiro de cabeçalho `stdlib.h`) estão definidas três funções que permitem fazer essa conversão. As funções são:

`int atoi (const char *string );` – converte a `string` num `int`.

`long atol (const char *string );` – converte a `string` num `long`.

`double atof (const char *string );` – converte a `string` num `double`.

A seguir apresenta-se um exemplo da utilização destas funções

Listing 1.47: Exemplo da utilização de funções da biblioteca `cstring`

```
1 #include<iostream.h>
2 #include<stdlib.h>
3
4 #define SIZE 50
5
6 int main()
7 {
8   char buffer[SIZE];
9   int i;
10  long l;
11  double d;
12
13  cout<<"Digite um numero inteiro: "<<endl;
14  cin.getline(buffer,SIZE);
15  i=atoi(buffer);
16
17  cout<<"Digite um numero inteiro (long): "<<endl;
```

```
18    cin.getline(buffer,SIZE);
19    l=atol(buffer);
20
21    cout<<"Digite um numero fraccionario: "<<endl;
22    cin.getline(buffer,SIZE);
23    d=atof(buffer);
24
25    cout<<"Os valores inseridos foram: "<<i<<"(int): ";
26    cout<<l<<"(long): "<<d<<"(double)"<<endl;
27
28    return 0;
29 }
```

1.11.4 Exercícios resolvidos

1.11.4.1 Programa para manipulação de *strings* e caracteres

Desenvolva um programa que implemente as seguintes funções:

1. Uma função que recebe uma *string* e retorne o seu comprimento. O protótipo da função deverá ser `int mystrlen(char *s);`

2. Uma função que copie o conteúdo de uma *string* para a outra. Portanto a função recebe duas *strings*, uma designada de origem e a outra de destino. O objectivo é copiar o conteúdo da de origem para a de destino. Note que as strings podem ter tamanhos diferentes. Daí que, a função recebe as duas *strings* e um número com o comprimento da string de destino (`comp_strDest`). O objectivo da função é que copiar da *string* de origem para a *string* de destino no máximo `comp_strDest` –1 caracteres. A função deve retornar o número de caracteres efectivamente copiados. O protótipo da função deverá ser `int mystrncpy(char *strDest, char *strSource, int comp_strDest);`

3. Uma função que receba duas *strings* e retorne: 1 se as *strings* forem iguais; e 0 se forem diferentes. O protótipo da função deverá ser `int str1Igualstr2(char *s1, char *s`

4. Uma função que receba um carácter e caso seja maiúsculo retorne o correspondente minúsculo. O protótipo da função deverá ser `char mytolower(char s);`

5. Uma função que receba um carácter e caso seja minúsculo retorne o correspondente maiúsculo. O protótipo da função deverá ser `char mytoupper(char s);`

6. Uma função que elimine todos os espaços à esquerda do primeiro carácter(diferente de espaço). O protótipo da função deverá ser `void mylefttrim(char *s);`

7. Uma função que elimine todos os espaços á direita do último carácter (diferente de espaço). O protótipo da função deverá ser `void myrighttrim(char *s);`

8. Uma função que elimine todos os espaços múltiplos entre as palavras ou letras. O protótipo da função deverá ser `void removemultiplespace(char *s)`.

Listing 1.48: Exemplo de manipulação de *strings* []

```
#include<iostream.h>
#include<stdlib.h>
#include<stdio.h>
#define MAX 50

int menu();
int mystrlen(char *s);
int mystrncpy(char *strDest, char *strSource, int comp_strDest);
int str1Igualstr2(char *str1, char *str2);
char mytolower(char s);
char mytoupper(char s);
void mylefttrim(char *s);
void myrighttrim(char *s);
void removemultiplespace(char *s);

int main()
{
  int op,n;
  char str1[MAX],str2[MAX],c;
  do{
    op=menu();
    switch(op){
      case 1: cout<<"\nDigite a string: ";
          cin.getline(str1,MAX);
          n=mystrlen(str1);
          cout<<"A string tem "<<n<<" caracteres"<<endl;
        break;
      case 2:
          cout<<"\nDigite a string: ";
          cin.getline(str1,MAX);
          n=mystrncpy(str2, str1, MAX);
          cout<<"Copiou "<<n<<" caracteres"<<endl;
          cout<<str2<<endl;
        break;
      case 3:
          cout<<"\nDigite a string 1: ";
          cin.getline(str1,MAX);
          cout<<"\nDigite a string 2: ";
          cin.getline(str2,MAX);
          n=str1Igualstr2(str2, str1);
          if(n==1)
            cout<<"As strings sao iguais"<<endl;
          else
            cout<<"As strings sao diferentes"<<endl;

        break;
      case 4:
          cout<<"\nDigite o caracter: "<<endl;
          c=getchar();
          cout<<mytoupper(c)<<endl;
```

```
51          break;
52        case 5:
53            cout<<"\nDigite o caracter: "<<endl;
54            c=getchar();
55            cout<<mytolower(c)<<endl;
56          break;
57        case 6:
58            cout<<"\nDigite a string: ";
59            cin.getline(str1,MAX);
60            mylefttrim(str1);
61            cout<<str1<<endl;
62          break;
63        case 7:
64            cout<<"\nDigite a string: ";
65            cin.getline(str1,MAX);
66            myrighttrim(str1);
67            cout<<str1<<endl;
68          break;
69        case 8:
70            cout<<"\nDigite a string: ";
71            cin.getline(str1,MAX);
72            removemultiplespace(str1);
73            cout<<str1<<endl;
74          break;
75      }
76    }while(op!=0);
77    return 0;
78  }
79
80  int menu()
81  {
82    int op;
83    char buffer[MAX];
84    do{
85      cout<<"Menu\n";
86      cout<<"1 - Determinar o comprimento de uma string\n";
87      cout<<"2 - Copiar uma string\n";
88      cout<<"3 - Verificar se duas strings sao iguais\n";
89      cout<<"4 - Converter um caracter minusculo em maiusculo\n";
90      cout<<"5 - Converter um caracter maiusculo em minusculo\n";
91      cout<<"6 - Eliminar os espacos a esquerda\n";
92      cout<<"7 - Eliminar os espacos a direita\n";
93      cout<<"8 - Remover multiplos espacos\n";
94      cout<<"0 - Sair\n";
95      cout<<"\nDigite a opcao: ";
96      cin.getline(buffer,MAX);
97      op=atoi(buffer);
98    }while(op<0 || op>8);
99    return op;
100 }
101
```

```
102 int mystrlen(char *s)
103 {
104   int i=0;
105   if(s!=NULL)
106   {
107     while(s[i]!='\0')
108       i++;
109   }
110   return i;
111 }
112 
113 int mystrncpy(char *strDest, char *strSource, int comp_strDest)
114 {
115   int x,i;
116   x=mystrlen(strSource);
117   for(i=0;i<comp_strDest-1 && i<x;i++)
118     strDest[i]=strSource[i];
119   strDest[i]='\0';
120   return i;
121 }
122 
123 int str1Igualstr2(char *s1, char *s2)
124 {
125   int i, x, y;
126   x=mystrlen(s1);
127   y=mystrlen(s2);
128   if(x!=y)
129     return 0;
130   for(i=0;i<x;i++)
131     if(s1[i]!=s2[i])
132       return 0;
133   return 1;
134 }
135 
136 char mytolower(char s)
137 {
138   if(s>='A' && s<='Z')
139     s+='a'-'A';
140   return s;
141 }
142 
143 char mytoupper(char s)
144 {
145   if(s>='a' && s<='z')
146     s-='a'-'A';
147   return s;
148 }
149 
150 void mylefttrim(char *s)
151 {
152   int i,x;
```

```
153     if(s!=NULL)
154     {
155         while(s[0]==' ')
156         {
157             x=mystrlen(s);
158             for(i=1;i<x;i++)
159                 s[i-1]=s[i];
160         }
161     }
162 }
163 
164 void myrighttrim(char *s)
165 {
166     int i;
167     i=mystrlen(s);
168     if(i>0)
169     {
170         while(s[i-1]==' ' && i>0)
171         {
172             s[--i]='\0';
173         }
174     }
175 }
176 
177 void removemultiplespace(char *s)
178 {
179     int i,j,x;
180 
181     i=0;
182     while(s[i]==' ')i++;
183 
184     j=mystrlen(s)-1;
185     while(s[j]==' ')j--;
186 
187     while(i<j)
188     {
189         if(s[i]==' ' && s[i+1]==' ')
190         {
191             for(x=i;x<mystrlen(s);x++)
192                 s[x]=s[x+1];
193             j--;
194         }
195         else
196             i++;
197     }
198 }
```

1.11.5 Exercícios propostos

1.11.5.1 Função que determine o número de ocorrências

Escreva uma função que receba duas *strings* e retorne o número de ocorrências de uma na outra. Por exemplo, a *string* "DEIDE DEDEI DEEI"tem duas ocorrências da *string* "DEI".

1.11.5.2 Função que verifique se uma *string* é inversa de outra

Escreva uma função que recebe duas *strings* e verifica se uma é inversa de outra. Caso seja deve retornar 1 e 0 caso não seja. Note que uma *strings* vazia não tem inversa.

1.11.5.3 Função que conta as palavras de uma *string*

Escreva uma função que recebe uma *string* e retorne o número de palavras. Entendendo-se por palavra cadeias de caracteres terminadas pelo carácter espaço. As palavras têm que ter mais do que um carácter.

1.11.5.4 Função que formate uma *string*

Escreva uma função que receba uma *string* e a formate da seguinte forma:

1. As palavras começam sempre com letras maiúscula e o resto com letra minúscula;

2. As palavras têm que ter mais do que um carácter;

3. As letras (caracteres isolados em letra minúscula).

1.12 Ponteiros

As variáveis na linguagem C++ são definidas através de um identificador e do tipo de dados. No entanto, estas estão armazenadas algures na memória. A memória é uma lista de bytes, na qual cada um tem endereço único. Assim, as variáveis podem também ser referenciadas através do endereço de memória onde estão armazenadas. A memória de um computador com 512 `MBytes` (512*1024*1024=536870912 `bytes`) é ilustrada na figura 1.13. Todos os bytes são endereçáveis (o endereço está escrito em notação hexadecimal).

1.12.1 Operador endereço `&`

Sempre que se declara uma variável, ela é armazenada num local concreto da memória, sendo que o local é definido pelo sistema operativo em tempo de execução. Uma vez atribuído o local de armazenamento, por vezes é necessário aceder directamente a esse local. Isto pode ser feito colocando o `&` antes do identificador da variável. A listagem 1.49 apresenta um programa que permite mostrar no ecrã o endereço de memória onde a variável está declarada.

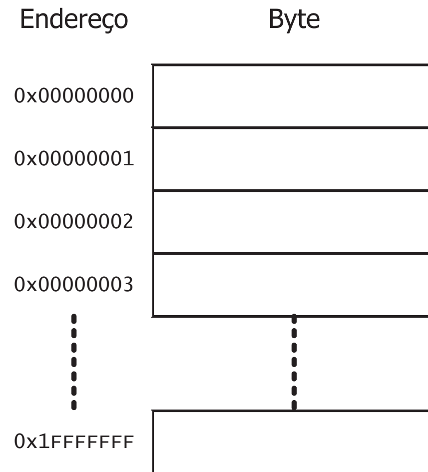


Figura 1.13: Representação da memória do computador

Listing 1.49: Endereço de uma variável

```
1  #include<iostream.h>
2
3  int main()
4  {
5    int i;
6
7    cout<<&i<<endl;
8
9    return 0;
10 }
```

A colocação do `&` antes do identificador da variável significa o **"endereço de"**. No caso do exemplo da listagem 1.49 a instrução `cout<<&i<<endl;` imprime no ecrã o endereço da variável `i`.

1.12.2 Operador de referência *

Os ponteiros são variáveis para que armazenam endereços. A forma de definir variáveis do tipo ponteiro é a seguinte:
`<tipo-de-dados> * <identificador>;`

A única diferença em relação à forma de definir variáveis é a colocação do caracter `*` entre o `tipo-de-dados` e o `identificador`.

- `<tipo-de-dados>`– é um qualquer tipo de dados válido;
- `<identificador>` – tem o mesmo significado que o `identificador` de uma variável de um qualquer tipo de dados estudado até agora.

A seguir apresenta–se a definição de duas variáveis do tipo ponteiro.

```
int *pi;
char *pc;
```

Uma variável do tipo ponteiro tem sempre o mesmo tamanho (em número de bytes) quer aponte para uma variável do tipo `int` ou para uma do tipo `char`. No entanto, na definição é necessário definir qual o tipo de dados que o ponteiro vai apontar. Isto porque, como já foi referido, os tipos de dados não ocupam todos o mesmo número de bytes. Daí que, o compilador precisa de saber qual o tipo de dados que o ponteiro aponta, porque quando faz uma escrita ou uma leitura através do ponteiro tem que saber em quantos bytes vai escrever ou ler. A listagem 1.50 ilustra a definição, atribuição e o acesso (leitura e escrita) a uma variável do tipo `int` através de um ponteiro.

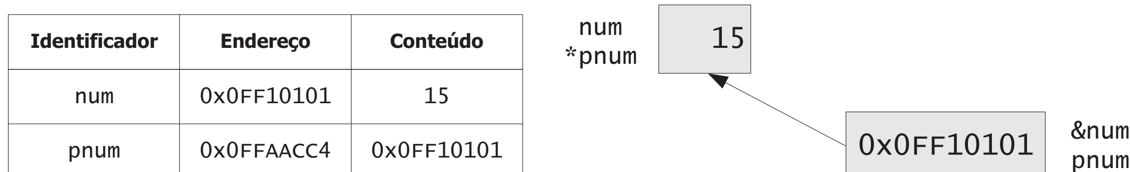
Listing 1.50: Exemplo da utilização de ponteiros

```
#include<iostream.h>

int main()
{
  int *pi, i=100;
  //atribuição do endereço
  //da variável i ao ponteiro
  pi=&i;
  //aceder ao valor da
  //variável i através do ponteiro
  cout<<*pi<<endl;
  //alterar o valor da
  //variável i através do ponteiro
  *pi=200;
  //aceder ao valor da
  //variável i através do ponteiro
  cout<<*pi<<endl;

  return 0;
}
```

Como já foi referido a definição de um ponteiro faz–se com a utilização do caracter `*` entre o tipo de dados e o identificador. No entanto, para aceder ao conteúdo da variável apontado pelo ponteiro também é usado o caracter`*` antes do identificador. No exemplo, da listagem 1.50 a instrução `pi=&i;` atribuí ao ponteiro `pi` o endereço da variável `i`. A instrução `cout<<*pi<<endl;` vai imprimir no ecrã o valor da variável `i` o valor do apontado pelo ponteiro `pi`. O significado do caracter`*` neste caso significa o `valor apontado por`. A figura 1.14 ilustra o resultado da atribuição de uma endereço de uma variável a uma ponteiro.



Identificador	Endereço	Conteúdo
num	0x0FF10101	15
pnum	0x0FFAACC4	0x0FF10101

Figura 1.14: Ponteiro

Como se pode ver o valor da variável `pnum` é o endereço da variável `num`. Portanto

através do ponteiro `pi` ((valor apontado por `pnum`) `*pnum`) é possível aceder ao valor da variável `num`.

Sendo um ponteiro uma variável, que armazena endereços de variáveis, o seu valor pode ser alterado. Isto é, é possível atribuir o endereço de uma variável a um ponteiro e depois alterar esse valor com o endereço de outra variável. Uma boa prática de programação é colocar o valor de uma ponteiro a `NULL` quando não está apontar para nenhuma variável.

Listing 1.51: Exemplo da utilização de ponteiros

```
#include<iostream.h>

int main()
{
  int *p1=NULL,*p2=NULL;
  int v1=100,v2=200;
  //os ponteiros apontam todos para a mesma variável
  p1=p2=&v1;
  //alterar o valor variável v1 através dos ponteiros
  *p1+=100;
  *p2+=100;
  cout<<v1<<endl;
  //alterar o ponteiro p2 que passa
  //apontar para a variavel v2
  p2=&v2;
  cout<<*p2<<endl;
  //atribui à variável v2 o valor do
  //apontador por p1(o valor da varivél v1)
  v2=*p1;
  cout<<v2<<endl;
  //altera o valor variável v1
  //( o ponteiro p1 aponta para a varivael v1)
  v1+=50;
  cout<<*p1<<endl;

  return 0;
}
```

As variáveis `v1` e `v2` são iniciados com o valor `100` e `200` respectivamente. A instrução `p1=p2=&v1;` faz com que os dois ponteiros apontem para a variável `v1`. O valor desta variável é alterado através dos dois ponteiros nas instruções `*p1+=100;` e `*p2+=100;` pelo que o valor da variável `v1` passa a ser de `300`. A instrução `p2=&v2;` altera o ponteiro `p2`, que deixa de apontar para a variável `v1` e passa a apontar para a variável `v2`. A instrução `v2=*p1;` altera o valor da variável `v2` com o valor da variável `v1` (uma vez que o ponteiro `p1` aponta para a variável `v1`). O valor da variável `v1` é alterado com a instrução `v1+=50;`. Assim, a saída deste programa é a seguinte:

```
300
200
300
350
Press any key to continue
```

Note-se que o valor das variáveis pode ser alterado através dos ponteiros ou através das próprias variáveis.

1.12.3 Ponteiros e vectores

Um identificador de um vector é equivalente a um ponteiro para a primeira posição desse vector. Por exemplo, a seguir apresenta–se a declaração de um vector e de um ponteiro;

```
int vec[10];
int *p;
```

a seguinte instrução é valida
`p=vec;`
esta instrução faz com que o ponteiro `p` e `vec` sejam equivalentes, e tem as mesmas propriedades. A única diferença está relacionada com o facto de poder ser atribuído outro valor ao ponteiro `p` enquanto que ao ponteiro `vec` não. Portanto, `p` é uma variável do tipo ponteiro enquanto que `vec` é uma variável do tipo ponteiro constante. Daí que a seguinte instrução é inválida:
`vec=p;`.

A seguir apresenta–se um exemplo que mostra a utilização de ponteiro para aceder aos elementos de um vector.

Listing 1.52: Exemplo da utilização de ponteiros

```
1 #include<iostream.h>
2
3 int main()
4 {
5    int *p=NULL;
6    int vec[10]={0,1,2,3,4,5,6,7,8,9};
7    p=vec;
8    for(int i=0;i<10;i++)
9       cout<<p[i]<<endl;
10   return 0;
11 }
```

1.12.4 Ponteiros para ponteiros

A linguagem C++ permite a utilização de ponteiros para ponteiros no qual o último aponta para uma variável. Para tal basta acrescentar um asterisco por cada referência. Por exemplo:

Listing 1.53: Exemplo da utilização de ponteiros

```
1 #include<iostream.h>
2
3 int main()
4 {
5    int **pp=NULL,*p=NULL, a=10;
6
7    p=&a;
```

```
8     pp=&p;
9
10    cout<<"endereco de a: "<<&a<<endl;
11    cout<<"endereco de p: "<<&p<<endl;
12    cout<<"endereco de pp: "<<&pp<<endl;
13
14    cout<<"O valor da variavel a: "<<a<<endl;
15    cout<<"O valor da variavel p: "<<p<<endl;
16    cout<<"O valor da variavel pp: "<<pp<<endl;
17
18    cout<<"O valor da variavel a: "<<a<<endl;
19    cout<<"O valor apontado por p: "<<*p<<endl;
20    cout<<"O valor apontado pelo apontado por pp: "<<**pp<<endl;
21
22    return 0;
23 }
```

A saída do programa anterior poderia ser a seguinte.

```
endereco de a: 0x0012FF74
endereco de p: 0x0012FF78
endereco de pp:0x0012FF7C
O valor da variavel a: 10
O valor da variavel p:0x0012FF74
O valor da variavel pp: 0x0012FF78
O valor da variavel a: 10
O valor apontado por p: 10
O valor apontado pelo apontado por pp: 10

Press any key to continue
```

Como se pode ver, o valor do ponteiro `pp` é o endereço do ponteiro `p` que por sua vez tem o endereço da variável `a`. A figura 1.15 apresenta graficamente a interacção entre as variáveis presentes no programa da listagem 1.53.

Figura 1.15: Ponteiro para ponteiro

1.12.5 Ponteiros do tipo `void`

Os ponteiros de tipo `void` são um tipo especial de ponteiros que permite que um ponteiro possa apontar para variáveis de qualquer tipo. O único problema é que

para se aceder ao valor da variável apontado por este tipo de ponteiros é necessário fazer a conversão do tipo de dados (*cast*). Este tipo de ponteiro é muito interessante para o desenvolvimento de funções genéricas, que podem receber vários tipos de dados diferentes. O exemplo 1.54 mostra a utilização dos ponteiros de tipo `void`.

Listing 1.54: Exemplo da utilização de ponteiros do tipo void

```
1 #include <iostream.h>
2 #define TPCHAR    1
3 #define TPINT   2
4 #define TPFLOAT   3
5
6 void incremento (void* ap, int tipo)
7 {
8   switch (tipo)
9   {
10    case TPCHAR : (*((char*)ap))++; break;
11    case TPINT: (*((long*)ap))++; break;
12    case TPFLOAT : (*((float*)ap))++; break;
13  }
14 }
15 int main ()
16 {
17   char a = 'a';
18   int b = 9;
19   float c = 7.5;
20
21   incremento (&a,TPCHAR);
22   incremento (&b,TPINT);
23   incremento (&c,TPFLOAT);
24
25   cout << a << ", " << b << ", " << c<<endl;
26
27   return 0;
28 }
```

A saída deste programa é a seguinte:

```
b, 10, 8.5
Press any key to continue
```

1.13 Tipos de dados não nativos

Nesta secção é apresentada a forma de definir novos tipos de dados na linguagem C++.

1.13.1 Estruturas de dados – instrução `struct`

Uma estrutura de dados é um conjunto de dados de variáveis de diversos tipos de dados agrupados sob um único identificador. A seguir apresenta–se a forma de definir uma estrutura:

```
struct <identificador-da-estrutura>
{
   <tipo-de-dados> <identificador 1>;
   <tipo-de-dados> <identificador 2>;
   ...
   <tipo-de-dados> <identificador N>;
}<identificador-da-variavel>;
```

na qual

- `<identificador-da-estrutura>` – é o nome pelo qual este conjunto de variáveis é identificado;
- `<tipo-de-dados> <identificador ...>;` – consiste na declaração de uma variável dentro da estrutura (opcional).

Na prática a instrução `struct` permite definir um novo tipo de dados. A partir do momento que é definido é possível usar da mesma forma que os tipos de dados primitivos tais como o `int`, `char`, `long` e outros. `<identificador-da-variavel>` é opcional e consiste em definir uma variável do tipo `<identificador-da-estrutura>`.

A seguir apresenta–se a declaração de uma estrutura:

```
struct produtos
{
   char nome[30];
   float preco;
};
```

A instrução anterior consiste na definição de um novo tipo de dados. De seguida é apresentada a declaração de variáveis do tipo `produtos`:
`produtos lapis, caneta, borracha;`

Como foi referido anteriormente é também possível definir variáveis do tipo da estrutura no momento da sua definição, conforme o exemplo seguinte.

```
struct produtos
{
   char nome[30];
   float preco;
}lapis, caneta, borracha;
```

A definição das estruturas pode ser realizada em qualquer parte do código, dentro ou fora das funções. Sendo que, quando é definida dentro de uma função este tipo de dados só é conhecido dentro dessa função.

Após a definição do novo tipo de dados para aceder aos diferentes campos que o compõe utiliza-se operador `.` (ponto). Por exemplo, a instrução `lapis.preco=0.5;` permite atribuir o valor `0.5` ao campo `preco` da variável `lapis`. Para realizar a leitura /atribuição o processo é semelhante:

```
float x=lapis.preco;
cin>>lapis.preco;
```

Tal como para os outro tipo de dados também é possível definir ponteiros para estruturas. As regras são as mesmas que para qualquer outro tipo de dados. De seguida é apresentada a forma de definir de um ponteiro para uma estrutura.
`<identificador-da-estrutura> * <identificador>;`

O acesso aos elementos constituintes da estrutura é feita através do operador `->`. Conforme o exemplo (1.55) apresentado de seguida.

Listing 1.55: Exemplo da utilização de ponteiros para estruturas

```
1 #include <iostream.h>
2 #include <stdlib.h>
3
4 struct filmes_t {
5   char titulo [50];
6   int ano;
7 };
8
9 int main ()
10 {
11   char buffer[50];
12   filmes_t afilme;
13   filmes_t * pfilme;
14
15   //atribuir ao ponteiro o endereço da variavel afilme
16   pfilme = & afilme;
17
18   cout << "Digite o titulo: ";
19   //armazenar o titulo na variavel atraves do ponteiro
20   cin.getline (pfilme->titulo,50);
21
22   cout << "Digite o ano: ";
23   cin.getline (buffer,50);
24   //armazenar o ano na variavel atraves do ponteiro
25   pfilme->ano = atoi (buffer);
26
27   //ler os dados armazenados na
28   //variavel atraves do ponteiro
29   cout << pfilme->titulo;
30   cout << " (" << pfilme->ano << ")\n";
31
32   return 0;
33 }
```

O operador `->` é exclusivamente utilizado para aceder aos elementos das estruturas através de ponteiros. No entanto, também é possível a utilização do operador `.` para aceder aos elementos de um ponteiro para uma estrutura. A seguir apresenta–se diferentes formas de aceder aos elementos de uma estrutura através de um ponteiro.

```
filmes_t afilme;
filmes_t * pfilme;
pfilme=&afilme;
```

A instrução seguinte permite aceder ao elemento `titulo` da estrutura apontado pelo ponteiro `pfilme`.

`pfilme->titulo;`

A instrução seguinte é equivalente à anterior.

`(*pfilme).titulo;`

A definição de estruturas dentro de estruturas é muito útil e comum. O programa da listagem 1.56 contém duas estruturas: `data_t` e `filmes_t`, note–se que um dos elementos da estrutura `filmes_t` é ele próprio uma estrutura (`data_estreia`.

Listing 1.56: Estruturas dentro de estruturas

```
#include <iostream.h>
#include <stdlib.h>

struct data_t
{
    int dia,mes,ano;
};

struct filmes_t
{
    char titulo [50];
    data_t data_estreia;
};

int main ()
{
    char buffer[50];
    filmes_t afilme;

    cout << "Digite o titulo: ";
    cin.getline (afilme.titulo,50);
    cout << "Digite a data de estreia(dia/mes/ano): ";
    cin.getline (buffer,50);
    afilme.data_estreia.dia = atoi (buffer);
    cin.getline (buffer,50);
    afilme.data_estreia.mes = atoi (buffer);
    cin.getline (buffer,50);
    afilme.data_estreia.ano = atoi (buffer);

    cout << afilme.titulo;
    cout << " (" << afilme.data_estreia.dia<<":"
    cout <<afilme.data_estreia.mes<<":";
    cout<<afilme.data_estreia.ano<<")\n";

    return 0;
}
```

Da mesma forma que é possível definir vectores de tipos de dados primitivos, também é possível definir vectores, tanto uni–dimensionais como multi–dimensionais, de estruturas. Assim como, os elementos de uma estrutura também pode ser vectores. O programa da listagem 1.57 mostra como definir estruturas nas quais alguns dos seus elementos são também estruturas e vectores de estruturas. O objectivo deste exemplo é mostrar a forma de aceder/utilizar os elementos de uma estrutura deste tipo.

Listing 1.57: Vectores de estruturas

```
#include <iostream.h>

#define STR_LEN     50
#define NUM_DISC    30
#define NUM_ALUNOS   100

struct data_t
{
  int dia,mes,ano;
};

struct disciplina_t
{
  char disciplina[STR_LEN]
  int classificacao;
};

struct aluno_t
{
  char nome [STR_LEN];
  char morada [STR_LEN * 2];
  data_t data_nasc;
  disciplina_t disciplinas[NUM_DISC];
};

void listarAlunos(aluno_t al[]);

int main()
{
  aluno_t alunos[NUM_ALUNOS];

  listarAlunos(alunos);

  return 0;
}

void listarAlunos(aluno_t al[])
{
  for(int i=0;i<NUM_ALUNOS;i++)
  {
    cout<<"Nome: "<<al[i].nome<<endl;
    cout<<"Morada: "<<al[i].morada<<endl;
    cout<<"Data de nascimento: "<<al[i].data_nasc.dia<<":";
    cout<<<<al[i].data_nasc.mes<<":";
    cout<<<<al[i].data_nasc.ano<<endl;
    for(int j=0;j<NUM_DISC;j++)
    {
      cout<<"\tDisciplina: "<<al[i].disciplinas[j].disciplina<<endl;
      cout<<"\tDisciplina: "<<al[i].disciplinas[j].classificacao<<endl;
    }
```

```
51     }
52 }
```

1.13.2 Definição de tipos – instrução typedef

A linguagem C++ permite a definição de tipos de dados não nativos através da instrução `typedef`. A forma para o fazer é a seguinte:

```
typedef <tipo-de-dados-existente> <novo-tipo-dados>;
```

no qual

- `<tipo-de-dados-existente>` – é um tipo de dados já definido, por exemplo um tipo de dados primitivos ou uma tipo de dados composto (estrutura);
- `<novo-tipo-dados>` – a designação do novo tipo de dados.

A seguir apresenta-se vários exemplos da definição de novos tipos de dados.

```
typedef char CARACTER;
typedef int INTEIRO;
typedef float REAL;
typedef char STRING [50];
```

A partir destas definições é possível usar os novos tipos da mesma forma que se podem todos os outros. No extracto de código seguinte são definidoas variáveis com base nos tipos anteriores.

```
CARACTER a, b,c='A';
INTEIRO x=5;
REAL f=4.9;
STRING str="Bom dia mundo!!";
```

1.13.3 União – instrução union

Uma união[5] permite fazer a definição de um tipo de dados de forma disjunta, isto significa que em termos práticos uma variável deste tipo pode ser de qualquer dos sub–tipos utilizados na definição. A sintaxe de uma união é muito semelhante com a definição de uma estrutura:

```
union <identificador-da-união>
{
   <tipo-de-dados> <identificador 1>;
   <tipo-de-dados> <identificador 2>;
   ...
   <tipo-de-dados> <identificador N>;
}<identificador-da-variavel>;
```

na qual

- `<identificador-da-união>`– é a designação do novo tipo de dados;

[5] do anglo-saxónico *union*.

- `<identificador-da-variavel>`– define uma variável do novo tipo de dados (opcional).

Uma das grandes diferenças entre uma estrutura e uma união, consiste no tamanho (número de bytes) que cada uma ocupa. O tamanho de uma estrutura corresponde ao somatório dos tamanhos dos seus elementos e consequentemente numa estrutura pode armazenara dados de todos os seus elementos em simultâneo. No caso das uniões, o tamanho de uma união corresponde ao tamanho do elemento que ocupa o maior número de bytes. Por exemplo, a seguinte união `mtipos_t` ocupa 8 bytes, uma vez que o tamanho de um `char` é 1 byte, de um `int` são 4 bytes e de um `float` são 8 bytes.

```
union mtipos_t
{
   char c;
   int i;
   float f;
};
```

A figura 1.16 apresenta graficamente a união `mtipos_t`. Como se pode ver, uma união, de cada vez, só pode armazenar informação de um dos seus tipos, uma vez que o mesmo espaço é usado para todos eles. Nesta representação assume-se que os tipos de dados `char`, `int` e `float` ocupam um, dois e quatro bytes, respectivamente.

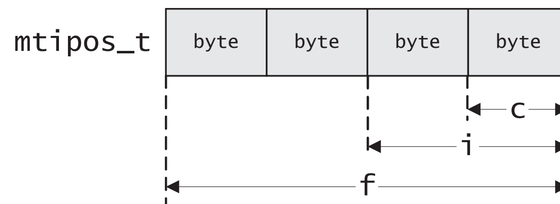


Figura 1.16: Representação de uma união

Para aceder aos elementos de uma união é usado o operador `.` (ponto). Por exemplo, a seguir apresenta–se a forma de definir uma união e de aceder aos seus elementos.

```
mtipos_t un;
un.c='a';
cout<<un.c;
un.i=1000;
cout<<un.1;
```

Importa referir, que sendo utilizado o mesmo espaço para armazenar a informação dos diferentes elementos, a modificação num dos elementos afecta a informação dos outros. Na linguagem C++ também existe a possibilidade de definir uniões anónimas, isto é, se se definir uma união dentro de uma estrutura sem `identificador-da-variável` é possivel aceder aos elementos da união sem referenciar o `identificador-da-variavel` (que de facto não existe). A seguir é apresentada a definição de duas estruturas, uma com uma outra união simples e uma anónima. A união simples:

```
struct livro
{
   char titulo[50];
   char autor[50];
   union{
       float dolar;
       int iene;
       }preco;
};
```

A união anónima:

```
struct livro
{
   char titulo[50];
   char autor[50];
   union
   {
       float dolar;
       int iene;
   };
};
```

Considere–se que existe uma variável do tipo `livro` definida com o identificador `meu_livro`. O acesso ao elemento `dolar` da união simples faz–se da seguinte forma:
`meu_livro.preco.dolar`
enquanto que no caso, da união anónima faz-se da seguinte forma:
`meu_livro.dolar`
O elemento `dolar` é acedido com se fosse um elemento da estrutura e não da união. Note–se que independentemente de serem ou não anónima, as uniões ocupam o mesmo espaço.

1.13.4 Enumeradores – instrução `enum`

Os enumeradores servem para criar tipos de dados definidos à custa de uma lista ordenada de valores possíveis. A forma de definir um enumerador é a seguinte:

```
enum <id-do-enumerador>
{
    <valor 1>,
    <valor 2>,
    ...
    <valor N>
}<id-da-variavel>;
```

Por exemplo, a instrução seguinte define um novo tipo de dados chamado `cor_t`, para armazenar as cores.

`enum cor_t{preto, branco, azul,vermelho, verde, amarelo};`

Nos elementos do enumerador `cor_t` não existe nenhuma referência a qualquer tipo de dados. Portanto, os enumeradores permitem definir novos tipos de dados

sem referenciar qualquer outro tipo de dados já existente. No entanto, uma qualquer variável do tipo `cor_t` só pode assumir os valores escritos entre `{}`. De facto os enumeradores são compilados como inteiros, em que se nada for referido o primeiro elemento assume o valor `0` e o seguinte `1` e assim por diante. No entanto, pode–se definir qual o valor inicial. Por exemplo, a instrução seguinte cria o tipo de dados `mes_t` em que o primeiro elemento assume o valor `1`.

`mes_t {jan=1,fev,mar,abr,mai,jun,jul,ago,set,out,nov,dez};`

O exemplo 1.58 mostra a utilização dos tipos de dados definidos através de `enum`, `struct` e `union`.

Listing 1.58: Exemplo da utilização de tipos de dados definidos

```
1 #include <iostream.h>
2
3 #define PI  3.1415
4
5 enum tipo_figura_t{RECTANGULO, CIRCULO};
6
7 struct rectangulo_dimensoes_t
8 {
9   double comprimento;
10  double largura;
11 };
12 struct circulo_dimensoes_t
13 {
14   double raio;
15 };
16 struct figura_t
17 {
18   tipo_figura_t tag;
19   union
20   {
21     rectangulo_dimensoes_t rectangulo;
22     circulo_dimensoes_t circulo;
23   } dimensoes;
24 };
25 double area(figura_t *fig)
26 {
27   switch(fig->tag)
28   {
29     case RECTANGULO:
30       {
31       double c = fig->dimensoes.rectangulo.comprimento;
32       double l = fig->dimensoes.rectangulo.largura;
33       return c * l;
34       }
35     case CIRCULO:
36       {
37       double r = fig->dimensoes.circulo.raio;
38       return PI * (r*r);
39       }
40
```

```
41      default: return -1.0; /* tag invalida */
42    }
43 }
44 int main()
45 {
46    figura_t fig;
47
48    fig.tag=RECTANGULO;
49    fig.dimensoes.rectangulo.comprimento=3.0;
50    fig.dimensoes.rectangulo.largura=2.0;
51
52    cout<<area(&fig)<<endl;
53
54    return 0;
55 }
```

A figura 1.17 apresenta um resumo dos principais tipos de dados em C++.

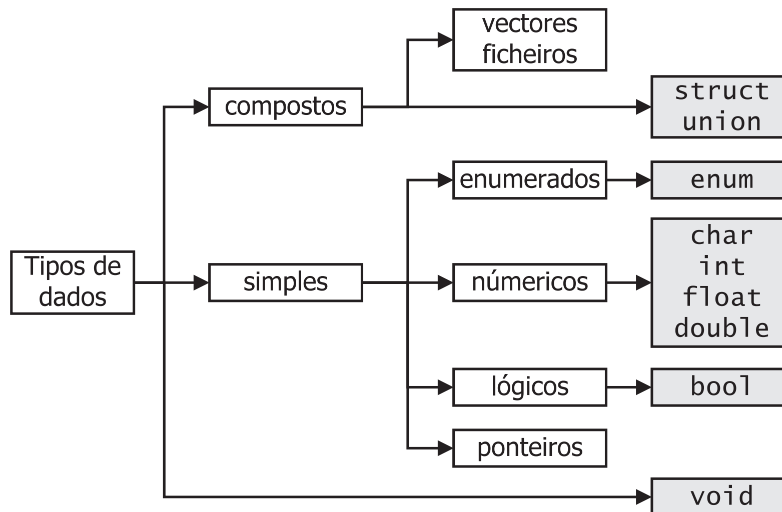


Figura 1.17: Tipos de dados

1.13.5 Exercícios resolvidos

1.13.5.1 Ponto e recta

Sabendo que um ponto é constituído por três coordenadas e uma recta pode ser definida por dois pontos. Desenvolva um programa que implemente as seguintes estruturas e funções.

1. Defina as estruturas ponto e recta;
2. Desenvolva uma função que permita definir um ponto. A função deverá ter o seguinte protótipo: `void inserirPonto(Ponto &p);`
3. Desenvolva uma função que mostre o valor das coordenadas de um ponto. A função deverá ter o seguinte protótipo: `void imprimirPonto(Ponto p);`

4. Desenvolva uma função que permita alterar valor das coordenadas de um ponto. A função deverá ter o seguinte protótipo: `void alterarPonto(Ponto *p);`

5. Desenvolva uma função que permita definir uma recta. A função deverá ter o seguinte protótipo: `void inserirRecta(Recta *r);`

6. Desenvolva a função que mostre o valor das coordenadas dos pontos de uma recta. A função deverá ter o seguinte protótipo: `void inserirRecta(Recta &r);`

7. Desenvolva a função que calcule o comprimento de uma recta. A função deverá ter o seguinte protótipo: `double comprimentoRecta(Recta r);`

Listing 1.59: Exercicio de manipulação de estruturas

```
1 #include <iostream.h>
2 #include <math.h>
3
4 struct Ponto
5 {
6     double x;
7     double y;
8     double z;
9 };
10 struct Recta
11 {
12     Ponto p1;
13     Ponto p2;
14 };
15
16 void inserirPonto(Ponto &p);
17 void imprimirPonto(Ponto p);
18 void alterarPonto(Ponto *p)
19 void inserirRecta(Recta &r);
20 void imprimirRecta(Recta &r);
21 double comprimentoRecta(Recta r);
22
23 int main()
24 {
25     Ponto p;
26     Recta r;
27     inserirPonto(p);
28     imprimirPonto(p);
29     alterarPonto(&p);
30     imprimirPonto(p);
31     inserirRecta(r);
32     imprimirRecta(r);
33     cout<<"comprimento da recta : "<<comprimentoRecta(r)<<endl;
34     return 0;
35 }
36
37 void inserirPonto(Ponto &p)
38 {
```

```
39     cout<<"Coordenadas (x,y,z) do ponto="<<endl;
40     cin>>p.x>>p.y>>p.z;
41 }
42
43 void imprimirPonto(Ponto p)
44 {
45     cout<<p.x<<":"<<p.y<<":"<<p.z<<endl;
46 }
47
48 void alterarPonto(Ponto *p)
49 {
50     cout<<"Novas coordenadas (x,y,z) do ponto="<<endl;
51     cin>>p->x>>p->y>>p->z;
52 }
53
54 void inserirRecta(Recta &r)
55 {
56     cout<<"Ponto 1"<<endl;
57     inserirPonto(r.p1);
58     cout<<"Ponto 2"<<endl;
59     inserirPonto(r.p2);
60 }
61
62 void imprimirRecta(Recta &r)
63 {
64     cout<<"Ponto 1"<<endl;
65     imprimirPonto(r.p1);
66     cout<<"Ponto 2"<<endl;
67     imprimirPonto(r.p2);
68 }
69
70 double comprimentoRecta(Recta r)
71 {
72     return sqrt(
73         pow(r.p1.x-r.p2.x,2)+
74         pow(r.p1.y-r.p2.y,2)+
75         pow(r.p1.z-r.p2.z,2));
76 }
```

1.13.5.2 Gestão de clientes de uma discoteca

Pretende–se um programa para a gestão de clientes de uma discoteca. A cada cliente é dado, à entrada, um cartão com um número. Os cartões estão previamente numerados e existe uma lista de produtos (código do produto, descrição e preço) também previamente definida. De cada vez que o cliente consome algum produto é registado no cartão o código do produto assim como a quantidade. É necessário verificar se o código do produto introduzido é valido. Se o cliente exceder os 10 consumos terá que liquidar a conta e pedir novo cartão. Quando um cliente sai, o programa deverá calcular o preço a pagar pelo cliente e registar esse cartão como pago (deverá apresentar a relação dos consumos com totais parciais e totais).

1. Defina as estruturas necessárias para a resolução do problema;
2. Crie uma função que numere os cartões a partir de um número dado pelo utilizador, supondo que serão necessários, no máximo 600;
3. Crie uma função que inicie a lista de produtos, supondo que existem 10 produtos;
4. Crie uma função para registar a entrada dos clientes;
5. Crie uma função para inserir consumos;
6. Crie uma função para calcular a despesa de um cliente;
7. Crie uma função para indicar o número de clientes na discoteca;
8. Crie uma função que permita listar os produtos.

1.13.6 Exercícios propostos

1.13.6.1 Empresa de construção civil

Uma empresa de construção civil pretende uma aplicação informática de gestão de recursos humanos. A empresa não prevê ultrapassar os 100 funcionários. Os dados dos funcionários são os seguintes: o número, o nome, a categoria, o vencimento e a data de entrada dos funcionários da empresa

1. Defina as estruturas de dados;
2. Escreva uma função para ler os dados de um funcionário;
3. Escreva uma função para listar os dados de um funcionário;
4. Escreva uma função para ler os dados dos `n` funcionários da empresa;
5. Escreva uma função para listar os dados dos `n` funcionários da empresa.

1.13.6.2 Empresa de construção civil

Dado número elevado de alunos inscritos na disciplina de Introdução à Programação foi pedido ao programador que desenvolvesse uma aplicação para gerir a disciplina. Por motivos ainda desconhecidos o programador em causa foi raptado. Partindo do que já está (1.60) desenvolvido termine a aplicação . Pressupostos a considerar na elaboração do programa:

1. Os alunos são identificados pelo número;
2. Um aluno só pode estar inscrito a uma turma;
3. Não podem existir turmas com o mesmo número;
4. A avaliação consiste em duas componentes, componente Frequência e a Prova escrita. As notas da Frequência e da Prova escrita tem um peso de 50% cada na nota final. A nota da frequência é determinada em função das classificações dos 4 Mini–testes (cada com um peso de 10%) e de um trabalho (peso de 60%). Além disso e pela ordem que se apresenta, os alunos que:

(a) Faltem a mais de 30% das aulas efectivamente dadas são classificados com "NF"e reprovam;

(b) Os alunos com nota inferior a 8 valores na nota da Frequência ou da Prova Escrita são classificados com "NC"e reprovam;

(c) Os alunos com nota superior ou igual a 9.5 são classificados como aprovados outros como reprovados.

O valor de qualquer classificação está compreendido entre 0 e 20 valores. Desenvolva:

- Relativamente as turmas:

  1. Uma função que permita inserir uma turma. Preferencialmente inserção ordenada;
  2. Uma função que liste todas as turmas.Listar só a informação referente à turma (número, professor, aulas dadas e numero de alunos);
  3. Uma função que liste uma dada turma. Informação da turma assim como dos alunos;
  4. Uma função que permita inserir as aulas efectivamente dadas aquela turma.

- Relativamente aos alunos:

  1. Uma função que permita inserir um aluno. Preferencialmente inserção ordenada;
  2. Uma função que mostre os dados de um aluno (número, nome, faltas, mini–testes, trabalho e exame);
  3. Uma função que permita inserir as faltas e as classificações;
  4. Uma função que elimine um aluno;
  5. Uma função que calcule a classificação final de um aluno.

Listing 1.60: Base para o programa

```
1 #include <iostream>
2 #include <iomanip>
3 #include <ctype.h>
4 #include <time.h>
5
6 using namespace std;
7
8 #define NOME_TAM       40
9 #define NUM_TURMA_TAM  4
10 #define NUM_ALUNOS    20
11 #define NUM_MINI_TESTES 4
12 #define NUM_TURMAS    10
13 #define IGNORE        10
14
15 struct FREQ
16 {
17   //posicao 0 para o 1º MT, posicao 1 para o 2ºMT ...
```

```
18    int miniteste[NUM_MINI_TESTES];
19    int trabalho;
20 };
21
22 struct ALUNO
23 {
24    long numero;
25    char nome[NOME_TAM];
26    FREQ frequencia;
27    int exame;
28    int faltas;
29 };
30
31 struct TURMA
32 {
33    char numero[NUM_TURMA_TAM];
34    char professor[NOME_TAM];
35    int num_alunos;
36    ALUNO alunos[NUM_ALUNOS];
37    int aulas_dadas;
38 };
39
40 int menuAlunos();
41 int menuTurmas()
42 int menu();
43 void manutencaoAlunos(TURMA t[], int &n);
44 void manutencaoTurma(TURMA t[], int &n);
45
46 void main()
47 {
48    TURMA turmas[NUM_TURMAS];
49    int num_turmas=0;
50    int opcao;
51    do
52    {
53       opcao=menu();
54       switch(opcao)
55       {
56          case 1: manutencaoTurma(turmas, num_turmas);
57          break;
58          case 2: manutencaoAlunos(turmas, num_turmas);
59          break;
60       }
61    }while(opcao);
62 }
63
64 int menuAlunos()
65 {
66    int opcao;
67    do
68    {
```

```
69       cout<<"\n\nALUNOS"<<endl;
70       cout<<" 1 - Inserir Aluno\n";
71       cout<<" 2 - Listar Aluno\n";
72       cout<<" 3 - Inserir Faltas-Classificacoes\n";
73       cout<<" 4 - Eliminar Aluno\n";
74       cout<<" 5 - Calcular Classificacao\n";
75       cout<<" 0 - Voltar \n";
76       cout<<" Opcao --> ";
77       cin>>opcao;
78     }while(opcao<0 || opcao>5);
79     cin.ignore(IGNORE,'\n');
80     return opcao;
81 }
82 
83 int menuTurmas()
84 {
85     int opcao;
86     do
87     {
88       cout<<"\n\nTURMAS"<<endl;
89       cout<<" 1 - Inserir Turma\n";
90       cout<<" 2 - Listar Turmas(Todas)\n";
91       cout<<" 3 - Listar Turma(Uma)\n";
92       cout<<" 4 - Inserir Aulas Dadas\n";
93       cout<<" 0 - Voltar \n";
94       cout<<" Opcao --> ";
95       cin>>opcao;
96     }while(opcao<0 || opcao>4);
97     cin.ignore(IGNORE,'\n');
98     return opcao;
99 }
100 
101 int menu()
102 {
103     int opcao;
104     do
105     {
106       cout<<"\n\nALGORITMIA E PROGRAMACAO"<<endl;
107       cout<<" 1 - Turmas \n";
108       cout<<" 2 - Alunos \n";
109       cout<<" 0 - Sair \n";
110       cout<<" Opcao --> ";
111       cin>>opcao;
112     }while(opcao<0 || opcao>2);
113     cin.ignore(IGNORE,'\n');
114     return opcao;
115 }
116 
117 void manutencaoAlunos(TURMA t[], int &n)
118 {
119     int opcao;
```

```
120   do
121   {
122      opcao=menuAlunos();
123      switch(opcao)
124      {
125         case 1:
126         break;
127         case 2:
128         break;
129         case 3:
130         break;
131         case 4:
132         break;
133         case 5:
134         break;
135      }
136   }while(opcao);
137 }
138
139 void manutencaoTurma(TURMA t[], int &n)
140 {
141   int opcao;
142   do
143   {
144      opcao=menuTurmas();
145      switch(opcao)
146      {
147         case 1:
148         break;
149         case 2:
150         break;
151         case 3:
152         break;
153         case 4:
154         break;
155      }
156   }while(opcao);
157 }
```

1.14 Programas de grandes dimensões

Nos exemplos apresentados até esta parte, o código de cada programa está escrito num só ficheiro com a extensão **".cpp"**. No entanto, quando os programas são de grandes dimensões, é uma boa prática dividi–los em ficheiros de menor dimensão. Esta, divisão traz enormes vantagens: embora a no entanto obriga a alguns cuidados. Como vantagem, pode-se referir a manutenção do programa, isto é, é mais fácil analisar uma ficheiro com 100 linhas de código do que com 10000. Por outro, lado pode–se agrupar o código que de alguma forma esteja relacionado nos mesmos ficheiros, levando, portanto,à criação de módulos. A criação de módulos permite a reutilização desses módulos noutros programas. Como desvantagem, pode–se referir

a necessidade de gerir as referências a variáveis externas, a inclusão recursiva de módulos entre outros. Esta separação em módulos obriga a que a compilação de cada módulo. Por um lado, pode ser considerado uma desvantagem porque é necessário compilar todos os módulos separadamente, por outro lado, pode ser considerado uma vantagem, uma que compilado, só é necessário voltar a compilar caso o módulo tenha sido alterado.

1.14.1 Divisão em módulos

Um módulo, geralmente, consiste em dois ficheiros: um ficheiro de cabeçalhos com a extensão **".h"** e outro com a implementação (código) com a extensão **".cpp"** ou **"cc"**. Por uma questão de simplicidade, geralmente os nomes dos dois ficheiros são iguais, variando apenas a extensão. Um aspecto a ter em consideração, é que é necessário fazer o inclusão do ficheiro de cabeçalhos no ficheiro de código (ver figura 1.18).

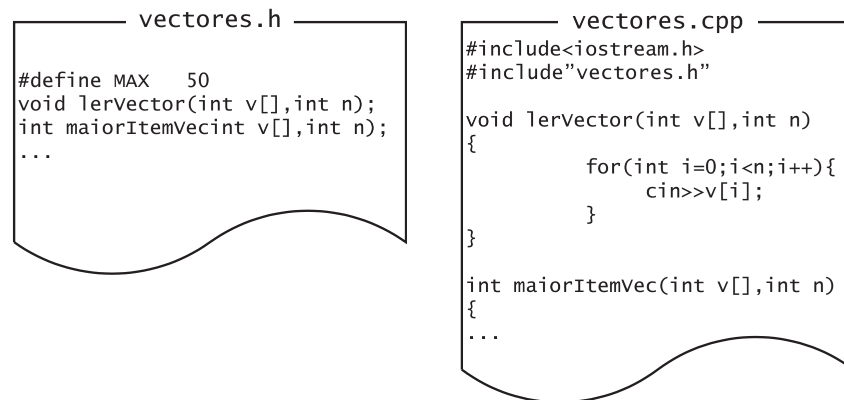


Figura 1.18: Representação gráfica de um módulo

Quando um programa é dividido em módulos, por vezes existe a necessidade de num determinado módulo invocar funções ou aceder a variáveis definidas noutros módulos. Este problema é resolvido através da directiva de pré–processador `#include`. No entanto, este mecanismo de inclusão de ficheiros pode originar erros de *linker* uma vez que pode acontecer que um ficheiro de cabeçalho seja incluído noutro mais do que uma vez. Na figura 1.19 é ilustrado este problema. O ficheiro `appMain.cpp` faz a inclusão de dois ficheiros (`vectores.h` e `matrizes.h`) sendo que por sua vez cada um deles faz a inclusão do ficheiro `defs.h`. Ora, o pré–processador irá incluir no ficheiro `appMain.cpp` duas vezes o conteúdo do ficheiro `defs.h`), o que dará origem a um erro de *linker*.

A resolução deste problema pode ser feita através da directiva de pré–processador `#if` (inclusão condicional). O `#if` avalia uma expressão inteira e executa uma acção baseada no resultado (interpretado como `true` ou `false`). Esta directiva é sempre terminada com a palavra `#endif`. Entre `#if` e `#endif` poderão estar as directivas `#else` e `#elif`. Outras formas da directiva `#if` são:

[`ifdef` <identificador>] verifica se o identificador está definido com uma directiva `define`.

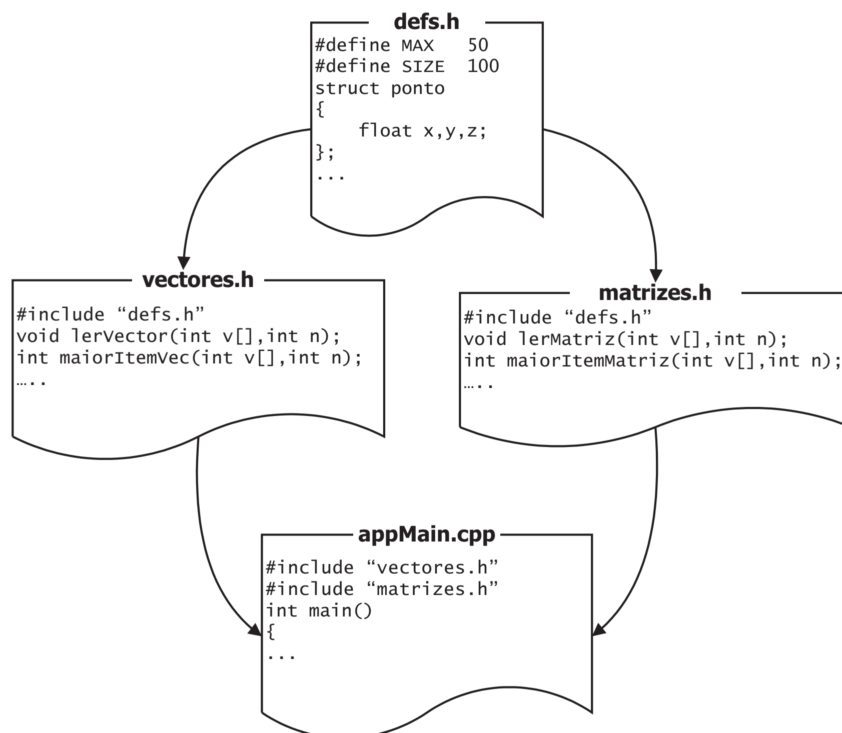


Figura 1.19: Inclusão de um ficheiro

[`ifndef` <identificador>] verifica se o identificador não está definido com uma directiva `define`.

Por exemplo, a seguir define-se uma constante que contenha o número de bits de um inteiro que possa ser compilado correctamente em MS-DOS no compilador "Turbo C"no Unix.

```
#ifdef TURBOC
   #define INT_SIZE 16
#else
   #define INT_SIZE 32
#endif
```

Portanto, para resolver o erro do `linker` é necessário alterar os ficheiros de cabeçalho `defs.h`, `vectores.h` e `matrizes.h`. A figura 1.20 apresenta uma solução.

A desvantagem da necessidade da compilação separada dos módulos, é facilmente superável através de "makefiles"ou com a utilização de uma Ambiente Integrado de Desenvolvimento.

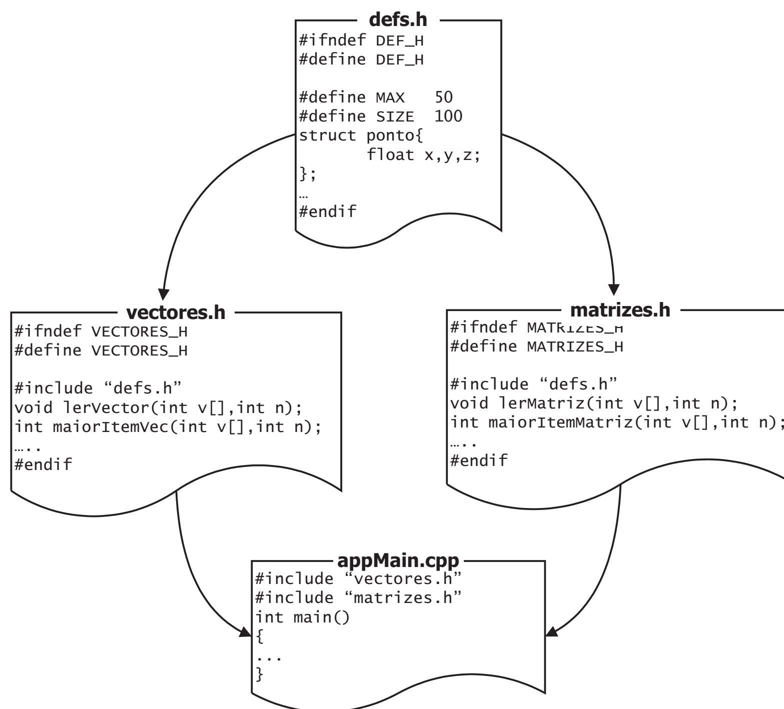


Figura 1.20: Inclusão de um ficheiro (directivas de pré-processamento)

Bibliografia

[CCT, 2001] CCT. *C Programming - Foundation Level, Training Manual & Exercises*. Cheltenham Computer Training, Gloucester/UK, 2001.

[CPP,]

[Kernighan e Ritchie, 1988] Brian W. Kernighan e Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, Inc., 1988.

[Mosich, 1988] D. Mosich. *Advanced Turbo C Programmer's Guide*. Wiley, John & Sons, 1988.

[Sampaio e Sampaio, 1998] Isabel Sampaio e Alberto Sampaio. *Fundamental da Programação em C*. FCA- Editora Informática, 1998.

[Stroustrup, 2000] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Pub Co, 3rd edition edition, February 2000.