
# INTRODUÇÃO À PROGRAMAÇÃO

## MANUAL DE APOIO À UNIDADE CURRICULAR DE INTRODUÇÃO À PROGRAMAÇÃO

| | |
|---|---|
| **Curso** | Engenharia Informática |
| **Unidade Curricular** | Introdução à Programação |
| **Ano Letivo** | 2020 - 2021 |
| **Docente** | José Alberto Quitério Figueiredo |
| **Data** | Outubro 2020 |

# Índice

# 0 Prefácio

Este manual tem como objetivo principal servir de referência à unidade curricular de Introdução à Programação, do curso de Engenharia Informática, do Instituto Politécnico da Guarda.

Na elaboração deste manual procuramos utilizar muita na experiência adquirida ao longo dos 25 anos da interação com os alunos que iniciam o estudo da programação. Incluímos, também, algumas referências e adaptações do que achamos mais importante da inúmera bibliografia existente na área.

Iniciamos com uma pequena homenagem a Dennis M. Ritchie, um dos criadores do sistema Unix e pai da linguagem de programação C, com uma adaptação do prefácio da primeira edição do livro "The C Programming Language", referência para muitos dos que iniciaram o estudo da programação em C (Kernighan & Ritchie, 1988).

*Figura 1 - Dennis M. Ritchie, 1941 - 2011*

A linguagem C é uma linguagem de programação de utilização geral com recursos eficientes de expressões, de controlo de fluxo, de estrutura de dados e um excelente conjunto de operadores. C não é uma linguagem de alto nível e, não é especializada para nenhuma área de aplicação em particular. Mas a sua ausência de restrições e a sua generalidade atribuí-lhe um estatuto muito conveniente e eficaz para muitas tarefas, comparando com muitas linguagens supostamente mais poderosas.

# 1 Introdução

## 1.1 Programar. O que é?

Algumas definições do que é programar.

Programar é basicamente um processo de tradução de uma linguagem favorável para os humanos numa linguagem favorável para os computadores, em (Blackwell, 2002) citando McCracken.

Programar é instruir um computador a fazer qualquer coisa para nós com a ajuda de uma linguagem de programação (Schneider, et al., 1997).

Programar é um processo criativo, efetuado por programadores, que ensina um computador de como deve fazer uma tarefa (Bolton, 2011).

Programar é a arte de fazer com que um computador faça o que nós quisermos que faça (Gauld, 2007).

Programar é um processo de transformação de um plano mental de termos familiares e correntes em termos compatíveis com o computador. A programação é a arte de transformar requisitos em formalidades que o computador pode executar (Hoc & Nguyen-Xuan, 1990).

Segundo (Caspersen, 2007) programar é um processo criativo. Quando desenvolvemos um programa para a resolução de um determinado problema, os programadores são livres de inventar as estruturas que quiserem em função das suas necessidades.

Em resumo, podemos enunciar que programar é um processo de criação, escrita, teste e manutenção de código de um programa de computador. Este código é escrito numa linguagem de programação. O objetivo da programação é produzir um programa que resolva um determinado problema ou concretize um plano.

## 1.2 Linguagem C é importante.

A linguagem C foi até a alguns anos atrás a linguagem mais popular entre os programadores. O código C podia ser utilizado nas mais diferentes máquinas e sistemas operativos. No entanto, com a evolução das tecnologias de informação e comunicação a utilização da linguagem C têm sofrido um considerável decréscimo. Tudo isto devido ao surgir e crescimento exponencial da Internet onde os desafios e necessidades requeridos são diferentes dos requisitos exigidos no passado recente.

No entanto, muitas empresas valorizam os conhecimentos e competências adquiridas na linguagem C, porque sabem que esses programadores rapidamente conseguem aprender uma nova linguagem. A aprendizagem da linguagem de programação C é a base para iniciar a programar. Facilita a aprendizagem de linguagem como C++ e Java, bem como muitas outras linguagens de programação.

## 1.3 Um pouco de história

A linguagem C surge nos laboratórios AT&T Bell Labs entre 1969 e 1973, sendo o seu criador Dennis M. Ritchie. A linguagem C derivou da linguagem B, desenvolvida por Ken Thompsom, que se baseou na linguagem BCPL. Na altura nos laboratórios AT&T Bell estava em curso o

desenvolvimento de um novo sistema operativo UNIX. Até aqui o sistema operativo UNIX era totalmente desenvolvido em Assembly. Durante os anos 80 surgem muitas versões da linguagem C. Por volta de 1990 surge a primeira versão padrão ANSI C.

# 2 Benvindo à programação em C

Em primeiro lugar, para começar a escrever código em linguagem C, é necessário um editor de texto onde poderá escrever o código, e de um compilador de C, que interpretará e produzirá um programa executável com os objetivos especificados. Existem muitos editores e compiladores de C disponíveis, no entanto existem também IDE, ambientes de desenvolvimento integrados, que combinam as duas necessidades. Ao longo deste manual vamos utilizar o Pelles C.

Pelles C: para informação e download utilizar: http://www.christian-heffner.de/index.php?page=start&lang=en

A utilização desta aplicação é apenas uma sugestão, poderá ser utilizado qualquer outro ambiente de desenvolvimento. Utilizamos o Pelles C por ser uma aplicação bastante simples, completa e bastante eficiente. E com isto, não queremos desviar a atenção para pormenores de utilização do ambiente de desenvolvimento, e centrar a nossa atenção unicamente na programação em C.

## 2.1 Primeiro programa

A única forma de aprender a programar é fazer programas, muitos programas.

Vamos Iniciar com o tradicional primeiro programa "Olá Mundo!". Ou seja, o objetivo do programa é escrever uma mensagem "Olá Mundo!".

Para escrever, compilar e executar o programa vamos utilizar o Pelles C. No ecrã inicial escolhemos a opção ***"Start a new project"***, como mostra a figura:

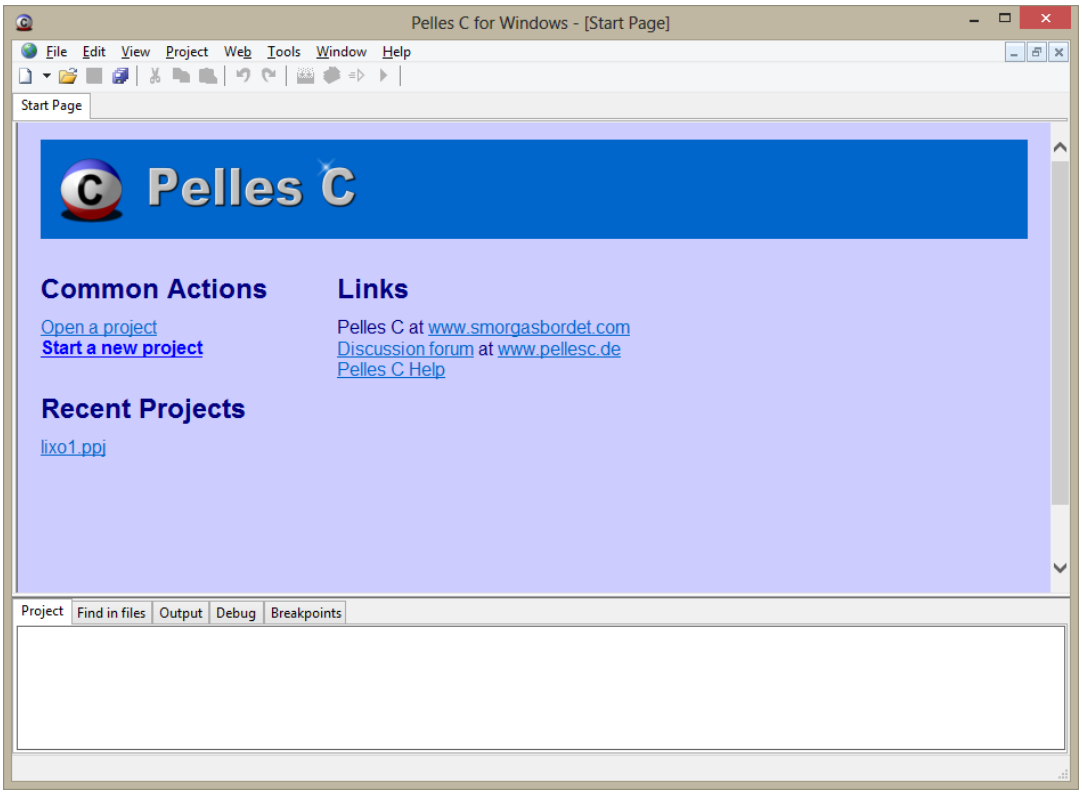



*Figura 2 - Janela inicial do Pelles C*

De seguida, escolhemos o tipo de projeto: ***Console application***, e damos um nome ao projeto: ***Proj01***. Como ilustra a figura.

Normalmente o Pelles C, cria uma pasta, de nome ***Pelles C Projects***, dentro da pasta *Documentos* do seu computador, onde serão guardados todos os projetos.

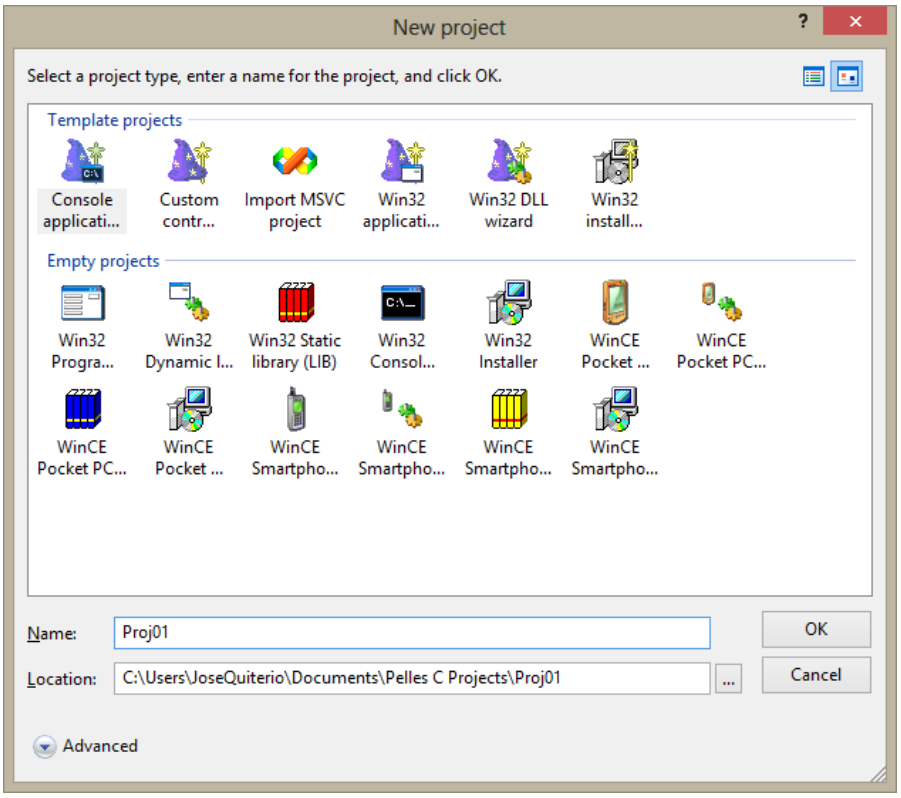



*Figura 3 - Novo projeto. Escolha do tipo de projeto e nome.*

No caso deste ambiente de desenvolvimento, o programa sugere um modelo de programa pré-definido, no caso: um ***simple program*** ou um ***"Hello, World!" program***. No nosso caso vamos escolher ***simple program***. Selecionar ***Seguinte***, como na Figura.

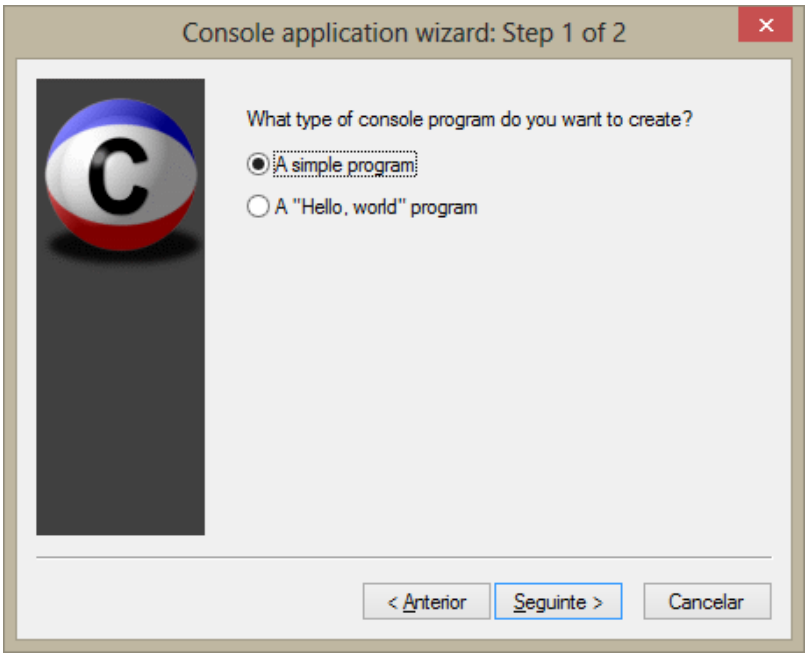
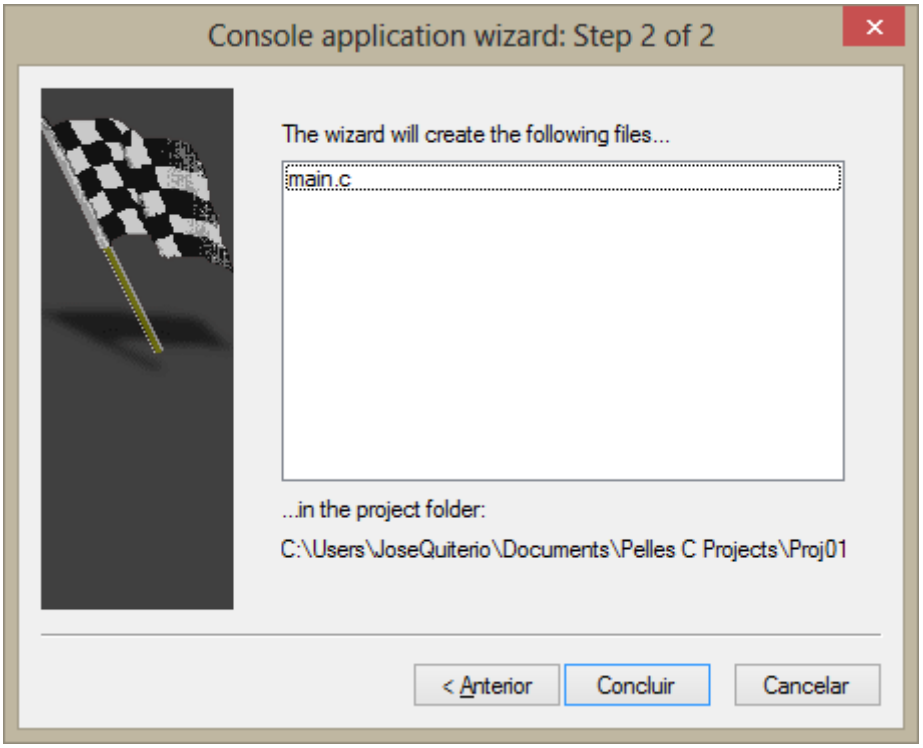
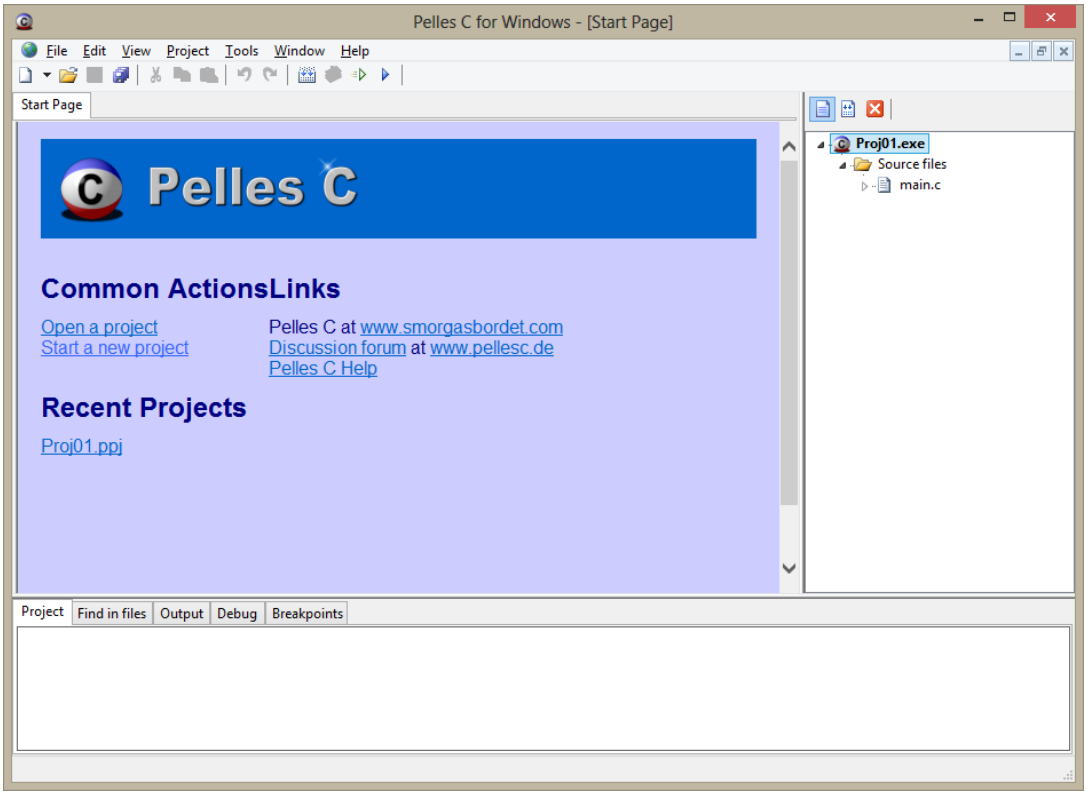


*Figura 4 - Janela de escolha do tipo de aplicação.*

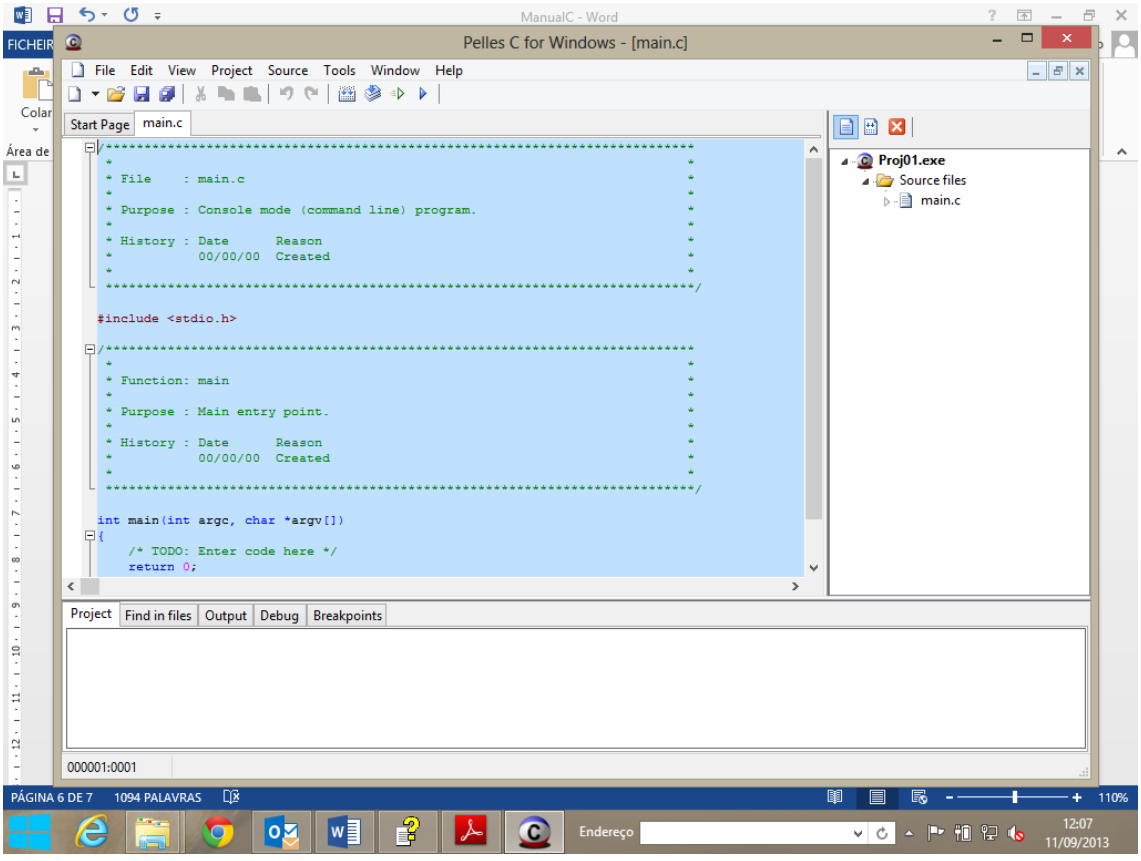
Na janela seguinte, figura, é-nos informado que vai ser adicionado ao nosso projeto o ficheiro ***main.c***. Selecionar ***Concluir***.




Por fim, surge no ecrã o nosso projeto. No lado direito, devemos escolher o ficheiro ***main.c,*** como mostra a figura.




Depois de selecionar o ficheiro **main.c** surge o respetivo código, na área de introdução de código, como na figura.



Sem nos preocuparmos muito sobre o significado de cada instrução, vamos copiar para a nossa aplicação o código seguinte. Deve ter em especial atenção que todos os carateres, maiúsculos ou minúsculos, sinais, espaços, bem como a disposição das instruções são importantes. Assim sendo deve copiar exatamente as instruções seguintes. Este pequeno exemplo serve para começarmos a adaptar à escrita de programas em C e exemplificar a utilização do nosso ambiente de programação.

Verifique que o Pelles C já adicionou algum código, nomeadamente uma área de comentário, onde podemos escrever alguma descrição sobre o programa que estamos a elaborar, e ainda, a função principal: ***int main(int argc, char *argv[])***.

**Este modelo servirá para todos os programas que vamos efetuar**. Será sempre assim para efetuar um novo programa. Criar um projeto e escolher o tipo de programa.

**PROJ01**:

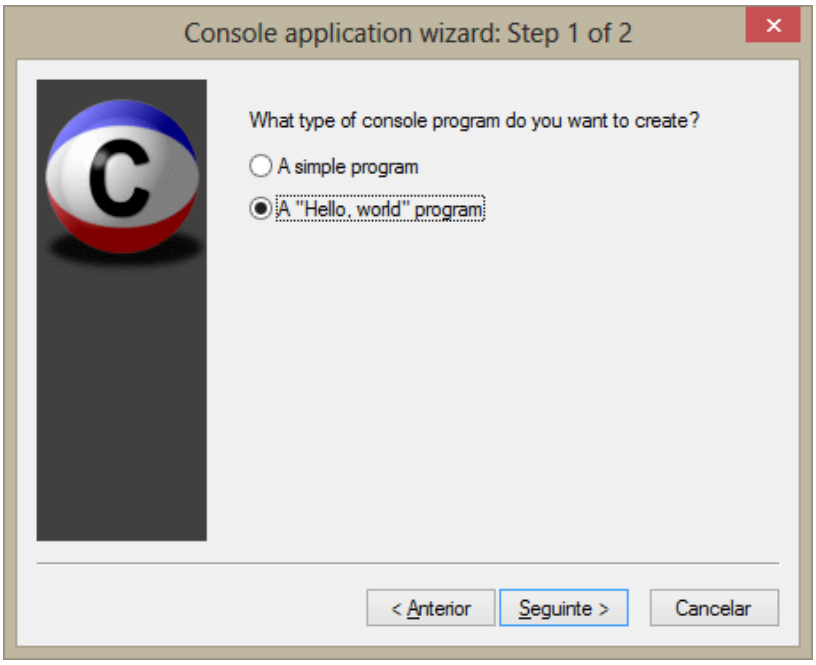
```
#include <stdio.h>
int main(int argc, char *argv[])
{
      printf("Olá, Mundo!!!\n");
      return 0;
}
```

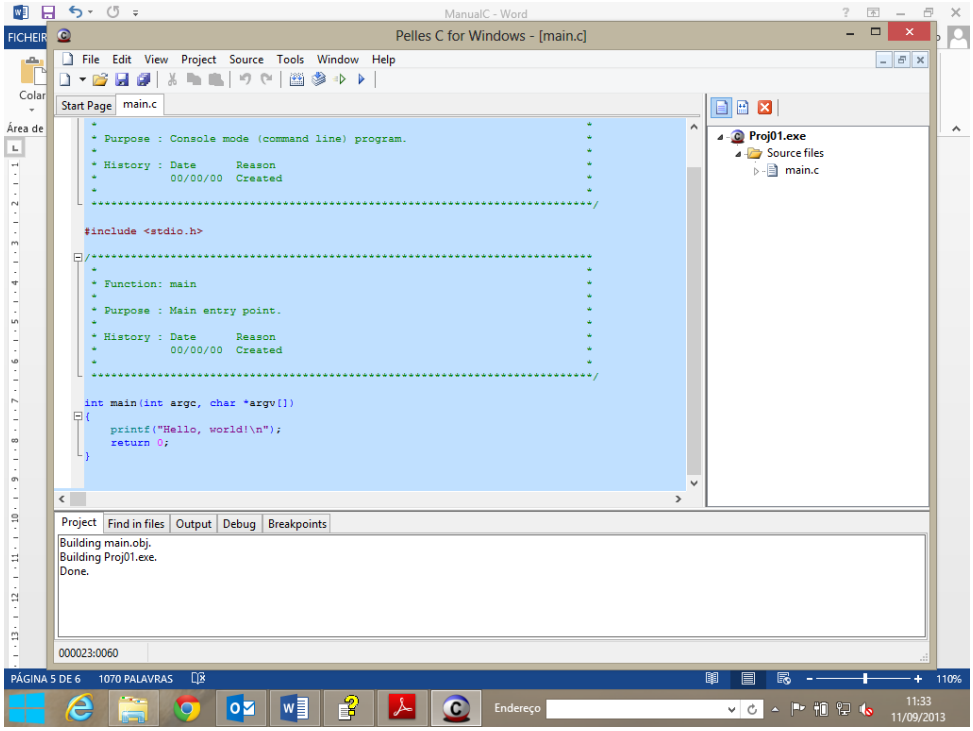
Porque estamos a utilizar o Pelles C, verifique que apenas tem de introduzir a instrução:

```
printf("Olá, Mundo!!!\n");
```

onde está: ***/* TODO: Enter code here */***.

No caso de escolher a opção ***"Hello, world" program***, o código seria exatamente igual ao pretendido para este exemplo, como pode constatar pela figura x e figura y.





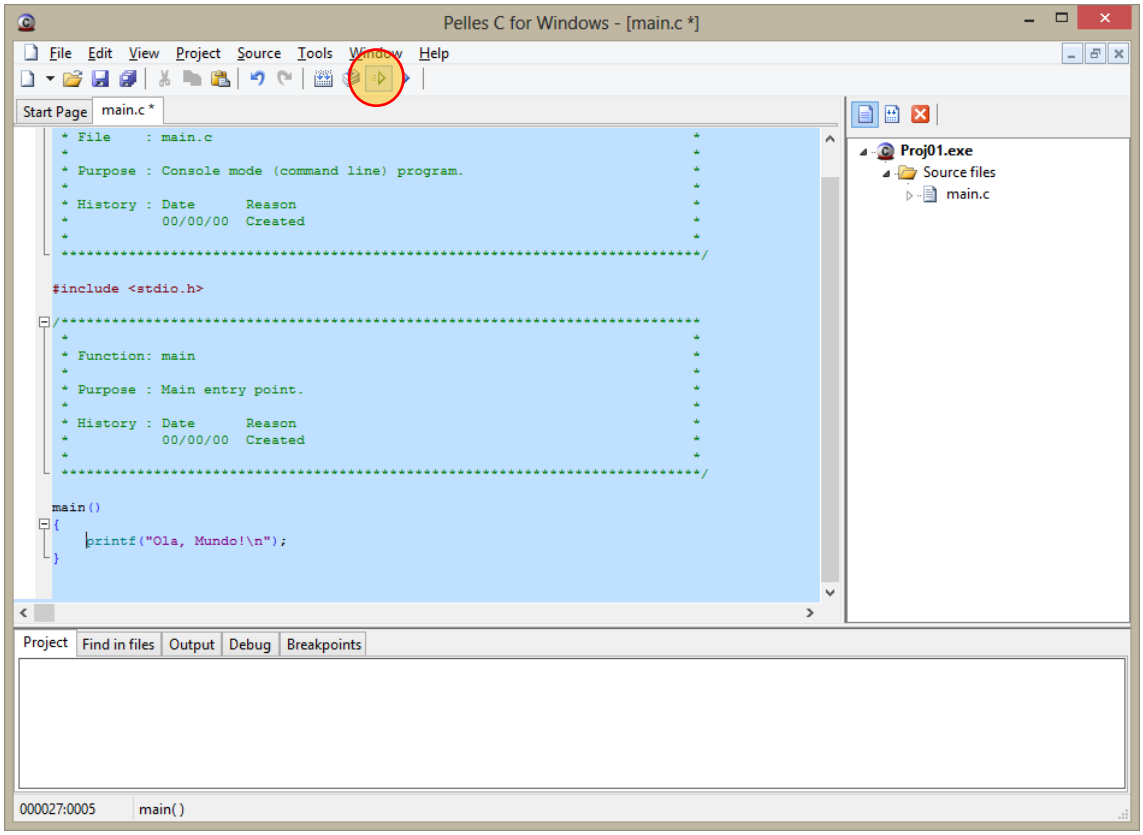

Outras formas de codificação deste primeiro programa podem ser encontradas em diferentes bibliografias, que na sua generalidade são semelhantes e com o mesmo objetivo.

```
#include <stdio.h>
main()
{
      printf("Olá Mundo!\n");
}
```

No modelo sugerido pelo Pelles C, simplesmente podemos apagar tudo e escrever o código sugerido.

## 2.2 Execução do programa

Depois de escrever o primeiro programa vamos executar de modo a podermos visualizar o seu resultado. Para tal, no Pelles C, escolhemos a opção assinalada na figura.




O resultado será uma mensagem no ecrã com a mensagem “Olá, Mundo!”, como na figura.

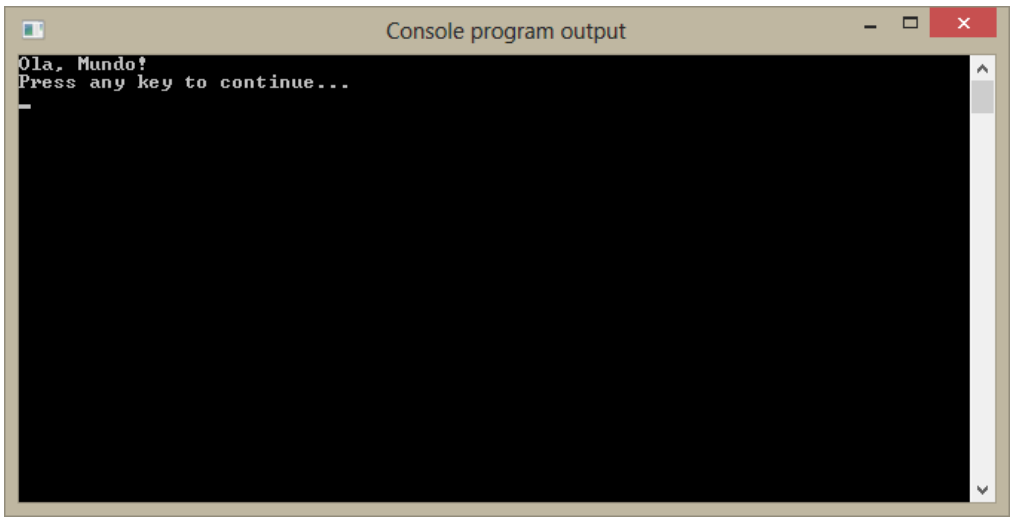



*Figura 5 - Janela de output do programa.*

## 2.3 Compilação e erros

Quando efetuamos a operação de execução (ou Run), em primeiro lugar o programa é compilado, é verificado se as regras da linguagem foram cumpridas. Se todas as regras foram cumpridas o programa é executado normalmente e produzido o resultado, ou output, de acordo com os objetivos esperados.

No entanto, se existirem erros no programa a execução é interrompida e são apresentados os erros existentes no código do nosso programa.

Por exemplo, vamos retirar o ";" - ponto e vírgula, existente no final da instrução:

```
printf("Olá Mundo!\n")
```

De seguida executar a aplicação. Verificamos que não surge a janela de output, mas sim uma área com os erros encontrados no código, como podemos ver pela figura.

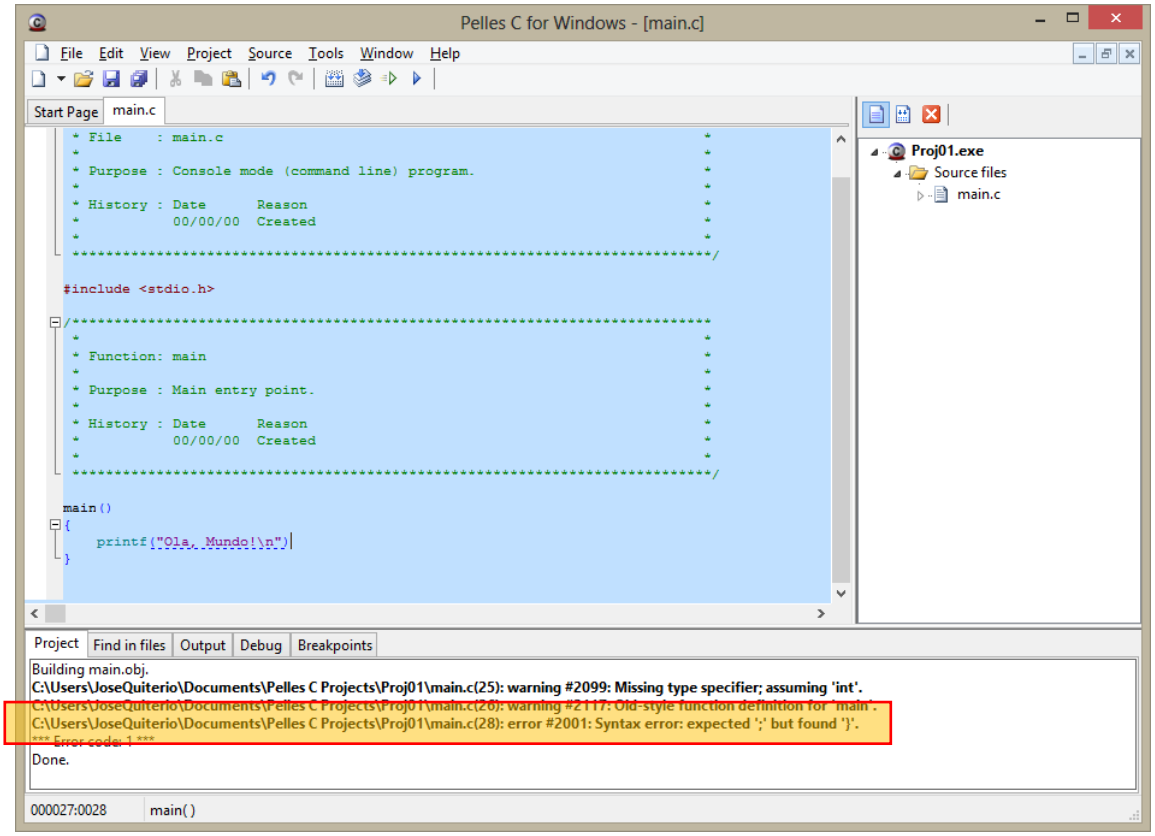



*Figura 6 - Janela com indicação de erros.*

Nesta situação devemos rever o código de modo a corrigir os erros assinalados e, de seguida voltar a executar a aplicação.

## 2.4 Explicação do primeiro programa

Um programa em C é constituído por funções e variáveis. As funções contêm instruções que especificam as operações que devem ser efetuadas. As variáveis guardam valores que são utilizados durante a execução do programa.

No nosso exemplo existe uma função de nome ***main***. Normalmente podemos especificar o nome que desejarmos a uma função. No entanto, a função ***main*** é especial, significa que será executado o que existir na função ***main*** e pela ordem especificada. Isto significa que em todos os programas tem de existir a função principal ***main***.

A função principal ***main*** deverá ser usada para chamar outras funções. Algumas dessas funções escritas por nós e outras existentes na própria linguagem C. Por exemplo, no nosso programa a primeira linha:

```
#include <stdio.h>
```

indica ao compilador que deve incluir um conjunto de funções standard de input e output.

As instruções de uma função devem ser incluídas entre { }. No nosso programa existe uma única instrução na função main:

```
printf("Olá Mundo!\n");
```

Uma função é executada através da evocação do seu nome seguida de um conjunto de parâmetros. No nosso caso a função chama-se ***printf*** e os seus parâmetros são **"Olá Mundo!\n"**. A função ***printf*** é uma função existente na biblioteca de funções ***stdio.h***, e tem como caraterística a escrita de um conjunto de carateres entre aspas no ecrã.

O carácter **\n** existente no final é um carácter especial com a função de efetuar uma mudança de linha, ou mais especificamente com a função de "*newline*".

# 3 Introdução à programação

## 3.1 Linguagem de programação: sintaxe e semântica

Para resolver um problema de uma forma computacional é necessário usar uma linguagem precisa e exata. Essa linguagem deve ser constituída por um conjunto de primitivas e regras formando uma linguagem de programação. Quando essas primitivas e regras são organizadas num determinado método permitem solucionar o problema. As primitivas e regras constituem a sintaxe e semântica de uma linguagem.

Ou seja, a sintaxe é a forma como as instruções de uma linguagem são escritas. A semântica diz respeito ao significado, ou comportamento que um computador segue pela execução das instruções válidas de um programa.

Nos capítulos seguintes vamos apresentar as regras da linguagem de programação C.

## 3.2 Estrutura de um programa

Um programa em C é constituído por uma sequência de funções que no seu conjunto vão permitir a resolução de um problema. Cada uma dessas funções é constituída por um conjunto de instruções que especificam o que o processador deve executar. Cada instrução termina por ';' (ponto e vírgula). O programa inicia na função principal ***main()***.

Na figura seguinte é apresentada a estrutura base de um programa em C.

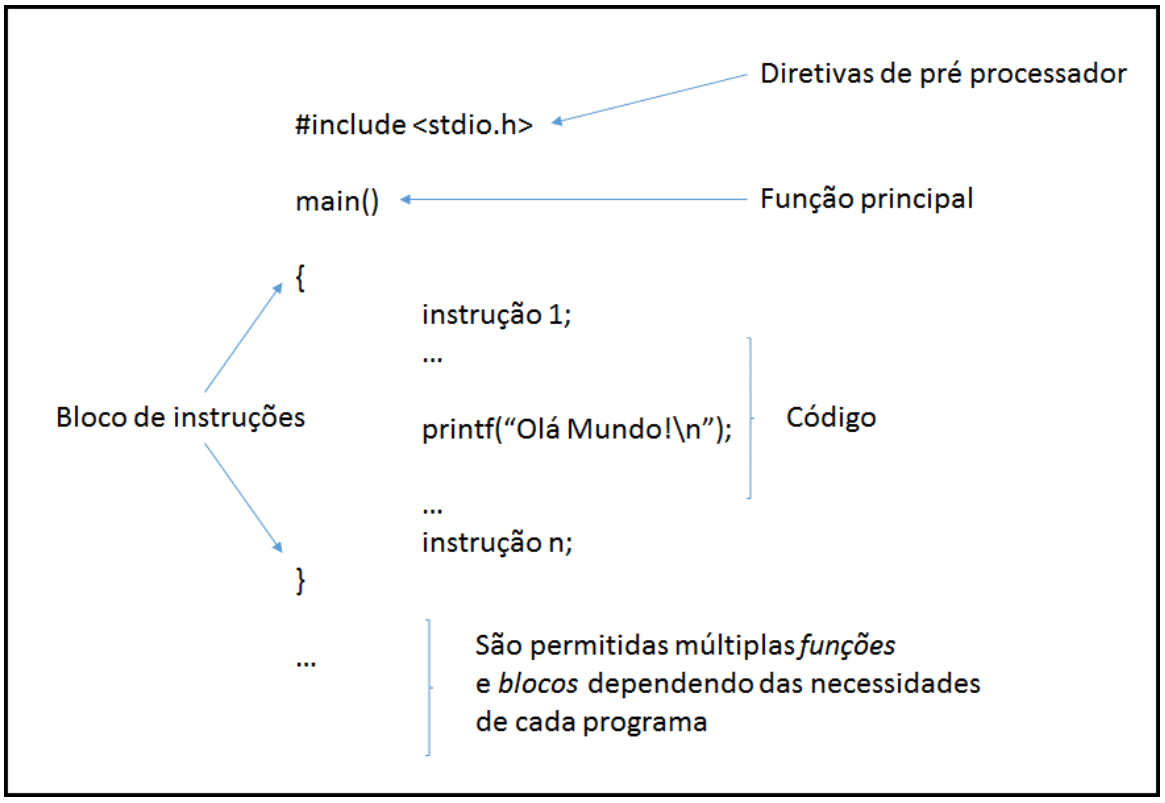



*Figura 7 - Estrutura de um programa em C.*

## 3.3 Tokens

Token é um elemento básico reconhecido pelo compilador de c. No código de um programa em c, existem vários tipos de token's: palavras reservadas, identificadores, constantes, cadeias de carateres (string), pontuações ou símbolos. Todos os token's em C são ***case sensitive***. Significa isto que é diferente escrever caracteres minúsculos ou maiúsculos.

Por exemplo, para a linguagem C as palavras: Soma, soma, SOMA são todas diferentes.

Devemos evitar este tipo de escrita (Soma, soma, SOMA) pois causam bastante confusão ao longo da codificação de um programa.

### 3.3.1 Palavras reservadas

Palavras reservadas é uma palavra que não pode ser utilizada como identificador ou variável pois é de uso da própria linguagem. **As palavras reservadas são sempre escritas em minúsculas**. Na figura seguinte são apresentadas as palavras reservadas da linguagem C.

| | | | |
|---|---|---|---|
| *auto* | *double* | *int* | *struct* |
| *break* | *else* | *long* | *switch* |
| *case* | *enum* | *register* | *typedef* |
| *char* | *extern* | *return* | *union* |
| *const* | *float* | *short* | *unsigned* |
| *continue* | *for* | *signed* | *void* |
| *default* | *goto* | *sizeof* | *volatile* |
| *do* | *if* | *static* | *while* |

*Figura 8 - Palavras reservadas da linguagem C.*

### 3.3.2 Identificadores

Identificadores são os nomes atribuídos a variáveis, funções, tipos de dados ou nomes de programas.

As regras básicas de construção de um identificador são:

1. Os identificadores podem ser constituídos por letras maiúsculas ou minúsculas, números ou o carácter de sublinhado '_';
2. O primeiro carácter não pode ser um número;
3. Não são permitidos espaços em branco e caracteres especiais, como: @, $, +, –, %, !;
4. Os identificadores não podem ser palavras reservadas.

```
Exemplos corretos:

     total, soma, iva, _taxa, media, cod_postal, x1, y3


Exemplos incorretos:

     1lugar     - porque começa por um número

     auto       - porque é uma palavra reservada
```

```
preco total      - porque contem um espaço
soma+mensal      - porque contem o carácter +
```

Tradicionalmente em C, as variáveis são escritas em letras minúsculas. As letras todas maiúsculas são utilizadas para a declaração de constantes.

## 3.4 Tipos de dados

Existem 4 tipos de dados básicos em C, a saber: `char`, `int`, `float` e `double`.

| Tipo de dado | Significado | Tamanho (bytes) | Intervalo de valores |
|---|---|---|---|
| char | Caractere | 1 | de -128 a 127 |
| int | Inteiro | 4 (em processador 32 bits) | -2147483648 a 2147483647 |
| float | Flutuante (real) | 4 | 3.4E-38 a 3.4E38 |
| double | Duplo Flutuante | 8 | 3.4E-4932 a 3.4E4932 |

Alguns autores defendem, também, a existência de um tipo especial de dados – ***void***, que significa vazio, "nada", ou ainda, sem valor definido.

### 3.4.1 char

Representa um carácter do conjunto de caracteres local. Um byte. O carácter é representado entre ' '.

Exemplos de caracteres: 'a' 'A' '3' '+' ' ' (espaço)

### 3.4.2 int

Representa o conjunto de valores inteiros.

Exemplos de valores inteiros: 3 -123 321 0 2014

### 3.4.3 float

Representa o conjunto dos valores reais. Valores com casa decimal.

Exemplos de valores float: 0.01234 3.1415 -9.56 3.0

### 3.4.4 double

Igual ao *float* mas de dupla precisão.

### 3.4.5 Modificadores de estado

Um modificador é utilizado para alterar o significado de um tipo básico e assim adaptá-lo melhor em função das necessidades. Os modificadores de tipo do C são quatro: *long*, *short*, *signed* e *unsigned*.

- ***long***: Dobra o número de bytes de um tipo básico.

- ***short***: Reduz pela metade o número de bytes de um tipo básico.
- ***signed***: Determina que o tipo básico deve reservar um bit para o controle de sinal.
- ***unsigned***: Determina que não deve ser reservado um bit para o controle do sinal do tipo básico.

Regras de aplicação dos modificadores de estado:

- Ao float não se pode aplicar nenhum.
- Ao double pode-se aplicar apenas o long.
- Os quatro podem ser aplicados a inteiros

Normalmente o resultado da combinação dos modificadores de estado com os tipos de dados básicos é a tabela X seguinte.

| Tipo de dado | Tamanho (bytes) | Intervalo de valores |
|---|---|---|
| char | 1 | de -128 a 127 |
| unsigned char | 1 | de 0 a 255 |
| signed char | 1 | de -128 a 127 |
| int | 4 (em processador 32 bits) | -2147483648 a 2147483647 |
| unsigned int | 4 (em processador 32 bits) | 0 a 4294967295 |
| signed int | 4 (em processador 32 bits) | -2147483648 a 2147483647 |
| short int | 2 (no minimo 16 bits) | -32768 a 32767 |
| unsigned short int | 2 (no minimo 16 bits) | 0 a 65535 |
| signed short int | 2 (no minimo 16 bits) | -32768 a 32767 |
| long int | 4 (em processador 32 bits) | -2147483648 a 2147483647 |
| float | 4 | 3.4E-38 a 3.4E+38 |
| double | 8 | 0.00E-01 a 1.7E+308 |
| long double | 80 bits | 3.4E-4932 a 3.4E+4932 |

Esta tabela serve de referência, pois diferentes processadores podem apresentar valores diferentes. Esta noção é importante na escolha do tipo de dados, de acordo com os requisitos do nosso problema devemos escolher tipos de dados capazes de representar esses valores.

## 3.5 Variáveis

Sempre que seja necessário guardar um valor, que não seja um valor fixo, devemos utilizar variáveis. Uma variável é um nome que é dado a uma determinada posição de memória para conter um valor de um determinado tipo de dados.

Tal como o próprio nome indica, o valor contido numa variável, pode variar ao longo da execução de um programa.

Uma variável deve ser sempre definida antes de ser utilizada.

A definição de uma variável indica ao compilador qual o tipo de dado que fica atribuído ao nome que indicamos.

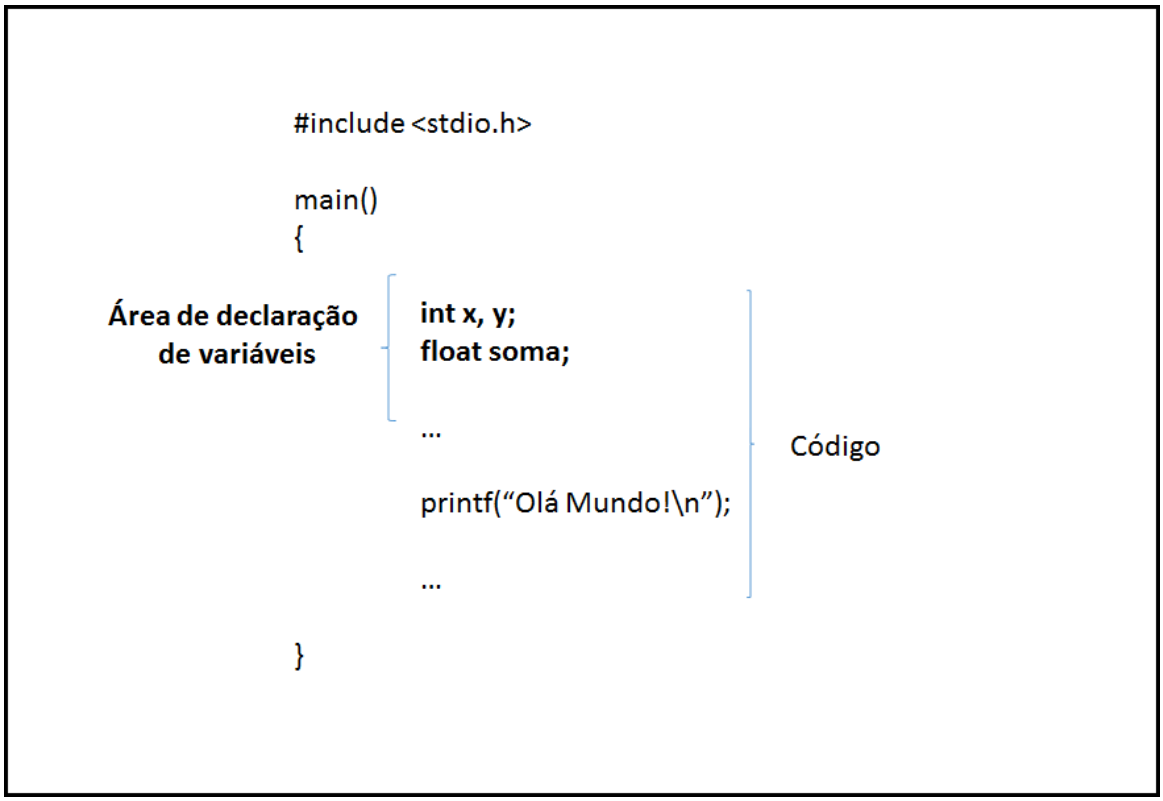
Para a declaração de uma variável utilizamos a seguinte sintaxe:

```
Tipo de dados variável1;  ou  Tipo de dados variável1, variável2;
```

Os nomes das variáveis seguem a regra dos identificadores. É boa regra de programação utilizar nomes sugestivos daquilo que representam.

```
Exemplos de nomes corretos:
     idade, morada, cod_postal, total, Numero, SALDO
     x, y, z, i, k, _alfa, _Beta, X35, H2O, _123, _999
Exemplos de declaração de variáveis:
     int idade;
     float total;
     int x, y, z, i, k;
     char _alfa, _Beta;
     double X35, H2O;
```

Apesar de apenas ser necessário declarar a variável antes de ser utilizada, é uma boa regra declarar as variáveis todas numa mesma área logo no início do programa, ver figura x.




## 3.6 Constantes

Tal como as variáveis, existem diferentes tipos de constantes em C. As constantes não alteram o seu valor ao longo do programa.

### 3.6.1 Constantes inteiras

As constantes inteiras representam valores inteiros, ou seja, números inteiros sem ponto decimal. Especial atenção deve ser dada aos valores inteiros precedidos de 0 (zero). Uma constante inteira precedida de zero representa um valor no sistema octal (base 8). A constante 024 não representa o valor 24 mas sim o valor inteiro 20 no sistema decimal (24 na base 8 -> 20 na base 10).

As constantes inteiras iniciadas por 0x são valores representados no sistema hexadecimal (base 16).

Exemplo de constantes inteiras representadas em hexadecimal:

```
0x10        0xFFFF          0x7A        0x7DD
```

### 3.6.2 Constantes reais

Valores reais ou de vírgula flutuante são representados com ponto decimal. Qualquer valor que tenha parte decimal deve ser guardado numa variável do tipo real.

Exemplos de constantes reais:

```
1.234       -90.05          2.0         .0987654
```

### 3.6.3 Constantes char

Todos os caracteres em C são representados entre ‘ ‘ (plica ou aspa simples).

Alguns exemplos de caracteres: ‘a’ ‘A’ ‘S’ ‘8’ ‘#’ ‘;’

Quando falamos em caracteres numa linguagem de programação, muitas vezes, é associada a referência à tabela ASCII (figurax). ASCII é o acrônimo para *American Standard Code for Information Interchange*. Ou seja, código padrão americano para a troca de informação.

| Dec | Hx | Oct | Char | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr | Dec | Hx | Oct | Html | Chr |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 000 | NUL (null) | 32 | 20 | 040 |   | Space | 64 | 40 | 100 | @ | @ | 96 | 60 | 140 | ` | ` |
| 1 | 1 | 001 | SOH (start of heading) | 33 | 21 | 041 | ! | ! | 65 | 41 | 101 | A | A | 97 | 61 | 141 | a | a |
| 2 | 2 | 002 | STX (start of text) | 34 | 22 | 042 | " | " | 66 | 42 | 102 | B | B | 98 | 62 | 142 | b | b |
| 3 | 3 | 003 | ETX (end of text) | 35 | 23 | 043 | # | # | 67 | 43 | 103 | C | C | 99 | 63 | 143 | c | c |
| 4 | 4 | 004 | EOT (end of transmission) | 36 | 24 | 044 | $ | $ | 68 | 44 | 104 | D | D | 100 | 64 | 144 | d | d |
| 5 | 5 | 005 | ENQ (enquiry) | 37 | 25 | 045 | % | % | 69 | 45 | 105 | E | E | 101 | 65 | 145 | e | e |
| 6 | 6 | 006 | ACK (acknowledge) | 38 | 26 | 046 | & | & | 70 | 46 | 106 | F | F | 102 | 66 | 146 | f | f |
| 7 | 7 | 007 | BEL (bell) | 39 | 27 | 047 | ' | ' | 71 | 47 | 107 | G | G | 103 | 67 | 147 | g | g |
| 8 | 8 | 010 | BS (backspace) | 40 | 28 | 050 | ( | ( | 72 | 48 | 110 | H | H | 104 | 68 | 150 | h | h |
| 9 | 9 | 011 | TAB (horizontal tab) | 41 | 29 | 051 | ) | ) | 73 | 49 | 111 | I | I | 105 | 69 | 151 | i | i |
| 10 | A | 012 | LF (NL line feed, new line) | 42 | 2A | 052 | * | * | 74 | 4A | 112 | J | J | 106 | 6A | 152 | j | j |
| 11 | B | 013 | VT (vertical tab) | 43 | 2B | 053 | + | + | 75 | 4B | 113 | K | K | 107 | 6B | 153 | k | k |
| 12 | C | 014 | FF (NP form feed, new page) | 44 | 2C | 054 | , | , | 76 | 4C | 114 | L | L | 108 | 6C | 154 | l | l |
| 13 | D | 015 | CR (carriage return) | 45 | 2D | 055 | - | - | 77 | 4D | 115 | M | M | 109 | 6D | 155 | m | m |
| 14 | E | 016 | SO (shift out) | 46 | 2E | 056 | . | . | 78 | 4E | 116 | N | N | 110 | 6E | 156 | n | n |
| 15 | F | 017 | SI (shift in) | 47 | 2F | 057 | / | / | 79 | 4F | 117 | O | O | 111 | 6F | 157 | o | o |
| 16 | 10 | 020 | DLE (data link escape) | 48 | 30 | 060 | 0 | 0 | 80 | 50 | 120 | P | P | 112 | 70 | 160 | p | p |
| 17 | 11 | 021 | DC1 (device control 1) | 49 | 31 | 061 | 1 | 1 | 81 | 51 | 121 | Q | Q | 113 | 71 | 161 | q | q |
| 18 | 12 | 022 | DC2 (device control 2) | 50 | 32 | 062 | 2 | 2 | 82 | 52 | 122 | R | R | 114 | 72 | 162 | r | r |
| 19 | 13 | 023 | DC3 (device control 3) | 51 | 33 | 063 | 3 | 3 | 83 | 53 | 123 | S | S | 115 | 73 | 163 | s | s |
| 20 | 14 | 024 | DC4 (device control 4) | 52 | 34 | 064 | 4 | 4 | 84 | 54 | 124 | T | T | 116 | 74 | 164 | t | t |
| 21 | 15 | 025 | NAK (negative acknowledge) | 53 | 35 | 065 | 5 | 5 | 85 | 55 | 125 | U | U | 117 | 75 | 165 | u | u |
| 22 | 16 | 026 | SYN (synchronous idle) | 54 | 36 | 066 | 6 | 6 | 86 | 56 | 126 | V | V | 118 | 76 | 166 | v | v |
| 23 | 17 | 027 | ETB (end of trans. block) | 55 | 37 | 067 | 7 | 7 | 87 | 57 | 127 | W | W | 119 | 77 | 167 | w | w |
| 24 | 18 | 030 | CAN (cancel) | 56 | 38 | 070 | 8 | 8 | 88 | 58 | 130 | X | X | 120 | 78 | 170 | x | x |
| 25 | 19 | 031 | EM (end of medium) | 57 | 39 | 071 | 9 | 9 | 89 | 59 | 131 | Y | Y | 121 | 79 | 171 | y | y |
| 26 | 1A | 032 | SUB (substitute) | 58 | 3A | 072 | : | : | 90 | 5A | 132 | Z | Z | 122 | 7A | 172 | z | z |
| 27 | 1B | 033 | ESC (escape) | 59 | 3B | 073 | ; | ; | 91 | 5B | 133 | [ | [ | 123 | 7B | 173 | { | { |
| 28 | 1C | 034 | FS (file separator) | 60 | 3C | 074 | < | < | 92 | 5C | 134 | \ | \ | 124 | 7C | 174 | | | \| |
| 29 | 1D | 035 | GS (group separator) | 61 | 3D | 075 | = | = | 93 | 5D | 135 | ] | ] | 125 | 7D | 175 | } | } |
| 30 | 1E | 036 | RS (record separator) | 62 | 3E | 076 | > | > | 94 | 5E | 136 | ^ | ^ | 126 | 7E | 176 | ~ | ~ |
| 31 | 1F | 037 | US (unit separator) | 63 | 3F | 077 | ? | ? | 95 | 5F | 137 | _ | _ | 127 | 7F | 177 | &#127; | DEL |

Source: www.LookupTables.com

*Figura 9 - Tabela ASCII*

### 3.6.4 Constantes String

Uma constante *string* é sempre representada entre " " (aspas). Uma string é um conjunto de caracteres. Exemplos de constantes *string*:

"Olá Mundo!"

"Programar em C é espetacular"

"6300 Guarda"

"2013" " " "Z" "1"

Na figura seguinte podemos verificar a representação em memória de uma *string*. De salientar o limitador de string 0 (zero), que tem a função de indicar o fim da *string*.

| | O | l | á | | M | u | n | d | o | ! | 0 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

*Figura 10 - Representação em memória de uma string.*

Todas as *string's* terminam por zero '0'. O carácter não é visível. Não é necessário introduzir este carácter, porque sempre que o compilador analisa um conjunto de caracteres entre " " (aspas), introduz automaticamente o limitador de *string*. O limitador de *string* é diferente do carácter '0'. Verificando a tabela ASCII, podemos constatar que para o limitador de string corresponde o valor decimal 0 (null) e o carácter '0' corresponde o valor decimal 48.

## 3.7 Comentários

Os comentários não se destinam a ser executados pelo compilador. Os comentários destinam-se unicamente a documentar o programa, de forma a facilitar a sua compreensão.

Um comentário em C é delimitado pelos caracteres:

/* - Início de comentário

*/ - Fim do comentário

Exemplo:

```
/* Primeiro programa, escreve no ecrã
   a mensagem: Olá Mundo!              */
#include <stdio.h>
main() /* inicio da função principal */
{
     printf("Olá Mundo!\n");
}
```

## 3.8 Atribuição

Sempre que uma variável é declarada, estamos a solicitar ao compilador para reservar um espaço em memória para a armazenar. O tamanho do espaço a esse armazenamento será definido pelo tipo de dados.

A atribuição de um valor a uma variável e efetuado através do sinal de atribuição: **=** (igual).

A atribuição é sempre efetuada através da seguinte expressão:

```
variável = expressão
```

Exemplos de atribuição:

```
float x = 2.5;

int i = 5, y = -8;

int a = b = c = d = 5;

char letra;

letra = ´A´;

float iva;

iva = preco * 0.23;
```

## 3.9 Operadores simples

Os operadores em C são os símbolos usados para as operações matemáticas usuais conhecidas como a adição, a subtração, a multiplicação e a divisão.

Na figura , estão representadas os operadores simples e o seu significado.

| Símbolo | Significado |
|---|---|
| + | Adição |
| - | Subtração |
| * | Multiplicação |
| / | Divisão |
| % | Resto da divisão |

*Figura 11 - Operadores em C*

Na figura , são apresentadas exemplos de operações com operandos inteiros. De salientar a operação de divisão inteira. Designamos de divisão inteira pelo facto de envolver dois operandos inteiros (21 e 4) o resultado continua a ser um valor inteiro (5), e não 5.25. Especial atenção deve ser dada a esta operação, pois podem resultar valores muito diferentes do esperado.

| Operação | Descrição | Exemplo | Resultado |
|---|---|---|---|
| + | Soma | 21 + 4 | 25 |
| - | Subtracção | 21 - 4 | 17 |
| * | Multiplicação | 21 * 4 | 84 |
| / | Divisão inteira | 21 / 4 | 5 |
| % | Resto da divisão inteira | 21 % 4 | 1 |

*Figura 12 - Operações com inteiros.*

### 3.9.1 Atribuição composta

É utilizada com bastante frequência a atualização de uma variável com um novo valor. Esta operação de atualização pode ser efetuada com qualquer um dos operadores descritos. Alguns exemplos:

| Operador | Exemplo | Expressão equivalente |
|---|---|---|
| += | soma += 50 | soma = soma + 50 |
| -= | valor -= 5 | valor = valor - 5 |
| *= | produto *= 3 | produto = produto * 3 |
| /= | preco /= 12 | preco = preco / 12 |
| %= | resto %= 2 | resto = resto % 2 |

A expressão geral será:

***valor operador= expressão***

para expressões do tipo:

***valor = valor operador (expressão)***

### 3.9.2 Precedência dos operadores

Tal como na matemática, os operadores têm uma ordem de precedência de execução. A ordem de execução é exatamente igual à matemática, a saber:

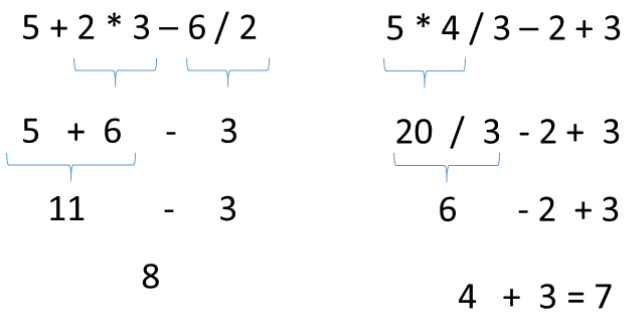
- E primeiro lugar: Multiplicação, divisão e resto da divisão.
- Em segundo lugar: Soma e subtração.

Sejamos o exemplo seguinte:

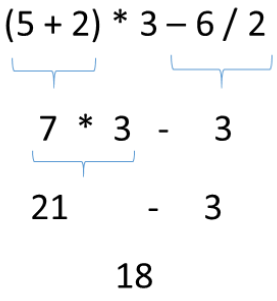
```
2 + 3 * 4
```

O resultado é 14 (1º: 3*4 = 12, 2º: 2 + 12 = 14). No entanto se efetuarmos a operação sem precedência de operadores o resultado seria 20 (2 + 3 = 5 * 4 = 20).

Exemplos:




A ***utilização de parêntesis*** permite alterar a ordem de precedência estabelecida. Em primeiro lugar é efetuado o que estiver dentro de parêntesis. Exemplo:




## 3.10 Input e Output

Na maioria dos programas é necessário a introdução e visualização de dados. Uma das formas de introduzir dados num programa (input) ou de mostrar no ecrã os resultados de um programa (output) é através da utilização funções standard de input e output disponibilizada pela linguagem C. Deste modo, para a introdução de dados num programa vamos utilizar a função ***scanf()*** e para escrever no ecrã vamos usar a função ***printf()***, funções existentes na biblioteca de funções da linguagem C - ***stdio.h***.

### 3.10.1 printf

A função ***printf()*** permite enviar para o output, normalmente o ecrã, texto formatado de acordo com os seus parâmetros.

Uma das formas mais simples de escrever no ecrã é incluir nos parâmetros da função uma constante do tipo *string*. Por exemplo:

```
printf("Olá Mundo!");      - escreve no ecrã o valor da string "Olá Mundo!";

printf("2013");            - escreve no ecrã o valor da string "2013";
```

Outro exemplo:

```
printf("Linha 1");
printf("Linha 2");
printf("Linha 3");
```

Este conjunto de instruções produz o output seguinte:

```
Linha 1Linha 2Linha 3
```

O resultado produzido, provavelmente, não é o pretendido. Para produzir o output:

```
Linha 1
Linha 2
Linha 3
```

É necessário incluir em cada uma das instruções o carácter especial **\n**, que significa mudança de linha.

```
printf("Linha 1\n");
printf("Linha 2\n");
printf("Linha 3\n");
```

Verifique o output da seguinte instrução:

```
printf("\n\n.1\n..2\n...3\n");
```

**Output:**

```
- (linha em branco)
- (linha em branco)
.1
..2
...3
-
```

Como vemos, para além dos caracteres normais incluídos na *string*, existem caracteres especiais precedidos por **** (barra invertida). Este conjunto de caracteres é denominado de "*escape sequences*". Um dos exemplos já utilizados é a mudança de linha \n. Alguns exemplos de *escape sequences*:

```
\7    - sinal sonoro
\a    - sinal sonoro
\b    - BackSpace
\n    - new line
\\    - forma de representar \
\'    - Carácter '
```

```
\”    - Carácter ”
\t    - tab
```

Outra variação possível é a utilização dos valores de variáveis. Neste caso utilizamos o carácter de controlo de variável **%** (sinal de percentagem) seguido do carácter que corresponde o tipo de dados a apresentar. Deste modo, numa *string* o sinal de % indica que nessa posição vai ser substituído por um valor. Para os diferentes tipos de dados são utilizadas caracteres distintos, a saber:

```
%s    - Conjunto de caracteres (String)
%c    - Carácter
%d    - Inteiro
%f    - Real (Floating-point)
%e    - Real (notação exponencial)
%o    - Valores inteiros em Octal
%x    - Valores inteiros em Hexadecimal
%%    - Escreve o símbolo % (percentagem)
```

Exemplos:

```
1.  int idade = 20;
2.  float iva = .23;
3.  char letra = 'Z';
4.  int valor = 123;
5.  int simbolo = 65;

6.  printf("Idade = %d anos\n", idade);
7.  printf("Valor do IVA = %f\n", iva);
8.  printf("Letra = %c \n", letra);
9.  printf("123(base 10) = %o (base 8) = %x (base 16)\n",valor, valor);
10. printf(“Inteiro = %d \t Real = %f \n”, 8, 5.25);
11. printf(“Caracter = %c\n”, simbolo);
12. printf(“Valor da letra = %d\n”, letra);
13. printf("Programar em C \n");
14. printf("%s \n", "Programar em C");
```

Output

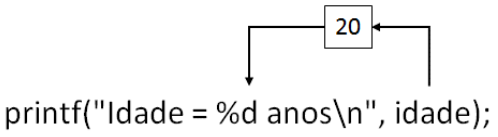
```
6.   Idade = 20 anos
7.   Valor do IVA = 0.230000
8.   Letra = Z
9.   123(base 10) = 173 (base 8) = 7b (base 16)
10.  Inteiro = 8   Real = 5.250000
11.  Caracter = A
12.  Valor da letra = 90
13.  Programar em C
14.  Programar em C
```

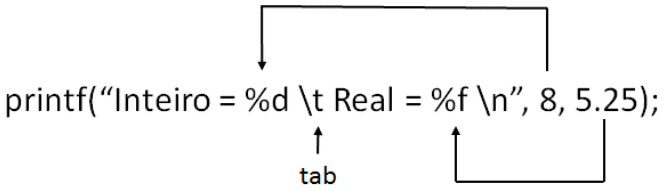
No caso da instrução 11 é dada a instrução para escrever um valor do tipo carácter **%c,** no entanto o valor da variável *simbolo* é o inteiro 65, neste caso o valor inteiro é convertido para o respetivo

carácter da tabela ASCII cujo valor inteiro é 65, o que corresponde a 'A'. Na instrução 12 é executada a operação inversa, ou seja, mandamos escrever um inteiro **%d** mas é colocado um carácter, pelo que o carácter **'Z'** é substituído pelo seu valor da tabela ASCII, o valor 90.

Verifique que as instruções 13 e 14 são idênticas. Pelo que a utilização de **%s** é redundante, nestes casos devemos utilizar simplesmente: **`printf("Programar em C \n");`**

Nas figuras seguintes, podemos observar o funcionamento da função ***printf***.




Muitas variações de formatação do output podem ser utilizadas, e que serão apresentadas ao longo deste manual. No entanto, para iniciar o estudo da programação não é necessário conhecer todas as formas possíveis, mas sim algumas das mais comuns.

Uma das utilizações mais comuns é alinhar à esquerda ou à direita o valor a escrever. Repare no exemplo seguinte, na linha 3 é associado ao controlo de variável o valor inteiro 5, com o significado de alinhar à direita com o tamanho de 5 caracteres, ou com a referência de tamanho 5. A utilização do formato **%-5d** significa alinhar à esquerda reservando o 5 espaços de caracteres. Se nada for especificado o resultado será alinhado à esquerda com o espaço necessário a escrever o valor.

```
1. printf("12345678\n");
2. printf("%d\n",21);
3. printf("%5d\n",21);
4. printf("%-5d\n",21);
5. printf("%5.2f\n",5.38);
```

Output

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| 2 | 1 | | | | | | |
| | | | 2 | 1 | | | |
| 2 | 1 | | | | | | |
| | 5 | . | 3 | 8 | | | |

Por último, é de salientar o formato especificado na instrução da linha 5. Formato de grande utilidade, pois permite formatar um valor real, ver figura x.

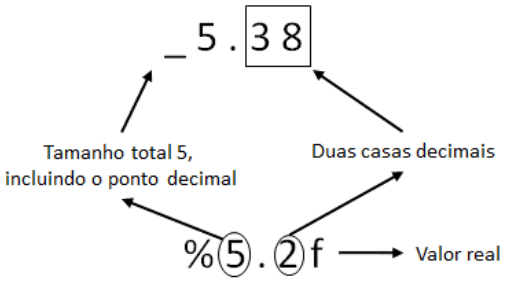



*Figura 13 - Formatação output com valor real.*

Vejamos o exemplo seguinte:

```
                                                    Output
printf("%6.2f\n", 1.618033989);                      1.62
```

É de salientar que o valor foi arredondado na 2 casa decimal. Vejamos outro exemplo:

```
                                                    Output
printf("%6.2f\n", 12345.618033989);                  12345.62
```

Neste caso, foi especificado o tamanho máximo de 6 **(3 para a parte inteira + 1 para o ponto decimal + 2 para a parte decimal)**, mas como a parte inteira são necessários 5, é criado automaticamente o tamanho necessário à representação da parte inteira. Por este motivo muitos programadores ignoram este formato, especificando apenas o valor para a parte decimal. Por exemplo:

```
                                                    Output
printf("%.2f\n", 12345.618033989);                   12345.62
```

### 3.10.2 scanf()

A função *scanf* é uma das formas de introdução de dados no programa. A função *scanf* funciona de modo semelhante ao *printf*. O primeiro argumento da função *scanf* é o formato do tipo de variável que queremos ler. O segundo argumento é a variável que queremos ler precedida do sinal **&**, exceto se for uma *string* (ver figura X). Este sinal indica que é para colocar no endereço da variável.

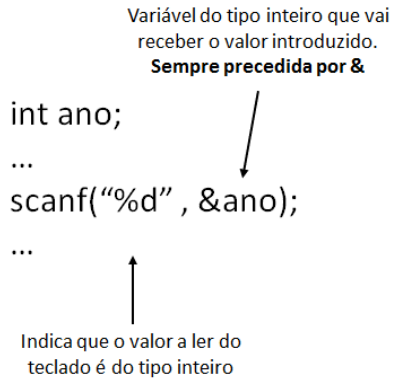
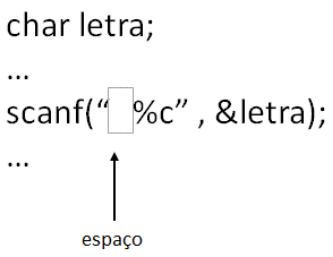



*Figura 14 - Exemplo de utilização de scanf().*

Exemplos:

```
scanf("%d", &idade);       - leitura de um valor int para a variável idade.
scanf("%f", &preco);       - leitura de um valor float para a variável preco.
scanf("%c", &letra);       - leitura de um valor char para a variável letra.
scanf("%s", nome);         - leitura de um valor string para a variável nome.
                             Nota: Não tem & porque se trata de uma string
```

Especial atenção deve ser dada a leitura de valores do tipo *char* e *string*. Por vezes quando existe uma instrução de leitura deste tipo, ao executar o código constatamos que o programa parece não executar essas instruções. Isto é devido ao *buffer* do teclado conter caracteres, pelo que são atribuídos de imediato a estas variáveis. Uma das formas de resolver este problema é incluir a instrução **`fflush(stdin);`** antes da instrução *scanf*. Esta instrução permite limpar o conteúdo do *buffer* do teclado (entrada padrão/*stdin*). Outra das formas de resolver o problema descrito é deixar um espaço antes da especificação do tipo de dados, como na figura x.




## 3.11 Exercícios resolvidos

Com os conteúdos transmitidos, até aqui, estamos em condições de começar a fazer alguns programas.

De seguida apresentamos alguns exercícios de programação resolvidos.

*Exercicio01 - Faça um programa que permita a soma de dois valores inteiros introduzidos pelo utilizador.*

1. Comece por criar um projeto, utilizando o Pelles C, com o nome **exercicio01**. Ver capítulo 3.
2. Depois de criar o projeto e o ficheiro *main.c* está pronto a introduzir o código em C necessário à resolução do problema. Introduzir o código logo a seguir à primeira chaveta.

3. Comece por pensar na resolução do problema. Se não souber resolver, será impossível efetuar qualquer programa independentemente da linguagem de programação.
4. De seguida escolha os nomes das variáveis que vai utilizar e dos respetivos tipos de dados.
5. Neste caso, vamos necessitar de 3 variáveis: uma para um dos valores, outra para o segundo valor introduzido, e ainda, uma outra para guardar o resultado final.
6. É boa regra escolher nomes que estejam de acordo com o problema. Por exemplo, será um bom nome: *valor1* e *valor2*, ou *numero1* e *numero2*. Obedecem às regras definidas para os identificadores e de alguma forma representam o problema. Já se a escolha incidir sobre o nome *texto*, pode ser utilizado não está de acordo com a sua função. E não será uma boa prática.
7. Variáveis a utilizar são: ***valor1*** e ***valor2*** para cada um dos valores a introduzir pelo utilizador; e ***soma*** para guardar o resultado da adição. O tipo de dados será ***int***, tal como o próprio enunciado indica.
8. Até aqui o programa ficará:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
      int valor1, valor2;
      int soma;



    return 0;
}
```

9. O passo seguinte ler os valores introduzidos pelo utilizador através do teclado. Como acabamos de estudar a instrução que faz essa operação é a função ***scanf()***.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
      int valor1, valor2;
      int soma;

      scanf("%d",&valor1);
      scanf("%d",&valor2);

    return 0;
}
```

10. Experimente executar o programa. Verifique que parece que o programa não está a fazer nada. No entanto, o programa está à espera que sejam introduzido os valores. Nestas situações devemos enviar para o ecrã uma mensagem a dizer que o utilizador deve introduzir dados no programa.
11. Para escrever uma mensagem no ecrã utilizamos a função ***printf***.

```
#include <stdio.h>
```

```
int main(int argc, char *argv[])
{
      int valor1, valor2;
      int soma;

      printf("Qual o primeiro valor? ");
      scanf("%d",&valor1);
      printf("Qual o segundo valor? ");
      scanf("%d",&valor2);

    return 0;
}
```

12. O passo seguinte é efetuar a soma dos dois valores e guardar o resultado na variável soma.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
      int valor1, valor2;
      int soma;

      printf("Qual o primeiro valor? ");
      scanf("%d",&valor1);
      printf("Qual o segundo valor? ");
      scanf("%d",&valor2);
      soma = valor1 + valor2;

    return 0;
}
```

13. Por último, falta escrever o resultado no ecrã.

```
#include <stdio.h>

int main(int argc, char *argv[])
{
      int valor1, valor2;
      int soma;

      printf("Qual o primeiro valor? ");
      scanf("%d",&valor1);
      printf("Qual o segundo valor? ");
      scanf("%d",&valor2);
      soma = valor1 + valor2;
      printf("%d",soma);

    return 0;
}
```

14. Podemos formatar o output de acordo com os objetivos ou necessidade do problema. Para este exemplo, vamos formatar o output de acordo com o aspeto seguinte: **3 + 5 = 8**

```
#include <stdio.h>

int main(int argc, char *argv[])
{
      int valor1, valor2;
      int soma;

      printf("Qual o primeiro valor? ");
      scanf("%d",&valor1);
      printf("Qual o segundo valor? ");
      scanf("%d",&valor2);
      soma = valor1 + valor2;
      printf("%d + %d = %d \n",valor1, valor2, soma);

      return 0;
}
```

15. O programa está finalizado. É de salientar que as instruções devem ser escritas uma por linha e alinhadas, como no programa anterior. Verifique o programa seguinte:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
      int valor1, valor2;int soma;

      printf("Qual o primeiro valor? ");
scanf("%d",&valor1);
      printf("Qual o segundo valor? ");scanf("%d",&valor2);
soma = valor1 + valor2;
            printf("%d + %d = %d \n",valor1, valor2, soma);

return 0;
}
```

16. Apesar de o programa ser exatamente igual ao anterior e ter o mesmo comportamento e resultado não é de fácil leitura, pelo que deve ser evitado.

*Exercicio02 - Faça um programa que permita o cálculo da área de um quadrado.*

Para a resolução deste exercício procure seguir os passos do exemplo anterior.

1. Área do quadrado = lado x lado.
2. Nomes e tipos de dados das variáveis, por exemplo: ***lado*** do tipo real (float), ***area*** do tipo real (float).

```
#include <stdio.h>
int main(int argc, char *argv[])
{
      float lado;
      float area;

      printf("Qual comprimento do lado do quadrado? ");
      scanf("%f",&lado);

      area = lado * lado;
      printf("A área do quadrado de lado %.2f é igual a %.2f \n",lado,area);

      return 0;
}
```

*Exercicio03 - Dado o preço de um produto, calcule o valor final a pagar quando aplicado um desconto de 5%.*

1. Calcular o valor de desconto: desconto = preço * valor de desconto
2. Valor final a pagar = preço – desconto

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    float preco;
     float desconto;
     float valor_desconto = 0.05; /* valor de desconto 5%
                                     ou seja 5/100 = 0.05    */
     float valorFinal;

     printf("Preço do produto? ");
     scanf("%f",&preco);

     desconto = preco * valor_desconto;
     valorFinal = preco - desconto;

     printf("Valor a Pagar = %.2f\n",valorFinal);

    return 0;
}
```

## 3.12 Exercícios propostos

***Exercicio04*** – Faça um programa que dada uma temperatura em graus celsius a converta em graus Fahrenheit.

***Exercicio05*** – Faça um programa que converta uma temperatura em graus celsius para graus kelvin.

***Exercicio06*** – Pretende-se um programa que dado um valor em euros apresente uma tabela com as respetivas conversões para as diferentes moedas, que a seguir se apresentam. Os valores de output devem surgir com duas casas decimais.

- DÓLAR DOS ESTADOS UNIDOS
- REAL DO BRASIL
- DÓLAR CANADIANO
- IENE JAPONÊS
- RUBLO RUSSO
- KUANZA

***Exercicio07*** – Faça um programa que permita a conversão de um valor inteiro introduzido, no seu respetivo valor **octal** (base 8) e **hexadecimal** (base 16).

***Exercicio08*** – Pretende-se um programa que permita a introdução de um carácter e escreva o seu valor inteiro, segundo a tabela ASCII.

***Exercicio09*** – Construa um programa que permita calcular e apresentar o número de notas e moedas mínimo existente num valor dado.

***Exercicio10*** – Conceba um programa que permita determinar a hora de chegada (horas, minutos, segundos) mediante a introdução de uma hora de partida e o tempo de viagem. Apresentar o resultado com o aspeto de **hh:mm:ss**.

***Exercicio11*** – Faça um programa que permita somar os dígitos de um valor inteiro introduzido, pertencente ao intervalo [100, 999].

***Exercicio12*** – Criar um programa que permita inverter um valor inteiro introduzido, pertencente ao intervalo [100, 999].

# 4 Estruturas de controlo

As estruturas de controlo são fundamentais para qualquer linguagem de programação. São as estruturas de controlo que permitem alterar o normal fluxo de execução de um programa. Antes de conhecer as diferentes estruturas de controlo é necessário saber utilizar os operadores relacionais.

## 4.1 Operadores relacionais

Os operadores relacionais permitem comparar ordens de grandeza. Possibilitam, por exemplo, determinar se um número é maior que outro, ou se é menor, ou até igual. Na figura seguinte são apresentados os operadores relacionais.

| Operadores relacionais | Significado |
|---|---|
| == | Igual |
| < | Menor que |
| <= | Menor ou igual que |
| > | Maior que |
| >= | Maior ou igual que |
| != | Diferente |

Alguns exemplos:

```
int a = 10;
int b = 8;
int c = 12;
int d = 8;
```

As expressões seguintes são **verdadeiras**:

```
a == b
b < c
b <= a
c > a
b >= d
a != c
```

Estas instruções são operações lógicas de comparação de valores entre duas variáveis. Na linguagem C, o resultado destas expressões é ***0*** (zero) para um resultado ***falso*** (*False*) e diferente de zero para um resultado **verdadeiro** (*True*), em muitos casos é associado o valor **1** (Um) a um resultado verdadeiro.

## 4.2 Instrução if

Os operadores relacionais, atrás mencionados, estão diretamente relacionados com a instrução ***if***. A instrução ***if*** é utilizada para a tomada de decisões. Esta instrução testa o resultado de uma comparação entre dois valores, utilizando operadores relacionais. Baseado no resultado, a instrução ***if***, faz decisão de executar ou não um conjunto de instruções.

### 4.2.1 Instrução if simples

A sintaxe mais simples de utilização da instrução ***if*** é a seguinte:

```
if (condição){
```

```
        Instrução ou conjunto de instruções;
}
```

A ***condição*** expressa uma qualquer expressão de comparação usando operadores relacionais, sempre entre parêntesis.

Entre chavetas seguem-se instruções em linguagem C, que serão executadas caso a condição seja verdadeira, ou como já vimos, o resultado seja diferente de zero. No caso de o resultado da condição ser falsa, igual a zero, as instruções entre chavetas não são executadas. O programa segue a sua execução normal logo a seguir ao fechar chaveta associado à instrução ***if***. Se se tratar de uma única instrução podemos omitir as chavetas. No entanto é boa regra incluir sempre as chavetas.

Atenção ao erro muitas vezes cometido pelos principiantes à programação, a introdução de ponto e vírgula (sinal de fim de instrução) depois da condição. Por exemplo:

```
if (condição);
{
        Instrução ou conjunto de instruções;
}
```

Neste caso, o bloco de instruções é sempre executado. A instrução ***if*** termina com o **;**.

*Alguns exemplos:*

Se o valor total a pagar for superior a 1000€ então o cliente terá um desconto de 5%.

```
if (precoTotal > 1000) {
        precoTotal = precoTotal – precoTotal * 0.05;
}
```

Escrever “Maior de idade” se a idade for maior ou igual a 18.

```
if (idade >= 18) {
        printf(“Maior de idade”);
}
```

Verifica e escreve se um valor é negativo.

```
if (valor < 0) {
        printf(“O valor é negativo”);
}
```

### 4.2.2 Instrução if - else

Outra forma de utilização da instrução **if** é a sua combinação com **else**. O *else* não pode existir sem *if*. O formato de utilização desta combinação ***if – else*** é a seguinte:

```
if (condição){
        Instrução ou conjunto de instruções A;
} else {
        Instrução ou conjunto de instruções B;
}
```

A primeira parte desta combinação é exatamente igual à anterior, ou seja, se a condição for verdadeira (1) é executada a instrução ou conjunto de instruções A. No caso de a condição ser falsa (0) será executada a instrução ou conjunto de instruções B associado ao ***else***.

Exemplos de utilização ***if-else***:

Escrever se um valor é positivo ou negativo.

```
if (valor >= 0) {
        printf("O valor é positivo");
} else {
        printf("O valor é negativo");
}
```

Uma loja em promoções está a fazer descontos de 10% em todos os produtos, no entanto se o valor total de compras for superior a 150€ faz um desconto de 25%.

```
if (total > 150) {
        desconto = total * 0.25;
} else {
        desconto = total * 0.10;
}
total = total – desconto;
```

### 4.2.3 if - else encadeado

Por vezes existe a necessidade de implementar uma decisão múltipla, ou seja, com diversas condições. Uma forma de simplificar a sua implementação é utilizar a sequência seguinte:

```
if (condição){
        Instrução ou conjunto de instruções;
} else {
        if (condição){
                Instrução ou conjunto de instruções;
        } else {
                if (condição){
                        Instrução ou conjunto de instruções;
                } else {
                        Instrução ou conjunto de instruções;
                }
        }
}
```

No entanto a forma mais usual de escrever esta sequência é:

```
if (condição){
        Instrução ou conjunto de instruções;
} else if (condição){
        Instrução ou conjunto de instruções;
} else if (condição){
        Instrução ou conjunto de instruções;
} else {
        Instrução ou conjunto de instruções;
}
```

O compilador inicia a avaliação das condições e logo que encontre uma verdadeira executa a instrução ou conjunto de instruções que lhe está associada, ignorando o resto da sequência. Se nenhuma das condições for verdadeira então é executada a instrução correspondente ao último *else*.

*Exemplos de aplicação de if encadeado.*

Determinar se dois valores são iguais ou qual é o maior.

```
if (valor1 == valor2){
        printf("Os valores são iguais");
 } else if (valor1 > valor2){
        printf(" o valor %d é maior que %d \n", valor1, valor2);
} else {
        printf(" o valor %d é maior que %d \n", valor2, valor1);
}
```

Escrever a respetiva nota qualitativa de acordo com a seguinte tabela:

| Nota | Valor Qualitativo |
|---|---|
| [0, 45] | Não Satisfaz |
| ]45, 55] | Satisfaz Pouco |
| ]55, 75] | Satisfaz |
| ]75, 85] | Bom |
| ]85, 100] | Muito Bom |

```
if (nota <= 45){
        printf("Não Satisfaz");
 } else if (nota <= 55){
        printf("Satisfaz Pouco");
} else if (nota <= 75){
        printf("Satisfaz");
} else if (nota <= 85){
        printf("Bom");
} else {
        printf("Muito Bom");
}
```

## 4.3 Operadores lógicos

Em muitas situações existe a necessidade de testar mais do que um conjunto de valores. Para testar mais do que uma condição relacional temos de utilizar os operadores lógicos (ver Figura 15).

| Operador Lógico | Significado |
|---|---|
| & & | E ou AND |
| ¦ ¦ | OU ou OR |
| ! | Negação ou NOT |

*Figura 15 - Operadores lógicos*

Na Figura 16 podemos verificar a tabela de verdade dos operadores lógicos, ou seja, como funcionam os operadores lógicos. Os valores de A e B representam condições relacionais.

| A | B | A && B | A \|\| B | !A | !B |
|---|---|---|---|---|---|
| VERDADEIRO | FALSO | FALSO | VERDADEIRO | FALSO | VERDADEIRO |
| VERDADEIRO | VERDADEIRO | VERDADEIRO | VERDADEIRO | FALSO | FALSO |
| FALSO | FALSO | FALSO | FALSO | VERDADEIRO | VERDADEIRO |
| FALSO | VERDADEIRO | FALSO | VERDADEIRO | VERDADEIRO | FALSO |

*Figura 16 - Tabela de verdade*

Exemplos:

Calcular o vencimento final a pagar depois de aplicada a taxa de imposto a cobrar, de acordo com a tabela seguinte:

| Vencimento | Taxa |
|---|---|
| ]0, 1000] | 0% |
| ]1000, 2000] | 3% |
| ]2000, 5000] | 5% |

```
if ((vencimento > 0) && (vencimento <= 1000)){
        taxa = 0;
} else if ((vencimento > 1000) && (vencimento <= 2000)){
        taxa = 0.03;
} else if ((vencimento > 2000) && (vencimento <= 5000)){
        taxa = 0.05;
}
valorFinal = vencimento – vencimento * taxa;
```

Neste exemplo, foram expressos os intervalos de acordo com a tabela definida. No entanto, podemos simplificar a representação dos intervalos como mostra o código seguinte:

```
if ((vencimento > 0) && (vencimento <= 1000)){
        taxa = 0;
} else if (vencimento <= 2000){
        taxa = 0.03;
```

```
} else if (vencimento <= 5000){
        taxa = 0.05;
}
valorFinal = vencimento – vencimento * taxa;
```

Escrever se um dado carácter introduzido é um dígito, letra ou outro.

```
if ((carater >= '0') && (carater <= '9')){
        printf("Digito");
} else if (((carater >='A') && (carater <= 'Z')) || ((carater >='a') && (carater <= 'z'))){
        printf("Letra");
} else {
        printf("Outro");
}
```

## 4.4 Exercícios propostos

***Exercicio13*** – Faça um programa que permita escrever no ecrã dois valores introduzidos ordenados por ordem crescente.

***Exercicio14*** – Construa um programa que escreve no ecrã o maior de dois valores introduzidos, com o output seguinte:

**Input: 5 3 Output: O maior de 5 e 3 = 5**

***Exercicio15*** – Produza um programa que escreve no ecrã o maior de três valores introduzidos, com o output seguinte:

**Input: 5 3 9 Output: O maior de 5 e 3 e 9 = 9**

***Exercicio16*** – Escrever a respetiva nota qualitativa de acordo com a seguinte tabela:

| Nota | Valor Qualitativo |
|---|---|
| [0, 45] | Não Satisfaz |
| ]45, 55] | Satisfaz Pouco |
| ]55, 75] | Satisfaz |
| ]75, 85] | Bom |
| ]85, 100] | Muito Bom |

***Exercicio17*** – Faça um programa em linguagem C, que permita determinar o valor do Imposto Único de Circulação (IUC) a pagar para um determinado veículo matriculado a antes de 1 de Julho de 2007.
De acordo com os dados da tabela abaixo (tabela simplificada para este exercício), calcule o valor do IUC a pagar, de um veículo a gasolina, dado o valor da cilindrada e o ano da matrícula.
O output do programa deve ter o aspeto seguinte:

| Gasolina Cilindrada | Posterior a 1995 | De 1990 a 1995 | De 1981 a 1989 |
|---|---|---|---|
| Até 1.000 | 16,50 € | 10,40 € | 7,30 € |
| Entre 1.001 e 1.300 | 33,10 € | 18,60 € | 10,40 € |
| Mais de 1.300 | 51,70 € | 28,90 € | 14,50 € |

Cilindrada do veículo: 1200 cm3
Ano da matrícula: 1991
Total a Pagar 18,60€

***Exercicio18*** – Criar um programa que permita determinar se um valor inteiro introduzido, pertencente ao intervalo [100, 999], é uma capicua.

***Exercicio19*** – Faça um programa que permita somar os dígitos pares de um valor inteiro introduzido, pertencente ao intervalo [100, 999].

***Exercicio20*** – Faça programa que permita determinar se um número é divisível por 3, aplicando a regra seguinte: um número é divisível por 3, se a soma dos seus dígitos é divisível por 3.

Exemplo: 381 (3 + 8 + 1 = 12 / 3 = 4 ) Sim

217 (2 + 1 + 7 = 10 / 3 = 3,33) Não

***Exercicio21*** – Faça programa que permita determinar se um número é divisível por 4, aplicando a regra seguinte: um número é divisível por 4, se os dois últimos dígitos são um número divisível por 4.

Exemplos: 1312 (12 / 4 = 3) Sim

7018 (18 / 4 = 4,5) Não

***Exercicio22*** – Faça programa que permita determinar se um número é divisível por 5, aplicando a regra seguinte: um número é divisível por 5, se o último dígito é 0 ou 5.

Exemplo: 175 Sim

809 Não

***Exercicio23*** - Efetuar um programa que permita converter uma letra minúscula em maiúscula e vice-versa.

***Exercicio24*** – Partindo do Exercicio09, Faça um programa que permita calcular e apresentar o número de notas e moedas mínimo existente num valor dado, mas também seja possível definir o número de notas e moedas máximo possível. Ou seja, se for definido que o número máximo de notas de 500 é 1, no valor 1000 não posso atribuir 2 notas de 500.

***Exercicio25*** - Existem pelo menos 3 formas diferentes de determinar se um valor inteiro é par ou ímpar. Implemente as 3 formas num programa.

## 4.5 Operador condicional

O operador condicional é o único operador ternário da linguagem C. Este operador pode substituir a instrução ***if*** em algumas situações. A sintaxe de utilização é a seguinte:

Expressão condicional **?** expressão1 **:** expressão2

A ***expressão condicional*** é uma qualquer expressão em C, de onde resulta o valor verdade (*true* ou *valor diferente de zero*) ao falso (*false* ou *valor zero*). Se o resultado da expressão condicional for verdadeiro a ***expressão1*** é executada. Por outro lado, se o resultado da expressão condicional for falso é executada a ***expressão2***.

Exemplo:

```
(valor >= 0) ? printf("O valor é positivo") : printf("O valor é negativo");
```

Que seria idêntico à utilização seguinte, utilizando a instrução if.

```
if (valor >= 0) {
        printf("O valor é positivo");
} else {
        printf("O valor é negativo");
}
```

Exemplo:

```
total = total - ((total > 150)? total * 0.25 : total * 0.10);
```

Seria semelhante a utilizar:

```
if (total > 150) {
        desconto = total * 0.25;
} else {
        desconto = total * 0.10;
}
        total = total – desconto;
```

Vejamos ainda o exemplo seguinte:

```
printf("%d Aluno%c \n",n,((n>1) ? 's' : ' '));
```

Onde é desejado escrever "*Aluno*" no plural ou no singular de acordo com o valor de ***n***.

Mais exemplos:

```
menor = (valor1 < valor2) ? valor1 : valor2;
maior = (valor1 > valor2) ? valor1 : valor2;
```

## 4.6 Operador incremento e decremento

O operador incremento e decremento são operadores unários que adicionam ou subtraem 1 ao operando. Os sinais utilizados para o incremento são **++**. Para o decremento é utilizado **--**.

Na figura seguinte, podemos verificar as diferentes utilizações deste operador. De salientar a notação *pós-fixada (postfix)* e *pré-fixada (prefix)*, caso o operador venha depois ou antes do operador, respetivamente.

| Operador | Exemplo | Significado | Expressão equivalente |
|---|---|---|---|
| `++` | `i++;` | postfix | `i = i + 1; i += 1;` |
| `++` | `++i;` | prefix | `i = i + 1; i += 1;` |
| `--` | `i--;` | postfix | `i = i - 1; i -= 1;` |
| `--` | `--i;` | prefix | `i = i - 1; i -= 1;` |

Sempre que for necessário o incremento ou decremento de 1 a uma variável podemos utilizar a notação:

**var++; ou ++var;**

**var--; ou --var;**

No entanto especial atenção deve ser dada quando a notação pós-fixada ou pré-fixada é utilizada numa expressão. Vejamos o exemplo seguinte:

```
x = 10;
y = ++x;
printf(" x = %d y = %d ", x,y);
```

***Output***:

```
x = 11 y = 11
```

```
x = 10;
y = x++;
printf(" x = %d y = %d ", x,y);
```

***Output***:

```
x = 11 y = 10
```

No primeiro exemplo, a instrução **y = ++x**; a expressão **++x** é executada incrementado de 1 o valor de **x**, de seguida atribuído o resultado à variável **y**. No outro exemplo, na instrução **y = x++**; em primeiro lugar é atribuído o valor de **x** à variável **y**, só depois é efetuado o incremento da variável **x**.

Para estas situações são utilizadas as seguintes regras:

- Se numa variável é utilizada o operador ++ ou -- com notação pré-fixada, o incremento ou decremento ocorre **antes** do valor da variável ser usado no resto da expressão.
- Se numa variável é utilizada o operador ++ ou -- com notação pós-fixada, o incremento ou decremento ocorre **depois** do valor da variável ser usado no resto da expressão.

Exemplos:

```
int i = 1;
int j = 2;
int k = 3;
int final;
final = i++*j- --k;
```

**Output**:

```
i = 2
j = 2
k = 2
final = 0
```

```
int i = 1;
int j = 2;
int k = 3;
int final;
final = ++i*j- k--;
```

**Output**:

```
i = 2
j = 2
k = 2
final = 1
```

## 4.7 Instrução switch

A instrução ***switch*** permite a escolha de várias opções. Vejamos a sua sintaxe:

***switch*** (expressão) {

***case*** (expressão1) **:** uma ou mais instruções; ***break***;

***case*** (expressão2) **:** uma ou mais instruções; ***break***;

***case*** (expressão2) **:** uma ou mais instruções; ***break***;

…

***default*** **:** uma ou mais instruções;

}

Todas as expressões da estrutura ***switch*** podem ser inteiras, carateres, constante, ou uma variável. O número de expressões ***case*** é determinado pelos requisitos do programa. A linha de ***default*** é opcional.

Se o resultado da ***expressão*** coincide com o valor da ***expressão1***, as instruções nela incluídas são executadas. Se o resultado da ***expressão*** coincide com o valor da ***expressão2***, as instruções nela incluídas são executadas. E assim sucessivamente. Se não coincidir com nenhuma expressão é efetuado o conjunto de instruções pertencentes ao ***default*** se existir.

Exemplo:

Escrever por extenso um número inteiro introduzido até 10.

```
switch(numero){
        case 1: printf("Um \n");break;
        case 2: printf("Dois \n");break;
        case 3: printf("Três \n");break;
        case 4: printf("Quatro \n"); break;
        case 5: printf("Cinco \n"); break;
        case 6: printf("Seis \n"); break;
        case 7: printf("Sete \n"); break;
        case 8: printf("Oito \n"); break;
        case 9: printf("Nove \n"); break;
        case 10: printf("Dez \n"); break;
        default: printf("Outro valor \n");
}
```

Outro exemplo:

Escreva se um carácter introduzido é ou não uma vogal.

```
switch( ch ) {
            case 'a':
            case 'A':
```

```
            case 'e':
            case 'E':
            case 'i':
            case 'I':
            case 'o':
            case 'O':
            case 'u':
            case 'U':
              printf("%c é uma vogal.\n", ch);
              break;
            default:
              printf("%c não é vogal.\n", ch);
}
```

Exemplo:

Escreva o número de dias de um dado mês [1, 12] de um ano comum.

```
switch( mes ) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12: dias = 31; break;
            case 4:
            case 6:
            case 9:
            case 11: dias = 30; break;
            case 2: dias = 28; break;
}
```

## 4.8 Estruturas iterativas

Estruturas iterativas, estruturas de repetição ou ciclos são nomes usuais para descrever a utilização das estruturas como: ***for***, ***do-while***, e ***while***; que vamos abordar nesta secção.

### 4.8.1 Ciclo for

A estrutura iterativa ou ciclo ***for*** permite a repetição de um conjunto de instruções um número específico de vezes. A estrutura de utilização do ciclo for é a seguinte:

```
for (valor inicial; condição de teste; incremento){
        Instrução ou instruções;
}
```

O ***valor inicial*** é executada antes de o ciclo ter início e só é executada uma vez. É normalmente utilizada para iniciar o valor da ***variável de controlo do ciclo***. Por exemplo:

```
int i;
for (i = 1 ; i<=100 ; i++){
        printf("%d \n", i);
}
```

No exemplo apresentado chamamos à variável ***i*** a ***variável de controlo do ciclo***.

A ***condição de teste*** determina o número de vezes que será executado o ciclo. Ou seja, o ciclo será executado enquanto a condição de teste for verdade (diferente de zero). O ***incremento*** é executado sempre que o ciclo é executado, afetando a variável de controlo do ciclo, normalmente incrementando ou decrementado o valor da variável. No exemplo apresentado incrementa de uma unidade (i++). No exemplo o output será a apresentação no ecrã dos números de 1 até 100, um valor por linha.

Exemplos de utilização do ciclo for:

```
char c;
for (c = 'A' ; c <= 'Z'; c++){
        printf("%c ", c);
}
```

Output:

A B C D E F G H I J L K M N O P Q R S T U V W X Y Z

```
int i;
for (i=5;i<=100;i+=5){
        printf("%d  ",i);
}
```

Output:

5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100

```
int i;
for (i=50;i>1;i-=2){
        printf("%d  ",i);
}
```

Output

50 48 46 44 42 40 38 36 34 32 30 28 26 24 22 20 18 16 14 12 10 8 6 4 2

### 4.8.2 Ciclo do-while

O ciclo ***do-while*** é executado pelo menos uma vez. A estrutura do ciclo *do-while* é a seguinte:

```
do{
        conjunto de instruções;
} while( condição );
```

O conjunto de instruções do ciclo *do-while* será executado enquanto a ***condição*** for verdadeira.

De seguida apresentamos alguns exemplos de utilização do ciclo do-while e o código equivalente com o ciclo for.

```
int i;
i = 1;
do {
        printf("%d \n", i);
        i++;
} while ( i<=100);
```

```
int i;

for (i = 1 ; i<=100 ; i++){
        printf("%d \n", i);
}
```

```
char c;
c = 'A';
do {
        printf("%c ", c);
        c++;
} while(c <= 'Z');
```

```
char c;

for (c = 'A' ; c <= 'Z'; c++){
        printf("%c ", c);
}
```

```
int i;
i = 5;
do {
        printf("%d  ",i);
        i += 5;
} while (i<=100);
```

```
int i;

for (i=5;i<=100;i+=5){
        printf("%d  ",i);
}
```

```
int i;

i = 50;
do {
        printf("%d  ",i);
        i -= 2;
} while (i>1);
```

```
int i;

for (i=50;i>1;i-=2){
        printf("%d  ",i);
}
```

A **validação de dados de entrada** é um fator muito importante na programação. A estrutura iterativa *do-while*, devido às suas caraterísticas, é muito útil para este tipo de utilização. Vejamos o exemplo seguinte:

```
do {
        printf("Introduza um valor no intervalo [0, 20] ? ");
        scanf("%d", &valor);
} while ((valor < 0) || (valor > 20));
```

No exemplo apresentado, pretende-se a leitura de um valor no intervalo [0, 20]. No caso de o valor introduzido seja fora do intervalo definido, o ciclo repete. Ou seja, enquanto o valor não for válido o programa não sai do ciclo. Na Figura 17, podemos ver o exemplo em execução.

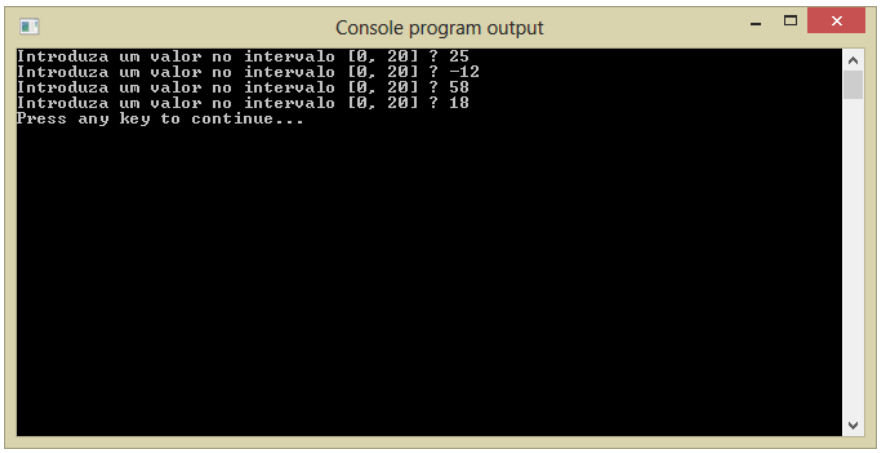



*Figura 17 - Exemplo do-while. Validação.*

Para completar o exemplo anterior é utilizado, frequentemente, o envio de uma mensagem para o ecrã alertando para a introdução de um valor incorreto, ou fora do intervalo. Assim, devemos completar o exemplo anterior com as instruções assinaladas.

```
do {
        printf("Introduza um valor no intervalo [0, 20] ? ");
        scanf("%d", &valor);
        if ((valor < 0) || (valor > 20)) {
                printf(" Valor incorreto! \n ");
        }
} while ((valor < 0) || (valor > 20));
```

Nos exemplos anteriores controlamos o número de vezes que é efetuado o ciclo utilizando um contador. Ou seja definimos o número máximo que desejamos que o ciclo seja executado. Outra forma de controlar o número de vezes que é realizado o ciclo é a utilização de um valor de controlo. Vejamos um exemplo de utilização deste valor de controlo:

```
do {
        printf("Valor ? (Para finalizar -1) ");
        scanf("%d", &valor);
        if (valor != -1){
                soma += valor;
        }
} while( valor != -1);
```

O valor -1 é utilizado para terminar a execução do ciclo. O valor selecionado **não pode fazer parte** do intervalo de valores de leitura.

Uma outra forma de repetir o ciclo é perguntar ao utilizador se deseja continuar a introdução de valores. No exemplo seguinte mostramos como implementar este método:

```
int valor;
int soma = 0;
int continuar;
do {
        printf("Valor ? ");
        scanf("%d", &valor);
        soma += valor;
        printf("Deseja continuar ? (1-Sim/0-Não) ");
        scanf("%d",&continuar);
} while( continuar != 0);
printf("Fim \n");
```

Neste caso foi utilizada uma variável do tipo inteiro, **continuar**, para verificar a condição de fim de ciclo (valor da variável *continuar* igual a zero). Podemos reescrever o código utilizando uma variável do tipo ***char***, sugerindo ao utilizador para a introdução do caractere ***s*** para sim e ***n*** para não.

```
int valor;
int soma = 0;
char continuar;
do {
        printf("\nValor ? ");
        scanf("%d", &valor);
        soma += valor;
        printf("Deseja continuar ? (s/n) ");
        scanf(" %c", &continuar);
} while( continuar != 'n');
printf("Fim\n");
```

Não esquecer a particularidade de leitura de carateres já referida (ver *scanf*: leitura de carateres pag. 30). Incluir a instrução ***fflush(stdin)*** ou espaço no ***scanf*** antes de %c.

### 4.8.3 Ciclo *while*

O ciclo *while* é mais uma das estruturas iterativas utilizadas na linguagem de C. Muito semelhante ao ciclo *do-while*, o ciclo *while* faz o teste da condição antes de iniciar o conjunto de instruções. Ou seja, o ciclo *while* pode nunca ser executado e o ciclo *do-while* é executado pelo menos uma vez.

Estrutura do ciclo *while*:

```
while ( condição ) {
        conjunto de instruções;
}
```

O conjunto de instruções é repetido enquanto a ***condição*** é verdadeira. De seguida apresentamos alguns exemplos:

```
int i;
i = 1;
while ( i<=100) {
        printf("%d \n", i);
        i++;
}
```

```
int i;
i = 1;
do {
        printf("%d \n", i);
        i++;
} while ( i<=100);
```

```
int i;

for (i = 1 ; i<=100 ; i++){
        printf("%d \n", i);
}
```

```
char c;
c = 'A';
while(c <= 'Z'){
        printf("%c ", c);
        c++;
}
```

```
char c;
c = 'A';
do {
        printf("%c ", c);
        c++;
} while(c <= 'Z');
```

```
char c;

for (c = 'A' ; c <= 'Z'; c++){
        printf("%c ", c);
}
```

```
int i;
i = 5;
while (i<=100) {
        printf("%d  ",i);
        i += 5;
}
```

```
int i;
i = 5;
do {
        printf("%d  ",i);
        i += 5;
} while (i<=100);
```

```
int i;

for (i=5;i<=100;i+=5){
        printf("%d  ",i);
}
```

```
int i;
```

```
int i;
```

```
int i;
```

```
i = 50;
while (i>1) {
        printf("%d  ",i);
        i -= 2;
}
```

```
i = 50;
do {
        printf("%d  ",i);
        i -= 2;
} while (i>1);
```

```
for (i=50;i>1;i-=2){
        printf("%d  ",i);
}
```

Especial atenção deve ser dada à condição do ciclo, uma vez que é ela que controla o fim do ciclo. Se a condição for verdadeira e no seu interior não existir uma variável que faça parte da condição o seu valor nunca é alterado e o ciclo nunca termina, diz-se que entramos num ciclo infinito.

### 4.8.4 Instrução break

Por vezes existe a necessidade de terminar um ciclo sem testar a sua condição. A instrução ***break***, já utilizada na instrução *switch*(), pode ser utilizada para este fim. Desta forma, a instrução break pode ser usada para terminar um ciclo independentemente da condição de fim do ciclo for, do-while ou while. Vejamos o exemplo seguinte:

```
for (i = 1 ; i<=100 ; i++){
        printf("%d \n", i);
        if (i==25) break;
}
```

No exemplo apresentado, o ciclo for inicia em 1 e termina em 100. No entanto, a instrução ***if (i==25) break;*** fará terminar o ciclo quando o valor de ***i*** for igual a 25.

Especial atenção deve ser considerada à utilização da instrução break, pois apenas termina o ciclo mais interior. Vejamos o próximo exemplo:

```
for (k = 1 ; k<=5 ; k++){
        for (i = 1 ; i<=100 ; i++){
                printf("%d \n", i);
                if (i==25) break;
        }
}
```

Na situação apresentada o programa irá escrever os números de 1 a 25, 5 vezes. O ciclo for interior é interrompido sempre que o valor i == 25, continuando a execução do ciclo for mais exterior de k=1 até k=5.

Nos exemplos apresentados foi utilizado o ciclo for, mas com os ciclos *do-while* e *while* a situação é idêntica.

```
for (k = 1 ; k<=5 ; k++){
        i = 1;
        do {
                printf("%d \n", i);
                if (i==25) break;
                i++;
        } while (i<=100);
}
```

### 4.8.5 Exercícios propostos

***Exercicio26*** - Escrever a tabuada à escolha do utilizador.

***Exercicio27*** - Calcular o ***factorial*** de um valor inteiro introduzido.

***Exercicio28*** - Determinar os ***n*** números ***fibonacci***.

***Exercicio29*** -Determinar o número de vezes que aparece uma letra à escolha do utilizador num conjunto de carateres introduzidos pelo utilizador.

***Exercicio30*** - *Calcular* se um número inteiro introduzido é um número perfeito. A soma dos seus divisores é o próprio número.

***Exercicio31*** - Calcular os números primos entre 3 e 10000.

**Exercicio32** - Faça um programa que permita calcular o número de dias decorridos da data: dia 1, do mês 1, do ano 1, até uma data introduzida pelo utilizador (dia, mês, ano). Utilize a seguinte fórmula para o cálculo dos anos bissextos, é ano bissexto se a expressão for verdadeira:

(ano % 4 == 0) && (ano % 100 != 0) || (ano % 400 == 0)

# 5 Array's

Um *array* é um conjunto de elementos do mesmo tipo referenciado por um mesmo nome. Um *array* pode ter uma dimensão ou mais. Geralmente, um *array* unidimensional é designado por **vetor**, e ao *array* bidimensional é denominado de **tabela** ou **matriz**. É habitual representar visualmente o vetor e matriz com o aspeto da Figura 18:

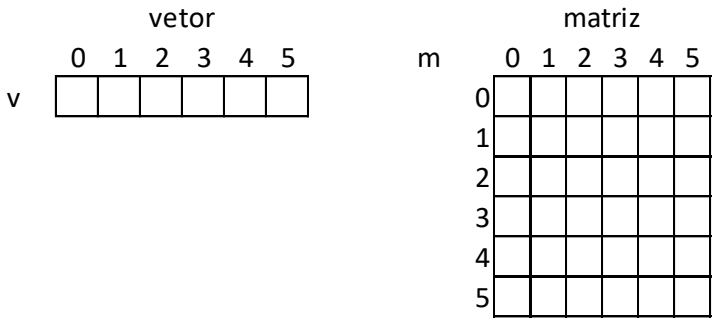
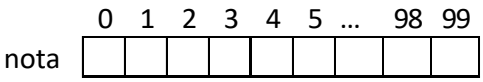



*Figura 18 - Array unidimensional e bidimensional.*

Em programação é normal existir a necessidade de guardar vários dados do mesmo tipo. Por exemplo, se for necessário guardar a nota de 100 alunos, até aqui, a única forma de registar estes dados seria declarar 100 variáveis (int nota1, nota2, nota3, …, nota100). Este método é impraticável. Para estas situações as linguagens de programação incluíram os ***arrays***. Desta forma, para o exemplo anterior, utilizamos a definição:

```
int nota[100];
```



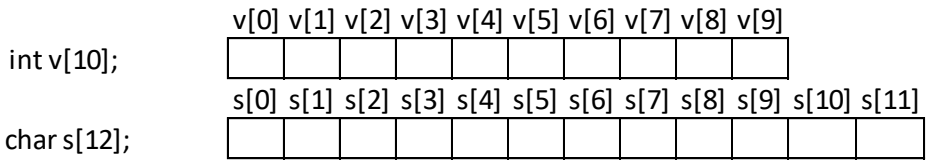

Os *arrays* são declarados da mesma forma que as outras variáveis acrescentando o seu tamanho entre parêntesis retos.

```
Tipo_de_dados nome_do_array [ tamanho];
```

Exemplo:

| | |
|---|---|
| int v[10]; | Array de inteiros de tamanho 10. |
| char s[12]; | Array de carateres de tamanho 12. |

A variável ***v*** pode guardar 10 valores do tipo inteiro, v[0] a v[9]. A variável ***s*** pode guardar 12 carateres. Nas variáveis do tipo *array*, podemos manipular individualmente cada uma das posições através da referência à sua posição, como mostra a figura:

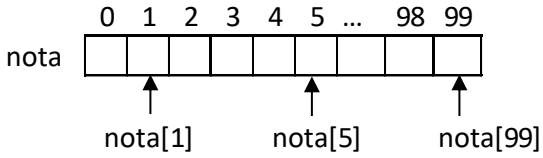


No momento da sua declaração podemos iniciar os *arrays* da seguinte forma:

int idade[5] = {18,25,20,21,19}; ou int idade[] = {18,25,20,21,19};

É de salientar que não é necessário indicar o tamanho do *array* quando efetuamos a iniciação. O tamanho é automático atribuído de acordo com o número de elementos atribuídos.

Associado a cada array existe um índice para cada posição. O índice começa sempre em zero. Como utilizamos na definição o tamanho, ou número de elementos do array, o último elemento terá o índice **tamanho -1**. Deste modo, para manipular cada um dos elementos do array, associamos o nome da variável com o respetivo índice, como podemos pela figura.




No exemplo seguinte mostramos como ler as notas dos 100 alunos.

```
int i = 0;
int nota[100];
for (i=0; i<100;i++){
        printf("Aluno[%d] – Nota ? ",i);
        scanf("%d",&nota[i]);
}
```

Para escrever as notas dos 100 alunos será:

```
for (i=0; i<100;i++){
        printf("Aluno[%d] – Nota = %3d \n",i,nota[i]);
}
```

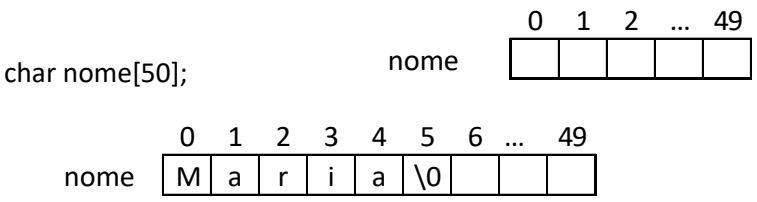
Uma chamada de atenção para a utilização de um ciclo para a manipulação dos dados do *array*, uma vez que será necessário percorre todas as posições do *array*. Ou seja, é necessário fazer variar o índice do *array*.

## 5.1 Strings

Uma *string* é um tipo especial de array. É um conjunto de carateres.

### 5.1.1 Declaração e valor inicial

A string é tratada como um vetor de carateres.



O último elemento é **'\0'**, introduzido automaticamente e representa o fim de *string*. Ver subcapítulo 3.6.4.

No momento de declaração da *string* podemos iniciar utilizando as seguintes instruções:

```
char s1[]="programacao";
char s2[12]="programacao";
char s3[]={'p','r','o','g','r','a','m','a','c','a','o','\0'};
char s4[12]={'p','r','o','g','r','a','m','a','c','a','o','\0'};
```

As instruções anteriores são todas equivalentes.

### 5.1.2 Leitura de dados

A função ***scanf*** também permite a leitura de uma *string* utilizando o formato %s. No entanto, a variável que recebe a string **não é precedida de &**. Exemplo:

```
scanf("%s",string);
```

A função ***scanf*** apenas faz a leitura de uma palavra. Ou seja, se forem introduzida duas ou mais palavras a função *scanf* apenas lê a primeira palavra. A leitura é efetuada até ao primeiro espaço. Devido a este tipo de comportamento da função *scanf*, devemos usar a função ***gets().***

A função **gets(nome_var)** faz a leitura do teclado de uma *string*. Para escrever podemos utilizar como habitualmente a função **printf("%s", string)** ou utilizar a função **puts(string).**

Exemplo de leitura da string nome: `gets(nome);`

### 5.1.3 Manipulação de strings

Atenção especial deve ser dada à manipulação de strings. Vejamos os exemplos seguintes:

- Não devemos atribuir o valor de uma string a outra string:

```
string2 = string1;     /* ERRADO */
```

Para efetuar esta operação de atribuição devemos utilizar a função **strcpy**(string2,string1);

Esta e outras funções de manipulação de *strings* pertencem à biblioteca de funções **string.h**. Apresentamos algumas da funções mais utilizadas:

- **strcpy**(string2,string1); copia a ***string1*** para a ***string2***.
- **strcat**(str_destino,str_origem); é acrescentada a ***str_origem*** no final da ***str_destino***.
- **strncat**(str_destino,str_origem,n); são acrescentados os ***n*** carateres de ***str_origem*** no final da ***str_destino.***
- **strlen**(string); comprimento da string (o caracter ´\0´ não é contado).
- **strcmp**(string1,string2); 0 (zero) se iguais e diferente de zero se diferentes.
- **strncmp**(string1,string2,n); Semelhante à anterior, compara os ***n*** carateres das duas strings.

De seguida apresentamos algumas funções para a manipulação de carateres, pertencentes à biblioteca ***ctype.h***.

- **isalpha**(c): Devolve true se c é uma letra.
- **islower**(c): Devolve true se c é uma letra minúscula.
- **isupper**(c): Devolve true se c é uma letra maiúscula.
- **isdigit**(c): Devolve true se c é um dígito (0 a 9).
- **isxdigit**(c): Devolve true se c é um dígito hexadecimal (0 a 9 ou A, B, C, D, E, F, a, b, c, d, e, ou f).
- **isalnum**(c): Devolve true se c é um carater alfanumérico (0 a 9 ou letra).
- **int tolower**(int c): converte uma letra maiúscula, c, em minúscula.
- **int toupper**(int c): converte uma letra minúscula, c, em maiúscula.

### 5.1.4 Exercícios propostos

***Exercicio33*** - Determinar o número de vezes que aparece uma letra à escolha do utilizador numa frase.

***Exercicio34*** - Verificar se uma palavra introduzida é políndromo. Ou seja, é a mesma palavra escrita da esquerda para a direita ou da direita para a esquerda.

***Exercicio35*** - Faça um programa que permita ler uma frase e escrever por ordem inversa.

Frase? Programar e espectacular
Output: ralucatcepse e ramargorP

***Exercicio36*** - Ler uma frase e um valor n, inteiro no intervalo [1,5]. Escrever os valores de n em n vezes da frase.

Frase? Programar e muito espectacular
Valor? 3
Output: oar i ptur

***Exercicio37*** - Ler uma frase e escrever os caracteres entre um valor inicial e final.

Frase? Programar e muito espectacular
Inicio? 5
Fim? 15
Output: ramar e muito e

***Exercicio38*** - Ler uma frase e codificar a frase com o código de char + n, sendo n um valor dado pelo utilizador que representa o número de carateres a somar ao original.

Frase? Programar e muito espectacular
Codigo? 5
Output: Uwtlwfrfw j rznyt jxujhyfhzqfw

***Exercicio39*** – Determinar o número de vezes que aparece cada letra (a..z) numa frase.

Frase? Estou contente porque vou para a aula de programacao.

Output:

a = 8

b = 0

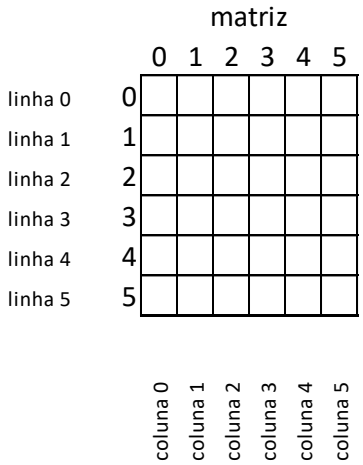
c = 2 … z = 0

## 5.2 Arrays multidimensionais

Em C é possível a utilização de *arrays* com mais de uma dimensão. Vejamos o exemplo seguinte:

```
int matriz[5][5];
```

Que visualmente podemos representar a definição anterior com o seguinte aspeto:




Como já vimos, os *arrays* com duas dimensões, linhas por colunas, são frequentemente denominados por matrizes ou tabelas.

Como nos exemplos anteriores, a inicialização de dados é feita da seguinte forma:

```
int m[3][3] = {{1,2,3}, {-1,-3,-5}, {4,-2,6}}; ou
int m[][3] = {{1,2,3}, {-1,-3,-5}, {4,-2,6}}; ou
int m[][] = {1,2,3,-1,-3,-5,4,-2,6};
```

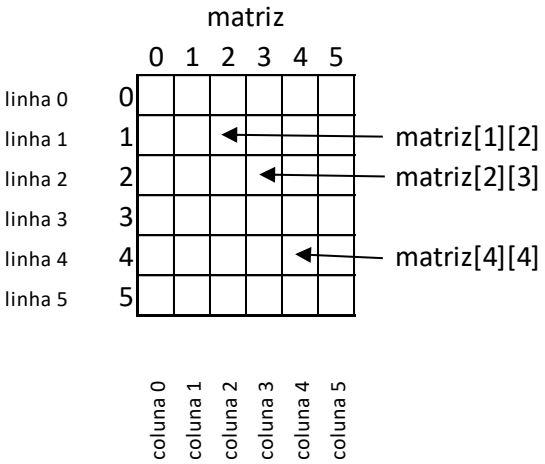
Outras dimensões são possíveis para a declaração de *arrays*, por exemplo:

```
float md3[2][3][3];      Dimensão 3
int md4[3][3][2][5];     Dimensão 4
```

Outras dimensões são possíveis declarar.

### 5.2.1 Manipulação de arrays multidimensionais

Para a manipulação de *arrays* multidimensionais vamos necessitar da referência da linha e da coluna. A conjugação dos dois elementos, linha e coluna, permitirá aceder à posição especificada. Por exemplo ***matriz[1][2]***: permite aceder à posição identificada por linha 1 e coluna 2; **matriz[2][3]:** permite aceder à posição identificada por linha 2 e coluna 3. Para percorrer toda a matriz será necessário um ciclo para cada uma das dimensões definidas. Para o exemplo seguinte, é necessário a utilização de dois ciclos, um que permitirá percorrer as linhas e outro que permitirá percorrer as colunas.




Apresentamos de seguida um conjunto de instruções de manipulação de matrizes.

#### *5.2.1.1 Leitura de dados para uma matriz*

Leitura de valores inteiros para uma matriz de dimensão ***n***, no intervalo [0, 20].

```
int m[10][10];
int n, i, j;

do{
        printf("Dimensão da matriz? ");
        scanf("%d",&n);
        if ((n<0)||(n>10)){
                printf("Valor incorreto!\n");
        }
} while ((n<0)||(n>10));

for (i=0;i<n;i++){
        for (j=0;j<n;j++){
                do{
                        printf(“m[%d][%d]? “,i,j);
                        scanf(“%d”,&m[i][j]);
                        if ((m[i][j]<0)||[m[i][j]>20)){
                                printf(“Valor incorreto!\n”);
                        }
                } while ((m[i][j]<0)||[m[i][j]>20));
```

```
        }
}
```

#### 5.2.1.2 Escrever dados de uma matriz

Escrever dados de uma matriz com o aspeto usual de matriz, linhas por colunas.

```
for (i=0;i<n;i++){
        for (j=0;j<n;j++){
                printf("%d \t", m[i][j]);
        }
        printf("\n");
}
```

#### 5.2.1.3 Escrever uma linha

Escrever uma linha à escolha do utilizador

```
int linha;

do{
        printf("Linha? ");
        scanf("%d",&linha);
        if ((linha<0)||(linha>=10)){
                printf("Valor incorreto!\n");
        }
} while ((linha<0)||(linha>=10));

for (i=0;i<n;i++){
        printf("%d \t", m[linha][i]);
}
```

#### 5.2.1.4 Escrever uma coluna

Escrever uma coluna à escolha do utilizador

```
int coluna;

do{
        printf("Coluna? ");
        scanf("%d",&coluna);
        if ((coluna<0)||(coluna>=10)){
                printf("Valor incorreto!\n");
        }
} while ((coluna<0)||(coluna>=10));

for (i=0;i<n;i++){
        printf("%d \t", m[i][coluna]);
}
```

#### 5.2.1.5 Diagonal principal

Escreva a soma da diagonal principal da matriz.

```
int soma = 0;
for (i=0;i<n;i++){
        soma = soma + m[i][i];
}
printf("Soma da diagonal principal: %d \n", soma);
```

#### 5.2.1.6 Diagonal secundária

Calcule e escreva a soma da diagonal secundária.

```
soma = 0;
for (i=0;i<n;i++){
        soma = soma + m[i][n-i-1];
}
printf("Soma da diagonal secundaria: %d \n", soma);
```

#### 5.2.1.7 Transposta da matriz

Escreva a transposta da matriz.

```
for (i=0;i<n;i++){
        for (j=0;j<n;j++){
                printf("%d \t", m[j][i]);
        }
        printf("\n");
}
```

### 5.2.2 Exercícios propostos

***Exercicio40*** – Faça um programa que permita a leitura de notas [0,20] de vários alunos.

***Exercicio40.1*** – Escreva os valores anteriores com o aspeto seguinte:

Aluno 1 – Nota: 15
Aluno 2 – Nota: 12
Aluno 3 – Nota: 10
…

***Exercicio40.2*** – Calcular e escrever a média das notas.

***Exercicio40.3*** – Determinar a maior nota.

***Exercicio40.4*** – Determinar o aluno com a maior nota.

***Exercicio40.5*** – Determinar a menor nota.

***Exercicio40.6*** – Determinar o aluno com a menor nota.

***Exercicio40.7*** – Escrever todos os alunos com nota inferior à média.

***Exercicio40.8*** – Escrever todos os alunos com nota superior à média.

***Exercicio40.9*** – Ordenar e escrever as notas por ordem crescente.

***Exercicio40.10*** - Escrever a respetiva nota qualitativa de acordo com a seguinte tabela:

[0,6] - Muito Fraco
]6,10[ - Fraco
[10,14[ - Razoável
[14,17[ - Bom
[17,20] - Muito Bom

Exemplo: Aluno 1 (15): Bom
Aluno 2 (12): Razoável
Aluno 3 (10): Razoável

***Exercicio40.11*** – Escreva um gráfico, com o sinal +, para as notas.

Exemplo: Aluno 1 (15): +++++++++++++++
Aluno 2 (12): ++++++++++++
Aluno 3 (10): ++++++++++

***Exercicio40.12*** – Criar um novo vetor com **1,** se nota é maior ou igual a 10, e **0** se a nota é inferior a 10.

***Exercicio40.13*** – Faça uma ***pesquisa linear*** de um uma dada nota.

***Exercicio40.14*** – Faça uma ***pesquisa binária*** de um uma dada nota.

***Exercicio55*** – Leitura de dados para uma matriz de n x m, de valores reais no intervalo [-50, 50].

***Exercicio55.1*** – Escrever os dados da matriz anterior, no formato usual de matriz.

***Exercicio55.2*** – Dada a matriz m, do *exercício55*, determinar e escrever o menor valor da matriz e respetiva posição.

***Exercicio55.3*** – Dada a matriz ***m***, do *exercício55*, determinar e escrever o maior valor da matriz e respetiva posição.

***Exercicio59*** – Introduzir dados numa matriz de dimensão **n**, de valores inteiros no intervalo [-100, 100].

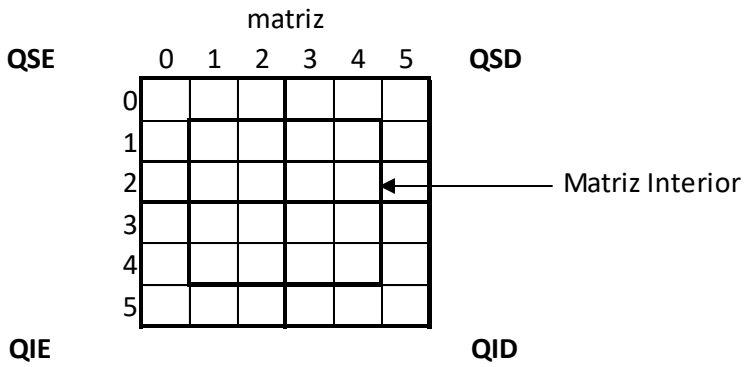
***Exercicio59.1*** – Visualizar os dados em formato usual de matriz.

***Exercicio59.2*** – Visualizar a diagonal principal.

***Exercicio59.3*** – Visualizar a diagonal secundária.

***Exercicio59.4*** – Maior elemento e respetiva posição.

*Exercicio59.5* – Menor elemento e respetiva posição.
*Exercicio59.6* – Trocar duas linhas à escolha do utilizador.
*Exercicio59.7* – Trocar duas colunas à escolha do utilizador
*Exercicio59.8* - Calcular a soma de uma linha à escolha do utilizador.
*Exercicio59.9* - Calcular a soma de uma coluna à escolha do utilizador.
*Exercicio59.10* - Calcular a soma da diagonal principal.
*Exercicio59.11* - Calcular a soma da diagonal secundária.
*Exercicio59.12*- Escreva o quadrante superior esquerdo, **QSE**, da matriz.
*Exercicio59.13*- Escreva o quadrante superior direito, **QSD**, da matriz.
*Exercicio59.14*- Escreva o quadrante inferior esquerdo, **QIE**, da matriz.
*Exercicio59.15*- Escreva o quadrante inferior direito, **QID**, da matriz.
*Exercicio59.16*- Escreva a matriz interior.




*Exercicio59.17*- Criar uma nova matriz, com o valor 0 se os valores da matriz menores que zero, e 1 se os valores da matriz maiores ou iguais a zero.
*Exercicio59.18*- Colocar os valores da matriz num vetor.
*Exercicio59.19*- Se a matriz é uma matriz simétrica, matriz igual a matriz transposta, então é possível guardar os seus valores num vetor como mostra a figura. Faça o código necessário para guardar os dados num vetor se possível.

| | | | |
|---|---|---|---|
| **1** | 5 | 6 | 7 |
| **5** | **2** | 8 | 9 |
| **6** | **8** | **3** | 5 |
| **7** | **9** | **5** | **4** |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| **1** | **5** | **2** | **6** | **8** | **3** | **7** | **9** | **5** | **4** |

*Exercicio79* - Faça um programa que permita o registo de temperaturas médias diárias ao longo de um mês, de acordo com a seguinte aspeto de *input*.

Dia 1 - Temperatura? 16.0
Dia 2 - Temperatura? 12.2
…
Dia 30 - Temperatura? 12.1

Com os dados obtidos, resolva as operações seguintes:

***Exercicio79.1*** - Apresente as temperaturas no ecrã, sete valores por linha, como na figura seguinte:

16,0 12,2 13,9 12,6 13,7 14,8 13,0
15,6 16,4 12,3 12,8 15,5 13,6 15,4
13,4 14,9 13,8 12,4 14,1 15,5 13,2
13,0 15,4 14,6 15,1 15,7 12,3 12,0
15,5 12,1

***Exercicio79.2***.- Determine a temperatura máxima e respetivo dia.

***Exercicio79.3*** - Determine a amplitude térmica desse mês. ***Amplitude térmica = temperatura máxima – temperatura mínima***.

***Exercicio79.4*** - Sabendo que o dia 1 foi domingo e que uma semana vai de domingo a sábado. Determine a média das temperaturas das quartas-feiras.

***Exercicio79.5*** - Escreva uma semana à escolha do utilizador, com validação [1, 4].

# 6 Introdução às funções em C

Dos exercícios e exemplos já apresentados ao longo deste manual, com certeza já verificou que algum do código se repete em várias situações. Podemos tirar vantagens da natureza repetitiva desse código e escrevê-lo com outra organização. Escrever esse código sobre a forma de funções.

Uma das formas possíveis, e já utilizada, é recorrer a funções já definidas pela linguagem C, como por exemplo: ***toupper()***, da biblioteca *ctype.h*, que permite converter um caracter minúsculo em maiúsculo; ou ainda, ***sin()***, da biblioteca *math.h*, que permite calcular o seno de um dado valor. No entanto, é também possível a definição das nossas próprias funções. Desta forma sempre que um conjunto de instruções se possa repetir ao longo do um programa, ou em outros programas, é uma boa solução tentar adaptar esse mesmo código a uma função.

Até aqui, construímos os nossos programas baseados apenas na função principal ***main()***, e isto, apenas com o propósito de dar a conhecer os conceitos fundamentais da introdução à programação. Entretanto, à medida que a complexidade aumenta existe a necessidade de recorrer a algumas estratégias que permitam resolver os problemas. Uma dessas técnicas de resolução de problemas é a simplificação, ou a divisão, de um problema complexo em pequenos problemas mais simples, que no seu conjunto resolvem o problema maior e mais complexo. Esta técnica é denominada por **programação modular**. Podemos então ver a função principal *main()* como a função que reúne todos as rotinas, ou subprogramas, ou ainda todas funções que resolvem o problema.

A estrutura base de definição de uma função é a seguinte:

```
tipo de dados nome([parâmetros]) { /* cabeçalho da função*/
        /* bloco de instruções da função */
        instruções;
        [return expressão;]
}
```

Como podemos verificar a definição anterior é muito semelhante ao já utilizado na definição da função principal ***main***. No entanto, o ***nome*** é a designação da função que será utilizado para a chamada, ou evocação, desta função. ***Tipo de dados*** é o valor devolvido pela função. Na função pode existir uma lista de ***parâmetros***, conjunto de valores a serem passados para a função, ou conjunto de valores necessários à manipulação das operações a realizar na função. O bloco de instruções da função são as instruções necessárias à sua implementação. A palavra reservada ***return*** é a instrução necessária à devolução do valor calculado na função, caso exista. Neste caso, se existir valor a devolver na função este tem de ser igual ao definido no tipo de dados da função.

A partir de agora, podemos resolver os nossos problemas segundo a técnica da programação modular. Desta forma a resolução do exercicio59 pode ser divido da forma seguinte:

```
void main(){
        lerMatriz();
        verMatriz();
        verDiagonalPrincipal();
        verDiagonalSecundaria();
        maiorValor();
```

```
            menorValor();
            …
        }

    void lerMatriz(){
        conjunto de instruções para leitura e validação dos dados da matriz
    }
    void verMatriz(){
        conjunto de instruções para visualização dos dados da matriz
    }

    void verDiagonalPrincipal(){
        conjunto de instruções para visualização da diagonal principal
    }

    void verDiagonalSecundaria()(){
        conjunto de instruções para visualização da diagonal secundária
    }

    int maiorValor(){
        conjunto de instruções para calcular o maior valor da matriz
        return maiorvalor
    }

    int menorValor(){
        conjunto de instruções para calcular o menor valor da matriz
        return menorvalor
    }
```

No exemplo seguinte construímos uma função para cálculo do fatorial. Esta função pode ser utilizada em outros programas, em que seja necessário o cálculo do fatorial.

```
        long fatorial(int n){
            long f = 1;
            int i;
            for (i=2;i<=n;i++){
                f = f * i;
            }
            return f;
        }
        int main(int argc, char *argv[])
        {
            int n;
            printf("Valor ? ");
            scanf("%d",&n);
            printf("%d!= %d\n",n,fatorial(n));
            return 0;
```

```
}
```

No exemplo seguinte, transformamos a leitura e escrita de valores de um vetor de valores de inteiros numa função.

```
void lerVetor(int v[],int tamanho){
        int i;
        for (i=0;i<tamanho;i++){
                printf("V[%d]? ",i);
                scanf("%d",&v[i]);
        }
}

void escreveVetor(int v[],int tamanho){
        int i;
        for (i=0;i<tamanho;i++){
                printf("%d \t ",v[i]);
        }
        printf("\n");
}

int main(int argc, char *argv[])
{
        int vetor[10];

        lerVetor(vetor,5);
        escreveVetor(vetor,5);

        return 0;
}
```

Outras hipóteses são possíveis na construção das funções anteriores.
Vamos agora construir uma função para a leitura de dados inteiros para uma matriz, no intervalo [*valorMin*, *valorMax*], de *n* linhas por *m* colunas.

```
void lerMatriz(int matriz[10][10],int n,int m, int valorMin, int valorMax){
        int i, j;
        for (i=0;i<n;i++){
                for (j=0;j<m;j++){
                        do{
                                printf("M[%d][%d] ? ",i,j);
                                scanf("%d",&matriz[i][j]);
                                if ((matriz[i][j]<valorMin)||(matriz[i][j]>valorMax)){
                                        printf("Valor incorreto!!!\n");
                                }
                        } while ((matriz[i][j]<valorMin)||(matriz[i][j]>valorMax));
                }
        }
}
```

No exemplo seguinte apresentamos uma função para a escrita de uma matriz.

```
void escreveMatriz(int matriz[10][10], int n, int m){
        int i, j;
        for (i=0;i<n;i++){
                for (j=0;j<m;j++){
                        printf("%d\t",matriz[i][j]);
                }
        }
}
```

Quando estamos perante um problema a resolver não devemos começar logo a escrever código. Devemos pensar um pouco e delinear uma estratégia, um plano, para a sua resolução. Uma das estratégias, como já vimos, é a divisão, ou decomposição do problema em problemas mais pequenos e de fácil resolução. Nunca esquecendo o objetivo final a que nos propomos.

A resolução do nosso problema pode ser a junção de um conjunto de peças, que no seu conjunto resolvem o grande problema. Essas peças são as nossas funções. Podem ser criadas novas funções, como podem ser funções já utilizadas em outros problemas.

## 6.1 Domínio das variáveis

O domínio das variáveis é a área onde a variável pode ser utilizada. O domínio das variáveis tem especial significado quando iniciamos o estudo das funções. Com o domínio das variáveis é possível definir se uma função reconhece, ou não, uma determinada variável. O domínio das variáveis protege as variáveis de uma função em relação a outras funções. Se uma dada função não necessita, ou não utiliza uma variável específica, essa variável não tem de estar visível, ou acessível, para essa mesma função.

Esta noção de domínio das variáveis dá lugar a um novo conceito de variáveis globais e variáveis locais, que vamos abordar de seguida.

### 6.1.1 Variável Global e local

Variável global é uma variável que é reconhecida por qualquer instrução do programa. Algum cuidado especial deve ser dado a utilização de variáveis globais, pois qualquer parte do programa pode inadvertidamente modificar o valor dessa variável.

Uma variável local unicamente pode ser modificada na função em que foi definida. Ou seja, uma variável local é definida dentro de uma função. O que significa que o domínio dessa variável será restrito apenas à área definida pela função. Uma variável local está limitada unicamente à função onde foi definida, estando protegida de ser utilizada, modificada ou apagada por qualquer outra função.

Até aqui, aprendemos a declarar variáveis em dois locais: depois da primeira chaveta, logo no início da função; ou, como é o caso das funções, entre os parênteses na definição da função, como argumentos da função.

Todas as variáveis definidas até agora, neste manual, foram definidas como locais.

No exemplo seguinte exemplificamos a definição de variáveis locais.

```
#include <stdio.h>
main(){
        int i,j;
        …
}
```

Variáveis ***i, j*** definidas como **locais** uma vez que foram definidas dentro da função **main()**.

```
#include <stdio.h>
int g, h;
main(){
        …
}
```

As variáveis ***g, h*** foram definidas fora da função main(), são visíveis para todas as funções que venham depois da sua definição. Como tal, chamamos de variáveis **globais**.

Outro exemplo.

```
#include <stdio.h>
int valor1, valor2, valor3;                    /* variáveis globais */

int soma() {
        int resultado;                         /* variável local */
        resultado = valor1 + valor2 + valor3;
        return resultado;
}

main()
{
        int resultado;                         /* variável local */
        valor1 = 10; valor2 = 20; valor3 = 30;
        resultado = soma();
        printf(´´A soma de %d+%d+%d = d\n´´,valor1,valor2,valor3,resultado);
}
```

Vejamos, ainda, o exemplo seguinte:

```
int n1;
void funcao1()
{
        n1 = 10;
        n2 = 20;
}
int n2, n3;
void funcao2()
{
        n2 = 20;
        n3 = 30;
```

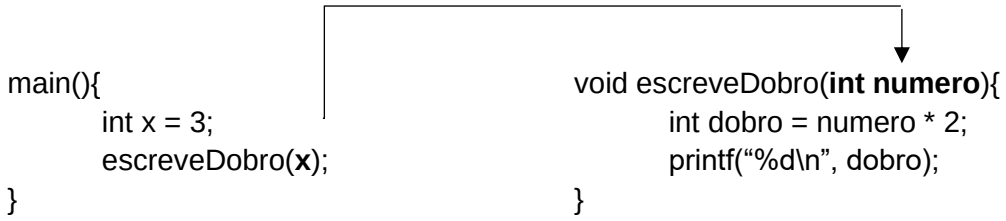
```
}
main(){
        …
}
```

Neste caso, na *funcao1* existe um erro pela não definição da variável ***n2***. As variáveis ***n2*** e ***n3*** apesar de globais foram definidas depois da função **funcao1**, logo não visíveis para esta função.

## 6.2 Passagem por valor

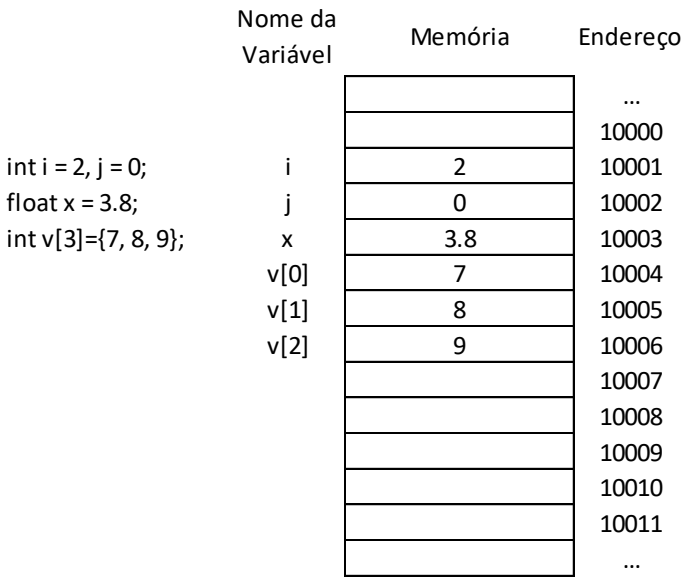
Como já vimos é possível passar valores para as funções. Estes valores são passados para as funções através da utilização de parâmetros declarados nas funções. Quando um argumento é passado por valor, uma cópia do valor da variável é enviada e atribuída ao parâmetro da função.

Na figura seguinte está representado a passagem do valor. O valor da variável **x** é passado para a função através do parâmetro da função **numero**. Ou seja, o valor da variável **x** é atribuído, ou copiado, à variável **numero** da função.




## 6.3 Passagem por referência

Passagem por referência ou endereço. Todas as variáveis estão guardadas em memória num endereço específico. Quando definimos uma variável é reservado um espaço em memória para essa variável, como representado na figura seguinte.




A cada variável é atribuído um endereço específico em memória. Quando utilizamos arrays como parâmetro é sempre utilizado a passagem por endereço. Enquanto no caso anterior quando

passamos uma variável para a função existem sempre duas cópias da variável em memória, uma para cada variável local. No caso dos arrays apenas existe uma cópia em memória. O que é passado por parâmetro da função é sempre o endereço desse array. Ou seja, se a referência a essa variável é sempre o endereço, significa que estamos a manipular o mesmo espaço em memória. Se manipulamos o mesmo espaço em memória, significa que quando alteramos o valor de um array dentro de uma função esse valor é também alterado fora da função. Verifique que é o que acontece quando utilizamos as funções anteriormente apresentadas para leitura de arrays, por exemplo:

```
void lerMatriz(int matriz[10][10],int n,int m, int valorMin, int valorMax){
        int i, j;
        for (i=0;i<n;i++){
                for (j=0;j<m;j++){
                        do{
                                printf("M[%d][%d] ? ",i,j);
                                scanf("%d",&matriz[i][j]);
                                if ((matriz[i][j]<valorMin)||(matriz[i][j]>valorMax)){
                                        printf(“Valor incorreto!!!\n”);
                                }
                        } while ((matriz[i][j]<valorMin)||(matriz[i][j]>valorMax));
                }
        }
}


main(){
        int tabela[10][10];
        int i = 3, j = 3;
        int min = 0, max = 200;
        lerMatriz(tabela,i,j,min,max);
        escreveMatriz(tabela,i,j);
}
```

De salientar, que o valor os dados da **tabela** são inseridos na função **lerMatriz(), utilizando a variável matriz,** e esses valores são os mesmos na função principal.

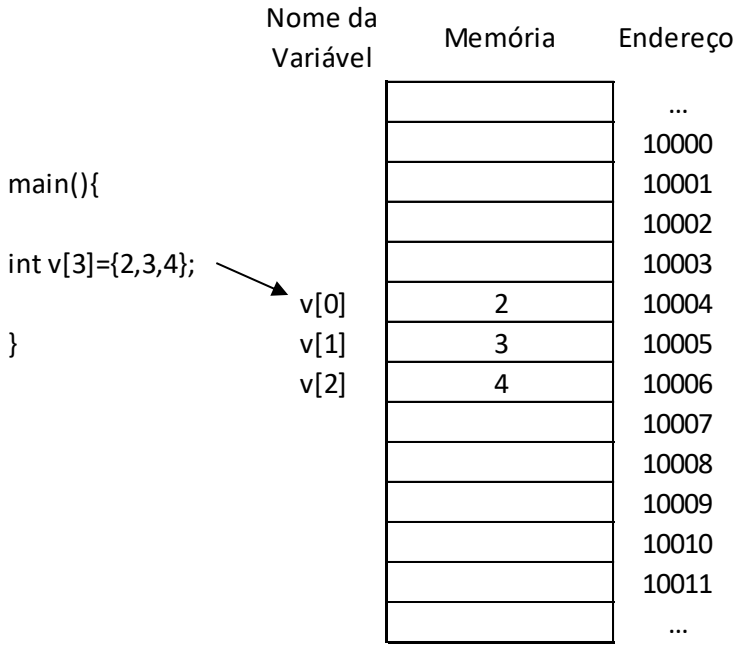
Outro exemplo:

```
void dobro(int vetor[3]){
        int i;
        for (i=0;i<3;i++){
                vetor[i] = vetor[i] * 2;
        }
}
main(){
        int v[3] = {2,3,4};
        dobro(v);
        for (i=0;i<3;i++){
                printf(“%d\t”,v[i]);
        }
```

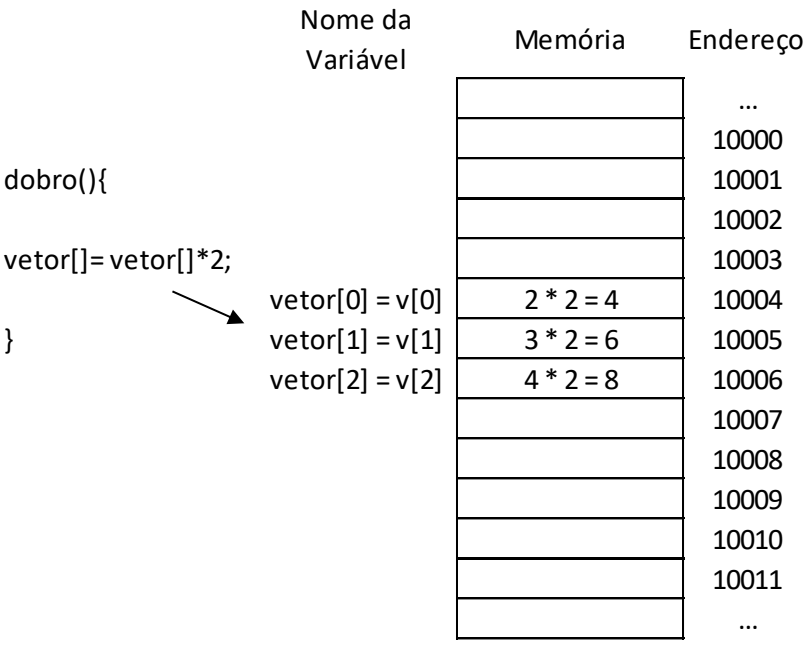
Output:

4 6 8

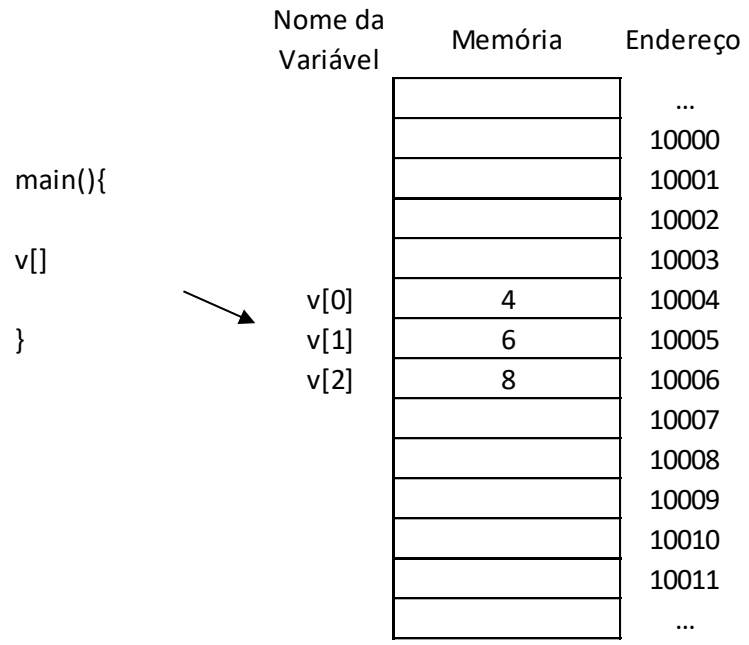
No exemplo apresentado, no programa principal o vetor, v[], foi inicializado com os valores 2, 3, 4.




Na função, **dobro**(), os valores foram alterados para 4, 6, 8.




Na função principal são escritos os valores do vetor v[], que são 4, 6 e 8.



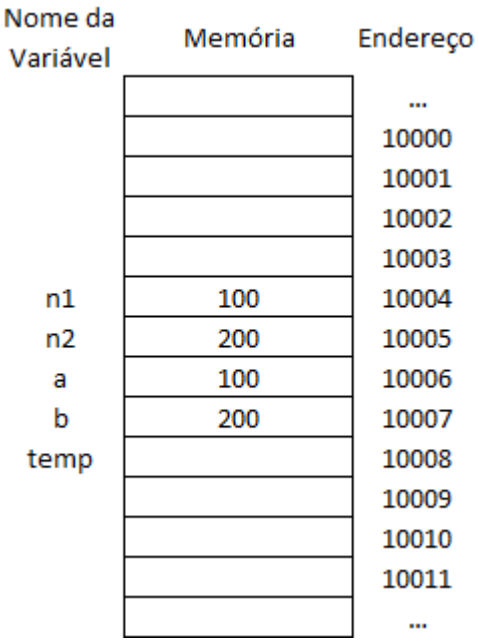

Como acabados de descrever os **arrays são sempre passados por referência**. As variáveis, não arrays, podem ser passadas por valor mas também podem ser passadas por referência.

Vejamos o exemplo seguinte:

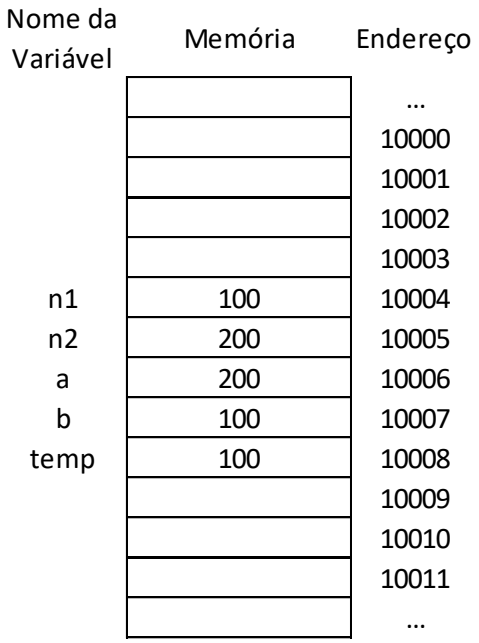
```
#include <stdio.h>

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

```
main()
{
        int n1,n2;
        n1 = 100;
        n2 = 200;
        troca(n1,n2);
        printf(´´%d %d\n´´,n1,n2);
}
```

Na figura seguinte é esquematizada uma simulação de uma possível configuração da memória para o exemplo apresentado. De salientar a existência das variáveis n1, n2, a e b. Deste modo, quando na função principal é evocada a função ***troca***, com os valores ***n1 = 100*** e **n2 = 200**, estes valores são atribuídos respetivamente às variáveis **a** e **b**.




Na função troca, estes valores são trocados, ou seja, o valor de a passa a ser 200, e o valor de b passa a ser 100. Como vemos na figura seguinte. No entanto,




O que resultará como output do programa os valores: 100 e 200. Ou seja, os valores apesar de terem sido trocados na função, no programa principal essa troca não teve qualquer influência.

Vejamos agora o caso em que desejamos que os valores no programa principal sejam passados para a função por referência, com o objetivo de a alteração efetuada dentro da função influencie os valores no local de chamada da função. Para que isto aconteça, temos de substituir os valores das variáveis em vez de os atribuir. Ou seja, os parâmetros da função têm de ser declarados com ***** (asterisco) antes do nome da variável, e na chamada da função os parâmetros correspondentes têm de ser precedidos por **&.** O asterisco significa que a variável guarda um endereço de uma

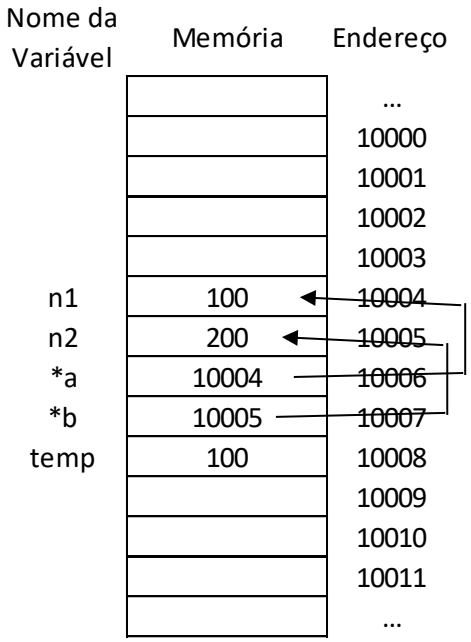
posição de memória. O & já utilizado com a instrução *scanf()*, significa endereço da variável. Ou seja, indicamos que uma variável fica a apontar para o endereço da outra. Resultando isto em que as duas variáveis fazem referência ao mesmo valor.

Vejamos o programa anterior modificado para a passagem de valores por referência.

```
#include <stdio.h>

void troca(int *a,int *b)
{       int temp;
        temp = *a;
        *a = *b;
        *b = temp;
}
```

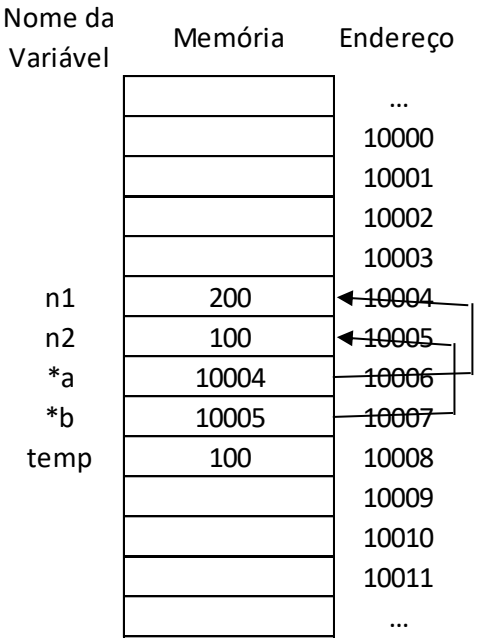
```
main()
{
        int n1,n2;
        n1 = 100;
        n2 = 200;
        troca(&n1,&n2);
        printf(´´%d %d\n´´,n1,n2);
}
```

O respetivo esquema da representação em memória ficará com o aspeto seguinte:




De salientar que agora a posição correspondente à variável **a**, passou a ser ***a**, ou seja, guarda o endereço da variável **n1**, representado por **&n1**. O mesmo se passa para a variável **b**.

No código da função, na instrução **temp = *a;** estamos a atribuir à variável **temp** o valor do endereço da variável **a**, ou seja, o valor no endereço **10004** é o valor **100**. Na instrução ***a = *b;** estamos a colocar o valor do endereço **10005**, no endereço **10004**. No final da execução da função troca temos os seguintes valores em memória:




Os valores das variáveis **n1** e **n2** são 200 e 100 respetivamente, pelo que o output do programa principal será 200 e 100.

## 6.4 Recursividade

Em linguagem C, uma função pode ser chamada de um modo recursivo, isto é, uma função pode chamar-se a si própria. Tipicamente no meio académico o exemplo utilizado para exemplificar a recursividade é o cálculo do fatorial. Para calcular o fatorial sabemos que:

$$n! = n \times (n-1)! \times (n-2)! \times \ldots \times 3 \times 2 \times 1$$

ou seja,

$0! = 1$

$1! = 1 \times 0! = 1$

$2! = 2 \times 1 = 2$

$3! = 3 \times 2 \times 1 \times 0! = 6$

$4! = 4 \times 3 \times 2 \times 1 = 24$

$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Em função do descrito podemos descrever o cálculo do fatorial em função de:

| | |
|---|---|
| Caso base: $0! = 1$ | Valor conhecido |
| Caso geral: $n! = n \times (n-1)!$ | Expressão geral |

De acordo com o acabamos de ver, podemos escrever a função em C para o cálculo do fatorial com o código seguinte:

```
int fatorial (int n){
        if (n == 0) return 1;
        else return n * fatorial(n-1);
}
```

Outro exemplo de recursividade. Pretende-se calcular a soma ***n*** valores inteiros, utilizando recursividade.

Para calcular a soma de n valores inteiros fazemos: 1 + 2 + 3 + 4 + … + n = soma

Ou seja, para o valor de **n = 5**. Seria:

1 + 2 + 3 + 4 + 5 = 15 ou 5 + 4 + 3 + 2 + 1 = 15

Em função do descrito podemos descrever o cálculo do fatorial em função de:

| | |
|---|---|
| Caso base: soma(1) = 1 | Valor conhecido |
| Caso geral: soma(5) = 5 + soma(4) | Expressão geral |

De acordo com o acabamos de descrever, podemos escrever a função em C para o cálculo do fatorial com o código seguinte:

```
int soma(int n){
        if (n == 1) return 1;
        else return n + soma(n-1);
}
```

Para melhor compreensão vejamos o seguinte exemplo:

```
soma(5)
= 5 + soma(4)
= 5 + 4 + soma(3)
= 5 + 4 + 3 + soma(2)
= 5 + 4 + 3 + 2 + soma(1)
= 5 + 4 + 3 + 2 + 1
= 5 + 4 + 3 + 3
= 5 + 4 + 6
= 5 + 10
= 15
```

### 6.4.1 Vantagens e desvantagens da recursividade.

A utilização da recursividade é mais elegante e simples. Ver o exemplo seguinte: calculo da sequência de fibonacci. A recursividade pode ser utilizada para simplificar código complexo. Por outro lado, é bastante mais difícil pensar na lógica de um problema recursivo. O debug do código é igualmente difícil.

Exemplo recursivo para o cálculo da sequência de fibonacci.

A sequência de fibonacci é dada pela seguinte sequência: 1;1;2;3;5;8;13;21;34;55;89;144; …

Ou seja, a sequência de fibonacci é dada pela expressão seguinte:

F(1) = 1
F(2) = 1
F(3) = F(2)+F(1) = 1 + 1 = 2
F(4) = F(3) + F(2) = 2 + 1 = 3
F(5) = F(4) + F(3) = 3 + 2 = 5

O termo geral é o seguinte: F(n) = F(n-1) + F(n-2)

Podemos agora escrever a função em C que permita calcular o F(n).

```
int fibonacci(int n){
        if ((n==1)||(n==2)) return 1;
        else return fibonacci(n-1) + fibonacci(n-2);
}
```

O programa principal para escrever a sequência de fibonacci de n valores, ficaria:

```
int  main() {
   int i;
   for (i = 0; i < 25; i++) {
      printf("%d\t ", fibonacci(i));
   }
   return 0;
}
```

O exemplo utilizando uma solução não recursiva seria:

```
int fibonacciNR(int n){
    int f1=1,f2=0,fn;
    while(n>0){
        fn = f1 + f2;
        f1 = f2;
        f2 = fn;
        n--;
    }
    return fn;
}
```

# 7 Estruturas

Uma estrutura é um conjunto de dados do mesmo tipo ou de tipos diferentes referenciados por um identificador comum. Uma estrutura em C é também designada por registo. Podemos fazer a analogia com os dados de um ficheiro, para guardar dados de vários alunos, o registo será a ficha de cada aluno, os diferentes dados do aluno, que fazem parte da sua ficha, são designados por campos, por exemplo: o nome, o número e o curso. O conjunto de vários registos será um ficheiro.

## 7.1 Definição de estruturas

Suponhamos que queremos guardar o nome, o número e a nota de resultados de uma unidade curricular. Ou seja, vamos ter três variáveis:

| | |
|---|---|
| Nome: | array de carateres, tamanho 60; |
| Numero: | valor inteiro, int; |
| Nota: | valor real, float; |

A definição da estrutura de dados em C tem a seguinte estrutura:

```
struct [nome_estrutura]
{
        tipo1 campo1, campo2;
        tipo2 campo3;
                ...
        tipon campo;
} [uma ou mais variáveis estrutura];
```

Para o exemplo referido ficaria:

```
struct Informacao{
        char nome[60];
        int numero;
        float nota;
} aluno1, aluno2, aluno3;
```

Para utilizar este novo tipo de dados fazemos referência à variável do tipo da estrutura e o campo. O campo é especificado utilizando o ponto, ou seja, **aluno1.numero**, ou **aluno2.nota**,ou ainda, **aluno3.nome**.

Outra forma de possível utilização é definir variáveis do tipo definido como **struct**. As variáveis **aluno1**, **aluno2** e **aluno3** foram definidas logo na declaração do tipo **Informacao**. No entanto podíamos omitir estas variáveis de declarar de forma idêntica às outras variáveis. Vejamos o exemplo seguinte.

```
struct Informacao{
        char nome[60];
        int numero;
        float nota;
};

main(){
    struct Informacao aluno;
    …
    scanf("%d",&aluno.numero);
    …
}
```

## 7.2 Utilização typedef

Com a palavra reservada **typedef** é possível definir um novo nome para um determinado tipo de dados. Por exemplo:

```
typedef int Numero;

main(){

    Numero x;

    printf("Numero? ");

    scanf("%d",&x);

}
```

Com a utilização do typedef as definições da estrutura anterior ficaria:

```
typedef struct Informacao{
        char nome[60];
        int numero;
        float nota;
} INFORMACAO;


main(){

    INFORMACAO aluno;
    …
    scanf("%d",&aluno.numero);
    …
}
```

Na definição typedef struct **Informacao**, este último nome pode ser omisso. Ficando a definição com a definição seguinte:

```
typedef struct {
        char nome[60];
        int numero;
        float nota;
} INFORMACAO;
```

## 7.3 Exercícios propostos

***Exercicio85*** – Pretende-se um programa que permita a gestão dos dados (nome, número, nota) dos alunos de uma unidade curricular. Faça a leitura do nome e número de vários alunos.
***Exercicio85.1*** – Faça a leitura das notas dos diversos alunos da unidade curricular. As notas devem pertencer ao intervalo [0, 20]. A introdução deve ter o aspeto seguinte:

1009876 Bruno Gomes Nota? 15.3

***Exercicio85.2*** – Construa uma função que permita a ordenação por nome.
***Exercicio85.3*** – Produza uma função que permita a visualização dos dados.
***Exercicio85.4*** – Faça uma função que permita a ordenação por número.
***Exercicio85.5*** – Construa uma função que permita o cálculo da média das notas.
***Exercicio85.6*** – Efetue uma função que calcule o número de notas superiores a um valor dado por parâmetro.
***Exercicio85.7*** – Faça uma função que calcule o número de notas num dado intervalo.
***Exercicio85.8*** – Construa uma função que calcule a percentagem dos alunos aprovados.
***Exercicio85.9*** – Produza um menu que permita executar as diferentes operações anteriores.

# 8 Ficheiros

Quando um programa termina toda a informação associada é perdida. A utilização de ficheiros permite preservar a informação mesmo que finalizado o programa. Se existir a necessidade de introdução de uma grande quantidade de dados, pode não ser viável a introdução de dados pelo utilizador. Pelo que, a leitura de dados de um ficheiro pode ser uma das hipóteses de solução.

Através de um programa em C, deve ser possível gravar dados em ficheiros ou enviar dados para dispositivos físicos, como monitores, discos ou impressoras. E, também deve permitir, ler dados de ficheiros ou de dispositivos de entrada de dados, como o teclado.

A comunicação de dados do computador com o exterior é efetuada utilizando sequências de dados designadas por ***ficheiros***. Em C, os ficheiros e dispositivos para input e output são representados logicamente como uma sequência de bytes designadas por ***streams***.

***Data stream***, ou fluxo de dados, é uma sequência de bytes usados para transmitir ou receber informações que estão em processo de transmissão.

A biblioteca padrão do C, **stdio.h**, fornece o conjunto de funções com a finalidade de comunicação de dados do computador com o exterior.

Neste capítulo vamos apresentar alguns exemplos para a manipulação de ficheiros.

## 8.1 Text vs Binary Streams

**Text Streams**. Uma sequência de dados de texto, text stream, transporta dados do tipo carácter de um texto dividido em linhas. Uma linha de texto, consiste numa sequência de caracteres que termina por um carácter especial de fim de linha ou mudança de linha. Estes carateres diferem de acordo com o sistema em que estamos a trabalhar, sistema Windows ou Não-Windows.

**Binary Streams**. Binary stream é uma sequência de bytes que são transmitidos sem modificação. Ou seja, as funções que operam a manipulação deste tipo de streams não fazem nenhum tipo de interpretação dos carateres de controlo. Os fluxos de dados binários são normalmente usados para manipular dados binários, como por exemplo escrever e ler dados de registos de um ficheiro de dados.

## 8.2 Files

Um ficheiro, ***File***, representa uma sequência de bytes. A função ***fopen()***, associa um ficheiro com uma *stream* e inicia um objeto do tipo **FILE**, o qual contém toda a informação necessária ao controlo da *stream*. Estas informações incluem um ponteiro para o buffer utilizado, um indicador de posição no ficheiro e, ainda, indicadores de erro e fim de ficheiro.

### 8.2.1 Tipos de ficheiros

Tal como as *streams*, quando trabalhamos com ficheiros, podemos considerar dois tipos de ficheiros: os ficheiros de texto e ficheiros binários.

Os ficheiros de texto, ***text files***, são normalmente identificados pela extensão **.txt**. Este tipo de ficheiro pode ser facilmente criado utilizando um qualquer editor de texto, como por exemplo o

*Notepad*. Quando abrimos este tipo de ficheiro, visualizamos o seu conteúdo como um texto. O seu conteúdo pode facilmente ser editado ou apagado.

Os ficheiros binários, ***binary files***, normalmente identificados pela extensão **.bin** ou **.dat**, são guardados em formato binário. Estes ficheiros, contrariamente aos ficheiros de texto, não são facilmente legíveis. Este tipo de ficheiros tem de ser manipulado por uma aplicação especifica, que proporcionará a edição, a alteração e a eliminação de conjuntos de bytes do ficheiro.

### 8.2.2 Operações com ficheiros

Em C, destacamos as principais operações sobre ficheiros de texto ou binários, como:

- Criar um novo ficheiro.
- Abrir um ficheiro existente.
- Fechar o ficheiro.
- Ler e um ficheiro e escrever para um ficheiro.

Nas seções seguintes, destacamos as principais funções para operações com ficheiros.

### 8.2.3 FILE e fopen()

Quando trabalhamos com ficheiros é necessário declarar um ponteiro para o tipo de ficheiro. Esta declaração é necessária para a comunicação entre o ficheiro e o programa.

```
FILE *f;
```

Para a abertura do ficheiro é utilizada a função **fopen()**, definida no *header file* ***stdio.h***.

A sintaxe para abrir um ficheiro é:

```
f = fopen("NomeDoFicheiro", "Modo");
```

Exemplos:

```
f = fopen("c:\CPrograms\novoficheiro.txt", "w");
f = fopen("ficheiro.bin", "rb");
```

Se o ficheiro "novoficheiro.txt" não existir na localização c:\CPrograms. A primeira instrução cria um ficheiro de nome "novoficheiro.txt", e abre-o para escrita, como indica o modo "w".

Se o ficheiro binário "*ficheiro.bin*" existe, na localização do programa. A segunda instrução abre o ficheiro para leitura. O modo de leitura "*rb*", apenas permite a leitura do ficheiro, não permite escrever no ficheiro.

Na Figura 19, podemos ver os diferentes modos de abertura de ficheiro para as mais diversas operações.

### 8.2.4 Instrução fclose()

Depois de qualquer manipulação de ficheiro é necessário fechar o ficheiro, seja ficheiro de texto ou binário. Para fechar o ficheiro é utilizada a função ***fclose(f)*** , sendo ***f*** o ponteiro associado ao ficheiro que queremos fechar. Ou seja, a variável declarada na instrução **FILE *f;** do ficheiro que desejamos terminar a sua manipulação.

| Modo | Significado | Ação |
|---|---|---|
| r | Abre o ficheiro para leitura | Se o ficheiro não existir, fopen() devolve NULL |
| rb | Abre o ficheiro para leitura, em modo binário | Se o ficheiro não existir, fopen() devolve NULL |
| w | Abre o ficheiro para escrita | Se o ficheiro existe, o seu conteúdo é apagado |
| | | Se o ficheiro não existir, o ficheiro será criado |
| wb | Abre o ficheiro para escrita, em modo binário | Se o ficheiro existe, o seu conteúdo é apagado |
| | | Se o ficheiro não existir, o ficheiro será criado |
| a | Abre o ficheiro para acrescentar | Se o ficheiro não existir, o ficheiro será criado |
| | Os dados são acrescentados no final do ficheiro | |
| ab | Abre o ficheiro binário para acrescentar | Se o ficheiro não existir, o ficheiro será criado |
| | Os dados são acrescentados no final do ficheiro | |
| r+ | Abre o ficheiro para leitura e escrita | Se o ficheiro não existir, fopen() devolve NULL |
| rb+ | Abre o ficheiro para leitura e escrita, em modo binário | Se o ficheiro não existir, fopen() devolve NULL |
| w+ | Abre o ficheiro para leitura e escrita | Se o ficheiro existe, o seu conteúdo é apagado |
| | | Se o ficheiro não existir, o ficheiro será criado |
| wb+ | Abre o ficheiro para leitura e escrita, em modo binário | Se o ficheiro existe, o seu conteúdo é apagado |
| | | Se o ficheiro não existir, o ficheiro será criado |
| a+ | Abre o ficheiro para leitura e acrescentar | Se o ficheiro não existir, o ficheiro será criado |
| ab+ | Abre o ficheiro para leitura e acrescentar, em modo binário | Se o ficheiro não existir, o ficheiro será criado |

*Figura 19 – Diferentes modos de operação com ficheiros, a utilizar como parâmetro de fopen().*

### 8.2.5 Ler e escrever num ficheiro de texto: fscanf() e fprintf()

Para ler o ficheiro de texto e escrever usamos, respetivamente, as funções ***fscanf()*** e ***fprintf()***.

As funções são de utilização semelhante às funções ***scanf()*** e ***printf()***, com a diferença de ser necessário indicar o ponteiro para o ficheiro.

No exemplo seguinte, o programa permite a introdução de um número inteiro pelo utilizador e escreve-o num ficheiro de texto, de nome "***Numero.txt"***.

```
#include <stdio.h>

int main()
{
   int num;
   FILE *fptr;
   fptr = fopen("Numero.txt","w");
   if(fptr == NULL)
   {
      printf("Error!");
      exit(1);
   }
   printf("Número? ");
   scanf("%d",&num);
   fprintf(fptr,"%d",num);
   fclose(fptr);
   return 0;
}
```

Depois de executar o programa anterior, é criado um ficheiro, na mesma localização do programa, com o nome "Numero.txt". Podemos visualizar o conteúdo do ficheiro, através da utilização de um editor de texto (por exemplo *Notepad*), e verificar que o número introduzido está registado no ficheiro.

No exemplo seguinte, o programa permite a leitura do valor inteiro do ficheiro "Numero.txt", e escreve-o no ecrã.

```
#include <stdio.h>

int main()
{
   int num;
   FILE *fptr;
   if ((fptr = fopen("Numero.txt","r")) == NULL){
       printf("Error! opening file");
       // Program exits if the file pointer returns NULL.
       exit(1);
   }
   fscanf(fptr,"%d", &num);
   printf("Valor do ficheiro n= %d \n", num);
   fclose(fptr);

   return 0;
}
```

A função do exemplo seguinte, permite a cópia, carácter a carácter, do ficheiro: ***infile***, para o ficheiro: ***outfile***.

```
int copyfile(char *infile,char *outfile)
{
      FILE *fp1,*fp2;
      if (( fp1 = fopen(infile,"r")) ==NULL)
            return 0; /* Erro */
      if (( fp2 = fopen(outfile,"w")) ==NULL)
      {
            fclose(fp1);
            return 0;
      }
      while (!feof(fp1))
            putc(getc(fp1),fp2);
      fclose(fp1);
      fclose(fp2);
      return 1;  /* Sucesso */
}
```

### 8.2.6 Ler e escrever para ficheiros binários

As funções ***fread()*** e ***fwrite()*** são usadas para a leitura e escrita de dados em ficheiros binários.

Para escrever dados num ficheiro binário usamos a função ***fwrite()***. A função ***fwrite()*** é constituída por 4 parâmetros:

```
fwrite(addressData, sizeData, numbersData, pointerToFile);
```

`addressData`: Endereço dos dados a gravar;
`sizeData`: Tamanho dos dados a gravar;
`numbersData`: Número desse tipo de dados a gravar;
`pointerToFile`: Ponteiro para o ficheiro em que deseja gravar.

Exemplo de utilização da função ***fwrite()***:

```
#include <stdio.h>

typedef struct{
   int numero;
   char nome[25];
   float taxa;
}DADOS;

int main()
{
   int n;
   DADOS reg;
   FILE *fptr;
   if ((fptr = fopen("Dados.bin","wb")) == NULL){
       printf("Error! opening file");
       // Program exits if the file pointer returns NULL.
       exit(1);
   }
   for(n = 1; n < 5; n++)
   {
      reg.numero = n;
      reg.taxa = n / 10.0;
      strcpy(reg.nome,"Nome");
      fwrite(&reg, sizeof(DADOS), 1, fptr);
   }
   fclose(fptr);

   return 0;
}
```

A função ***fread()*** é utilizada para ler dados de um ficheiro binário. A função fread(), muito semelhante à função ***fwrite()***, também, é constituída por 4 argumentos:

```
fread(addressData, sizeData, numbersData, pointerToFile);
```

onde:

`addressData`: Endereço da variável que vai receber os dados a ler;
`sizeData`: Tamanho dos dados a ler;
`numbersData`: Número desse tipo de dados a ler;
`pointerToFile`: Ponteiro para o ficheiro que desejamos ler.

Exemplo de utilização de fread(). O programa permite a leitura dos dados do ficheiro "Dados.bin", utilizado no programa anterior.

```
#include <stdio.h>
typedef struct dados
{
   int numero;
   char nome[25];
   float taxa;
}DADOS;

int main()
{
   int n;
   DADOS reg;
   FILE *fptr;
   if ((fptr = fopen("Dados.bin","rb")) == NULL){
       printf("Error! opening file");
       // Program exits if the file pointer returns NULL.
       exit(1);
   }
   for(n = 1; n < 5; n++)
   {
     fread(&reg, sizeof(DADOS), 1, fptr);
     printf("Numero: %d\n", reg.numero);
     printf("Nome: %s\n", reg.nome);
     printf("Taxa: %.2f\n", reg.taxa);
   }
   fclose(fptr);

   return 0;
}
```

### 8.2.7 Ler dados com posicionamento no ficheiro: fseek().

Se existirem vários registos no ficheiro e desejamos ler um registo específico, devemos utilizar a função ***fseek()*** para posicionar o cursor do ficheiro no registo a ler.

A sintaxe de utilização da função fseek(), é a seguinte:

```
fseek(FILE * stream, long int offset, int whence);
```

O primeiro parâmetro `stream` é o ponteiro para o ficheiro. O segundo parâmetro é a posição do registo que pretendemos, por último, `whence` é o local de partida para o deslocamento do `offset`.

Na Figura 20, podemos consultar as diferentes opções para o parâmetro `whence`.

| De onde | Significado |
|---|---|
| SEEK_SET | Inicio do ficheiro |
| SEEK_END | Fim do ficheiro |
| SEEK_CUR | Posição corrente do ficheiro |

*Figura 20 - Diferentes opções para posicionamento no ficheiro.*

O programa seguinte permite a leitura do ficheiro “Dados.bin” por ordem inversa.

```
#include <stdio.h>
typedef struct dados
{
   int numero;
   char nome[25];
   float taxa;
}DADOS;

int main()
{
   int n;
   DADOS reg;
   FILE *fptr;
   if ((fptr = fopen("Dados.bin","rb")) == NULL){
       printf("Error! opening file");
       // Program exits if the file pointer returns NULL.
       exit(1);
   }
   // Move o cursor do ficheiro para o fim (SEEK_END)
   fseek(fptr,-1*sizeof(DADOS),SEEK_END);
   for(n = 1; n < 5; n++)
   {
      fread(&reg, sizeof(DADOS), 1, fptr);
      printf("Numero: %d\n", reg.numero);
      printf("Nome: %s\n", reg.nome);
      printf("Taxa: %.2f\n", reg.taxa);
      // Move a partir da posição corrente (SEEK_CUR)
      fseek(fptr, -2*sizeof(DADOS), SEEK_CUR);
   }
   fclose(fptr);

   return 0;
}
```

Na manipulação de ficheiros por vezes não sabemos quantos registos existem no ficheiro, pelo que não podemos utilizar o ciclo ***for***. No entanto, com um cálculo simples podemos determinar o número de registos do ficheiro. A instrução seguinte permite determinar o número de registos do ficheiro:

```
fseek(fptr, 0, SEEK_END);

int nr = ftell(fptr) / sizeof(DADOS);
```

Onde, a instrução `fseek(fptr, 0, SEEK_END);` posiciona o ponteiro do ficheiro no fim do ficheiro. A instrução `ftell(fptr)` devolve a posição atual do ponteiro do ficheiro em bytes; `sizeof(DADOS)` dá-nos o valor em bytes de cada registo. Ou seja, dividindo o total de bytes do ficheiro pelo valor de cada registo, o resultado é o número de registos do ficheiro.

Utilizando o cálculo do número de registos, o ciclo de leitura do ficheiro fica:

```
nr = ftell(fptr) / (sizeof(DADOS));
fseek(fptr,0,SEEK_SET);
for (n = 0; n < nr; n++){
     fread(&reg, sizeof(DADOS), 1, fptr);
     printf("Numero: %d\n", reg.numero);
```

```
            printf("Nome: %s\n", reg.nome);
            printf("Taxa: %.2f\n", reg.taxa);
      }
```

No exemplo seguinte, apresentamos uma outra forma de efetuar a leitura do ficheiro.

```
      while (fread(&reg, sizeof(DADOS), 1, fptr)){
            printf("Numero: %d\n", reg.numero);
            printf("Nome: %s\n", reg.nome);
            printf("Taxa: %.2f\n", reg.taxa);
}
```

De salientar, que a função ***fread()***, devolve o valor 1 enquanto tem dados para ler, ou seja, faz o ciclo enquanto não chega ao fim do ficheiro. De destacar, também, que a função ***fread()***, lê um bloco de dados e posiciona o ponteiro do ficheiro na posição seguinte.

## 8.3 Exercícios propostos

***Exercicio86*** – Utilizando o Microsoft Excel criar uma matriz de 20 x 20, com valores inteiros entre [-200, 200]. Exportar essa matriz para um ficheiro de texto separada por tabulações de nome Matriz.txt. Faça um programa em C que permita a leitura da matriz para um array bidimensional, e escreva a matriz no ecrã com o aspeto usual de matriz.
***Exercicio87*** – Faça um programa que partindo da matriz anterior, escreva a matriz num ficheiro de texto, de nome *NovaMatriz.txt*, onde os valores negativos são substituídos por 0 (zero).
***Exercicio88*** – Utilize o programa do **Exercício85**. Na gestão dos dados (nome, número, nota) dos alunos de uma unidade curricular. Faça a leitura do nome e número de vários alunos e escreva-os num ficheiro binário. A nota deve iniciar com o valor -1 (significa que não tem nota atribuída).
***Exercicio88.1*** – Pretende-se lançar as notas dos alunos. Faça a leitura dos alunos do ficheiro e atualize as notas. As notas devem pertencer ao intervalo [0, 20]. A introdução deve ter o aspeto seguinte:

1009876 Bruno Gomes Nota? 15.3

***Exercicio88.2*** – Utilizando a função de ordenação, Exercício 85.2, ordenar os alunos pelo nome e guarde os dados num novo ficheiro.
***Exercicio88.3*** – Faça uma função que permita alterar a nota de um aluno, dado o seu número, e atualizar essa informação no ficheiro.
***Exercicio88.4*** – Faça uma função que permita eliminar um aluno, dado o seu número, e atualizar essa informação no ficheiro. ***Sugestão****: não deve eliminar o aluno, coloque uma marca num dos campos do registo do aluno. Faça uma nova função para leitura do ficheiro e visualização, onde o aluno com a marca de eliminado não é mostrado.*

# 9 Referências


Blackwell, A. (2002). What is Programming? *14th Workshop of the Psychology of Programming Interest Group* (pp. 204-218). Brunel University: In J. Kuljis, L. Baldwin & R. Scoble (Eds). Proc. PPIG 14.

Bolton, D. (2011). *What is programming?* Obtido em 20 de Abril de 2011, de About.com: http://cplus.about.com/od/introductiontoprogramming/p/programmers.htm

Caspersen, M. E. (2007). *Educating novices in the skills of programming.* PhD dissertation PD-07-4, University of Aarhus, Department of Computer Science.

Gauld, A. (27 de Maio de 2007). Obtido em 20 de Abril de 2011, de Learning to Program: http://www.freenetpages.co.uk/hp/alan.gauld/tutwhat.htm

Hoc, J., & Nguyen-Xuan, A. (1990). Language semantics, mental models and analogy. *Eds. Psychology of Programming* (pp. 139-156). London: Academic Press.

Kernighan, B., & Ritchie, D. (1988). *The C Programming Language.* Prentice Hall Software Series.

Programiz.com. (29 de Outubro de 2013). *Tutorial To Learn Programming.* Obtido de Programiz.com: http://http://www.programiz.com/

Programming is simplified. (29 de Outubro de 2013). *C programming.* Obtido de Programming is simplified: http://www.programmingsimplified.com/

Schneider, D., Godard , R., Drozdowski, T., Glusman , G., Block , K., & Tennison, J. (17 de April de 1997). The evolving TecfaMOO Book - Part II: Technical Manual.