

# A linguagem C



PROGRAMA OPERACIONAL REGIONAL DO NORTE



UNIÃO EUROPEIA

Fundo Social Europeu



*#include<stdio.h>*

Baseado nos slides da U.C. Fundamentos de Programação (LEI|ESTG|PP) do ano letivo 2013/2014



# Introdução

# Linguagem

- Para que serve uma linguagem?
  - Para comunicar entre duas ou mais entidades.
- Linguagem natural:
  - Descreve ideias, ações, sentimentos, emoções, etc.
  - Apresenta um vocabulário rico e regras gramaticais complexas.
  - É muitas vezes ambígua.

# Linguagem

- Caracteriza-se por possuir:
  - **Semântica** – conjunto de termos, palavras ou sinais que assumem determinados significados.
  - **Sintaxe** – estipula o modo correto de utilizar e estruturar os termos da linguagem.

# Linguagem de programação

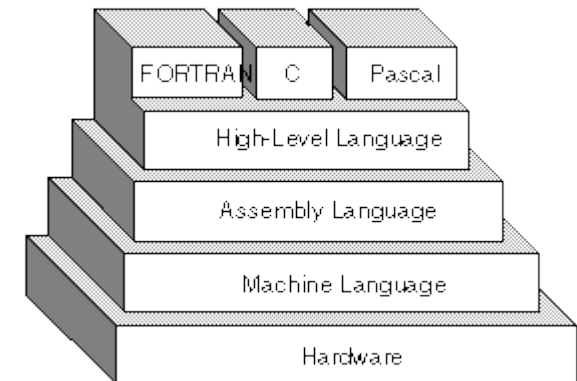
- Descreve operações a serem executadas por um dispositivo.
- Apresenta um vocabulário limitado e regras gramaticais simples.
- É sempre clara e concisa.
- Permite que um programador especifique precisamente sobre que dados um computador vai atuar, como serão armazenados ou transmitidos estes dados e que ações devem ser tomadas em circunstâncias bem definidas.

# Origens

- As primeiras linguagens de programação antecedem o próprio computador. (ex.: Linguagens de programação de máquinas)
- Métodos de programação primitivos:
  - Rodas dentadas e alavancas.
  - Tear de Jacquard.
  - Cartões perfurados de Hollerith.

# Baixo nível vs Alto nível

- As linguagens de programação de computadores são geralmente divididas em:
  - **Linguagens de baixo nível** - Mais próximas do hardware ou nível de máquina (ex.: Linguagem-máquina e Assembly).
  - **Linguagens de alto nível** - Mais próximas da linguagem humana.



# Linguagem- máquina

- Na década de 40 surgem os primeiros computadores digitais e com estes a linguagem-máquina.
- O hardware de um computador apenas entende a linguagem dos dígitos binários ou bits (0 e 1) devidamente codificados em sinais elétricos.
- Esta codificação baseada em agrupamentos de bits ou bytes constitui a linguagem-máquina ou código-máquina.



# Linguagem de baixo nível

- Cada arquitetura de processador possui uma linguagem-máquina própria, que consiste no conjunto de instruções (*instruction set*) que o CPU reconhece e sabe processar.
  - Isto obrigava o programador a ter de conhecer detalhadamente a estrutura e modo de funcionamento de cada modelo de CPU com que trabalhasse.
  - A programação era uma tarefa extremamente complexa e reservada a uma minoria de especialistas.

# Linguagem Assembly

- A linguagem Assembly consiste num conjunto de palavras (mnemónicas ou abreviaturas) que codificam as instruções de máquina (zeros e uns) com que determinado processador funciona.
- Surgiu para facilitar a programação de computadores, sendo mais inteligível e fácil de memorizar do que as sequências de zeros e uns da linguagem-máquina.
- Continua a existir uma correspondência direta entre cada linguagem Assembly e a linguagem-máquina de cada CPU, isto é, cada modelo de CPU tem a sua própria linguagem Assembly.

# Linguagem Assembly

- Apesar de mais inteligível do que a linguagem-máquina, continua a ser muito **complexa** e difícil de utilizar.
- Para cada instrução-máquina tem de se escrever uma instrução em Assembly, o que significa que os programas continuam a ser muito **extensos** e sujeitos a **erros** difíceis de detetar e corrigir.
- Continua a ser considerada linguagem de **baixo nível**.
- Hoje em dia são ainda utilizadas, principalmente, quando é necessária uma Interação mais direta com o hardware (ex.: controlador de dispositivo ou driver, BIOS).

# Linguagens de alto nível

- São assim designadas por estarem muito acima do nível do hardware ou da linguagem-máquina.
- Aproximarem da linguagem humana (inglês)
- Uma linguagem de alto nível descreve de forma mais abstrata as operações que a máquina deve executar, tendo mais em conta o significado das operações para nós, utilizadores, do que para o processador.
- Exemplos: BASIC, Pascal, C, COBOL, FORTRAN.

# Linguagens de alto nível

- A cada instrução numa linguagem de baixo nível corresponde uma única instrução em linguagem-máquina;
- Numa linguagem de alto nível, a cada instrução correspondem, normalmente, diversas operações em linguagem-máquina.
- Não existe correspondência direta entre uma linguagem de alto nível e a linguagem-máquina de cada processador, pelo que os programas criados através das primeiras são mais portáteis.

# Código-fonte, programa executável e software de tradução

- Um programa começa por ser escrito em texto inteligível para os programadores. A esse texto chamamos **código-fonte**. O código, depois de convertido (ou traduzido) em **código-máquina**, passa a designar-se por **programa executável**.
- Para converter o código-fonte em programa executável é necessário um tipo de software específico para fazer esse tipo de conversão ou tradução (das instruções na linguagem de programação para “zeros e uns”).

# Software de tradução

- Para converter um programa escrito numa linguagem Assembly para linguagem-máquina é necessário um software conhecido por **assemblador** (*assembler*).

# Software de tradução

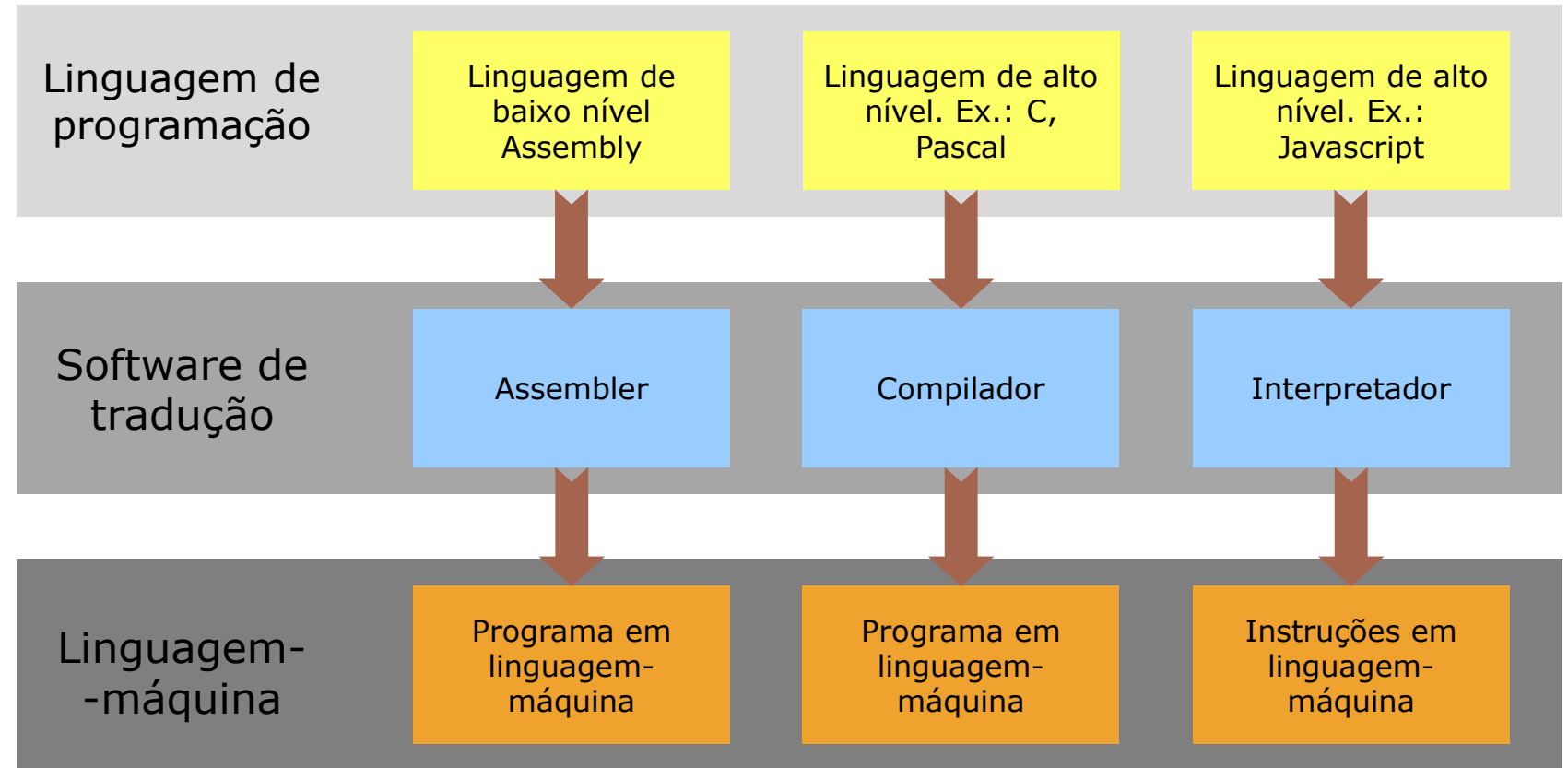
- Para converter um programa escrito numa linguagem de alto nível para código-máquina é necessário um programa *tradutor*, que pode ser de dois tipos diferentes:
  - **Compilador** – traduz a totalidade das instruções de alto nível para um programa em linguagem-máquina, o qual poderá ser executado de forma independente do software que faz a tradução.
  - **Interpretador** – traduz instrução por instrução, à medida que o programa vai sendo lido e executado, ficando o programa dependente do software que faz a tradução.



# Software de tradução

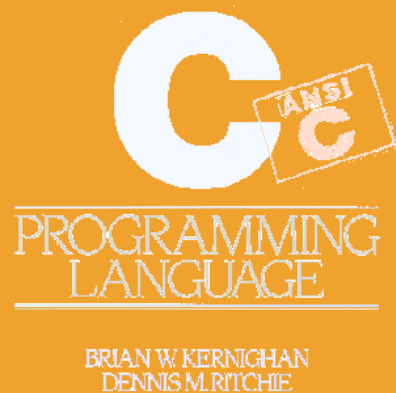
- **Transcompilador** (*Transcompiler* ou *transpiler*) referem-se a “*source-to-source compilers*” que tendo como input o código fonte numa linguagem de programação, produz como output código fonte equivalente numa outra linguagem de programação.

# Software de tradução



# Ambiente de programação

- É um conjunto de ferramentas de suporte destinadas a apoiar o desenvolvimento de programas ou projetos de software, desde a escrita dos programas à sua tradução para linguagem-máquina, passando pela detecção e correção de eventuais erros, bem como outras tarefas de apoio.
- Entre essas ferramentas incluem-se, tipicamente, editores, compiladores ou interpretadores, detecção e tratamento de erros, etc.
- Costuma designar-se por **IDE** (*Integrated Development Environment*).



# A linguagem C

# A linguagem C

- A linguagem C foi criada em 1972 nos Bell Telephone Laboratories por Dennis Ritchie.
- O objetivo era criar uma linguagem de alto nível que possibilitasse a criação de um sistema operativo evitando o recurso ao Assembly.
- Em 1983, o ANSI (American National Standards Institute) formou um comité para a definição de um padrão para a linguagem de modo a que todos os compiladores da linguagem tivessem um comportamento semelhante.
- O C é uma linguagem que se adapta a qualquer tipo de projeto (***General purpose***).

# Características

- **Rapidez** – consegue alcançar um desempenho semelhante ao Assembly usando instruções de alto nível.
- **Portável** – existe um standard (ANSI C) que define as características para os compiladores C, permitindo que o código escrito numa máquina possa ser transferido para outra máquina e compilado sem qualquer alteração (ou com um número reduzido de alterações).
- **Simples** – sintaxe bastante simples e um reduzido número de palavras reservadas, tipos de dados básicos e operadores.
  - Isto contribui para uma redução do tempo e esforço necessários para aprender a linguagem.

# Características

- **Bibliotecas poderosas** – a maior parte das capacidades que a linguagem tem são-lhe acrescentadas através da inclusão de funções existentes em bibliotecas adicionais.
- **Modular** – permite o desenvolvimento modular de aplicações, facilitando a separação de projetos em módulos distintos e independentes.
- **Focada no objetivo** – a forma como o código é escrito depende unicamente do gosto do programador.
  - O programador tem sobretudo que se preocupar com o objetivo e a correção do programa que está a desenvolver.

# Características

- **Alto nível** – considerado uma linguagem de alto nível, apesar de permitir incluir fragmentos de código em Assembly.
- **Popular** – universalmente conhecida e utilizada.
  - Está muito documentada em livros, manuais, fóruns, etc.
  - Existem compiladores para praticamente todo o tipo de arquiteturas e computadores.
- **Evolução** – é uma linguagem particularmente estável.
  - No entanto, evoluiu no sentido de acompanhar outras tendências de programação, nomeadamente a Programação Orientada a Objetos, dando origem à linguagem C++ e estando na base da linguagem Java.



# Tradução de código-fonte em programa executável

```
// ola.c  
#include <stdio.h>  
main() {  
    printf("HELLO!");  
}
```

pré-processador

ola.c

compilador

ola.asm

assembler

ola.obj

linker

ola.exe



# Tradução de código-fonte em programa executável

- **Pré-processador** – trata ficheiros fonte, remove comentários e interpreta instruções ou diretivas de pré-processamento tais como `#define` e `#include`.
- **Compilador** – verifica se o programa foi escrito corretamente, ou seja, verifica a sintaxe das instruções.
  - Além disso, deteta também situações que não são de erro, mas que levantam suspeitas (*warnings*).
  - Caso não existam erros (e mesmo que existam *warnings*) é criado um ficheiro em linguagem Assembly.

# Tradução de código-fonte em programa executável

- **Assembler** – converte um programa em linguagem Assembly para linguagem-máquina. O ficheiro criado é designado por código-objeto.
- **Linker** – se o código-fonte fizer referência a funções contidas em bibliotecas ou definidas noutras fontes, o Linker junta-as à função principal do programa e cria um ficheiro executável.
- A maior parte dos compiladores ocupa-se de todas estas tarefas.

# Compilação por linha de comando

- Compila e cria o executável a.exe<sup>1</sup> (a.out<sup>2</sup>)

```
$ gcc main.c
```

- Apenas compila o ficheiro main.c criando o ficheiro objeto main.o

```
$ gcc -c main.c
```

- Compila e cria o executável myprog.exe (myprog)

```
$ gcc main.c -o myprog
```

(1) nome dado por defeito em Windows

(2) nome dado por defeito em Linux

# Primitiva executável

- Expressão ou estrutura de controlo terminada por “;”.
  - Exemplo:  
`preco = 15;`
- Uma primitiva composta é um conjunto de primitivas.
  - Exemplo.:  

```
if (x == y) {  
    estado = “concluido”;  
    concluidos += 1;  
}
```

} primitiva composta

# *Case sensitive*

- O C é *case sensitive*, existindo uma diferenciação entre maiúsculas e minúsculas.
  - Ex: main ≠ Main ≠ MAIN
  - Todas as instruções de C são escritas em minúsculas.

# Finalização de instruções

- As instruções são finalizadas com “;” (ponto e vírgula), podendo ter na mesma linha várias instruções.
- Embora a disposição do código dependa apenas das preferências de cada programador, existem um conjunto de princípios ([C Style: Standards and Guidelines](#)) que são importantes para tornar o código mais legível e de fácil manutenção.

# Comentários

- Os comentários não têm interferência no programa, servem apenas para documentar o código, possibilitando uma mais fácil manutenção por parte do programador ou de terceiros.
- Um comentário é delimitado por `/* ... */` e pode estender-se por várias linhas.
- Podemos utilizar `//` para comentários com apenas uma linha.
- Exemplo:

```
/* Ficheiro: prog.c  
Autor: OAO  
Data: 01/01/2001 */
```

```
// Este é um comentário de uma linha
```



# Palavras reservadas

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



# Conceitos básicos

# Anatomia de um programa em C

```
/* inclusão de header files */  
/* definição de constantes */  
main() // Ponto de entrada (entry point) de um programa em C  
{      // inicio do bloco de código  
    ... /* declaração de variáveis */  
    ... /* primitivas executáveis */  
}      // fim do bloco de código
```

# Variáveis

- Sempre que temos que guardar um valor que não tenha um valor fixo, podemos recorrer a variáveis.
  - As variáveis são uma referência dada pelo programador a uma determinada posição de memória de modo a conter um valor com determinado tipo.
  - O seu valor poder variar ao longo da execução do programa e terá de ser definida antes de ser utilizada

```
tipo nomeVar1, nomeVar2, ...;
```
- O tipo que lhe está associado indica o número de bytes que irão ser utilizados para guardar o seu valor.

# Variáveis

## Nome - regras

- Apenas pode conter letras do alfabeto, dígitos e o *underscore* ( \_ )
- O primeiro carácter não pode ser um dígito
- Case sensitive (count  $\neq$  Count)
- Não se pode usar palavras reservadas do C para nomear variáveis

# Variáveis

## Nome - cuidados

- O nome de uma variável deve ser descritivo  
xpto vs. Num\_Aluno
- Deve-se evitar nomear variáveis com identificadores todo em maiúsculas
  - Os programadores tradicionalmente nomeiam totalmente constantes em maiúsculas
- Separe nomes que utilizam mais do que uma palavra  
numaluno vs. Num\_Aluno vs. numAluno
- Não iniciar o nome de variáveis com o *underscore*

# Variáveis

## Atribuição

- A atribuição é realizada utilizando o operador igual (=)  
`variável = expressão;`  
`numAluno = 12345;`
- As variáveis podem ser inicializadas quando declaradas.  
`int a, numAluno1 = 12345, numAluno1 = 12346, contador;`
- Podemos atribuir o valor de uma variável a uma outra variável  
`numAluno1 = numAlunoTemporario`

# Variáveis

## Atribuição

- é possível atribuir o mesmo valor a várias variáveis:

```
a = 0;
```

```
b = 0;
```

```
c = 0;
```

```
d = 0;
```

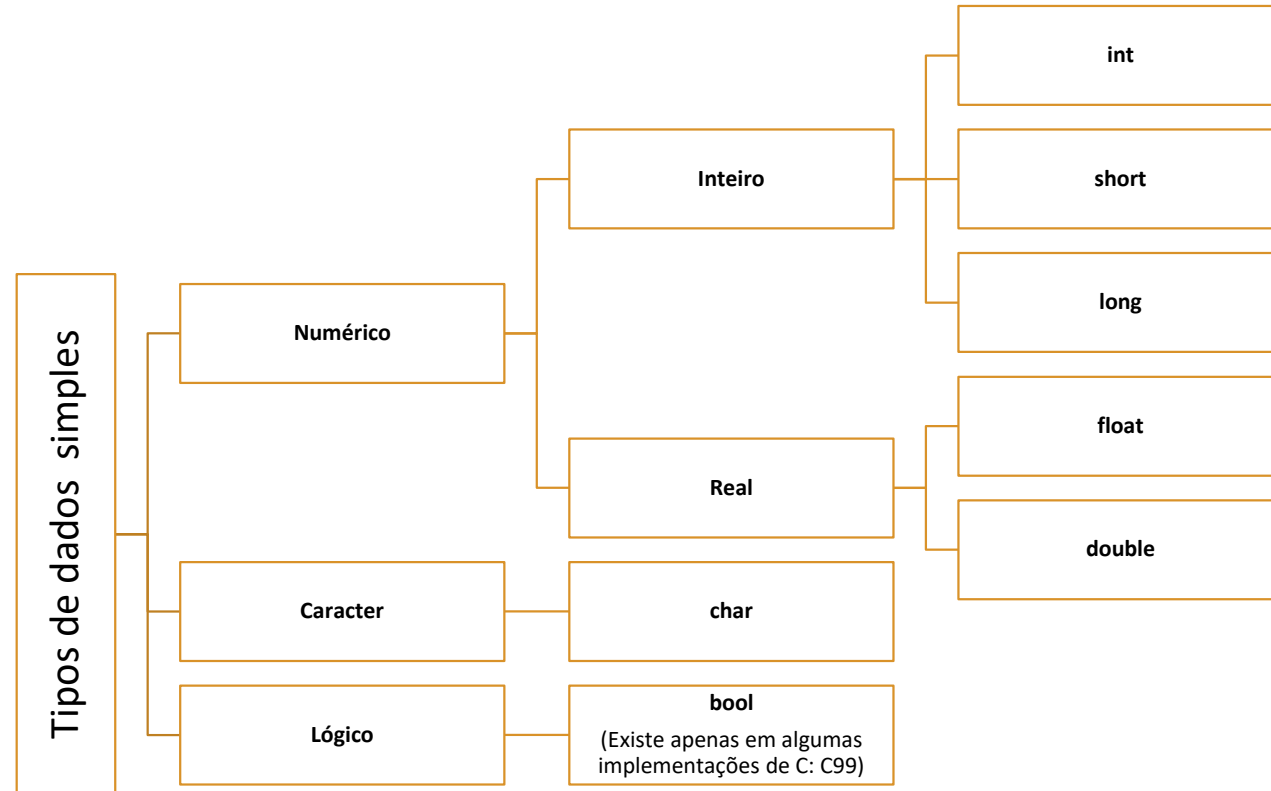
```
e = 0;
```



```
a = b = c = d = e = 0;
```



# Tipos de dados simples



# int

- Os dados do tipo **int** armazenam valores que pertencem ao conjunto dos números naturais (sem parte fracionária) positivos e negativos, incluindo o zero.

```
int soma;  
soma = 45;
```

ou

```
int soma = 45;
```

# int

## Variantes

Tipo	Nº bytes *	Valor mínimo *	Valor máximo *
short int	2	-32768	32768
int	4	-32768	32768
long	4	-2147483648	2147483648
unsigned int	2	0	65535
unsigned short int	2	0	65535
unsigned long int	4	0	4294967295

\* Depende do sistema

# int

## Operações

Operação	Descrição	Exemplo	Resultado
+	Soma	$21 + 4$	25
-	Subtração	$21 - 4$	17
*	Multiplicação	$21 * 4$	84
/	Divisão	$21 / 4$	5
%	Resto da divisão (módulo)	$21 \% 4$	1
++	Incremento	i++	Equivalente a $i = i + 1$
--	Decremento	i--	Equivalente a $i = i - 1$

int

- Qualquer operação entre inteiros devolve um inteiro
- Da divisão entre 21 e 4 não resulta em 5.25 mas sim 5
- **21/4** devolve o **quociente** da divisão de 21 por 4
- **21%4** devolve o **resto** da divisão de 21 por 4

$$\begin{array}{r|l} 21 & 4 \\ \hline 1 & 5 \end{array}$$

# float e double

- Os dados do tipo **float** e **double** armazenam valores numéricos com parte fracionária.  

```
float media;  
media = 14.57;
```
- A diferença entre dados do tipo float e double é o número de bytes que são reservados em memória para armazenar o valor e também a sua precisão.
- A manipulação de números reais pode ser realizada usando a notação científica, especificando uma base e um expoente (como nas calculadoras).
  - Ex.: 123.46E78 é o mesmo que  $123.46 * 10^{78}$

# float e double Operações

Operação	Descrição	Exemplo	Resultado
+	Soma	21.3 + 4	25.3
-	Subtração	21.7 - 4.8	16.9
*	Multiplicação	21.2 * 4.7	99.64
/	Divisão	21.0 / 4.0	5.25
%	Resto da divisão (módulo)	n.a.	

- Qualquer operação em que pelo menos um dos operandos seja real, produz um resultado do tipo real. Ex.:  $2 * 4.1 = 8.2$

# char

- Os dados do tipo char armazenam um único carácter.
  - Não permite armazenar cadeias de caracteres (*strings*).
- Um char é sempre armazenado num byte.
- É o tipo inteiro mais pequeno na linguagem.
  - Mas por uma questão de simplicidade será tratado como um tipo de dados totalmente diferente.
- A representação de caracteres faz-se utilizando plicas e não aspas.
- A cada carácter corresponde um valor inteiro. ([Tabela de ASCII](#))  

```
char letra;  
letra = 'A'; /* letra = 65; */
```



# Operadores compostos

- O operador de igualdade (=) pode ser combinado com outros para simplificar expressões. Notar que o outro operador deve sempre ser colocado à esquerda do sinal de igualdade.

Operador	Equivalência	Descrição
$a *= b$	$a = a * b$	Multiplicação
$a /= b$	$a = a / b$	Divisão
$a \% = b$	$a = a \% b$	Resto
$a += b$	$a = a + b$	Adição
$a -= b$	$a = a - b$	Subtração
$a << = b$	$a = a << b$	Desloc. Esquerda
$a >> = b$	$a = a >> b$	Desloc. direita
$a \& = b$	$a = a \& b$	E
$a \wedge = b$	$a = a \wedge b$	OU exclusivo
$a   = b$	$a = a   b$	OU inclusivo

# Casting

- Podemos promover (despromover) uma variável ou expressão com um determinado tipo para um tipo maior ou menor.

(tipo) <expressão>

- Exemplos:

```
short int soma = 21;
float media;
media = soma / 4;           /* Resultado: 5 */
media = 21.0 / 4;          /* Resultado: 5.25 */
media = (float)soma / 4;    /* Resultado: 5.25 */
media = (float)(soma / 4);  /* Resultado: 5 */
```

# sizeof()

- Para saber qual a dimensão de qualquer tipo de variável, o C disponibiliza um operador cuja sintaxe é semelhante à utilizada para invocar funções.

```
printf("tamanho em bytes de um inteiro: %d\n", sizeof(int));
```

# Constantes

- A definição de constantes é feita do seguinte modo:  
`const tipo NOME_CONSTANTE = valor;`  
`#define NOME_CONSTANTE valor`
- Exemplos:  
`#define PI 3.1416`  
`const double PI = 3.1416;`
- É boa prática, mas não obrigatório, que os nomes das constantes sejam escritos em maiúsculas. Assim, distinguem-se melhor das variáveis.

# printf

- O C não possui mecanismos de output incorporados, tendo de recorrer à bibliotecas para este tipos de funções.
- A função printf (***print*** + **formatted**) permite a escrita em ecrã.
- As strings em C são sempre delimitadas por aspas (“”).

```
/* Escrever a string “Hello Worlds” usando a função printf */  
#include <stdio.h> /* incluir funções de IO */  
main() {  
    printf(“Hello World”);  
}
```

# Exemplos

- `printf("Esta aula é divertida!");`  
`Esta aula é divertida!`
- `printf("Disciplina:\tLP1");`  
`Disciplina: LP1`
- `printf("2 + 2 = %d", 2 + 2);`  
`2 + 2 = 4`
- `printf("valor1: %d, valor2: %f", soma, media);`  
`valor1: 345, valor2: 47.5`

# Especificadores de formato

Tipo	Formatação
int	%d
long	%ld
unsigned int	%u
unsigned short int	%hu
char	%c
float	%f
diuble	%lf

## *Escape characters*

- De modo a conseguir apresentar vários caracteres que poderiam confundir-se com, por exemplo, delimitadores, o símbolo \ permite retirar o significado que o carácter tem e simplesmente ser considerado como carácter.

```
printf("0 meu texto com \"aspas\");  
0 meu texto com "aspas"
```



## *Escape characters*

- O símbolo \ permite ainda representar caracteres que muito dificilmente poderiam ser representados de outra forma, como por exemplo, o *new line* (\n) que efetua uma mudança de linha.

```
printf("O meu \ntexto em \nvárias linhas");
```

```
O meu
```

```
texto em
```

```
várias linhas
```

## *Escape characters*

- A tabela apresenta alguns dos caracteres especiais mais usados

<code>\b</code>	<i>backspace</i>
<code>\n</code>	<i>(new line)</i> mudança de linha
<code>\t</code>	tabulação horizontal
<code>\\</code>	carácter \
<code>\'</code>	aplica
<code>\"</code>	aspas
<code>\?</code>	?
<code>%</code>	%

# getchar() putchar(int)

- **int getchar (void)**
  - Lê um carácter do stdin (tipicamente o monitor) e devolve o valor inteiro correspondente (código ASCII).

```
char letra;  
letra = getchar();  
printf("%c", letra);
```
- **int putchar (int character)**
  - Escreve um carácter no stdout (tipicamente no ecrã). É necessário indiciar o valor ASCII do carácter.

```
putchar( 97 ); //equivalente a  
putchar( 'a' );
```

# Operações de entrada de dados

- A entrada de dados a partir do teclado é feita usando a função **scanf**.
- Exemplos:  
    `scanf("%f", &num);`  
    `scanf("%d%d", &x, &y);`  
    `scanf("%c", &letra);`
- ATENÇÃO: para ler o valor a armazenar numa variável do tipo `int`, `char`, etc. utilizando a função `scanf`, tem que se preceder o nome de cada variável com um `&`.

# Operações de entrada/saída de dados

- Para usar as funções `printf`, `scanf`, `getchar` e `putchar` é necessário incluir o *header file* `stdio.h`.
- [Mais informações](#)

## Cuidados na leitura de caracteres

- É conveniente limpar o *buffer* de entrada de dados para uma leitura correta de uma variável do tipo char.
- Exemplo (a limpeza poderia ser invocada antes da leitura):

```
char a, b;  
a = getchar();  
limparBufferEntrada();  
b = getchar();  
limparBufferEntrada();
```

# Limpar o *buffer*

```
/*
 * @brief Limpa o buffer de entrada de dados.
 * @warning Caso não exista nenhuma informação no buffer, o programa pode
 * ficar a aguardar que exista informação no buffer.
 *
 * Exemplo:
 * @code
 * char car:
 *
 * car = getchar();
 * limparBufferEntrada();
 * @endcode
 */
void limparBufferEntrada(){
    char ch;
    while ((ch = getchar()) != '\n' && ch != EOF);
}
```

# Operadores de Incremento / Decremento em C

## Operadores unários

- **Incremento: ++**
  - Adiciona 1 ao seu operando.
  - Ex: `i++;` // Equivalente a `i = i + 1;` ou `i += 1;`
- **Decremento: --**
  - Subtrai 1 ao seu operando.
  - Ex: `i--;` // Equivalente a `i = i - 1;` ou `i -= 1;`



# Operadores de Incremento / Decremento em C

- Sintaxe:
  - Pré-incremento: `++variável`
  - Pós-incremento: `variável++`
  - Pré-decremento: `--variável`
  - Pós-decremento: `variável--`
- Necessário utilizar variáveis na sua utilização
  - `i++;` `// Correto`
  - `(i + val)++;` `// Incorreto`

# Operadores de Incremento / Decremento em C

- Quando os prefixos `++` ou `--` são utilizados numa variável, a variável é primeiro incrementada ou decrementada, respetivamente, e só depois utilizada.

```
int i = 1;  
val = ++i; // variável val contém o valor "2"
```

- Quando os sufixos `++` ou `--` são utilizados numa variável, a variável é primeiro utilizada e só depois incrementada ou decrementada, respetivamente.

```
int i = 1;  
val = i++; // variável val contém o valor "1"
```

# Operadores de Incremento / Decremento em C

- Considerando a existência de duas variáveis do tipo inteiro:

```
int total, val = 10;
```

Operador	Instruções equivalentes	Valor de total	Valor de val
<code>total = val++;</code>	<code>total = val;</code> <code>val += 1;</code>	10	11
<code>total = ++val;</code>	<code>val += 1;</code> <code>total = val;</code>	11	11
<code>total = val--;</code>	<code>total = val;</code> <code>val -= 1;</code>	10	9
<code>total = --val;</code>	<code>val -= 1;</code> <code>total = val;</code>	9	9

# Operadores de Incremento / Decremento em C

- Considerando a existência de duas variáveis do tipo inteiro:

```
int num1 = 2, num2 = 1;
```

- Qual o valor final de res?

```
res = (num1--) * num2;
```

- Qual o valor final da variável num2?

```
if ((num1 == 2) || (num1 != ++num2)) { ... }
```

- Qual o valor impresso no ecrã?

```
printf ("%d", ++num1 + ++num1);
```



# Estruturas de controlo condicional

# Estruturas de controle condicional

- Na formulação de um programa, tal como no dia-a-dia, é muitas vezes necessário avaliar uma determinada condição, para decidir se deve ser executada uma ou outra ação (ou conjunto de ações).
- Teremos nestas situações que usar estruturas de controle condicional, que podem ser de decisão ou de seleção.

# O que é uma condição lógica?

- Permite a execução de um determinado conjunto de ações dependendo do valor devolvido por uma condição lógica.
- Devolve apenas um de dois possíveis valores: **VERDADEIRO** ou **FALSO**.
- Em C, o valor 0 (zero) corresponde a FALSO e um qualquer valor diferente de 0 é considerado VERDADEIRO.

# Tipo de dados lógico em C

- Antes do ISO C99 não existia o tipo de dados lógico em C. Este foi adicionado através do *header file* “stdbool.h”
- Alternativamente, definir uma enumeração:

```
typedef enum {false=0, true=1} bool;
```

```
int main(){  
    bool b = false;  
    (...)  
}
```



# Estruturas de decisão

- As estruturas de decisão assentam numa expressão do tipo “se... então... senão...”
- Permitem, com base numa condição, decidir sobre a execução ou não de uma determinada ação ou optar entre duas alternativas.
- Exemplos:
  - **se** está a chover **então** levo guarda-chuva.
  - **se** está a chover **então** calço botas **senão** calço sapatilhas.

“se...  
então...”

```
if (condição lógica)  
    acção1;
```

- Exemplo:

```
    if (nota >= 10) {  
        printf("O aluno teve positiva");  
    }
```

“se...  
então...  
senão...”

```
if (condição lógica)
```

```
    acção1;
```

```
else
```

```
    acção2;
```

- Exemplo:

```
    if (nota >= 10) {  
        printf("O aluno teve positiva");  
    } else {  
        printf("O aluno teve negativa");  
    }
```

Uma decisão  
pode executar  
várias ações

```
if (condição lógica) {  
    acção1; acção2; acção3;  
}  
else {  
    acção4; acção5; acção6;  
}
```

- Exemplo:

```
if (nota >= 10) {  
    printf("O aluno teve positiva");  
    aprovados++;  
}  
else {  
    printf("O aluno teve negativa");  
    reprovados++;  
}
```

# Operadores relacionais

Operador	Compação
==	Igual a ...
<	Menor que ...
<=	Menor ou igual que ...
>	Maior que ...
>=	Maior ou igual que ...
!=	Diferente de ...

# Exercícios

1. Leia dois números inteiros e escreva o maior deles.
2. Determine e escreva o montante de impostos a pagar sobre um salário anual lido, tendo em conta o seguinte:
  - Salário até 1500€ inclusive paga taxa de 20%
  - Salário superior a 1500€ paga taxa de 30%

# Estruturas de decisão aninhadas

- Por vezes não basta testar uma condição lógica para tomar uma decisão, por isso uma ação pode ser outro if (aninhado ou encadeado).

```
if (teste1 >= 10) {  
    if (teste2 >= 10) {  
        printf("Está dispensado de exame.");  
    } else {  
        printf("Tem de ir a exame.");  
    }  
} else {  
    printf("Tem de ir a exame.");  
}
```

# Exercício

3. Determine e escreva o montante de impostos a pagar sobre um salário anual lido, tendo em conta o seguinte:
  - Salário até 1500€ inclusive paga taxa de 20%
  - Salário de 1500€ a 2000€ inclusive paga taxa de 30%
  - Salário de 2000€ a 2500€ inclusive paga taxa de 35%
  - Salário superior a 2500€ paga taxa de 40%



# Exercício

4. Escreva um programa que leia o lugar em que terminou o piloto e escreva quantos pontos este ganhou. Os pontos no final de cada corrida são atribuídos da seguinte forma:
  - 1º lugar - 10 pontos
  - 2º lugar - 8 pontos
  - 3º lugar - 6 pontos
  - 4º lugar - 5 pontos
  - 5º lugar - 4 pontos
  - 6º lugar - 3 pontos
  - 7º lugar - 2 pontos
  - 8º lugar - 1 ponto
  - Os restantes têm zero pontos.

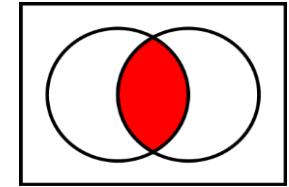
# Operadores lógicos

- Os operadores lógicos permitem obter como resultado apenas um de dois valores lógicos: verdadeiro ou falso.

Operador	Significado
&&	Conjunção (E)
!!	Disjunção (OU)
!	Negação (NÃO)

# Conjunção lógica (&&)

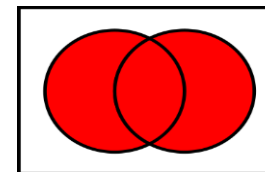
- Verdade se ambas as condições forem verdadeiras.
- Falso se alguma delas for falsa.



```
if (idade >= 18 && genero == 'M') {  
    printf("Adulto do sexo masculino");  
}
```

p	q	p && q
V	V	V
V	F	F
F	V	F
F	F	F

# Disjunção lógica (||)

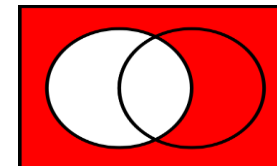


- Verdade se alguma das condições for verdadeira.
- Falsa se ambas forem falsas.

```
if (nota == 5 || nota == 4 || nota == 3) {  
    printf("Aprovado.");  
} else {  
    printf("Reprovado.");  
}
```

p	q	p    q
V	V	V
V	F	V
F	V	V
F	F	F

# Negação lógica (!)



- Verdade se a condição for falsa.
- Falsa se a condição for verdadeira.

```
if (!happy){  
    printf("Don't worry, be happy!");  
}
```

p	!p
F	V
V	F

# Resumo operadores lógicos

Operador	Significado	Exemplo	Resultado
&&	E	(cond1 && cond2)	Verdade se ambas as condições lógicas forem verdadeiras; Falso se alguma delas for falsa.
	OU	(cond1    cond2)	Verdade se alguma das condições lógicas for verdadeira; Falso se ambas forem falsas.
!	Negação	!(cond1)	Verdade se a condição lógica for falsa; Falso se a condição lógica for verdadeira.

## Exercício

5. Reescreva o seguinte programa de modo a não usar um if aninhado:

```
if (teste1 >= 10) {  
    if (teste2 >= 10) {  
        printf("Está dispensado de exame.");  
    } else {  
        printf("Tem de ir a exame.");  
    }  
} else {  
    printf("Tem de ir a exame.");  
}
```

## Exercício

(Soluções possíveis)

```
if (teste1 >= 10 && teste2 >= 10) {  
    printf("Está dispensado de exame.");  
} else {  
    printf("Tem de ir a exame.");  
}
```

```
if (teste1 < 10 || teste2 < 10) {  
    printf("Tem de ir a exame.");  
} else {  
    printf("Está dispensado de exame.");  
}
```



# Operador condicional (?)

- Em condições do tipo “SE... ENTÃO... SENÃO...” podemos usar o operador ?:

`condição ? acção1 : acção2;`

- Exemplo:

A instrução

```
if (nota >= 10) printf("Positiva");  
else printf("Negativa");
```

pode ser escrita de duas formas alternativas :

- `(nota >= 10) ? printf("Positiva") : printf("Negativa");`
- `printf("%s", (nota >= 10) ? "Positiva" : "Negativa");`

# Estruturas de seleção

(Escolha múltipla)

- As estruturas de seleção assentam numa expressão do tipo “escolhe... caso...”
- Permitem, mediante uma variável que funciona como um seletor, optar entre um determinado número de alternativas (casos) possíveis.
- As estruturas de decisão são particularmente úteis como possíveis alternativas a situações em que temos se aninhados.

# Estruturas de seleção

```
switch (variavel) {  
    case expressão1:  
        acção1;  
        break;  
    case expressão2:  
        acção2;  
        break;  
    ...  
    case expressãoN:  
        acçãoN;  
        break;  
    default:  
        acção_default;  
}
```

# Estruturas de seleção

- A instrução **switch** avalia uma variável e depois compara-a com as constantes de tipo inteiro ou caracter de cada case.
  - É selecionado o case com a constante igual ao valor de **variável**.
- A instrução **break** faz com que a estrutura switch termine de imediato.
  - Se esta instrução não estiver presente no final do caso selecionado, serão avaliados todos os casos seguintes.
- A inclusão de uma cláusula **default** é opcional e permite contemplar os casos que não verifiquem nenhuma das restantes condições.

# Exemplo

```
scanf("%d", &numero);  
switch (numero) {  
    case 1:  
        printf("Um");  
        break;  
    case 2:  
        printf("Dois");  
        break;  
    case 3:  
        printf("Três");  
        break;  
    default:  
        printf("Escolheu outro numero!");  
}
```

# Exemplo

```
switch (estado_civil) {  
    case 's': case 'S': printf("Solteiro"); break;  
    case 'c': case 'C': printf("Casado"); break;  
    case 'v': case 'V': printf("Viúvo"); break;  
    case 'd': case 'D': printf("Divorciado"); break;  
    default: printf("Estado civil incorreto!");  
}
```

# Exercício

6. Usando o `switch`, escreva um programa que leia o lugar em que terminou o piloto e escreva quantos pontos este ganhou. Os pontos no final de cada corrida são atribuídos da seguinte forma:
- 1º lugar - 10 pontos
  - 2º lugar - 8 pontos
  - 3º lugar - 6 pontos
  - 4º lugar - 5 pontos
  - 5º lugar - 4 pontos
  - 6º lugar - 3 pontos
  - 7º lugar - 2 pontos
  - 8º lugar - 1 ponto
  - Os restantes têm 0 pontos.



# Estruturas de repetição



# Estruturas de repetição

- É comum, no dia a dia, quando estamos a realizar uma determinada tarefa, termos que repetir uma determinada operação uma ou várias vezes até atingirmos um certo objetivo de uma tarefa.
- Exemplos:
  - Ao mudar uma lâmpada fundida, desapertá-la enquanto não se soltar do casquilho.
  - Ao fazer um bolo, bater claras até estas ficarem em castelo.

# Estruturas de repetição

- Esta repetição de operações é descrita através de estruturas de repetição (ou ciclos) que **permitem executar um conjunto de ações um certo número de vezes**, determinado pela avaliação de uma condição que devolve um valor lógico.
- A uma execução de um ciclo chamamos uma **iteração**.
- Vamos estudar os seguintes ciclos:
  - **while** (enquanto...faz)
  - **for** (para)
  - **do... while** (faz...enquanto)

# while

- O ciclo **while** executa uma ação ou bloco de ações enquanto for verdadeira uma condição lógica, avaliada antes da execução das ações.

```
while (condição lógica) {  
    acção1;  
    acção2;  
    ...  
    acçãoN;  
}
```

- Funcionamento:
  - A condição lógica é avaliada.
  - Se o resultado da avaliação for Falso, o ciclo termina.
  - Se o resultado da avaliação for Verdadeiro, são executadas as ações associadas ao ciclo, voltando-se depois à avaliação da condição lógica.

# while

```
int numero, limite, contador = 0;
scanf ("%d", &limite);
while (limite > 0) {
    scanf("%d", &numero);
    limite -= numero;
    contador++;
}
printf ("Foram introduzidos %d numeros!!!", contador);
```

- Este programa conta todos os números introduzidos pelo utilizador até ultrapassar um dado limite.
- Qual o resultado se a subtração da variável limite não existisse?

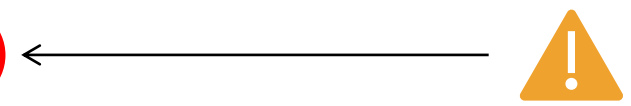
# Exercícios

7. Escreva um programa que imprima os números pares de 0 a 20 inclusive.
8. Escreva um programa que vá somando números introduzidos pelo utilizador enquanto a soma destes não chegar a 100.

# do ... while

- O ciclo **do...while** executa uma ação ou bloco de ações enquanto for verdadeira uma condição lógica, avaliada depois da execução das ações.

```
do {  
    acção1;  
    acção2;  
    ...  
    acçãoN;  
} while (condição lógica);
```



- Funcionamento:
  - São executadas as ações associadas ao ciclo.
  - A condição lógica é avaliada.
  - Se o resultado da avaliação for Falso, o ciclo termina.
  - Se o resultado da avaliação for Verdadeiro, executam-se novamente as ações associadas ao ciclo voltando-se depois à avaliação da condição lógica.

do  
...  
while

```
int numero = 0, soma = 0;  
do {  
    soma += numero;  
    scanf("%d", &numero);  
} while ( numero > 0 );
```

- Este programa calcula a soma dos números positivos introduzidos pelo utilizador.

# Exercícios

9. Escreva um programa que imprima os números pares de 0 a 20 inclusive.
10. Escreva um programa que vá somando números introduzidos pelo utilizador enquanto a soma destes não chegar a 100.



# for

- O ciclo **for** é um ciclo que deve ser utilizado quando conhecemos o número de iterações que o ciclo irá realizar (apesar de o mesmo permitir utilizar o ciclo de uma forma semelhante ao `while`)

```
for (inicializações; condição lógica; pós-instrução) {  
    acção1;  
    acção2;  
    ...  
    acçãoN;  
}
```

# for

- Funcionamento:
  - É executado o código presente em inicializações. É onde são geralmente inicializadas as variáveis do ciclo. Este código é executado uma única vez.
  - A condição lógica é avaliada.
  - Se o resultado da avaliação for Falso, o ciclo termina.
  - Se o resultado da avaliação for Verdadeiro, são executadas as ações associadas ao ciclo, sendo depois executada a pós-instrução, geralmente um incremento ou decremento.
  - Volta-se à avaliação da condição lógica.

# for

```
int numero;  
for (numero = 1; numero <= 10; ++numero) {  
    printf("%d\n", numero);  
}
```

- Este programa escreve os números de 1 a 10 inclusive.

# Exercícios

11. Escreva um programa que imprima os números pares de 0 a 20 inclusive.
12. Escreva um programa que vá somando números introduzidos pelo utilizador enquanto a soma destes não chegar a 100.

# Ciclos

- A escolha do ciclo a usar deve sempre ser feita com cuidado, devendo adotar-se a estrutura mais adequada ao contexto do problema que se pretende resolver.

	while	for	do...while
Executa a instrução	0 ou + vezes	0 ou + vezes	1 ou + vezes
Testa a condição	Antes da ação	Antes da ação	Depois da ação
Utilização	Frequente	Frequente	Pouco frequente

# Ciclos aninhados

- Uma ação de um ciclo de qualquer tipo pode ser outro ciclo de qualquer tipo (ciclo aninhado ou encadeado).

```
int num1 = 1, num2;  
while (num1 <= 5) {  
    for (num2 = 1; num2 <= num1; ++num2) {  
        printf("%d", num2);  
    }  
    printf("\n");  
    ++num1;  
}
```

```
1  
12  
123  
1234  
12345
```

# Ciclos infinitos

- Chama-se ciclo infinito a um ciclo que nunca termina. São, quase sempre, resultado de erros de programação.
- Exemplo:

```
int numero;  
numero =1;  
while (numero <= 10) {  
    printf("%d", numero);  
    --numero;  
}
```



$f(x)$

Funções



# Funções

- **Funções** são sequências de instruções que podem receber valores e que podem devolver um resultado.
- Deve preocupar-se apenas com a resolução de uma tarefa
- Podem tornar o desenvolvimento de programas mais fácil dividindo um problema em sub-problemas.
  - O programa resolve o problema e as suas funções resolvem os sub-problemas.
- Além de facilitarem bastante a reutilização de porções de códigos, aumentam a legibilidade e manutenção do programa.

# Funções

- Não é necessário conhecer a sua implementação, ou seja, o seu código. Apenas é necessário:
  - Conhecer o seu comportamento (o que faz);
  - Fornecer informação adicional (caso seja necessário) através dos seus parâmetros;
  - Determinar o tipo de retorno da mesma (Caso seja necessário).

# Exemplos

- **`sqrt(25)`** – é uma função incluída na biblioteca `math.h` e que recebe um parâmetro, um valor numérico, e devolve um valor numérico. Serve para calcular uma raiz quadrada.
- **`scanf("%d", &num)`** – é uma função incluída na biblioteca `stdio.h` e que recebe dois parâmetros, uma string de formatação e um apontador de memória, e devolve um valor inteiro.
- **`printf("teste")`** – é uma função incluída na biblioteca `stdio.h` e que recebe um parâmetro, uma string, e devolve um valor inteiro.
- **`int main()`** – é uma função que define o ponto inicial de execução de qualquer programa em C.

# Esqueleto geral de uma função

```
tipoDevolvido nomeFunção(listaParâmetros) {  
    Declaração de variáveis locais;  
    Primitivas executáveis;  
    Devolução de resultado;  
}
```

- **tipoDevolvido**: depende daquilo que a função devolve podendo ser, por exemplo, `int`, `float`, ou `char`.
- **nomeFunção**: deve representar uma Ação. Ignorando o significado semântico, segue as mesmas regras da declaração de variáveis.
- **listaParâmetros**: são os parâmetros que a função recebe de “quem” a invocou.
  - Os **parâmetros** vão funcionar como **variáveis locais** da função.
  - É preciso indicar o tipo e nome de cada parâmetro. Quando se invoca uma função, o número, tipo e ordem dos argumentos enviados deve ser coincidente com os parâmetros na declaração da função.

# Esqueleto geral de uma função

- **Declaração de variáveis locais:** declaração das variáveis que serão usadas na execução da função.
- **Devolução de resultados:** termina a função e devolve o seu resultado a “quem” a invocou.
  - O resultado pode ser um valor (exemplo: um número ou o conteúdo de uma variável) ou uma expressão (exemplo: um cálculo matemático).
  - Representa-se através da instrução `return` seguido do valor ou expressão a devolver.
    - Um programa/função pode ter várias instruções de devolução de resultados. No entanto, apenas uma delas é executada em cada invocação da função.

# Exemplo

```
long  somar(int num1, int num2){  
    long total;  
    total = num1 + num2;  
    return total;  
}
```

- Equivalente a:

```
long somar(int num1, int num2) {  
    return num1 + num2;  
}
```

# Invocação

```
#include <stdio.h>

long somar(int num1, int num2) {
    return num1 + num2;
}

main() {
    int a, b;
    long soma;
    scanf("%d", &a);
    scanf("%d", &b);
    soma = somar(a, b);
    printf("%ld", soma);
}
```

# Funcionamento

- Quando uma função é **invocada**, o bloco de código que a invoca é temporariamente “suspense”.
  - Ao invocar uma função podem ser enviados argumentos, que serão recebidos pela função. Estes serão armazenados em variáveis locais à função (parâmetros) que são automaticamente inicializadas com os argumentos enviados.
  - De seguida são executadas as instruções da função.
- Quando esta **termina**, o controlo da execução do programa volta ao ponto onde tinha sido invocada a função.



# Parâmetros vs. Argumentos

- Estes dois termos são por vezes usados de forma indiscriminada. Existe, no entanto, uma diferença:
  - **Parâmetros** surgem na definição das funções.
  - **Argumentos** aparecem na invocação da funções.
- Pode dizer-se, informalmente, que o parâmetro corresponde ao tipo e o argumento a uma instância desse tipo.

# Parâmetros vs. Argumentos

```
#include <stdio.h>

long somar(int num1, int num2) {
    return num1 + num2;
}

main() {
    int a, b;
    long soma;
    scanf("%d", &a);
    scanf("%d", &b);
    soma = somar(a, b);
    printf("%ld", soma);
}
```

num1 e num2 são parâmetros

a e b são argumentos

# Procedimento

- A uma função que não devolve nada, e logo não tem tipo de devolução, chamamos **procedimento**.
  - Em C usamos o tipo especial **void** para indicar que a função não tem tipo de devolução.
- Exemplo:

```
void escrevePrecoComIva(float preco) {  
    printf("Preco final: %.2f", preco * IVA);  
}
```
- Nota: uma função sem tipo de devolução declarado é considerada por defeito como sendo do tipo `int`.

# Onde implementar uma função

- Uma função deve estar implementada antes da função que a irá invocar.
  - Podemos contudo invocar a função caso o protótipo da função esteja definida no topo do programa.

# Onde implementar uma função

```
#include <stdio.h>
main(){
    printf("%ld", somar(1, 5));
}
long somar(const int num1, const int num2){
    return num1 + num2;
}
```

- Problema:

- Quando tentamos compilar o programa deparamo-nos com uma mensagem de erro semelhante a este:

**“...previous implicit declaration of ‘somar’ was here...”**

- Solução:

- Dar conhecimento que a função existe:
- Deve estar definida anteriormente
- Deve ser dado conhecimento através da definição do seu protótipo

# Solução 1

- Definir a função antes da sua invocação

```
#include <stdio.h>
long somar(const int num1, const int num2){
    return num1 + num2;
}
main() {
    printf("%ld", somar(1, 5));
}
```

- Em programas mais complexos, esta solução poderá tornar-se complexa de gerir senão mesmo impossível.

## Solução 2

- Definir o protótipo da função

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

```
long somar(int num1, int num2);  
// ou long somar(int, int);
```

```
main(){  
    printf("%ld", somar(1, 5));  
}
```

```
long somar(int num1, int num2){  
    return num1 + num2;  
}
```

# Resumo

- As funções podem evitar a repetição de código ao longo do programa e reduzem a sua complexidade.
- Uma função deve efetuar apenas uma tarefa bem definida.
- Cada função, tal como uma variável, tem que ter um nome único, que servirá para que possa ser invocada em qualquer ponto do programa a que pertence.
- Ao terminar a sua execução uma função pode devolver um valor ao bloco de código que a invocou.



# Resumo

- O código de uma função deve ser o mais independente possível do restante programa.
  - Se possível, deve também ser o mais genérico possível para que a função possa ser reutilizada noutros programas.
- A uma função sem tipo de devolução (e que não devolve nada) chamamos procedimento.

# Exercícios

13. Faça um programa que leia um valor inteiro e crie uma função que imprima numa linha um número de asteriscos igual ao valor inserido pelo utilizador.
14. Faça um programa que permita ao utilizador introduzir um número real e um carácter e que invoca depois uma função que recebe esses 2 parâmetros e que converte o número de Euros para Dólares ou de Dólares para Euros, caso o carácter seja 'E' ou 'D', respetivamente.
  - A mesma função deve permitir fazer as duas conversões.
  - O resultado é impresso na função principal do programa .
  - Use constantes para os fatores de conversão.

# Variáveis locais vs Variáveis globais

- As variáveis locais são declaradas dentro do corpo de uma função. Para o programa, essas variáveis apenas são “visíveis” dentro da própria função, daí a designação locais.
- Desde que estejam declaradas em funções distintas, podem existir variáveis locais com o mesmo nome, não existindo qualquer relação entre elas.
- Depois de terminada uma função, são “eliminadas” todas as suas variáveis locais.

# Variáveis locais vs Variáveis globais

- Sempre que possível devem usar-se variáveis locais, evitando assim eventuais efeitos colaterais que ocorrem quando se usam variáveis globais.
- As variáveis globais são declaradas logo no início do programa antes de qualquer função, sendo “visíveis” em todo o programa.
  - Qualquer alteração aos seus valores repercute-se em todo o programa.
- Uma variável local com o mesmo nome de uma variável global tem primazia sobre esta última.
- À “região de influência” de uma variável chamamos **escopo** da variável.

# Variáveis locais vs Variáveis globais

```
#include <stdio.h>
int a = 5, b;
```

```
void rotina(int num1) {
    printf("%d", num1);
    printf("%d", a);
    num1 = 10;
}
```

```
main() {
    int num1, a;
    a = 8;
    num1 = 2;
    rotina(num1);
    printf("%d", num1);
    printf("%d", a);
}
```

# Passagem por Valor

- Nos exemplos anteriores ao invocar funções com argumentos:
  - Na prática é feita uma cópia do valor da variável origem para a função.
  - As alterações feitas “à cópia” não são visíveis na variável original.
  - Chamado de **Passagem por Valor**

# Passagem por referência

- Existe outro conceito, a **Passagem por Referência**
  - As alterações feitas ao valor da “variável” na função são visíveis na variável original.
  - Não é feita uma cópia do valor, ou seja, estamos a trabalhar sobre o endereço de memória da variável original.

# Passagem por valor e por referência

- Se o parâmetro contiver um \*, a função estará à espera do **endereço** de uma variável do tipo indicado.

```
// Quero o endereço de duas variáveis do tipo float  
double somar(float *num1, float *num2)
```

- Se o parâmetro não contiver um \*, a função estará à espera de um **valor** do tipo indicado.

```
// Quero dois valores do tipo float  
double somar(float num1, float num2)
```



# Passagem por valor e por referência

- Exemplo 1

```
double somar(float num1, float num2) {...}
```

...

```
float num1 = 1.0, num2 = 5;
```

```
const double resultado = somar(num1, num2);
```

Valor da variável



- Exemplo 2

```
double somar(float *num1, float *num2) {...}
```

...

```
float num1 = 1.0, num2 = 5;
```

```
const double resultado = somar(&num1, &num2);
```

Endereço da variável





# *Header files*

# Biblioteca standard

- Facilita e simplifica o desenvolvimento de aplicações.
- Fornece mecanismos/funcionalidades como definições de tipos, macros ou funções para tarefas como entrada e saída de dados ou manipulação de texto.
- Composta por vários *header files*, em que cada *header file* é responsável um conjunto de tarefas comuns ou relacionadas.
  - Exemplo: o *header file* `stdio.h` é responsável pela leitura (*input*) e saída (*output*) de informação.

# Header Files

- Exemplos: `stdlib.h`, `stdio.h` e `math.h`
- Permitem a separação de elementos de um programa facilitando a sua reutilização em outros programas.
- Contêm normalmente a definição de constantes, estruturas, funções, etc.
- São, tipicamente, armazenadas em ficheiros com a extensão `.h`
- Incluem-se num programa através da diretiva `#include`, sendo o pré-processador responsável por este processo.
- Exemplo: `#include <math.h>`

# *Header files*

## Exemplos

- **stdio.h** - funções de input e output
- **stdlib.h** - funções para de conversão de dados, alocação de memória, entre outros
- **math.h** - funções matemáticas comuns
- **string.h** - funções de manipulação de strings
- **sdtbool.h** - define o tipo de dados booleano

# Criação de bibliotecas

- O programador pode criar as suas próprias bibliotecas.
- Incluem-se num programa também através da diretiva `#include`, sendo o pré-processador responsável por este processo.
- Exemplo: `#include "nome_ficheiro.h"`

# Programação modular

- No desenvolvimento de projetos com alguma dimensão, é usual dividir o código por diversos ficheiros a que se chama **módulos**.
  - Um módulo é uma coleção de funções que realizam tarefas relacionadas.
- Pode-se dividir um módulo em duas partes:
  - Parte pública do módulo:
    - Definição de estruturas de dados e funções que devem ser acedidas fora do módulo.
    - Estas definições estão no *header file* por convenção.
    - Os *header files* têm extensão .h.
  - Parte privada do módulo:
    - Tudo o que é interno ao módulo (não visível pelo mundo exterior).
    - Ficheiro com extensão .c.

# Programação modular

- A ideia é dividir um problema em secções diferentes e escrever um *header file* para cada um dos ficheiros C com as definições de todas as funções e variáveis globais desse ficheiro que devem ser usadas por outros módulos.
- Vantagens:
  - Estrutura do programa fica mais clara ao agrupar funções e variáveis relacionadas num mesmo ficheiro.
  - Possibilidade de compilar cada um dos módulos separadamente, poupando tempo.
  - A reutilização das funções é facilitada.

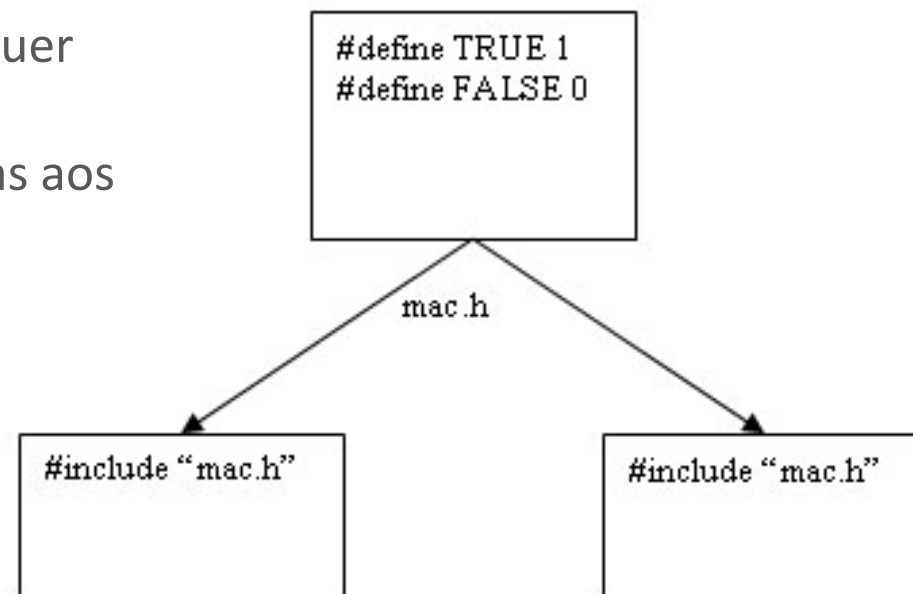


# Divisão de um programa em módulos

- Quando dividimos o código de um programa em diversos módulos, levantam-se algumas questões:
  - Como é que uma função de um módulo acede a uma outra de outro módulo?
  - Como é que uma função de um módulo acede a uma variável de outro módulo?
  - Como é que uma macro é partilhada por diversos módulos?

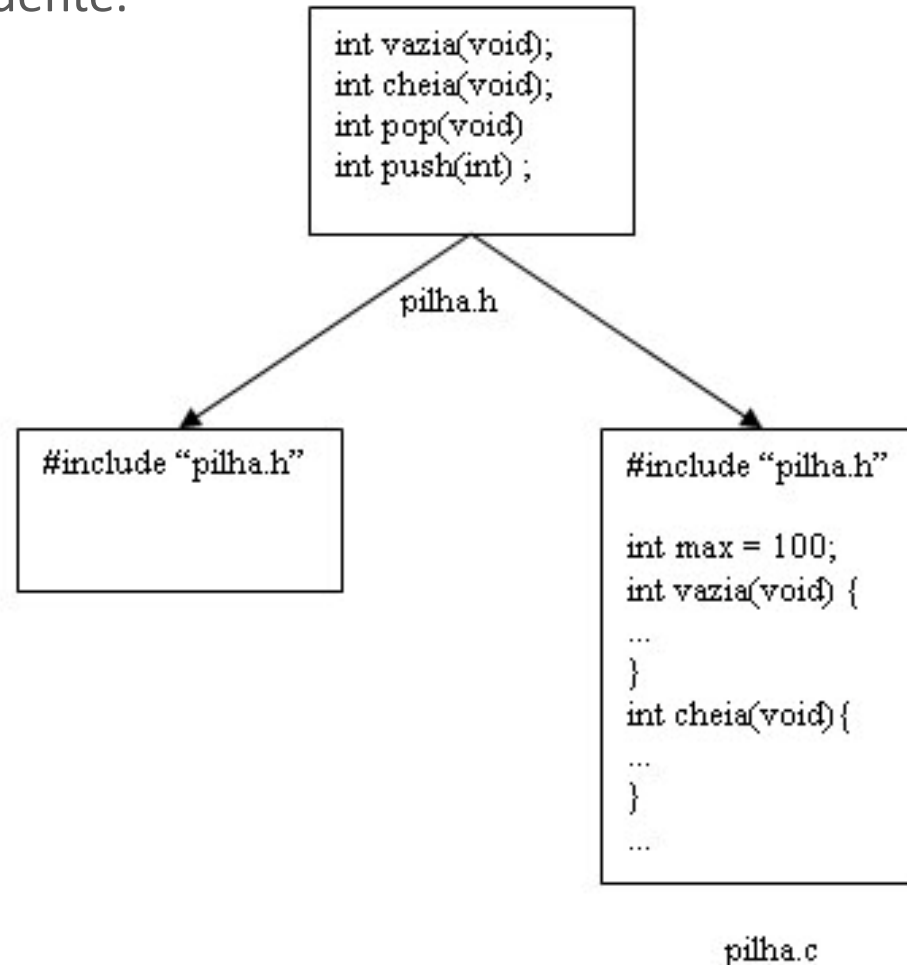
## Divisão de um programa em módulos

- A resposta está na diretiva `include` do pré-processador do C.
- A imagem seguinte ilustra a inclusão do ficheiro (*header file*) `mac.h` noutros dois:
- Aqui já se vê uma vantagem do uso de módulos, as definições comuns aos dois ficheiros estão armazenadas no *header file* `mac.h`, quaisquer alterações neste são automaticamente propagadas aos ficheiros que o incluem.



# Divisão de um programa em módulos

- No caso de um módulo necessitar de aceder a funções definidas noutra, os protótipos respetivos devem estar no *header file* correspondente.



# Extern

- Partilhar variáveis é semelhante a partilhar funções.
  - As funções são definidas num ficheiro e um protótipo é partilhado por outros módulos.
    - As variáveis também são definidas, por exemplo:  
`int i;`
    - é partilhada por outros módulos com a declaração:  
`extern int i;`
- O **extern** indica que uma variável está definida fora do módulo corrente (não a define!).
- Pode-se distinguir os seguintes casos:
  - `extern` variável: definida noutro ficheiro.
  - "nenhum" variável: pública.
  - `static` variável: local ao ficheiro (privada).

# Exemplo

```
// main.c
```

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

char *OutraString = "Ola a todos!";

main() {
    printf("\n Começo....\n");
    escreveString(A_STRING);
    printf("\n Fim....\n");
}
```

```
// escreve_string.h
```

```
#define A_STRING "Olá mundo"

void escreveString(char *);
```

```
// escreve_string.c
```

```
extern char *OutraString;

void escreveString(char *estaString) {
    printf("%s", estaString);
    printf("%s", OutraString);
}
```

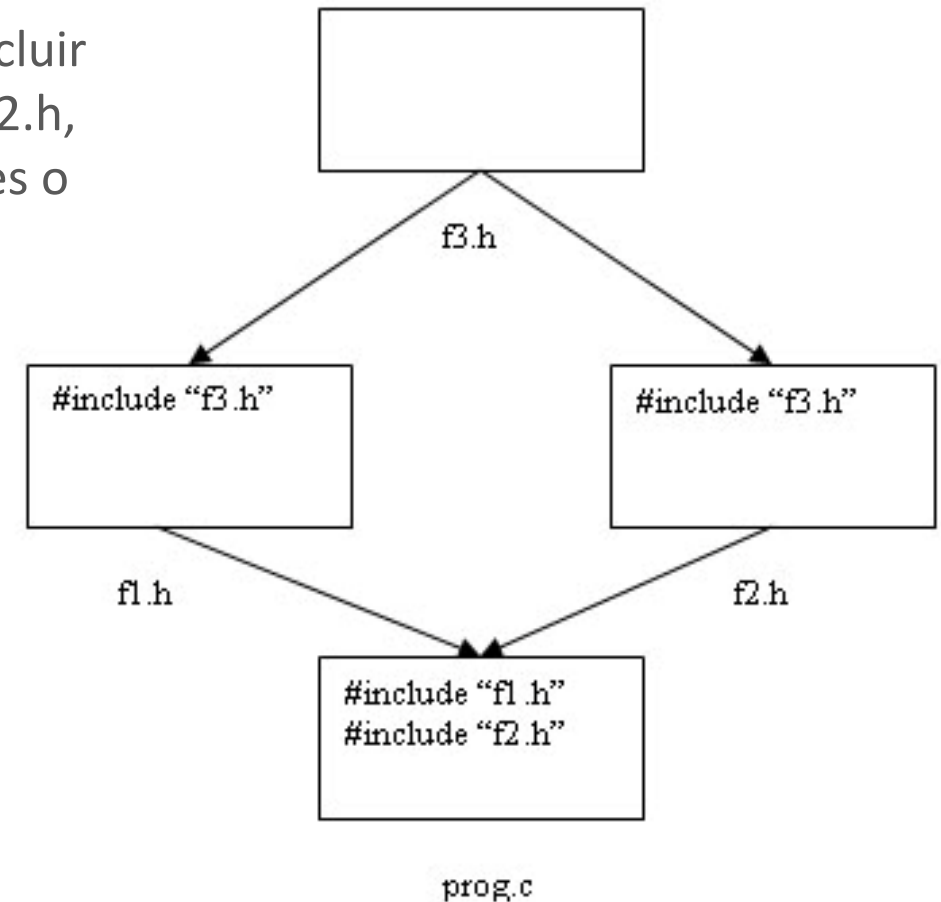
# Compilar

- A compilação dos diferentes ficheiros num só executável faz-se da seguinte forma:

```
$ gcc -o mod main.c escreve_string.c
```

# Proteção de *header files*

- A inclusão de um mesmo *header file* mais do que uma vez origina problemas.
- Se f1.h incluir f3.h e f2.h incluir f3.h e prog.c incluir f1.h e f2.h, f3.h será incluído duas vezes o que origina um erro de compilação.



## Proteção de *header files*

- Para resolver este problema utiliza-se o pré-processador. Por exemplo, para proteger o *header file* do exemplo anterior escreve-se o seguinte:

```
#ifndef F3_H
#define F3_H

#define A_STRING "Olá mundo"
void escreveString(char *);

#endif
```

- Este *header file* passa a ser identificado por F3\_H e só será incluído uma vez.



# Documentação de bibliotecas

- Não existe uma norma para a documentação de bibliotecas. Sugere-se a inclusão dos seguintes elementos (quando aplicável):
  - Nome do ficheiro
  - Copyright
  - Versão
  - Autor
  - Contacto
  - Data de Criação
  - Resumo
- É expectável que a documentação feita a um programa/biblioteca seja extraída por uma ferramenta específica.

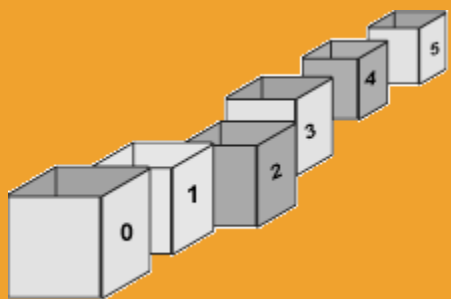
# Documentação de funções

- A documentação deve contar uma história.
- A documentação inclui:
  - Comentários ao funcionamento da função.
  - Comentários aos parâmetros da função.
  - Comentários ao que é devolvido pela função.

# Documentação de funções

- Sugere-se a utilização de cabeçalhos estruturados. Exemplo:

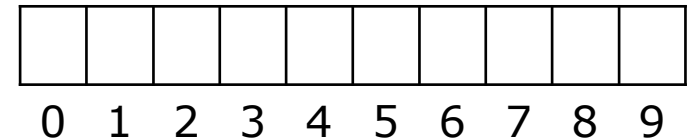
```
* Calcula a velocidade média constante dada uma distância e
* o tempo decorrido durante um percurso
*
* @param distancia A distância (em km) do percurso.
* @param tempo O tempo (em horas) que demorou a percorrer a distância.
* @return A velocidade média do percurso.
*
* @pre Tempo e Distância devem ser maiores que zero.
*
* Exemplo:
* float velMedia;
* velMedia = calcVelocidadeMediaConstante (200.5, 4.1);
*/
float calcVelocidadeMediaConstante(float distancia, float tempo) {
    // código da função
}/*
```



# Vetores

# Variável vs Vetor

- Podemos imaginar uma variável como sendo uma gaveta onde guardamos algo.
- Um vetor pode ser visto como uma fila de gavetas.
  - Um vetor, ou **array**, é um conjunto de dados de um mesmo tipo (homogêneos), que são armazenados de forma contígua, e a que é possível aceder individualmente através de um índice.
  - Exemplos de utilização para armazenamento:
    - Temperaturas médias de cada mês de um ano;
    - Comissões mensais de um vendedor;
    - Notas de um aluno a uma disciplina;
    - Movimentos de uma conta bancária.



# Declaração

```
tipo nome_array[nº_elementos];
```

- **tipo**: tipo de dados dos elementos do array (ex.: int, float ou char);
- **nome\_array** : nome a dar ao array seguindo as mesmas regras de atribuição de nomes a variáveis;
- **nº\_elementos**: número de elementos que o array irá conter (tem que ser um valor inteiro e positivo).
- Exemplos:
  - `float comissao[12];`
  - `unsigned short notaFinal[5];`
  - `int vetor[10];`

# Manipulação

`nome_array[índice]`

- **nome\_array**: nome que foi dado ao array;
- **índice**: posição do elemento no array;
  - **IMPORTANTE**: Os índices de um array com  $n$  elementos variam entre 0 e  $n-1$ . O  $n$ -ésimo elemento está na posição  $n-1$ .

- Exemplos:

- `vetor[4] = 2;`
- `vetor[7] = 5 - 1;`
- `vetor[0] = vetor[4] + vetor[7];`
- `vetor[ vetor[4] ] = 9;`
- `vetor[10] = 1;` ←

6		9		2			4		
0	1	2	3	4	5	6	7	8	9

IMPOSSÍVEL

# Manipulação

- A forma mais comum de percorrer o conteúdo de um array é usando ciclos.

- Exemplo:

```
const int ARRAY_TAM = 10;  
...  
int i, vetor[ARRAY_TAM];  
for (i = 0; i < ARRAY_TAM; ++i) {  
    vetor[i] = i + 1;  
}
```

1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9



## Exemplo de leitura

```
const int TAM_ARRAY = 10;  
...  
int i, vetor[TAM_ARRAY];  
for(i = 0; i < TAM_ARRAY; ++i) {  
    scanf("%d", &vetor[i]);  
}
```

## Particularidades

- Um array pode ser inicializado, na sua declaração, com o conjunto de valores que irá conter:  
`int v[3] = {2, 4, 6};`
- Se o número de inicializações for menor que o nº de elementos do array, os elementos em falta serão inicializados com o valor 0:  
`int w[5]={1,3,5};`  
`// equivalente a`  
`int w[5]={1,3,5,0,0};`

# Particularidades

- Os elementos de um array são armazenados em **posições contíguas da memória**.
- Quer durante a compilação, quer durante a execução, não é verificado se os índices usados estão de acordo com a dimensão declarada para o array.
  - É por isso possível declarar um array com, por exemplo, 3 elementos, e tentar depois aceder ao índice 5, o que levará a que ocorram problemas de acesso à memória.

# Exercícios

15. Faça um programa que preencha cada posição de um array (de 20 elementos inteiros) com o valor do respectivo índice. Deverá depois imprimir todo o conteúdo do array .
16. Faça um programa que preencha cada posição de um array (de 10 elementos reais) com 10 números introduzidos pelo utilizador. Deverá depois imprimir todo o conteúdo do array e o somatório de todos os seus elementos.
17. Faça um programa que preencha cada posição de um array (de 20 elementos inteiros) com os números de 5 até 25. Deverá depois imprimir todo o conteúdo do array.
  - Vê algum problema com o enunciado deste exercício?

## Passagem por valor: Arrays

```
void demonstrarPassagemValor(int n1){  
    n1 *= 2;  
    printf("Numero funcao: %d\n", n1);  
}  
  
int main(){  
    int n1 = 5;  
    printf("Numero ant. invocacao: %d\n", n1)  
    demonstrarPassagemValor(n1);  
    printf("Numero post. invocacao: %d\n", n1);  
}
```

Numero ant. invocação: 5

Numero funcao: 10

Numero post. invocação: 5

## Passagem por valor: Arrays

- Utilizado quando não queremos que o valor passado ao parâmetro seja alterado
- São criadas cópias das variáveis passadas
- As alterações feitas na função são visíveis apenas dentro da função
- Valores no *caller* (quem invocou a função) mantêm-se inalterados.

# Passagem por referência: Arrays

```
void calcularDobroArray(int dados[], int arrayTam) {  
    int i = 0;  
    for (; i < arrayTam; ++i) { dados[i] *= 2; }  
}  
  
int main() {  
    int i, nums[] = {1, 2, 3, 4, 5};  
    int arrayTam = sizeof(nums) / sizeof(nums[0]);  
    calcularDobroArray(nums, arrayTam);  
    for (i = 0; i < arrayTam; ++i){  
        printf("Indice: %d; Valor: %d\n", i, nums[i]);  
    }  
}
```

```
Indice: 0; Valor: 2  
Indice: 1; Valor: 4  
Indice: 2; Valor: 6  
Indice: 3; Valor: 8  
Indice: 4; Valor: 10
```

# Passagem por referência: Arrays

- No exemplo anterior, a função `calcularDobroArray` espera como argumentos:
  - `int dados[]`
    - O endereço de memória de um array de inteiros (o endereço do seu primeiro elemento).
      - Quando utilizamos o nome da variável que representa o array, estamos na realidade a “utilizar” o endereço de memória do primeiro elemento do array: `nums` é equivalente a `&nums[0]`
  - `int arrayTam` – Um valor inteiro



# Passagem por referência: Arrays

- No primeiro argumento (`nums`) as alterações feitas na função serão visíveis no seu *caller* (Passagem por Referência).
  - Não é feita nenhuma cópia da variável, mas sim enviada o local onde a mesma está armazenada
- No segundo argumento (`arrayTam`), é feita uma cópia da variável original, logo as alterações feitas na função são apenas visíveis nessa mesma função (Passagem por Valor).

# Passagem por referência: Arrays

- Representação simples da alocação de variáveis na função `main` (considere que cada célula representa um inteiro, e o endereço é sequencial):

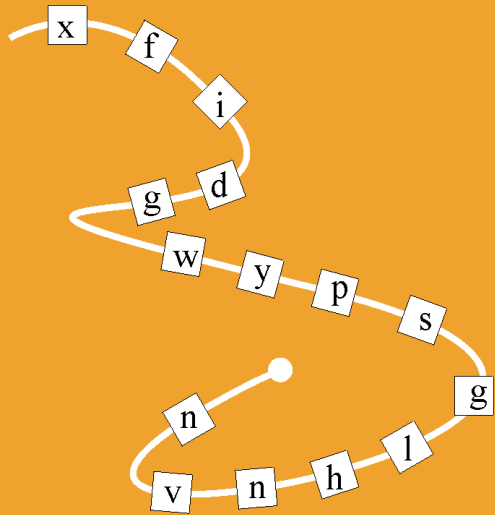
	nums[]						arrayTam		i
Endereço	0x28ac01	0x28ac02	0x28ac03	0x28ac04	0x28ac05		0x28ac07		0x28ac09
Valores:	1	2	3	4	5		5		?

- Quando invocamos a função `calcularDobroArray(nums, arrayTam)`, pode-se dizer que passamos a seguinte informação: `calcularDobroArray(0x28ac01, 5);`, onde `0x28ac01` é o endereço de memória do primeiro elemento do array.
- A representação da alocação de variáveis na função `calcularDobroArray` pode ser:

	dados[]						arrayTam		i
Endereço	0x28ac01	0x28ac02	0x28ac03	0x28ac04	0x28ac05		0x28ac10		0x28ac11
Valores:	1	2	3	4	5		5		?

# Passagem por referência: Arrays

- Utilizado quando pretendemos que as alterações feitas na função sejam visíveis no seu *caller*
- Neste cenário é necessário passar a referência da variável e a função trabalha sobre o endereço de memória dessa mesma variável.



# Strings

# Strings

- Exemplos de caracteres:
  - 'N'
  - '\_'
  - '1'
  - '\n'
- Exemplos de strings:
  - "Linguagens de Programação"
  - "N"
- Em C não existe o conceito de string. Para implementarmos uma string usamos uma cadeia de caracteres - um conjunto de caracteres armazenados num vetor.

# Strings vs Caracteres

- 'N' → 1 byte

- "N" → 2 bytes → 

N	\0
---	----

  
0    1

- O carácter especial `\0` indica o fim da string e deve ser tido em conta ao definir o tamanho do vetor.

# Declaração

- Declaração de uma string:
  - `char nome_string[nº_de_caracteres + 1];`
- Ex: declaração de string com 9 caracteres úteis:
  - `char palavra[10];`

0	1	2	3	4	5	6	7	8	9

- Inicialização automática de uma string:
  - `char palavra[10] = "ESTGF";` ou
  - `const char palavra[] = "ESTGF";`

E	S	T	G	F	\0				
0	1	2	3	4	5	6	7	8	9

# Output

- `printf("Texto a escrever");`
- `printf("Nome do aluno: %s", nome_aluno);`
- `puts("Texto a escrever");`
- `puts(nome_aluno);`
- A função `puts` escreve a string e faz automaticamente a mudança de linha.
  - `printf("Texto a escrever\n");`
  - `// é equivalente a`
  - `puts("Texto a escrever");`



## Outras operações

- Para utilizar algumas das seguintes funções é necessário incluir o *Header File* **string.h**.
- **strlen(p)** - devolve o número de caracteres existentes numa string p (sem contar o \0).
- **strcpy(destino, origem)** - copia a string origem para a string destino.
- **strcat(destino, origem)** - concatenação de duas strings, ou seja, coloca a origem imediatamente a seguir ao final de destino.
- **strcmp(p, q)** - compara alfabeticamente a string p com a string q, devolvendo um valor inteiro  $< 0$  se  $p < q$ ,  $0$  se  $p = q$  e  $> 0$  se  $p > q$

# Outras operações

- Para utilizar as funções que devolvem inteiros pode guardar o resultado obtido numa variável auxiliar do tipo inteiro.
- Exemplos:
  - `int comp = strcmp(p, q);`
  - `int qtdcar = strlen(p);`

# sprintf

- Envia output formatado para uma string “apontada” por **str**.  
`int sprintf(char *str, const char *format, ...)`
- Exemplo:  

```
char full_name[25];  
sprintf(full_name, "%s %s", first_name, last_name);  
puts(full_name);
```

# Exemplo

```
int main() {  
    // Conta os caracteres de uma string  
    int i = 0;  
    char palavra[] = "ESTGF";  
    while (palavra[i] != '\0'){ ++i; }  
}
```

# Input

- `scanf("%s", palavra);`
  - A leitura de caracteres termina quando é encontrado um espaço, TAB ou ENTER. Isto é, só é lida uma palavra.
- Porque razão não colocarmos o `&` na leitura de uma string???
  - A variável `palavra`, representa o array, referencia o endereço de memória para o primeiro elemento do array, ou seja `palavra` é equivalente a `&palavra[0]`
    - `scanf("%s", &palavra[0]);`

# gets

- `gets(frase);`
  - Não está limitada à leitura de uma só palavra.

# Input

- Caso o utilizador insira mais caracteres do que aqueles que a string pode armazenar, o programa pode dar erros de memória e ser terminado. Mesmo que não ocorra de imediato um erro, este estará latente.
- Tal como com os caracteres, antes de ler uma string deve limpar o buffer de entrada de dados.

# Problema com gets

- É possível ler uma string do utilizador através da função gets
- Problema:
  - Função gets assume que o buffer que passamos tem tamanho suficiente para guardar a informação introduzida pelo utilizador.
    - Exemplo:

```
char palavra[4 + 1];  
gets(palavra);
```
  - O que acontece caso o utilizador introduza a palavra “Programar” ?
    - O acesso, ou tentativa de acesso, a zonas de memória indevidas, podendo levar a um erro imediato, corrupção de informação noutras zonas de memória levando a outras anomalias, entre outras situações problemáticas.



# fgets

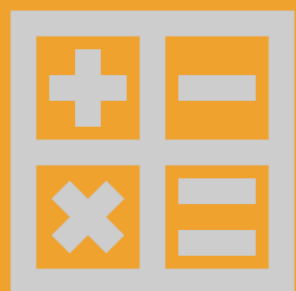
```
char * fgets (char * str, int num, FILE * stream);
```

- **str**: string para onde a sequência de caracteres lida será copiada.
- **num**: número máximo de caracteres a serem copiados para str (incluindo o caractere nulo final).
- **stream**: ponteiro para um objeto FILE que identifica um fluxo de entrada.
  - **stdin** pode ser usado como argumento para ler a partir da entrada padrão.
- Exemplo:

```
char user_input[10];  
fgets(user_input, 10, stdin);  
printf("%s", user_input);
```

# fgets

```
int get_user_input(char *str, int size) {  
    if (fgets(str, size, stdin) != NULL) {  
        int temp = strlen(str) - 1;  
        if (str[temp] == '\n') { str[temp] = '\0'; }  
        else { clean_buffer(); }  
        return 1;  
    }  
    return 0;  
}
```



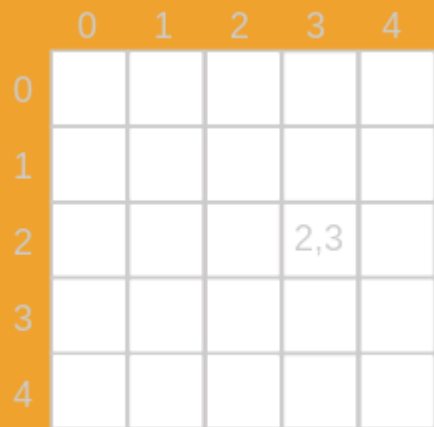
# Math

# Math

- **double acos(double x)**
  - retorna o arco cosseno de x em radianos.
- **double asin(double x)**
  - retorna o arco seno de x em radianos.
- **double atan(double x)**
  - retorna o arco tangente de x em radianos.
- **double cos(double x)**
  - retorna o cosseno de x.
- **double sin(double x)**
  - retorna o seno de x.

# Math

- **double pow(double x, double y)**
  - retorna a potência de x elevado a y.
- **double sqrt(double x)**
  - retorna a raiz quadrada de x.
- **double ceil(double x)**
  - retorna o inteiro mais pequeno que seja maior ou igual a x.
- **double fabs(double x)**
  - retorna o valor absoluto de x.
- **double floor(double x)**
  - retorna o maior inteiro que seja inferior ou igual a x.
- **double fmod(double x, double y)**
  - retorna o resto da divisão de x por y.



	0	1	2	3	4
0					
1					
2				2,3	
3					
4					

# Matrizes

# Vetor vs Matriz

- Se um vetor pode ser visto como uma fila de gavetas ...
- ... então uma matriz pode ser vista como várias filas de gavetas.

# Matriz

- Uma matriz não é mais do que um vetor bidimensional.
- Podemos pensar numa matriz como uma tabela.
- Uma matriz é um conjunto de dados de um mesmo tipo (homogêneos), distribuídos por linhas e colunas, e a que é possível aceder individualmente através de um par de índices.

					0
					1
					2
					3
					4
0	1	2	3	4	



# Declaração

```
tipo nome_matriz[nº_linhas][nº_colunas];
```

- **tipo**: tipo de dados dos elementos da matriz (ex.: int, float ou char);
- **nome\_matriz**: nome a dar à matriz, seguindo as mesmas regras de atribuição de nomes a variáveis;
- **[nº\_linhas][nº\_colunas]**: número de linhas e colunas da matriz (têm que ser valores inteiros e positivos).

- Exemplos:

- `char xadrez[8][8];`
- `int galo[3][3];`
- `float tabela[4][8];`
- `int matriz[5][5];`

					0
					1
					2
					3
					4
0	1	2	3	4	

# Manipulação

`nome_matriz[índice_linha][índice_coluna]`

- **nome\_matriz**: nome que foi dado à matriz;
- **[índice\_linha][índice\_coluna]**: posição do elemento na matriz;
- **IMPORTANTE**: Os índices linha e coluna de uma matriz variam entre 0 e  $n^{\circ}_{\text{linhas}}-1$  e 0 e  $n^{\circ}_{\text{colunas}}-1$ , respetivamente.

# Manipulação

- Exemplos:
  - `matriz[4][3] = 2;`
  - `matriz[1][2] = 5;`
  - `matriz[3][0] = matriz[4][3] + matriz[1][2];`
  - `matriz[matriz[4][3]][4] = 6;`
  - `matriz[2][5] = 9;` ← IMPOSSÍVEL

					0
		5			1
				6	2
7					3
			2		4
0	1	2	3	4	

# Manipulação

- A forma mais comum de percorrer o conteúdo de uma matriz é usando ciclos.
- Exemplo:

```
int x, y, matriz[5][5];  
for (x = 0 ; x < 5 ; ++x) {  
    for (y = 0 ; y < 5 ; ++y) {  
        matriz[x][y] = x + y;  
    }  
}
```

0	1	2	3	4	0
1	2	3	4	5	1
2	3	4	5	6	2
3	4	5	6	7	3
4	5	6	7	8	4
0	1	2	3	4	

# Leitura

- A forma mais comum de percorrer o conteúdo de uma matriz é usando ciclos. Exemplo:

```
const int TAM_COLUNAS 5;
const int TAM_LINHAS 5;
(...)
int i, j, matriz[TAM_LINHAS][TAM_COLUNAS];
for (i = 0; i < TAM_LINHAS; ++i) {
    for (j = 0; j < TAM_COLUNAS; ++j) {
        scanf("%d", &matriz[i][j]);
    }
}
```

# Particularidades

- Os elementos de uma matriz são armazenados em posições contíguas da memória.
- Quer durante a compilação, quer durante a execução, não é verificado se os índices usados estão de acordo com a dimensão declarada para a matriz.
  - É por isso possível declarar uma matriz com, por exemplo, a dimensão 3x3, e tentar depois aceder ao índice [5][1], o que levará a que ocorram problemas de acesso à memória.

# Matrizes como parâmetro de funções

- Necessário especificar o número de colunas.

```
void escreverMatriz(int mat[][TAMCOLS]){  
    int i = 0, j;  
    for (; i < TAMLINHAS; ++i)  
        for (j = 0; j < TAMCOLS; ++j)  
            printf("%d", mat[i][j]);  
}
```

# Exercícios

18. Faça um programa que preencha cada posição de uma matriz (3x3 elementos inteiros) com um valor introduzido pelo utilizador. Deverá depois imprimir todo o conteúdo da matriz da forma exemplificada:

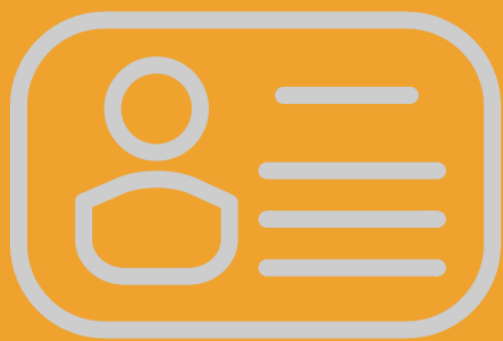
3 8 1

7 4 2

5 1 9

19. Faça um programa que preencha cada posição de uma matriz (2x3 elementos reais) com um valor introduzido pelo utilizador. Deverá depois indicar qual o maior e qual o menor dos valores guardados na matriz.





# Tipos estruturados

Enumerações e Registos

# Enumeração

- Uma enumeração é um conjunto de valores inteiros representados por identificadores, que têm que ser únicos.
- Esses valores são constantes simbólicas, tipicamente inteiros, cujo valor é automaticamente definido. Os valores iniciam-se em 0 e sendo depois incrementados em 1 unidade.

- Exemplos:

- `enum genero {M, F};`
- `enum estadoCivil {SOLTEIRO, CASADO, VIUVO, DIVORCIADO};`
- `enum combustivel {GASOLINA, DIESEL, OUTRO};`

A constante M fica definida com o valor zero e F com o valor 1.



# Enumeração

- Os valores da enumeração podem ser definidos explicitamente com o '='.

- Exemplo:

```
enum mes { JAN = 1, FEV, MAR, ABR, MAI, JUN, JUL,  
AGO, SET, OUT, NOV, DEZ};
```

- Cria uma nova enumeração mes em que os identificadores (JAN, FEV, etc.) correspondem aos inteiros de 1 a 12.

# Enumeração

```
#define MENSAGEM_ESTADO_CIVIL "O seu estado civil: %s"
enum EstadoCivil {SOLTEIRO, CASADO, VIUVO, DIVORCIADO};

int main() {
    enum EstadoCivil estado; //variável do tipo da enum EstadoCivil

    printf("Indique o seu estado civil [0 a 3]: ");
    scanf("%u", &estado);

    switch (estado){
        case SOLTEIRO:
            printf(MENSAGEM_ESTADO_CIVIL, "Solteiro"); break;
        case CASADO:
            printf(MENSAGEM_ESTADO_CIVIL, "Casado"); break;
        case VIUVO:
            printf(MENSAGEM_ESTADO_CIVIL, "Viuvo"); break;
        case DIVORCIADO:
            printf(MENSAGEM_ESTADO_CIVIL, "Divorciado"); break;
    }

    return 0;
}
```

# Registos

- Até agora, armazenamos todos os dados em variáveis de tipo **primitivo** (ex.: int, float ou char) e **conjuntos de valores homogêneos** (arrays e matrizes).
- Exemplos de dados sobre um automóvel:
  - `char matricula[9];`
  - `unsigned short ano;`
  - `enum combustivel comb[15];`
  - `unsigned short numero_velocidades`
  - `float cilindrada;`
  - `unsigned short potencia;`
  - `float ultimos_abastecimentos[ABASTECIMENTOS_TAM];`

# Registos (Estruturas)

- Os registos, também designados por tuplos ou estruturas, permitem armazenar numa única entidade elementos de diferentes tipos. Um registo não é mais do que um conjunto de uma ou mais variáveis que podem ser de diferentes tipos.

Matrícula: 17-50-BP					Ano: 2000				
Combustível: Diesel					Cilindrada: 1.9 L				
Nº velocidades: 5					Potência: 80 cv				
Últimos abastecimentos (litros)									
40,2	27,5	12,0	38,1	21,7					

# Definição

```
struct nome_registro {  
    tipo_do_elemento nome_do_elemento1;  
    tipo_do_elemento nome_do_elemento2;  
    ...  
    tipo_do_elemento nome_do_elementoN;  
};
```

- **nome\_registro**: nome a dar ao registro.
- **tipo\_do\_elemento**: tipo de dados do elemento (ex.: int, float, char, uma enumeração, ou um outro registro).
- **nome\_do\_elemento**: nome do elemento (variável), também conhecido por campo ou membro do registro.

# Exemplo

```
struct automovel {  
    char matricula[9];  
    int ano;  
    enum combustivel comb;  
    int numeroVelocidades;  
    float cilindrada;  
    int potencia;  
    float ultimosAbastecimentos[ABASTECIMENTOS_TAM];  
};
```

Matrícula: 17-50-BP					Ano: 2000				
Combustível: Diesel					Cilindrada: 1.9 L				
Nº velocidades: 5					Potência: 80 cv				
Últimos abastecimentos (litros)									
40,2	27,5	12,0	38,1	21,7					



# Definição

- À definição de uma estrutura não corresponde a declaração de uma nova variável, mas sim a definição de um **novo tipo de dados** que pode ser usado para declarar variáveis.
- Depois de definirmos uma estrutura podemos criar variáveis desse novo tipo de dados.
- Exemplos:
  - `struct automovel meu_carro, carro_empresa;`
  - `struct automovel frota[25];`

# Inicialização

- Uma variável de um tipo de estrutura pode ser inicializada quando é declarada.

```
struct nome_estr nome_var= {valor1, valor2, ..., valorN};
```

- Exemplo:

```
struct automovel meu_carro = {"17-50-BP\0", 2000,  
    DIESEL, 5, 1.9, 80,  
    {40.2, 27.5, 12.0, 38.1, 21.7} };
```

# Acesso a elementos e atribuição

- Operadores sobre estruturas:
  - Acesso a um elemento: . (ponto)
  - Atribuição de valor a um elemento: =
- Exemplos:
  - `meu_carro.ano = 2012;`
  - `frota[18].cilindrada = 1.2;`
  - `if (carro_empresa.potencia > 100) ...`
  - `printf("%f", carro_empresa.ultimos_abastecimentos[4]);`
  - `frota[2].ano = frota[3].ano;`

# Copiar

- Podemos copiar todo o conteúdo dos elementos de um registo para outro registo como se fosse uma qualquer variável usando a atribuição.
- Exemplo:  
`meu_carro = carro_empresa`

# Comparar

- Não é possível fazer comparações diretas entre registos. Apenas se podem comparar os seus elementos.
- Errado:

```
if (meu_carro > carro_empresa)
    puts("O meu carro é mais potente.");
```
- Correto:

```
if (meu_carro.potencia > carro_empresa.potencia)
    puts("O meu carro é mais potente.");
```

# Registos dentro de registos

- Exemplo:

```
struct automovel {  
    char matricula[9];  
    int ano;  
    enum combustivel comb;  
    int numeroVelocidades;  
    float cilindrada;  
    int potencia;  
    struct abastecimento ultimosAbastecimentos[ABASTECIMENTOS_TAM];  
};
```

```
struct abastecimento {  
    float quantidade;  
    float preco;  
    int kms;  
};
```

- Exemplo de acesso a um elemento de um registo dentro de outro:

```
meuCarro.ultimosAbastecimentos[3].quantidade = 38.1;
```

# Exemplo

```
int ABASTECIMENTOS_TAM = 10;
typedef enum combustivel { GASOLINA, DIESEL, OUTRO } Combustivel;
typedef struct abastecimento {
    float quantidade, preco;
    int kms; } Abastecimento;
typedef struct auto {
    char matricula[9];
    int ano, potencia, numeroVelocidades;
    float cilindrada;
    Combustivel comb;
    Abastecimento ultimosAbastecimentos[ABASTECIMENTOS_TAM];
} Automovel;
```

# Tipos de estruturas

```
struct abastecimento {  
    float quantidade, preco;  
    unsigned long kms;
```

```
};
```

```
typedef struct abastecimento Abastecimento;
```

```
// declaração de um array do tipo Abastecimento
```

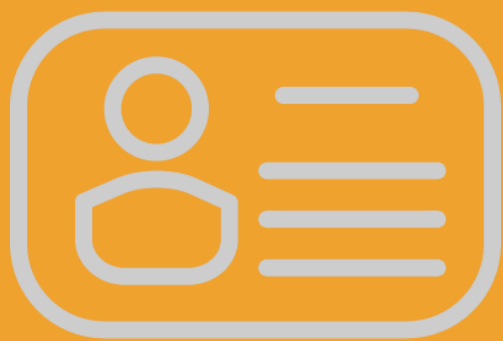
```
Abastecimento ultimos_abastecimentos[10];
```

```
typedef struct {  
    float quantidade, preco;  
    unsigned long kms;  
} Abastecimento;
```



# Declaração e utilização

- Função que lê um “Automovel” do utilizador  
`Automovel lerAutomovel();`
- Função que escreve o conteúdo de um “Automovel” no ecrã  
`void escreverAutomovel(const Automovel veic);`
- Declaração e utilização;  
`Automovel MeuCarro;  
MeuCarro = lerAutomovel();  
escreverAutomovel(MeuCarro);`



# Gestão de registos

Solução genérica

# Passos iniciais

- Declarar constantes
- Definir estrutura de dados
- Declarar vetor de registros
- Declarar contador para o vetor de registros

# Inserir

- Verificar se vetor está cheio
- Pedir identificador do registo
- Verificar se já existe esse identificador
- Inserir restantes dados
- Incrementar contador
- Ordenar (opcional)

# Remover

- Verificar se vetor está vazio
- Pedir identificador do registo
- Verificar se existe esse identificador
- Remover
- Decrementar contador

# Alterar

- Verificar se vetor está vazio
- Pedir identificador do registo
- Verificar se existe esse identificador
- Alterar dados (um ou mais elementos do registo)
- Verificar se existe novo identificador (caso este seja alterado)
- Ordenar (opcional)

# Listar

- Verificar se vetor está vazio
- Listar:
  - Tudo (todos os identificadores ou todos os elementos dos registros)Ou
  - Por pesquisa (de identificador ou outro elemento do registro)



Apontadores



# Apontadores

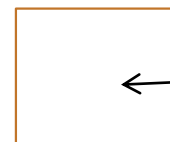
- A matéria de apontadores é dos tópicos mais delicados e importantes de programação.
- Normalmente, os alunos têm alguma dificuldade em dominar por completo esta matéria.
- O uso de apontadores requer bastante cuidado.
  - É fácil deixar escapar erros, pelo que devem testar convenientemente os programas.
- A sintaxe do C pode assustar um bocado, mas o conceito de apontador é simples.

# Representação de variáveis na memória

- Quando falamos no conceito de variável, dissemos que uma variável era como uma caixa ou gaveta em cujo interior podia ser guardado um valor.
- Por exemplo, `int n;` corresponde à figura:

Este é o nome pelo qual  
identificamos a “caixa”

→ **n**



Aqui dentro apenas  
podemos ter um  
valor do tipo **int**

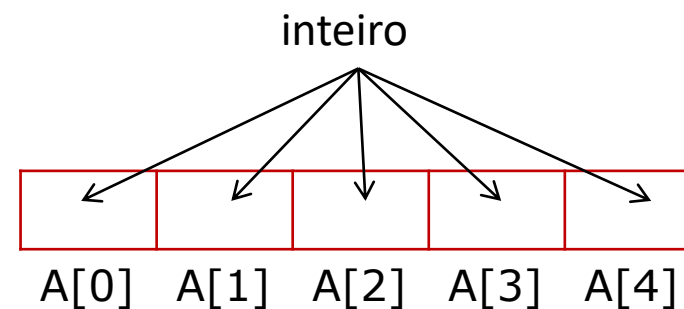
- Quando fazemos `n = 5;` ficamos com:

**n**

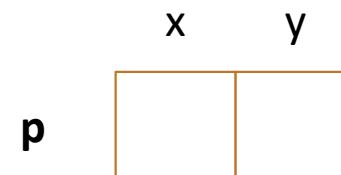


# Representação de variáveis na memória

- A representação de arrays e estruturas pode ser feita da mesma forma.
- `int a[5];` corresponde a cinco caixas contíguas, cada qual do tipo `int`, indexadas pelos índices 0, 1, 2, 3, 4.

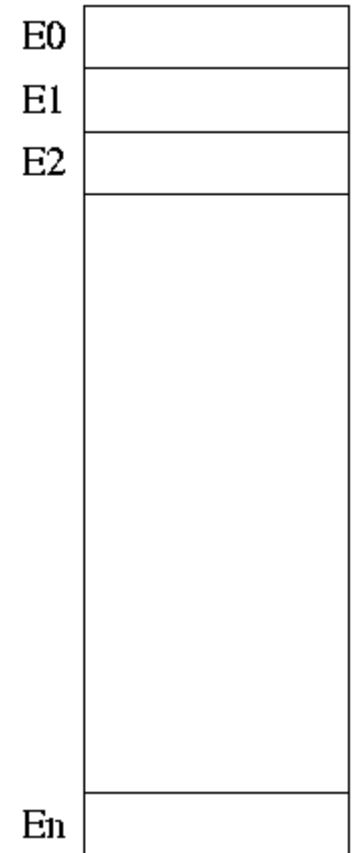


```
struct ponto { float x, y; };  
struct ponto p;
```



# Representação de variáveis na memória de um computador

- É útil pensar nas variáveis com se fossem caixas mas é óbvio que estas caixas não existem.
- Aquilo que existe na memória do computador (RAM) é algo que pode ser visto como um enorme array onde todos os dados são armazenados.
- Os índices das posições de memória chamam-se endereços e variam com a totalidade de memória disponível no computador.



# Representação de variáveis na memória de um computador

- Todas as variáveis declaradas nos programas ficam guardadas algures em posições da memória do computador.
  - Exemplo:

```
int a, b;  
a = 46;  
b = 17;
```
- Quando o programa é executado, o computador reserva espaço na memória para guardar as variáveis a e b. Estas variáveis são designadas por estáticas.
  - Por exemplo, o computador poderia guardar a variável a na posição E518 e a variável b na posição E833. Estas posições chamam-se endereços de memória.
- Resumindo, uma variável tem sempre 3 coisas: um **valor** (de um tipo), um **endereço** e um **nome** que é usado para a identificar.
  - No exemplo acima, uma das variáveis é identificada por a, tem o valor 46 e o seu endereço é E518.

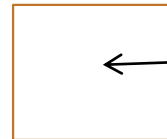
E0	
E1	
E2	
E518	46
E833	17
En	

# Apontadores

- Na linguagem C, existem variáveis “especiais” que permitem manipular endereços. Essas variáveis chamam-se **apontadores**.
- Tal como as variáveis têm um tipo de dados associado, os apontadores têm também um tipo de dados, como por exemplo: `int`, `float`, `char`, `struct`, etc.
- Exemplo:

```
int *p;
```

**p**



**endereço** de uma  
variável do tipo  
**int**

# Apontadores

- `int *p` : declaração de uma variável chamada `p` que é um apontador para um inteiro.
  - Por outras palavras, `p` é uma variável cujo valor é um **endereço** de uma variável do tipo `int`.
- Como qualquer outra variável, um apontador deve ser inicializado, ficando a apontar para:
  - Um endereço de memória;
  - “Nada”, representado através de `NULL` ou `0`.

# Apontadores

- Exemplo:

```
int a, b, *p = NULL;  
a = 46;  
b = 17;  
p = &a;  
printf("Endereco de a: %p\n  
      Valor de p: %p\n  
      Valor de a: %d\n  
      Valor de *p: %d\n", &a, p, a, *p);
```

Output:

```
Endereco de a: 0x28ac54  
Valor de p: 0x28ac54  
Valor de a: 46  
Valor de *p: 46
```

- O operador **&** devolve o endereço da variável que se lhe segue.
- No exemplo, se a variável a estiver a ser guardada no endereço 0x28ac54, o conteúdo da variável p vai passar a ser 0x28ac54. O valor de p indica a localização da variável a na memória do computador. Por isso, diz-se que p aponta para a.



# Apontadores

- É importante lembrar que `p` também é uma variável e, como tal, também tem um endereço.
  - Por exemplo, o computador poderia guardar `p` no endereço `E2`).
- Ou seja, tal como as outras variáveis, o apontador `p` também tem 3 coisas: um **nome** que é usado para o identificar, um **endereço** e um **valor**.
  - O endereço de `p` é `E2` e o seu valor é `E518`.

E0	
E1	
E2	E518
E518	46
E833	17
En	

# Apontadores

- Além do operador **&** existe também outro operador, cujo símbolo é um asterisco (**\***).
  - Se **p** for um apontador, **\*p** devolve o valor da variável que está no endereço de memória indicada por **p**. Por outras palavras, **\*p** é o valor que é apontado por **p**.

# Apontadores

- Exemplo:

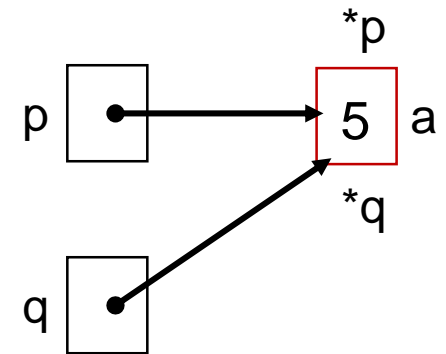
```
int a, b, *p = NULL;  
a = 46;  
b = 17;  
p = &a;  
b = *p;
```
- Quando o programa executa a instrução `b = *p;` o computador irá guardar na variável `b` (endereço E833) o valor que é apontado por `p`. Esse valor, que está no endereço E518, é 46. Ou seja, na altura em que o programa executa esta instrução, `*p` é o mesmo que `a`.

E0	
E1	
E2	E518
E518	46
E833	17
En	

# Exercício

- Represente todas as variáveis existentes no programa, bem como as alterações que elas sofrem ao longo da sua execução.

```
#include <stdio.h>
main( ) {
    int *p= NULL, *q = NULL, a = 1;
    q = &a;  p = q;
    *q = 3;  *p = 5;
    printf("%d %d %d", a, *p, *q);
}
```



# Para que servem?

- Em linguagem C, existe uma série de coisas que só se consegue fazer manipulando apontadores:
  - Mudar valores de variáveis externas, passadas como parâmetros, durante a execução de uma função.
  - Alocar memória dinamicamente.
    - Isso permite, entre outras coisas, criar e usar vetores de dimensão variável sem ter de especificar previamente o tamanho máximo do vetor.
  - Usar eficientemente estruturas de dados mais complexas como, por exemplo, stacks, queues e listas ligadas.

# Passagem de argumentos para funções

- Imaginemos que queremos fazer uma função para trocar o valor de 2 variáveis:

```
void trocar(char x, char y) {  
    char temp = x;  
    x = y;  
    y = temp;  
}
```

- A função principal poderia ser:

```
main() {  
    char a = 'P', b = 'F';  
    trocar(a, b);  
    printf("%c %c", a, b);  
}
```

# Passagem de parâmetros para funções por valor

- O objetivo seria trocar os valores das variáveis.
  - Como tal, o programa deveria escrever no ecrã: F P
  - Mas se executarmos o programa, o computador não irá trocar coisa nenhuma e irá escrever: P F
- O problema acontece porque aquilo que é passado quando a função troca é chamada são cópias dos valores dos argumentos a e b.
  - Isto é, x vai receber uma cópia do valor de a ('P' no exemplo) e y vai receber uma cópia do valor de b ('F' no exemplo).
  - Internamente na função, os valores de x e y vão ser trocados mas os valores de a e b manter-se-ão inalterados.
  - Lembre-se que isso aconteceria mesmo que na função troca tivéssemos chamado aos parâmetros a e b em vez de x e y, já que nesse caso, o a da função nada teria a ver com o a que estava declarado fora da função.

# Passagem de parâmetros para funções

- Uma forma de alterar o valor de variáveis externas dentro de funções é utilizando variáveis globais.
- Uma variável definida globalmente pode ser modificada dentro de uma função sem ter de ser passada como argumento.
- À primeira vista, esta estratégia pode parecer mais simples. No entanto, o uso de variáveis globais deve ser evitado sempre que possível porque as funções que usam variáveis globais deixam de ser genéricas.
- Uma função deve resolver um sub-problema específico e deve ser feita de tal modo que possa ser reutilizada noutro contexto. Se uma função depende de variáveis globais, não podemos reutilizá-la noutro programa.



# Passagem de parâmetros para funções por referência

- Outra forma de alterar o valor de variáveis externas dentro das funções é passando o endereço das variáveis e depois modificar o seu valor através do operador \*.

- Exemplo:

```
void trocar(char *x, char *y) {  
    char temp = *x;  
    *x = *y;  
    *y = temp;  
}  
  
main() {  
    char a = 'P' , b = 'F';  
    trocar(&a, &b);  
    printf("%c %c", a, b);  
}
```

# Passagem de parâmetros para funções por referência

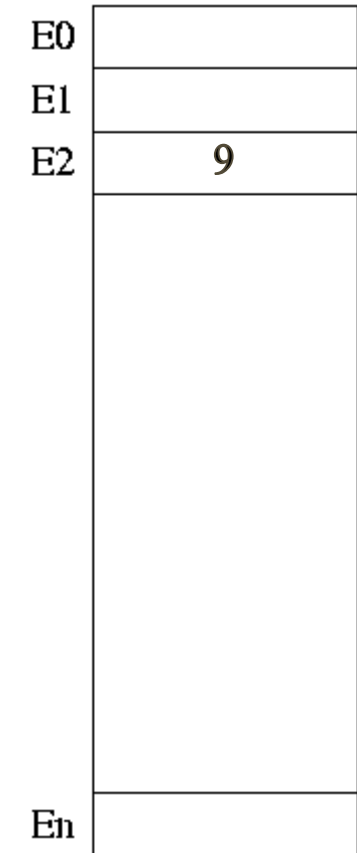
- Desta vez, a função troca vai trocar os valores que são apontados por x e y.
  - No exemplo concreto, o programa irá trocar o valor das variáveis a e b.
- Conclusão: sempre que quisermos modificar o valor de uma variável local de uma função no código de uma outra função devemos passar o endereço dessa variável como argumento.

# Visão mais detalhada da memória do computador

- Quando falamos da representação das variáveis na memória do computador, vimos que cada variável ficava guardada num endereço de memória. Por exemplo, se fizéssemos um programa que contivesse a seguinte declaração:

```
int x = 9;
```

- x** poderia ser armazenado no endereço de memória E2 quando o programa fosse executado.

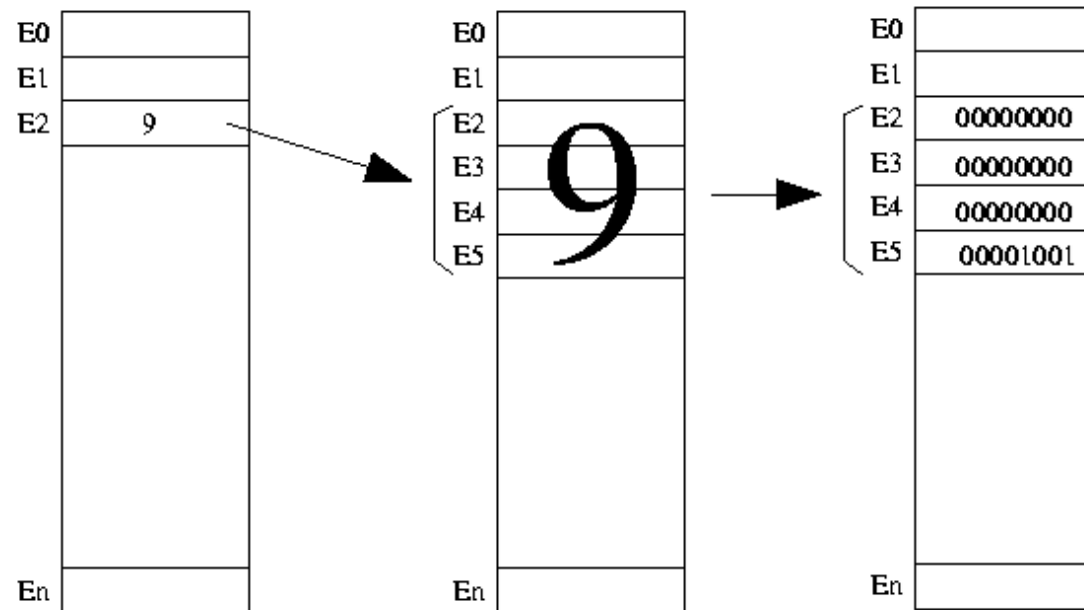


## Visão mais detalhada da memória do computador

- Na realidade, as coisas não são bem assim. Cada endereço de memória corresponde apenas a um byte e uma variável do tipo `int` pode ocupar, por exemplo, 2 ou 4 bytes (depende da arquitetura).
- Ou seja, o número 9 não vai ocupar apenas um endereço de memória. Assumindo que o tipo `int` ocupa 4 bytes, então o número 9 ocupará 4 endereços de memória.
- Aliás, aquilo que está nesses 4 endereços de memória é a representação em binário do número 9.

# Visão mais detalhada da memória do computador

- Apesar da variável x ocupar os endereços E2, E3, E4 e E5, diz-se que o endereço da variável x é E2. Isto é, o endereço de uma variável é o endereço do primeiro byte que ela ocupa.



# Apontadores e arrays

- E os vetores? Como são guardados?
- Se declararmos um vetor `a` de 10 inteiros, vai ser reservado um bloco de memória consecutivo que permite guardar esses 10 inteiros.
- Se um inteiro ocupar 4 bytes, o compilador terá de reservar um bloco de 40 bytes (por exemplo, do endereço E100 até ao endereço E139).
  - `a[0]` vai ocupar os endereços E100, E101, E102, E103
  - `a[1]` vai ocupar os endereços E104, E105, E106, E107
  - `a[2]` vai ocupar os endereços E108, E109, E110, E111
  - ...
  - `a[9]` vai ocupar os endereços E136, E137, E138, E139

# Apontadores e arrays

- Do mesmo modo que dizemos que o endereço da variável `x` é `E2`, podemos dizer que o endereço do vetor `a` é `E100`. Isto é, o endereço do vetor é o endereço do primeiro byte que o vetor ocupa.
- De facto, quando declaramos `int a[10];` o nome `a` é o endereço da primeira posição do vetor. Por outras palavras, `a` é sinónimo de `&a[0]`.
- Assim, é perfeitamente válido fazer:

```
int a[10], *p = NULL;  
p = a;  
*p = 4;    // equivalente a dizer a[0] = 4
```

## Passagem de arrays para funções por referência

- Pelo que vimos atrás, se invocarmos uma função e lhe passarmos como parâmetro um array, a função não receberá a totalidade do array mas apenas o seu endereço inicial (a é sinónimo de &a[0]).
- Assim, quando queremos receber um vetor como parâmetro de uma função, devemos recebê-lo num apontador para o tipo de elementos do vetor.
- Uma vez que apenas é apenas passado o endereço inicial, deve também ser enviada a dimensão do vetor ou o nº de elementos contidos neste.



# Passagem de arrays para funções por referência

```
#include <stdio.h>

void inicializar (float *vetor, int tam) {
    int i;
    for(i = 0; i < tam; i++) vetor[i] = 0;
}

main () {
    float a[10];
    inicializar(a, 10);
}
```

# Apontadores e registos

- Como já foi dito, um apontador também pode ser do tipo struct.
- Exemplo:

```
struct data { int dia, mês, ano; };
```

```
main () {  
    struct data hoje = { .dia=14, .mes=12, .ano=2012 };  
    struct data *p = &hoje;  
}
```

# Apontadores e registos

Como aceder a elementos de um registo apontado?

- Existem duas formas distintas mas equivalentes:

`(*apontador_struct).elemento`

ou

`apontador_struct -> elemento`

- Exemplos:

`printf(“%hu”, (*p).ano );`

ou

`printf(“%hu”, p -> ano );`

# Apontadores e registros

## Exemplo

```
struct data { int dia, mes, ano; };

void alterarData(struct data *pData) { pData->dia = 25; }

main () {
    struct data h = { .dia = 14, .mes = 12, .ano = 2012 };
    printf("Original: %02d-%02d-%d\n", h.dia, h.mes, h.ano);
    alterarData(&h);
    printf("Alterada: %02d-%02d-%d\n", h.dia, h.mes, h.ano);
}
```

Output:


```
Original: 14-12-2012
Alterada: 25-12-2012
```

## Qualificador `const` em apontadores

- A palavra reservada `const` já foi estudada. Indica que a variável se comportará como uma constante não devendo ser modificada durante a execução do programa.
  - Exemplo: `const double PI = 3.14159265359;`
- No caso de apontadores, podemos definir como constantes o seu valor (endereço), ou o valor para o qual aponta.


# Qualificador const em apontadores

- Apontadores constantes:
  - Apontador para uma localização de memória constante.
  - Sintaxe: `tipo *const nome_apontador;`
- Exemplo:

```
int num1 = 10, num2 = 20;  
int *const apontador = &num1;  
apontador = &num2;   
printf("%d", *apontador);
```
- Ao tentarmos alterar o valor do apontador irá ocorrer um erro, pois indicamos na sua declaração que o mesmo não deve ser alterado.



# Qualificador const em apontadores

- Apontadores para constantes:
  - Apontador referencia um valor constante (valor apontado pelo apontador não pode ser alterado).
  - Sintaxe: `const tipo *nome_apontador;`
- Exemplo:

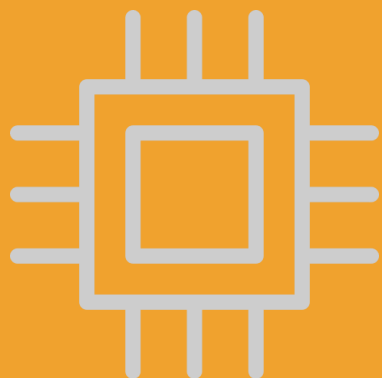
```
int num1 = 10, num2 = 20;  
const int *apontador = &num1;  
apontador = &num2;  
*apontador = 150;   
printf("%d", *apontador);
```
- Ao tentarmos alterar o valor apontado pelo apontador irá ocorrer um erro, pois indicamos na sua declaração que o valor apontado não deve ser alterado.

# Qualificador const em apontadores

- Apontadores constantes para constantes
  - Apontador para uma localização de memória constante cujo valor não pode ser alterado.
  - Sintaxe: **const tipo \*const** nome\_apontador;
- Exemplo:

```
int num1 = 10, num2 = 20;  
const int *const apontador = &num1;  
apontador = &num2;   
*apontador = 150;   
printf("%d", *apontador);
```
- Ao tentarmos alterar o valor do apontador ou o valor apontado pelo mesmo irá ocorrer um erro.





# Memória dinâmica

# Stack

- Uma **stack** assemelha-se a pilha onde inserimos o elemento no topo, e se pretendemos retirar o elemento, também retiramos o primeiro elemento visível (no topo).
- Cada programa (processo) tem a sua própria stack (assumindo apenas uma thread).
- Insere-se no topo e remove-se no topo (**LIFO – *Last In, First Out***).
- Elementos estão empilhados, e apenas se pode inserir ou remover no topo da pilha.
  - Quando um elemento da stack é removido, os seus recursos são automaticamente libertados.

# Stack

- Bloco contíguo de memória.
- Normalmente limitada em termos de espaço (memória).
- Mais rápida (em termos de alocação).
- Quando se pretende utilizar estruturas de maior dimensão, ou funções recursivas mais intrincadas, podem ocorrer erros do tipo *Stack Overflow*.

# Stack

## Exemplo simplificado

```
void funcaoC(){
    printf("Eu sou a funcaoC\n");
}
void funcaoB(){
    printf("Eu sou a funcaoB\n");
    funcaoC();
}
void funcaoA(){
    printf("Eu sou a funcaoA\n");
    funcaoB();
}
int main(){
    printf("Eu sou o ponto de entrada.\n");
    funcaoA();
    exit(0);
}
```



### **FuncaoC**

- Argumentos
- Var. Locais
- ....

### **FuncaoB**

- Argumentos
- Var. Locais
- ....

### **FuncaoA**

- Argumentos
- Var. Locais
- ....

### **Main**

- Argumentos
- Var. Locais
- ....

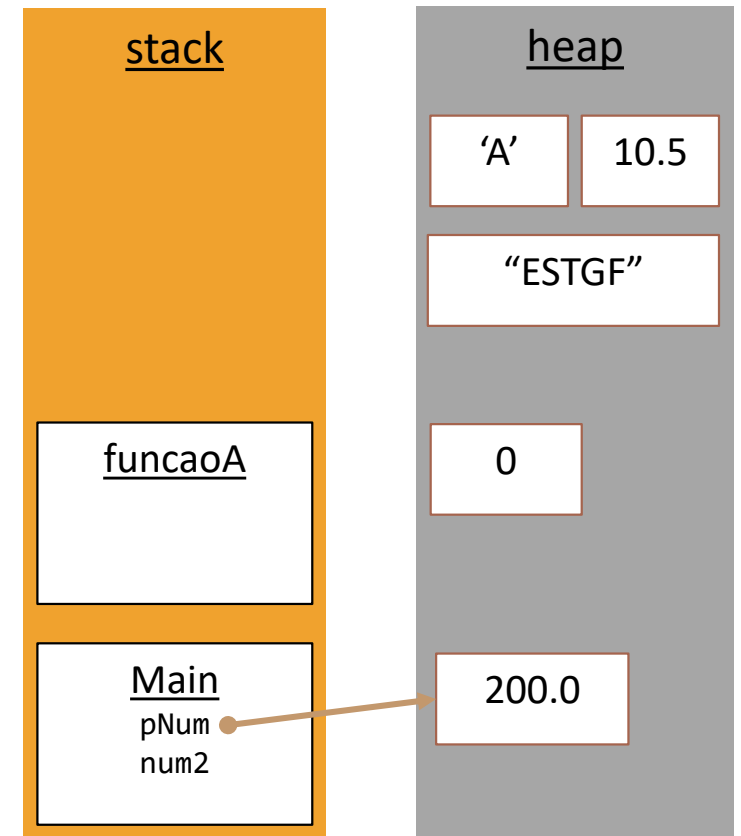
# Heap

- Grande espaço de memória disponível para alocação de memória
- Utilizado para alocação dinâmica de memória
- Na linguagem C, a palavra reservada `malloc` reserva espaço na heap
- Memória reservada existe durante toda a “vida” do programa, ou até que seja explicitamente removida

# Heap e Stack

## Exemplo simplificado

```
void funcaoA() {printf("FuncaoA\n"); }  
int main(){  
    double *pNum = (double*) malloc(sizeof(double));  
    double num2= 100.0;  
    *pNum = 200.0;  
    printf("%lf %lf", *pNum, num2);  
    printf("Entrada.\n");  
    funcaoA();  
    exit (0);  
}
```



# Heap e Stack

## Exemplo diferença

```
double * lerNumero(){  
    double num;  
    readDouble(&num);  
    return &num;  
}
```

ou

```
double * lerNumero(){  
    double *num = (double *) malloc(sizeof(double));  
    readDouble(num);  
    return num;  
}
```

# Stack vs Heap

- **Stack** - Elementos estão empilhados, e apenas se pode inserir ou remover no topo da pilha.
  - Quando um elemento da stack é removido, os seus recursos são automaticamente libertados.
- **Heap** - Elementos não estão alocados por uma ordem pré-definida. Elementos são visíveis (ou podem ser) em todo programa.
  - Necessário libertar memória de uma forma explícita (no caso da linguagem C e outras linguagens).



# Alocação dinâmica de memória

- Quando estudámos arrays, vimos que tínhamos que os declarar com uma dimensão máxima. Para fazer um programa em que se preenche um array com  $n$  números, em que  $n$  seria um número introduzido pelo utilizador, faríamos algo do género:

```
#define N 100
main() {
    int n, a[N];
    printf("Quantos números quer introduzir? ");
    scanf("%d", &n);
    if (n > N) {
        printf("Erro: a dimensão máxima do array é %d", N);
    } else {
        /* código para preencher um array com n números */
    }
}
```

# Alocação dinâmica de memória

- O programa apresentado tem limitações e desvantagens:
  - Podemos ter falta de espaço – o programa só funciona quando  $n \leq N$ .
    - O truque que usamos é o de definir  $N$  com um valor acima daquilo que é expectável e depois usar apenas parte do vetor.
    - Ainda assim, no exemplo, o utilizador poderá pretender introduzir para  $n$  o valor 110, não havendo, nesse caso, suficiente espaço de memória reservado.
  - Pode desperdiçar memória – estamos a reservar espaço para um vetor de 100 inteiros e o número  $n$  introduzido pelo utilizador poderá ser bastante inferior a isso como, por exemplo, 10 ou 20.

# Alocação dinâmica de memória

- O ideal seria reservarmos o espaço de memória exatamente necessário para guardar os dados que o utilizador pretende.
- Isso é possível se reservarmos o espaço de memória durante a própria execução do programa.
- A ideia é perguntar ao utilizador a dimensão do vetor.
  - Exemplo: se o utilizador introduzir 10, pedimos ao computador para reservar um bloco de memória que permita guardar exatamente um vetor de 10 inteiros.
- Deste modo, não temos falta de memória nem desperdício de memória.

# Alocação dinâmica de memória

- Programa com recurso a memória dinâmica:

```
main() {  
    int n, *a = NULL;  
    printf("Quantos números quer introduzir? ");  
    scanf("%d", &n);  
    a = (int *) malloc(n * sizeof(int));  
    if (a == NULL) {  
        printf("Erro: não há memória\n");  
    } else {  
        /* código para preencher um vetor com n números */  
        free(a);  
        a = NULL;  
    }  
}
```

# Alocação dinâmica de memória: malloc

- Em vez de declaramos a variável `a` como sendo um array de inteiros, declaramos `a` como sendo um apontador para um inteiro.
  - A ideia é que `a` vai ser o endereço do primeiro elemento do array.
  - No *header file* `stdlib.h` existe uma função chamada `malloc` (abreviatura de *memory allocation*) que permite reservar (alocar) um bloco de memória de `x` bytes.
  - Concentremo-nos então na linha do programa que tem o `malloc`. A linha parece estranha mas não é nada complicada.

```
a = (int *) malloc(n * sizeof(int));
```

# Alocação dinâmica de memória: `malloc`

```
a = (int *) malloc(n * sizeof(int));
```

- A função **malloc** recebe um argumento que é o número de bytes de memória que pretendemos utilizar.
  - Se quisermos guardar n inteiros temos de pedir n vezes o número de bytes que cada inteiro ocupa.
- O nº de bytes ocupado por um inteiro é dado por **sizeof(int)**. A função `sizeof(T)` devolve o nº de bytes que T (um tipo de dados ou o nome de uma variável) ocupa.
- `malloc` devolve o primeiro endereço do bloco de memória que foi reservado.
  - Caso não seja possível fazer a alocação de memória, `malloc` devolve **NULL**.
- O **(int \*)** que aparece antes do `malloc` indica o tipo de dados que serão apontados no bloco de memória.

# Alocação dinâmica de memória: malloc

- De um modo geral, se quisermos alocar memória para um vetor de n elementos de um tipo T escrevemos:

```
T *a;  
a = (T *) malloc(n * sizeof(T));
```

- O exemplo anterior refere-se a arrays, mas podemos alocar memória para um único valor.
  - Por exemplo, para alocar espaço para um caracter:

```
char *a;  
a = (char *) malloc( sizeof(char) );
```

# Alocação dinâmica de memória: `realloc`

- É muitas vezes útil poder aumentar ou diminuir o tamanho de um bloco de memória alocado dinamicamente através de `malloc`.
  - Isto seria interessante para, por exemplo, ir adaptando o tamanho de um array às necessidades do momento.
- Isto pode ser feito através da função `realloc`, também incluída no *header file* `stdlib.h`, que realoca blocos de memória dinâmica.
  - Exemplo:

```
b = (int *) realloc(a, 100);
```
  - A função recebe dois argumentos: o apontador que indica o atual bloco de memória e o novo tamanho pretendido em bytes.
  - Esta função devolve o primeiro endereço do bloco de memória realocado.
    - Caso não seja possível fazer a realocação de memória, `realloc` devolve `NULL` e mantém o anterior bloco e apontador inalterado.
  - O bloco realocado contém os dados existentes no bloco antes da alteração.
    - Caso o novo bloco seja menor, apenas são mantidos os dados existentes até ao novo tamanho definido.



# Libertar memória: free

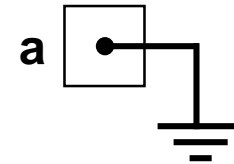
- Sempre que reservamos memória dinamicamente com `malloc` ou `realloc`, devemos libertar essa memória quando já não for necessária.
- Ao contrário do que acontece com a memória reservada para as variáveis estáticas, esta não é automaticamente libertada ao terminar a execução da função onde a memória foi reservada.
- Para libertarmos a memória alocada dinamicamente usamos a função `free`. Uma vez libertada a memória, esta pode ser reutilizada pelo sistema operativo.
- `free(a);` liberta o bloco de memória apontado por `a`.

# Libertar memória: free

- Depois de libertarmos memória alocada dinamicamente, o apontador que para ela aponta passa a apontar para “lixo”.

- Depois de `free( a );` 

- Para indicar que `a` já não aponta para nada, é boa prática definir `a = NULL`. `NULL` é um valor especial para apontador que simboliza “nada”.



- Tal como a função `malloc`, a função `free` e o valor especial `NULL` estão definidos no *header file* `stdlib.h`.

# Precauções no uso de memória dinâmica e apontadores

- Como foi dito no início, é fácil deixar escapar erros ao usar memória dinâmica e apontadores.
- Nem sempre esses erros são fáceis de detetar, em particular em programas extensos.
- A recomendação, que já antes foi dada, é que vão testando o programa cuidadosamente à medida que o desenvolvem.
- Os próximos slides indicam os erros mais frequentes na utilização de memória dinâmica e apontadores.

## Precaução: possível falha na alocação de memória

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

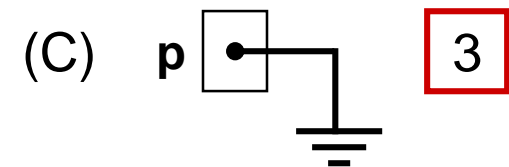
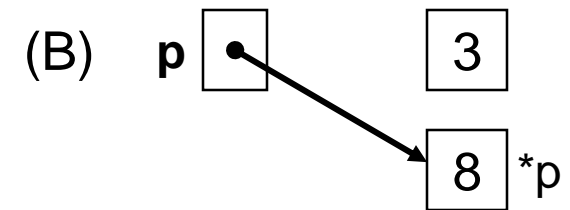
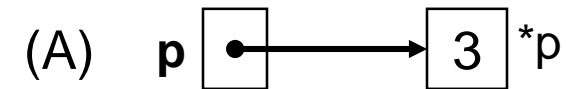
int main() {
    int *const a= (int*) malloc( sizeof(int) );
    if ( a != NULL ) {
        *a = 8;
        free( a );
        a = NULL;
    } else printf("Erro: memória não alocada!");
}
```

- Já foi dito que, caso não seja possível fazer a alocação de memória malloc e realloc devolvem NULL. Deve por isso ser verificado se a alocação de memória foi bem sucedida.

# Precaução: fugas de memória

```
#include <stdlib.h>

int main ( ) {
    int *p = NULL, *q = NULL;
    p = (int*) malloc( sizeof(int) );
    *p = 1 + 2; (A)
    p = (int*) malloc( sizeof(int) );
    *p = 8; (B)
    free( p );
    p = NULL; (C)
}
```



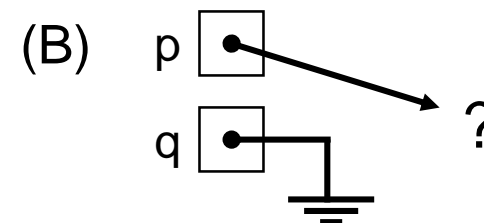
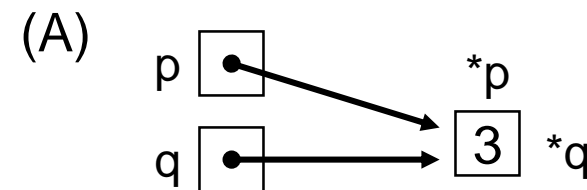
- Uma variável dinâmica tem que ser acedida através de um apontador. Assim, antes de alterar um apontador (exemplos: `p = q;` ou `p = (int*) malloc(...);` ou `p = NULL;`), é preciso ter cuidado para que `*p` não fique inacessível.

# Precaução: referências inválidas a um apontador

```
#include <stdlib.h>

int main ( ) {
    int *p = NULL, *q = NULL;
    p = (int*) malloc( sizeof(int) );
    q = p;
    *q = 3;    (A)
    free( q );
    q = NULL;  (B)
    printf("%d %d", *p, *q); // Errado
    *p = 999; // Errado
    *q = 999; // Errado
}
```

- Depois de `free( q )`, o bloco apontado por `*p` (e `*q`) é libertado e não pode mais ser acedido. Logo, já não podemos usar `*p` e `*q` que apontam para “lixo”.

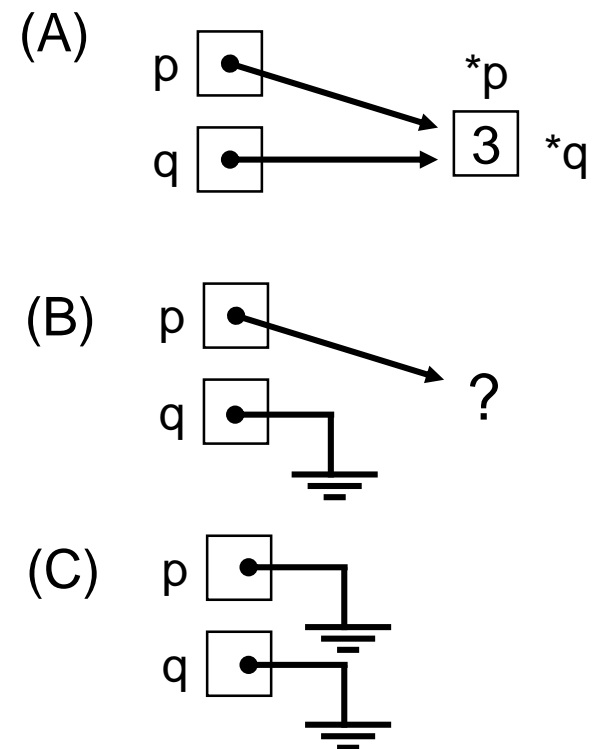


# Precaução: libertar bloco desnecessário uma única vez

```
#include <stdlib.h>

int main ( ) {
    int *p = NULL, *q = NULL;
    p = (int*) malloc( sizeof(int) );
    q = p;
    *q = 3;      (A)
    free( q );
    q = NULL;   (B)
    free( p );  // Errado
    p = NULL;   (C)
}
```

- Libertar cada bloco desnecessário uma única vez.
- Não libertar cada apontador uma vez.



## Precaução: não libertar apontadores indefinidos

```
#include <stdlib.h>

int main ( ) {
    int *p, *q;
    p = (int*) malloc( sizeof(int) );
    q = NULL;
    free( p );  p = NULL;
    free( q );  q = NULL;
}
```

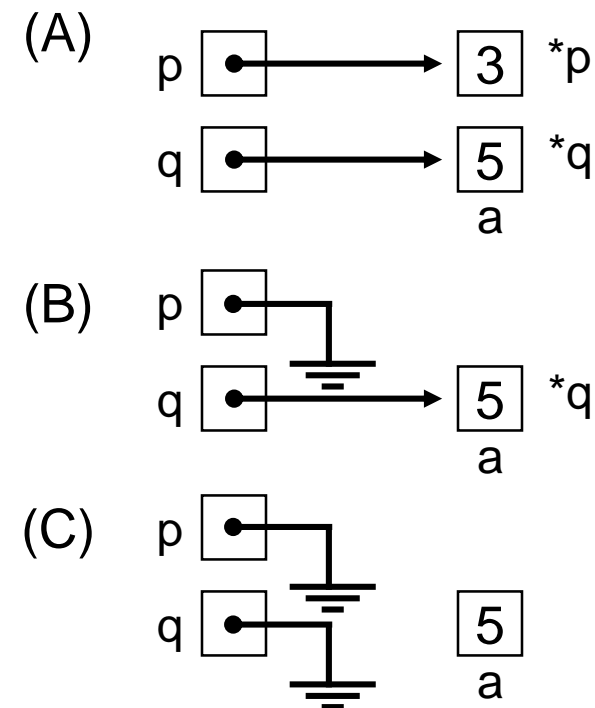
- Libertar um apontador que não esteja a apontar para um bloco de memória alocado dinamicamente provoca geralmente a interrupção súbita do funcionamento do programa ou leva a erros estranhos mais tarde.
- Para evitar este problema deve sempre inicializar um apontador com a alocação de memória ou com o valor NULL.



# Precaução: não libertar variáveis estáticas

```
#include <stdlib.h>

int main ( ) {
    int *p = NULL, *q = NULL;
    int a = 5;
    p = (int*) malloc( sizeof(int) );
    q = &a;
    *p = 3; (A)
    free( p );
    p = NULL; (B)
    free( q ); // Errado
    q = NULL; (C)
}
```



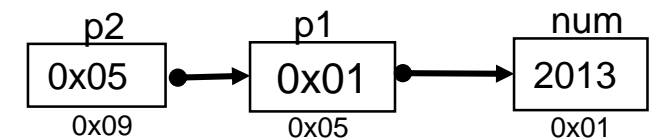
- Só é necessário libertar variáveis dinâmicas. Nunca libertar variáveis estáticas. Só libertamos (free) aquilo que alocamos (malloc).

# Apontadores para Apontadores

- Sabemos que um apontador armazena o endereço de outra variável. Essa outra variável pode ser um outro apontador!!!

- Exemplo:

1. `int num = 2013;`
2. `int *p1 = &num;`
3. `int **p2 = &p1;`



1. A variável num armazena um valor inteiro, neste caso “2013”;
  2. A variável p1 armazena o endereço de num;
  3. A variável p2 armazena o endereço de p1 (outro apontador).
- Aceder ao valor de num através de p2?
    - `**p2 = 2013;`

# Matrizes

```
#define LINHAS 100
```

```
#define COLUNAS 50
```

```
...
```

```
int i, **matriz;
```

```
matriz = (int **) malloc(LINHAS * sizeof(int *));
```

```
for (i = 0; i < LINHAS; ++i) {
```

```
    matriz[i] = (int *) malloc(COLUNAS * sizeof(int));
```

```
}
```

```
matriz[0][0] = 10;
```

```
printf("%d", matriz[0][0]);
```

```
for (i = 0; i < LINHAS; ++i) { free(matriz[i]); }
```

```
free(matriz);
```



# Ficheiros

# Streams

- Uma *stream* é qualquer fonte de input ou qualquer destino de output.
- Muitos pequenos programas obtêm o seu input do teclado e o colocam o seu output no ecrã.
- *Stream* normalmente representam ficheiros armazenados em diferentes tipos de média.
- *Stream* podem facilmente ser associados com outros dispositivos como *network ports* e impressoras.

# *File Pointers*

- Aceder a uma stream faz-se com um *file pointer*, cujo tipo é `FILE *`.
- O `FILE` está declarado no `<stdio.h>`.
- Podemos declarar apontadores para *streams* facilmente:

```
FILE *fp1, *fp2;
```

# Standard streams

- `stdio.h` fornece 3 *streams*. Estas *streams* não têm que ser inicializadas nem declaradas para serem utilizadas.

File Pointer	Stream	Significado
<code>stdin</code>	Standard input	Teclado
<code>stdout</code>	Standard output	Ecrã
<code>stderr</code>	Standard error	Ecrã

# Ficheiros de texto vs binários

- `stdio.h` suporta ficheiros de texto e binários.
- Os bytes num ficheiro de texto representam caracteres que permitem entender o conteúdo.
  - O código fonte de um programa C está num ficheiro de texto.
- Num ficheiro binário os bytes não representam necessariamente caracteres.
  - Grupos de bytes podem representar outros tipos de dados como inteiros, números de vírgula flutuante, etc.
  - Um programa executável feito em C é armazenado num ficheiro binário.



# Ficheiros de texto vs binários

- Os ficheiros texto possuem 2 características que não existem nos ficheiros binários:
  - Estão divididos em linhas
  - Podem conter um marcador de “fim de ficheiro”
- Num ficheiro binário, todos os bytes são tratados de igual forma.

# Ficheiros de texto vs binários

- Quando escrevemos num ficheiro podemos fazê-lo em texto ou em formato binário.
- Uma forma de guardar o número 32767 num ficheiro será a de escrever cada um dos seus caracteres 3, 2, 7, 6, e 7:

00110011	00110010	00110111	00110110	00110111
'3'	'2'	'7'	'6'	'7'

# Binário

- A outra opção é guardar em binário. O que utilizaria apenas 2 bytes: Guardar um número em binário poupará espaço!

01111111	11111111
----------	----------

# Ficheiros de texto vs binários

- Um programa que leia ou escreva num ficheiro tem que levar em conta o seu tipo.
- Um programa que mostra o conteúdo de um ficheiro no ecrã provavelmente assume que este é de texto.
- Um programa de copia de ficheiros não pode assumir que um ficheiro é de texto.
  - Se o fizer, um ficheiro binário que inclua o caracter de “fim de ficheiro”, não será totalmente copiado!

# Abrir ficheiro

- Chamada à função fopen:

```
FILE *fopen(const char * filename, const char * mode);
```

- `filename` é o nome do ficheiro a ser acedido (Pode incluir o path).
- `mode` especifica o tipo de operação que queremos executar sobre o ficheiro.

# Abrir ficheiro

- No Windows é preciso ter cuidado quando a chamada ao `fopen` ao incluir o caracter `\`.
- A chamada `fopen("c:\estgf\test1.dat", "r")` falha porque `\t` é tratado de forma especial.

- Uma solução é usar `\\` em vez de `\`:

```
fopen("c:\\estgf\\test1.dat", "r")
```

- Outra é usa o caracter `/` em vez de `\`:

```
fopen("c:/estgf/test1.dat", "r")
```

# Abrir ficheiro

- `fopen` retorna um *file pointer* que normalmente é guardado numa variável:

```
/* abre in.dat para leitura */  
fp = fopen("in.dat", "r");
```
- Quando a abertura falha, o `fopen` retorna `NULL`.

# Modos

- O modo do fopen depende:
  - Das operações que se pretende executar sobre o ficheiro.
  - Se o ficheiro contém dados de texto ou binários.



# Modos para ficheiros de texto

String	Significado
“r”	Abertura para leitura
“w”	Abertura para escrita (ficheiro não tem que existir)
“a”	Abertura para acrescentar (ficheiro não tem que existir)
“r+”	Abertura para leitura e escrita
“w+”	Abertura para leitura e escrita (cria se não existir)
“a+”	Abertura para leitura e escrita (adiciona no fim se já existir)

# Modos para ficheiros binários

String	Significado
“rb”	Abertura para leitura
“wb”	Abertura para escrita (ficheiro não tem que existir)
“ab”	Abertura para acrescentar (ficheiro não tem que existir)
“r+b” ou “rb+”	Abertura para leitura e escrita
“w+b” ou “wb+”	Abertura para leitura e escrita (cria se não existir)
“a+b” ou “ab+”	Abertura para leitura e escrita (adiciona no fim se já existir)

# Fechar um ficheiro

- A função `fclose` permite fechar um *file pointer* obtido na execução de um `fopen`.
  - `fclose` retorna zero se o ficheiro foi fechado com sucesso.
  - Noutros casos, retorna o código de erro EOF (macro definida em `stdio.h`).

# Fechar um ficheiro

```
#include <stdio.h>
#include <stdlib.h>
#define FILE_NAME "example.dat"
int main(void) {
    FILE *fp;
    fp = fopen(FILE_NAME, "r");
    if (fp == NULL) {
        printf("Can't open %s\n", FILE_NAME);
        exit(EXIT_FAILURE);
    }
    ...
    fclose(fp);
    return 0;
}
```

# Remove Rename

- As funções `remove` e `rename` permitem fazer as operações básicas sobre ficheiros, remover e renomear, respetivamente.
- `remove` e `rename` usam nomes de ficheiros em vez de ponteiros.
  - Ambas retornam zero se têm sucesso e um valor diferente de zero se falham.
- Exemplos:
  - `remove("c:/test.txt");`
  - ```
if (rename(filename, newfn) == 0) {  
    printf("%s has been renamed %s.\n", filename, newfn);  
}
```

# printf fprintf

- `printf` e `fprintf` permitem escrever dados formatados para um *stream*.  

```
int printf(const char * format, ...);  
int fprintf(FILE * stream, const char * format, ...);
```
- `printf` escreve para o `stdout` e `fprintf` escreve para o *stream* definido no primeiro argumento:  

```
printf("Total: %d\n", total);  
fprintf(fp, "Total: %d\n", total);
```

  - Uma chamada ao `printf` é equivalente a chamar o `fprintf` com `stdout` no primeiro argumento.
  - Ambas retornam o número de caracteres escritos ou um valor negativo se ocorreu um erro

# scanf fscanf

- scanf e fscanf permitem ler dados de uma *stream*:  
scanf("%d%d", &i, &j);  
fscanf(fp, "%d%d", &i, &j);
- Uma chamada ao scanf é equivalente a chamar o fscanf com `stdin` no primeiro argumento.

# Ficheiros de Texto

- A chamada `feof(fp)` retorna um valor diferente de zero se o fim de ficheiro foi detetado para o *file pointer* `fp`.



# fputc / putc fgetc / getc

- fputc e putc escrevem um caracter numa *stream*:  
    fputc(ch, fp);      /\* escreve ch em fp \*/  
    putc(ch, fp);      /\* escreve ch em fp \*/
- fgetc e getc lêem um caracter de uma *stream*:  
    ch = fgetc(fp);  
    ch = getc(fp);

## fcopy.c (1/2)

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

int main(int argc, char *argv[]) {
    FILE *source_fp, *dest_fp;
    int ch;
    if (argc != 3) {
        fprintf(stderr, "usage: fcopy source dest\n");
        exit(EXIT_FAILURE);
    }
    if ((source_fp = fopen(argv[1], "rb")) == NULL) {
        fprintf(stderr, "Can't open %s\n", argv[1]);
        exit(EXIT_FAILURE);
    }
}
```

## fcopy.c (2/2)

```
    if ((dest_fp = fopen(argv[2], "wb")) == NULL) {  
        fprintf(stderr, "Can't open %s\n", argv[2]);  
        fclose(source_fp);  
        exit(EXIT_FAILURE);  
    }  
    while ((ch = getc(source_fp)) != EOF)  
        putc(ch, dest_fp);  
    }  
    fclose(source_fp);  
    fclose(dest_fp);  
    return 0;  
}
```

fputs  
fgets

- Escrever uma string para uma *stream*:  
`fputs("Ola ESTGF!", fp);`
- Obter uma string de uma *stream*:  
`fgets(str, sizeof(str), fp);`

# Ficheiros binários

- Os ficheiros binários têm duas características que os distinguem dos ficheiros de texto:
  - Podemos saltar instantaneamente para qualquer registo, fornecendo acesso aleatório.
  - Podemos alterar o conteúdo de um registo em qualquer parte do ficheiro.
- Normalmente têm tempos de acesso mais reduzidos do que os ficheiros de texto.
- A principal desvantagem está relacionada com a portabilidade do código.

# Leitura/escrita de dados

- Cada operação de leitura sobre o apontador, lê a posição corrente e incrementa o apontador de um registo. O mesmo acontece com as operações de escrita.

```
read_size = fread(data_ptr,size,1,file);  
write_size = fwrite(data_ptr,size,1,file);
```

- `read_size / write_size`: Tamanho da informação que foi lida/escrita. Se o valor for inferior a 1 então chegamos ao EOF ou ocorreu um erro.
- `data_ptr`: É o ponteiro para a informação a ser lida/escrita.
- `size`: número de bytes que constituem o bloco de informação a ser lida/escrita.
- `1`: número de blocos a serem lidos/escritos.
- `file`: ficheiro de input/output.

# Posicionamento

```
int fseek(FILE *file_ptr, long numbytes, int origin);
```

- `file_ptr`: Ponteiro para o ficheiro que foi retornado pela chamada ao `fopen`.
- `num_bytes`: Número de bytes a partir de `origin` que nos dará a nova posição.
- `origin`: É uma das seguintes macros:
  - `SEEK_SET`: a partir do início do ficheiro;
  - `SEEK_CUR`: a partir da posição atual;
  - `SEEK_END`: a partir do fim (deslocamento negativo obrigatório).
- O `fseek` retorna zero em caso de sucesso e outro valor no caso de não obter sucesso
- O comando `ftell` retorna o valor atual do ponteiro para o ficheiro.

# Exemplo

## Declaração estrutura

```
#include <stdio.h>
struct reg {
    char nome[40];
    int num;
};
```



# Exemplo

Abrir ficheiro e colocar  
10 registos

```
int main() {  
    struct reg r;  
    FILE *f; int i,j;  
    f = fopen("lixo.dat","wb");  
    for(i=0;i<10;i++) {  
        r.num=i;  
        strcpy(r.nome,"John");  
        fwrite(&r,sizeof(struct reg),1,f);  
    }  
    fclose(f);  
}
```

# Exemplo

Abrir ficheiro e mostrar  
10 registos

```
f = fopen("lixo.dat","rb");  
for(i=0; i<10; ++i) {  
    fread(&r, sizeof(struct reg), 1, f);  
    printf("\n Lido: %d - %s \n", r.num, r.nome);  
}  
fclose(f);
```

# Exemplo

Ler o 4º registo e  
modificá-lo

```
f = fopen("lixo.dat", "r+b");

fseek(f, sizeof(struct reg)*3, SEEK_SET);
fread(&r, sizeof(struct reg), 1, f);
printf("\n Lido: %d - %s ", r.num, r.nome);
strcpy(r.nome, "Ze");
r.num = 500;

/* posiciona-se no inicio do registo */
fseek(f, -sizeof(struct reg), SEEK_CUR);
fwrite(&r, sizeof(struct reg), 1, f);
fclose(f);
}
```

```
return 1;
```



*#include<stdio.h>*