Imperative Programming

Programação Imperativa

(21 day course in Programming in C (PASCAL version also available))

Cursos: ESI, I

P. Stallinga, Universidade do Algarve, CEOT, OptoEl

---

aulas teóricas

1	1-Oct 1-Oct	Apresentação		
2	6-Oct 8-Oct	Computadores		
3	8-Oct 11-Oct	Memória		
4	13-Oct 15-Oct	A linguagem C		
5	15-Oct 18-Oct	Variáveis, **`printf`**		
6	20-Oct 22-Oct	atribuição `scanf` cálculo		
7	22-Oct 29-Oct	**`if`** ... **`else`** ..., comparações		
8	29-Oct 1-Nov	Álgebra Booleana, **`switch`**		
9	3-Nov 5-Nov	Ciclos I: **`for`**		
10	5-Nov 8-Nov	Ciclos II: **while**		
11	10-Nov 12-Nov	Programação Modular I Funções		
12	12-Nov 15-Nov	Programação Modular II Funçoes com input e output		

mini testes

1		
2		
3		
4		
5		
6		
7		
8		
9 10		
11 12		
14		
15		
16		
18		
19		
20		

aulas práticas

1	1:11-Oct 2:19-Oct 3:19-Oct 4:6-Oct 5:14-Oct	Linux 1 Linux 2 Internet	
2	1:18-Oct 2:26-Oct 3:26-Oct 4:13-Oct 5:21-Oct	Compilador gcc Input/output	soluções
3	1:25-Oct 2:2-Nov 3:2-Nov 4:20-Oct 5:28-Oct	Variáveis, `if`, `if..else` versão 1 versão 2 versão 3	soluções versão 1 versão 2 versão 3
4	1:8-Nov 2:9-Nov 3:9-Nov 4:27-Oct 5:4-Nov	`switch`, ciclos versão 1 versão 2 versão 3	soluções versão 1 versão 2 versão 3
5	1:15-Nov 2:16-Nov 3:16-Nov 4:3-Nov 5:11-Nov	ciclos versão 1 versão 2 versão 3	soluções versão 1 versão 2 versão 3
6	1:22-Nov 2:23-Nov 3:23-Nov 4:10-Nov 5:18-Nov	funções versão 1 versão 2 versão 3	soluções versão 1 versão 2 versão 3
7	1:29-Nov 2:30-Nov	arrays versão 1	soluções versão 1

13	17-Nov 19-Nov	`arrays`		
14	19-Nov 22-Nov	pointers		
15	24-Nov 26-Nov	Âmbito das variáveis. *Passing by value* / *Passing by reference*		
16	26-Nov 29-Nov	Programação recursiva		
17	3-Dez 3-Dez	strings		
18	10-Dez 6-Dez	**`struct`**		
19	10-Dez 10-Dez	**`typedef`**		
20	15-Dez 13-Dez	Ficheiros		
	17-Dez 17-Dez	aula de dúvidas		
		explicações de um teste		

Datas: azuis: I-T2, vermelhas: I-T1 e ESI

	3:30-Nov 4:17-Nov 5:25-Nov	versão 2 versão 3	versão 2 versão 3
8	1:6-Dez 2:7-Dez 3:7-Dez 4:24-Nov 5:2-Dez	strings versão 1 versão 2 versão 3	soluções versão 1 versão 2 versão 3
9	1:13-Dez 2:14-Dez 3:14-Dez 4:15-Dez 5:9-Dez	struct versão 1 versão 2 versão 3	soluções versão 1 versão 2 versão 3
10	5:16-Dez	ficheiros versão 1 versão 2 versão 3	soluções versão 1 versão 2 versão 3
11	----	quer mais ...?	soluções

1: I, turma A: terça-feira 14:00-17:00
2: I, turma B: segunda-feira 13:30-16:30
3: ESI, turma B: terça-feira 15:30-18:30
4: EI: quarta-feira 15:00-18:00
5: ESI, turma A: quinta-feira 13:30-16:30

azul: dada pelo Álvaro Barradas
verde: dada pelo Patrício Serendero
vermelho: dada pelo Peter Stallinga

Trabalho Prático 2004-2005

clique aqui

Avaliação

regras de avaliação

Datas dos exames

Exame época normal: 17/01/2005 solução

Exame de recurso: 3 de Fevereiro de 2005 9:00-11:00

Exemplos de exames de um ano passado:

Frequência: enunciado sol

Exame época normal: enunciado

Exame de recurso: enunciado

Mirror sites:

descontinuado (o servidor da FCT agora está disponível fora da universidade)

Documentação adicional

- Bibliotecas da linguagem C
- Documentação sobre Linux
- Compilador de C para MSDOS/Windows
- Dificuldades em aprender?
- GNU C
- getch() para Linux: kbstuff.h (para guardar na pasta de trabalho), exemplo de uso

---

compiladores C

miracle C
http://www.bloodshed.net/
mais compiladores: www.download.com

Peter Stallinga, 2002-2004. Última alteração: 8.11.2004

Programação Imperativa
Aula 1: Apresentação

Índice

Descrição e objectivos da cadeira

Esta é uma cadeira de introdução à computação de programação. Nas primeiras duas semanas de aulas iremos dar uma prespectiva global sobre as várias facetas do mundo da computação. Daremos as noções de hardware, software, sistema operativo, linguagens de programação, compiladores, programas de aplicação, a Internet e a sua utilização.

O resto das aulas vai incidir sobre os fundamentos de programação de computadores. Como linguagem de programação iremos usar a linguagem C mas os conceitos que vão aprender aplicam-se a quase todas as linguagens de programação. No final da cadeira os alunos devem saber os fundamentos de programação e devem ser capazes de escrever programas simples. A cadeira não requer conhecimentos prévios na área de informática.

Nos apontamentos:

- Textos em *italics* são informações gerais e não precisa de lembrar. O uso de *italics* pode também significar palavras estrangeiras (por exemplo: *file* em vez de ficheiro)
- Texto em `fixed-width font` é código C.

Professores

NOME	E-MAIL	HORÁRIO DE ATENDIMENTO	GABINETE
Peter Stallinga	pjotr@ualg.pt **não** recebo e-mails do `hotmail.com` e `yahoo.com`	segunda-f. 13:30-17:30 quinta-f. 10:00-12:00	2.78 / 2.68
Patricio Serendero	pserende@ualg.pt	segunda-f. 16:00-17:00 Quinta-f 15:00-18:00	2.63

Álvaro Barradas	abarra@ualg.pt	segunda-f. 16:30-18:00 terça-f. 13:30-14:00 terça-f. 17:00-18:00 quarta-f. 13:30-15:00	2.57

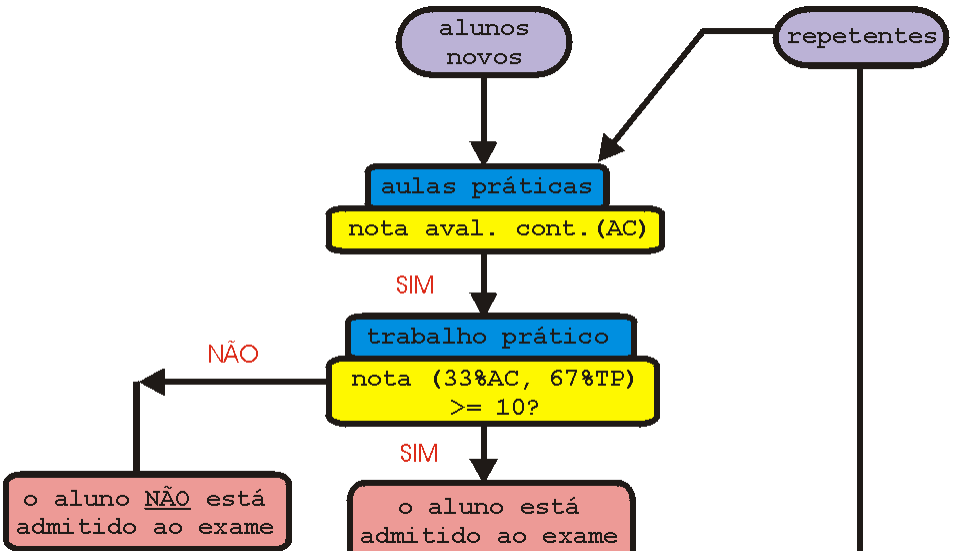
Os alunos devem tirar as suas dúvidas preferencialmente nas aulas. Só se a dúvida persistir é que devem então contactar os docentes no horário referido acima.

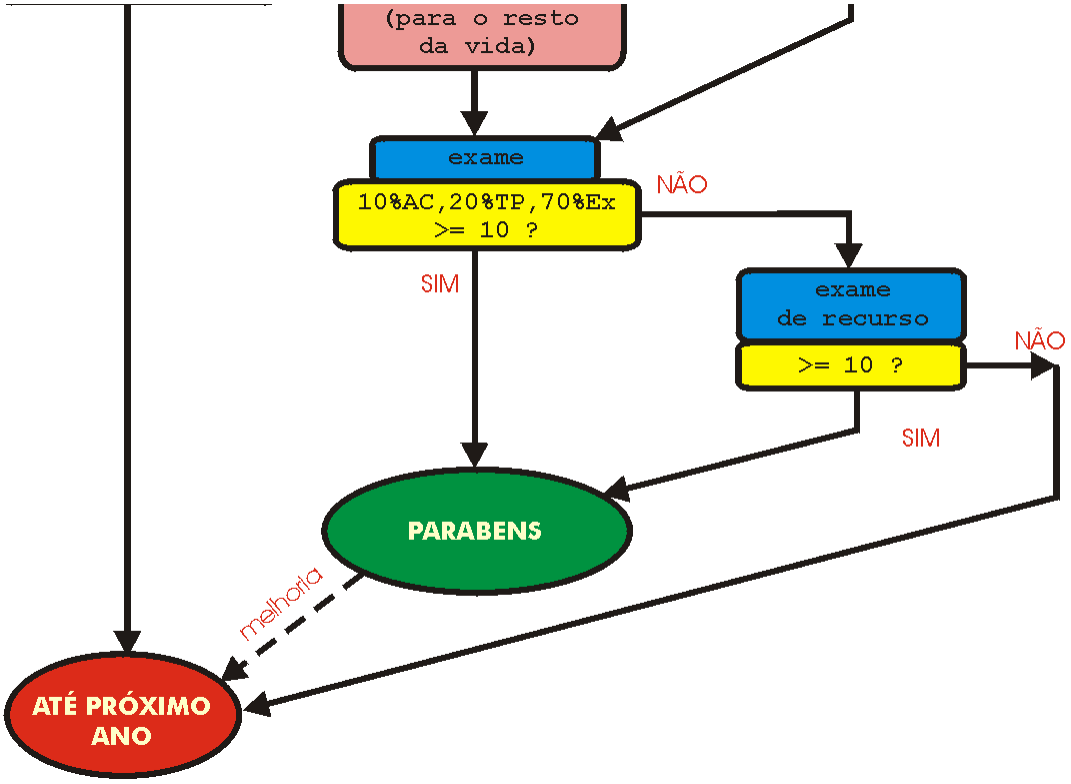
Bibliografia

- Apontamentos dados nas aulas estarão disponíveis na webpage da cadeira, http://diana.uceh.ualg.pt/P1_pjotr/ e *mirror site* http://w3.ualg.pt/~pjotr/p1/index.html
- "Programação com linguagem C", João Gonçalves, Edições Sílabo, ISBN 972-618-088-0
- "The C programming language", Brian W. Kernighan and Dennis M. Ritchie, Prentice Hall; ISBN: 0131103628
- http://www.strath.ac.uk/IT/Docs/Ccourse/
- http://www.eskimo.com/~scs/cclass/notes/top.html

Regras de avaliação (2004-2005)

- A avaliação da cadeira é baseada numa avaliação contnua nas aulas práticas, num trabalho prático e um exame com peso 10%, 20% e 70% respectivamente. A nota final deve ser igual ou superior a 10.
- Quem não passa o exame de época normal pode fazer o exame de recurso. A parte prática não conta para o exame de recurso; o peso de exame de recurso é 100%.
- Para estar admitido ao exame precisa uma nota da parte prática (33% avaliação contínua, 67% trabalho prático) igual ou superior a 10. Os alunos com notas abaixo de 10 serão classificados "sem frequência". Os alunos com nota igual ou superior a 15 podem ser chamados para explicar os seus trabalhos. Trabalhos copiados serão anulados.
- As entregas de trabalhos práticos devem ser feitas em disquette 3.5" até uma semana antes do exame.
- As regras para os 'repetentes' (que foram no mínimo uma fez inscritos na cadeira de Programação 1) são iguais às regras para os alunos novos (10%-20%-70%). Alternativamente podem optar para uma nota apenas baseada no exame (100%).





Data e local dos testes

veja no servidor dos horários da universidade

Copianço

Quem copiar, deixar copiar, ou fizer qualquer outro tipo de batota, durante os momentos de avaliação, terá zero valores e leva um processo disciplinar para o Conselho Pedagógico.

Programa

Apresentação, descrição e objectivos da cadeira.
Noções introdutórias sobre computadores: tipos de computadores, componentes de um computador, características de um computador, sistema operativo, linguagens de programação, compiladores, programas de aplicação.
Noções e utilização da Internet.
Noções básicas de programação: constantes, variáveis, expressões, operadores, instrução de atribuição, instruções de input/output, funções pré-definidas.
Noções de programação estruturada: sequência, selecção, iteração.
Instruções de selecção: if, if-else, switch.
Instruções de iteração: ciclos for, while, do-while.
Funções (programação modular).

Vectores e matrizes: arrays de uma e duas dimensões.
Caracteres e cadeias de caracteres.
Structs e definição de novos tipos de dados.
Recursividade.
Noção de algoritmo. Algoritmos de ordenação simples. Algoritmo de pesquisa sequencial e de pesquisa binária.
Apontadores. Passagem de parâmetros.
Input / *output* de ficheiros.

Recomendação para os alunos

A programação de computadores não é difícil. Pelo contrário, é uma tarefa relativamente fácil e divertida que envolve apenas meia dúzia de conceitos. No entanto, requer um tipo de raciocínio a que as pessoas normalmente não estão muito habituadas. Como tal, trata-se de uma tarefa que exige bastante prática e por isso recomendo que treinem bastante fora do horário das aulas. Se fizerem isso ao longo do semestre, nem sequer precisam de estudar para os testes e passarão à cadeira com boa nota quase de certeza.

Peter Stallinga. 24 setembro 2004

Aula 2: Computadores

Translated by Luís Pereira

O que é um computador? De acordo com a Columbia Encyclopedia é "um dispositivo capaz de executar um conjunto de operações aritméticas e lógicas. Um computador distingue-se de uma máquina de calcular, como por exemplo uma calculadora electrónica, por ser capaz de guardar um programa (para que possa repetir as suas operações e tomar decisões lógicas), devido ao número e complexidade das operações que podem ser executadas, e pela possibilidade de executar, guardar e aceder dados sem intervenção humana."

Existem vários tipos de computadores. Hoje, quando falamos em computadores, queremos dizer computadores digitais. Ao longo da história, também existiram computadores mecânicos e analógicos (electrónico).

O Primeiro computador mecânico foi criado por Charles Babbage no principio do século 19 (por volta de 1815)! Por esta razão, Babbage é frequentemente chamado "O pai da computação". Famosa é também a Difference Engine. Se quer saber mais sobre Charles Babbage, clique aqui.
O primeiro computador electrónico, processando dados no formato digital foi o ENIAC pouco antes da segunda grande guerra (1939). O primeiro computador comercial disponível foi o UNIVAC (1951). Nessa altura pensava-se que uma mão cheia de computadores, distribuídos pelo mundo, seriam suficientes para efectuarem todos os cálculos necessários. Hoje em dia, a grande maioria das pessoas no Ocidente tem um computador em casa, muito mais poderoso que esses primeiros computadores.

Os computadores estão organizados por tamanho e número de pessoas que os pode usar ao mesmo tempo:

Supercomputadores	Máquinas sofisticadas desenvolvidas para executarem calculos complexos a grande velocidade; são utilizados para modelar grandes sistemas dinâmicos, como por exemplo os padrões do tempo. Um exemplo é o Cray SV2 (ver figura), que tem as dimensões de uma sala média 
Mainframes	Os maiores e mais poderosos sistemas para fins comuns, são desenvolvidos para as necessidades de uma grande organização, servindo centenas de terminais ao mesmo tempo. Imaginem companhias de seguros com todos os seus documentos internos partilhados através de uma rede. Todos os empregados podem aceder e editar os mesmos dados.
Minicomputadores	Embora algo pequenos, também são computadores multiutilizador, pensados para satisfazer as necessidades de uma pequena empresa servindo até uma centena de terminais.

Microcomputers	Computadores servidos por um microprocessador, são divididos em computadores pessoais e estações de trabalho. Computadores pessoais são os que a maioria das pessoas têm em casa. Exemplos:  IBM PC e compatíveis com um microprocessador como o Intel Pentium IV, 1.5 Ghz. Computadores portáteis são uma variante dos micro-computadores.   Apple Macintosh incorporando processador da familia Motorola.
Processors	Muitos electrodomésticos, como as máquinas de lavar e fornos, contêm um pequeno processador para controlar o equipamento. São computadores muito pequenos que foram programados nas máquinas ainda na fábrica e não podem ser programados pelos utilizadores. Podem assim não ser considerados computadores, mas convem ser referidos. Alguns electrodomésticos mais avançados, como receptors de satellite ou sistemas de cinema, correm programas muito sofisticados que seguem as regras a seguir apresentadas nesta aula.

Nota: com a velocidade que a tecnologia avança podemos dizer que "os supercomputadores de hoje são os microcomputadores(pessoais) de amanhã"

Hardware vs. software

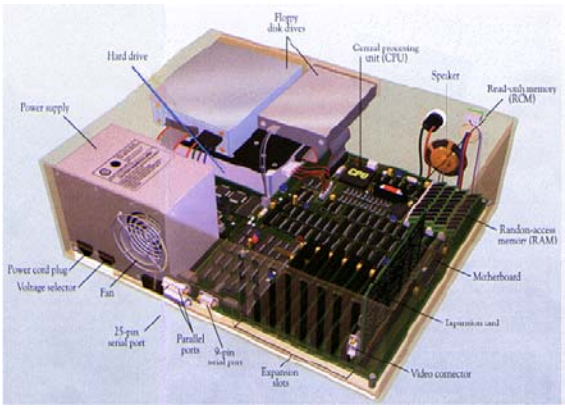
Dois termos que importa distinguir são hardware e software. Hardware é tudo o se pode tocar e sentir. Sofware são os programas que correm no hardware. Exemplos em baixo.

Todos os computadores são constituídos pelo menos por:

1) Um processador
2) Memória para armazenar o programa
3) Um dispositivo de entrada
4) Um programa

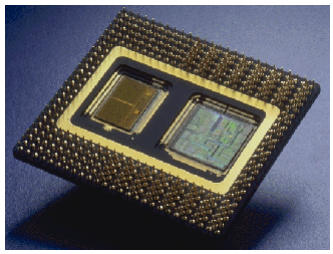
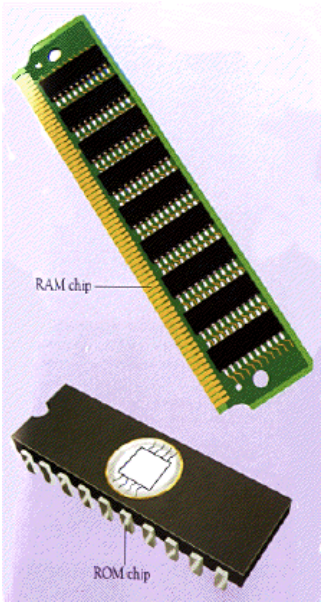
A maioria dos computadores também têm:

5) Um dispositivo de entrada, para mudar o programa, introduzir novos dados para serem processados ou controlar os processos correntes.



Vejamos alguns componentes do computador

Tabela: elementos de hardware

Rato		dispositivo de entrada	Para controlar os processos do computador
CPU		Processador	Central Processing Unit (Unidade Central de Processamento) Peça que faz o trabalho. Calcula, controlas os dados, etc.
Joystick		dispositivo de entrada	Controlador para jogos
Teclado		dispositivo de entrada	Para dar instruções ao computador ou introduzir dados para processamento
Memória		armazenamento	Guardar programas e dados para processamento
Monitor		Dispositivo de saída	Mostra os resultados dos processos
Impressora		Dispositivo de saída	Imprime os resultados dos processos

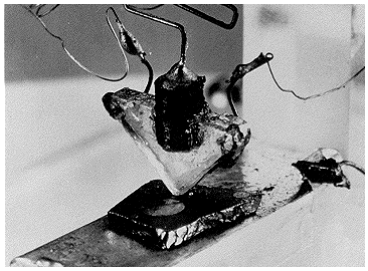
Modem		Dispositivo de entrada/saída	MOdulador-DEModulador comunicar com outros, computadores pela linha telefónica
Placa de rede		Dispositivo de entrada/saída	Comunicar com outros computadores através de uma rede
Disco rigido/disquete		armazenamento	Guardar dados e programas num formato não volátil (os dados permanecem quando se desliga o PC)
CD-ROM		dispositivo de entrada	Carregar programas ou dados na memória
Placa de som		Dispositivo de saída	Tocar música ou outros sons
Scanner		dispositivo de entrada	Digitalizar imagens

Camadas físicas e lógicas de um PC

Ao nível mais baixo que podemos chamar de **Físico**. Electrons são responsáveis pela condução electrica dos materiais. Os materiais usados
nos computadores chamam-se semicondutores, o que quer dizer que eles têm uma resistência entre os metais (como o cobre e o ouro) e os
isoladores (como o vidro e o plástico).

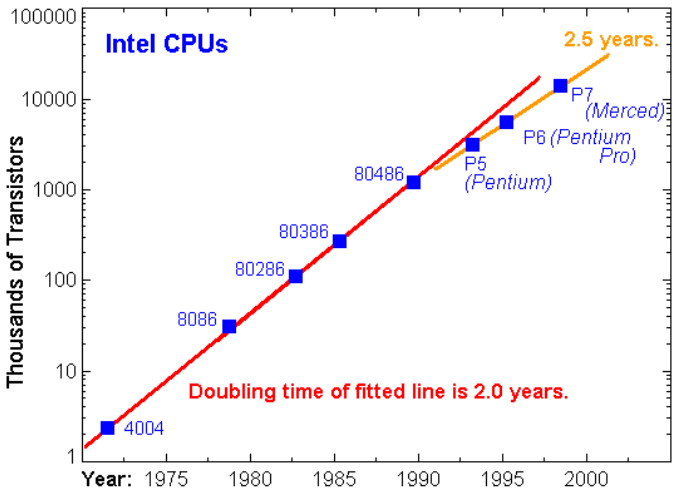
The picture shows the first

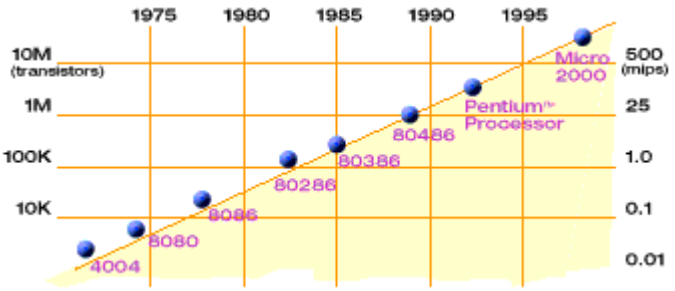
No nível seguinte temos a **Electronica**. A electrónica liga os materiais com propriedades diferentes transformados num componete para utilização.
Note que o diodo que conduz corrente numa só direcção, e o transistor,
cuja conductividade é controlada por voltagem exterior.

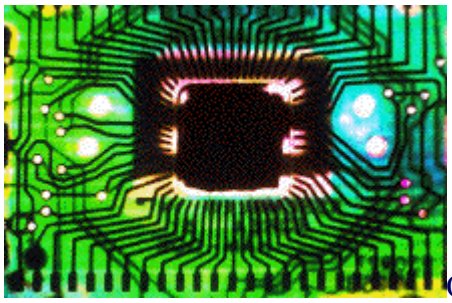


transistor, as invented at Bell Labs in 1947.

A seguir encontramos o nível da **Electrónica Digital**. São utilizadas as chamadas "gates": OR, NOR, AND, NAND, XOR, só para mencionar
algumas. Estas "gates" são feitas de componentes de electrónica básica como transistores. Gates are the cornerstone of digital computers. Temos
que nos lembrar que todos os calculos são feitos a este nível. Quando juntamos 2+2, algures no PC as gatesestão mudando e processando
calculus como "1 OR 1 = 1". Outro componente da electrónica digital é a memória. Também são feitos de transistors (e capacitadores no caso da RAM dinâmica) e podem guardar temporariamente informação.







O nível seguinte é o dos **Circuitos Integrados**. Nestes circuitos, milhões e milhões de gatesestão ligados o que permite que programas complexos corram. Desde o primeiro circuito integrado, que o número de de transistors num unico IC vem duplicando de dois em dois anos, aproximadamente. A esta regra chama-se lei de Moore que ainda é válida, embora os limites fisicos estejam á vista. Ver a figura à esquerda.

Para fazer uma comparação popular. Se o mesmo avanço se tivesse verificado na industria automovel, um carro moderno poderia andar a 5 milhões de km/h, consumia uma gota de combustivel aos 100.000 km e poderia levar sentadas 10000 pessoas.

No nível seguinte começa a programação real. Começamos com programas binaries, ou (em formato legivel) **Micro-assembler**. Isto é programação directa no processador: coloca endereço xxx no registo de endereço, active linhas de endereço, esperar xx ns, adicionar registo X ao registo Y, etc. ("registos" são pequenas unidades de memória dentro do processador)

No nível seguinte temos a **Linguagem da Máquina**. Um exemplo:
101000100000101000
O que pode significar: põe o conteudo do endereço 0000101000 no registo A. Esta linguagem é quase

imposível a ler para os humanos. Por isso, foram inventados os Macro-assembler o que já é um pouco melhor.
Mas lembra que quando um programa vai correr, o codigo da linguagem da máquina vai ser caregado (sem mais traduições) na memória and executed. Ficehiros com nomes finalizados com ".exe" e ".com" in MS-DOS and Windows são deste tipo.

Para o nível seguinte, que já se parece mais com programação real, temos o **Macro-assembler**. Neste nível nós instruimos o processador a executar os pequenos programas escritos em micro assembler. Esta é a forma

```
ADDA $2050
```

ou numa forma menos legivel (pelo menos para as pessoas);

```
1010001000000101000
```

o que significa "adicionar o conteúdo da memória do endereço $2050 ao registo A". Linguagens de programação modernas como o Pascal e C são traduzidas
para esta forma com a ajuda de compiladores.

No nível seguinte temos finalmente as **Linguagens de Programação**. Estas linguagens são regularmente chamadas de "linguagens de quarta geração", porque evoluiram das linguagens mais antigas como o assembler, etc. Muitas destas linguagens foram inventadas durante os anos 1960 e 1970. Para aplicação existe uma linguagem de programação. em 1995 existiam cerca de 2500 linguagens de programação diferentes. (para quem estiver interessado, ver http://cui_www.unige.ch/langlist). Para programadores profissionais exite o C e C++, para aplicações simples o BASIC. Para fins educacionais foi inventado o Pascal, especialmente para ensinar ideias de programação.

Presentemente, novas gerações de linguagens evoluem. Todas elas envolvem o conceito de **Programação Orientada a Objectos**, um conceito que não iremos usar nas nossas aulas, mas que se tornou indispensável nos ambientes de programação modernos. Podemos chamar-lhes a "quinta geração de linguagens"

As linguagens de programação modernas são flexiveis. Podemos escrever uma grande variedade de aplicações nestas linguagens. Podemos, por exemplo, escrever um programa que simule o funcionamento de um diodo, ou calcular um transistor ao nível fisico. Completamos assim o ciclo; podemos utilizar o computador para fazer os componentes básicos e melhores e mais rápidos computadores.

Exemplos:

BASIC	`IF A=20 THEN PRINT"Hello World"`
PASCAL	`if (a=20) then writeln('Hello World');`
C	`if (a==20) printf("Hello world\n");`
FORTRAN	`IF (A .EQ. 20) PRINT ,'Hello World'`

Não esquecer que se escrevermos em PASCAL

```
writeln('Hello world');
```

e correrrmos o programa, estamos de facto a controlar o fluxo de electrões
ao mais baixo nível. Isto podevos dar uma boa ideia de controlo ...

Compiladores

Como foi dito antes, as linguagens modernas de programação têm de ser traduzidas da forma que os utilizadores compreendem para a linguagem dos computadores. Quando escrevemos

```
  writeln('Benfica - Sporting   3 - 0');
```

Isto tem de ser traduzido em

```
  MOVAI $0102       ; carregar 'B' no registo A
  MOVAO $1245     ; mover o conteúdo do registo para a placa de video
```

ou, a um nível mais profundo

```
  00110111000111111100011110100011111
```

Para este propósito existem compiladores. Eles traduzem o que é legivel para as pessoas para a o que é executado pelo computador Quando começamos um ficheiro contendo o nosso programa meuprograma.pas, traduzimos com o compilador para um ficheiro chamado meuprograma.exe

```
  myprogram.exe
```

que pode ser executado escrevendo

```
  meuprograma
```

e o resultado aparecerá no ecran quando tudo correr correctamente

```
 Benfica - Sporting   3 - 0
```

Na maioria das versões modernas das linguagens de programação, iremos trabalhar no chamado IDE (integarted development environment - ambiente integrado de desenvolvimento), o que significa que podemos escrever um programa e com um simples toque podemos compilar o programa, ver os erros da nossa escrita, e, caso não existam erros durante a compilação, executar o programa e ver os resultados. Esses ambientes proporcionam grande velocidade de desenvolvimento de software, mas não nos devemos esquecer que o compilador está traduzindo o programa para nós. Iremos discutir os compiladores e o seu uso mais tarde nas aulas
práticas.

Sistemas Operativos

Sistemas Operativos são programas que estão constantemente a correr no nosso computador e estão interpretando os comandos que nós damos. Por exemplo, quando nós queremos "correr" o nosso programa, podemos escrever o seu nome ou clicar no seu icon, ou de qualquer outra forma. O sistema operativo vai então:

1) carregar o programa do disco para a memória
2) executa-lo

Quando o nosso programa está terminado, é removido da memória novamente, mas o sistema operativo continua a correr, esperando pela próxima instrução. De facto, um computador por si só não faz mais nada sem ser verificar se nós digitamos ou clicamos em algo. Que desperdicio de energia....

O Sistema operativo mais famoso terá sido possivelmente o MS-DOS da Microsoft. Este era uma sistema operativo em linha de comando, quer dizer, tinha que se escrever as instruções no computador através do teclado. Por exemplo

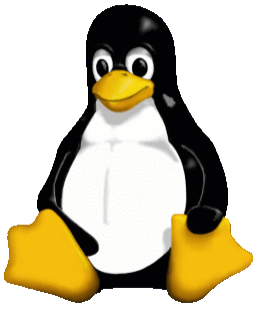
```
  DIR C:\
```

Mais tarde, uma interface gráfica foi adicionada ao MS-DOS e chamada de Windows. Por baixo dela, ainda tinhamos o mesmo sistema de linha de comandos do MS-DOS , mas os clicks do rato nos icones e objectos traduziam-se em comandos. Podiamos clicar numa 'pasta' e ver o seu conteudo. Clicar numa pasta era igual a escrever DIR mas com os resultados apresentados em modo gráfico. Ao longo dos anos, Windows tornou-se mais avançado e nos dias de hoje é um sistema operativo de multitarefa (o que quer dizer que mais do que um programa pode ser executado ao mesmo tempo) e é utilizado pela maioria das pessoas mundialmente.

Por causa do monopólio detido pela Microsoft, os 27% de acções que o co-fundador Bill Gates representam 20 biliões de dolares (20.000.000.000 dolares) em 1995. Em 2000 subiu para 65 biliões. Note

que se trata de 10 dolares por cada pessoa na terra, seja ele americano ou um chinês numa aldeia remota da China.

Alternativas para o Windows são o Unix/Linux, que tem a vantagem de ser gratuito, e o MacOS para os Macintosh.




.

Mini teste:

Para testar os conhecimentos adquiridos nesta aula, click aqui para um teste online. Note que esta não será a forma do teste final!

Peter Stallinga, Universidade do Algarve, 6 fevereiro 2002

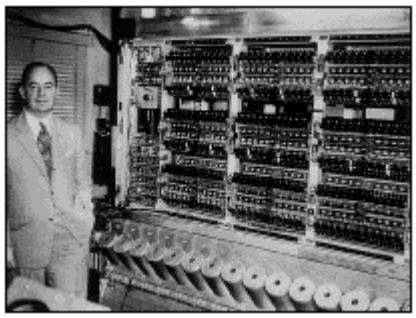
Aula 3: Unidades de Informação / Memória

Translated by Luís Pereira

Como descrito na aula anterior, a memória é uma parte essencial do computador. Guarda

- o programa
- os dados com que o programa está trabalhando

Nota: A ideia de separar o programa dos dados com que está a trabalhar e o hardware do software (ou "a máquina" e o "programa") vem de Von Neumann. Ele desenhou o primeiro computador electrónico capaz de correr um programa flexivel (1940-1952). Todos os computadores modernos são computadores Von Neumann.



Antes de escrever programas, é útil ver mais de perto a memória.

A memória está cheia de **informação**. Esta informação pode ser código de programação ou dados.

BIT

A menor quantidade de informação é um bit. Um bit pode conter a informação do tipo "TRUE ou FALSE". Por exemplo, pode conter a informação
"O aluno pagou as propinas, sim ou não?",
"O aluno pode fazer a frequência, sim ou não?",
"*x* é maior que *y*, sim ou não?".
Temos que nos lembrar que, ao nível da electrónica, o computador está calculando com estes bits de informação. Na aula anterior vimos como os componentes electrónicos digitais ("AND gates", etc) geriam estes bits de informação. Para estes componentes electrónicos, existem dois niveis: 0V e +5V (ou qualquer par de níveis de voltage). Na nossa linguagem, podemos chamar 'TRUE' e 'FALSE', ou '1' e '0', ou 'green' e 'red', ou qualquer par de nomes simbólicos que queiramos atribuir. Porque um bit de informação só pode ter dois valores, podemos chama-lhe unidade binária (**binary** unit). Como todas as unidades de informação derivam do bit, chama-mos ao computador calculadora binária. Embora se possam construir computadores baseados noutras unidades de informação (como por exemplo ternary or quaternary), todos os computadores modernos são do tipo calculadoras binárias.

Níveis electrónicosde AND gates	0 V	+5 V
binary	0	1
logical	FALSE	TRUE
bicolor	verde	vermelho

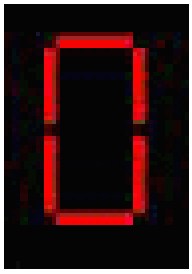
Estes principios podem ser misturados á nossa vontade. Por exemplo, PASCAL usa o principio 'TRUE' e 'FALSE' para calculos lógicos, enquanto o C usa '1' e '0'.

NIBBLE

A seguinte unidade de informação é o **nibble**. Trata-se de um conjunto de 4 bits. Nestes 4 bits podemos por exemplo guardar informação do tipo 0..9. Nibbles são usados em muitos mostradores digitais, como por exemplo relógios de alarme, em que cada digito é um nibble. Podemos chamar a este código binary-coded-decimal (**BCD**):

binary	*BCD*
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

binary	*BCD*
1000	8
1001	9
1010	not used
1011	not used
1100	not used
1101	not used
1110	not used
1111	not used



examplo dum display de LED

Pela tabela podemos ver que para o código binário

abcd

o código decimal é

a*8 + b*4 + c*2 + d

ou, mais genericamente:

$a*2^3 + b*2^2 + c*2 + d$

Esta, iremos ver, é sempre a relação entre binários e decimais.

Note também que algumas das combinações possiveis de bits não são usadas em BCD. Uma forma de representar estes quatro bits de informação com todas as combinações possiveis é o sistema **hexadecimal**. A combinação de bit de '1010' até '1111' são representadas pelas letras de A a F. Na representação hexadecimal temos a seguinte tabela de traduçã

binary	*hexa-decimal*
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7

binary	*hexa-decimal*
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

Como vemos todos os 16 binários têm um correspondente no sistema hexadecimal. O sistema hexadecimal é frequentemente usado na tecnologia dos computadores. Como exemplo: 21F no sistema hexadecimal é igual a $2*16^2 + 1*16^1 + 15*16^0 = 2*256 + 1*16 + 15*1 = 543$.

BYTE

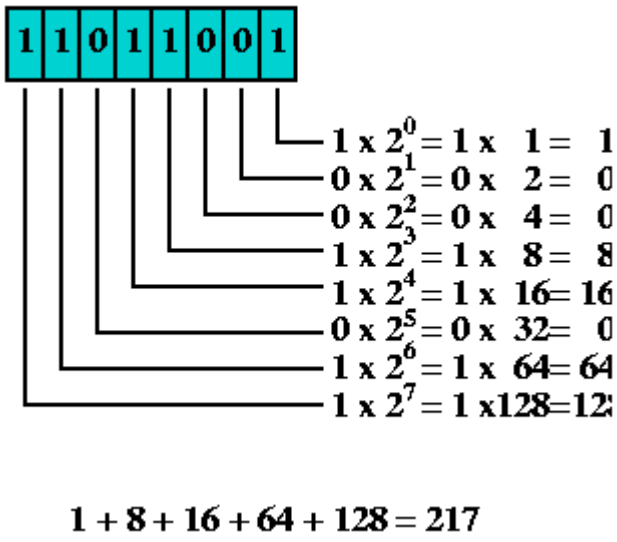
A seguinte unidade de informação é o byte. Byte é a combinação de dois nibbles e portanto de 8 bits. Nestes podemos guardar numeros de 0..255 porque

$00000000 = 0*2^7 + 0*2^6 + 0*2^5 + 0*2^4 + 0*2^3 + 0*2^2 + 0*2^1 + 0*2^0 = 0$

$11111111 = 1*2^7 + 1*2^6 + 1*2^5 + 1*2^4 + 1*2^3 + 1*2^2 + 1*2^1 + 1*2^0 =$
$= 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 =$
$= 255$

another example:

$11011001 = 1*2^7 + 1*2^6 + 0*2^5 + 1*2^4 + 1*2^3 + 0*2^2 + 0*2^1 + 1*2^0 =$
$= 128 + 64 + 0 + 16 + 8 + 0 + 0 + 1 =$
$= 217$



De outra forma, um byte pode representar todas as letras do alfabeto, em maiusculas ('A' .. 'Z') e minusculas('a' .. 'z'), mais todos os digitos '0' .. '9', alguns caractere especiais com '{', '}', '(', ')', space, etc, e outras coisas como códigos de controle. A mais usual de o fazer é através do **ASCII** (American Standard Code of Information Interchange) no qual por exemplo, 'A' é 65 (decimal), ou 01000001 (binário), ou 41 (hexadecimal). Outros exemplos são:

binary	*decimal*	*hexadecimal*	*ASCII*
01000001	65	41	'A'
01000010	66	42	'B'
01100001	97	61	'a'
00010000	32	20	' '

Memory

Em muitos computadores, o byte é a unidade mais pequena que pode ser 'endereçada'. Para percebermos esta ideia, vejamos como a memória está organizada. Imaginem a memória como uma rua (muito grande) com casas. Cada casa tem um endereço. Se quisermos colocar algo na casa, ou tirar, temos que indicar o endereço da casa. No computador, a memória toma o lugar da rua e o byte toma o lugar da casa. Em cada 'casa', vivem 8 bits, ou um byte.

Notem que as três unidades descritas até agora são também unidades de comida em Inglês. Bit, Nibble, Byte. As seguintes já não seguem este principio.

Tamanho de Memória dos Computadores

> Muitas vezes podemos ler em publicidade:
> Computador, com processador 1.7 GHz , 256 MB RAM, 40 GB harddisk, disquete 1.4 MB

1.7 GHz specifica a velocidade do processador (CPU - central processing unit). 1.7 GHz significa que pode fazer 1.7 milhões de instruções simples por segundo.
(Como a maioria dos comandos dados ao processador precisam de mais do que uma instrução, o numero efectivo de comandos por segundo é inferior. Por exemplo a adição de dois numeros de virgula flutuante pode consistir em 1) carregar o primeiro numero da memória, 2) carregar o segundo da memória, 3) adicionar o numero (possivelmente em vários passos), 4) guardar o resultado em memória. No entanto, a velocidade global do computador é determinada pela velocidade do processador e a velocidade com que ele pode carregar os dados da memória.)

Os outros numeros determinam o tamanho da memória do computador (RAM = Random-access memory), o disco e a disquete respectivamente, em numero de (B). Para dar uma ideia do que estes numeros representam, analisemos calmamente:

BYTE: A unidade básica para descrever a memória é um byte (B). Como foi dito antes, um byte é suficiente para conter uma letra, ou um numero de 0 .. 255.

KILOBYTE: 1024 bytes são um kilobyte (kB). Em ciência, 'kilo' quer dizer 1000, um numero redondo no sistema decimal. Para os computadores 'kilo' representa um pouco mais, 1024. Por ser baseado em 2^{10} que corresponde a 1024 ou em binário 10000000000. Para dar uma ideia de quanto é um megabyte: uma página de texto A4 tem aproximadamente 4 kB.

MEGABYTE: 1024 kilobytes são um megabyte (MB). É igual a 1024 x 1024 bytes, ou 1048476 bytes. Para dar uma ideia da dimensão de um megabyte: 250 páginas de texto, ou digamos um livro. A maioria das disquetes têm 1.4 MB seria suficiente para guardar um livro com 350 páginas. (só texto sem imagens etc.)

GIGABYTE: 1024 megabytes são um gigabyte (GB). Seria suficiente para guardar uma boa biblioteca com milhares de livros A maioria dos CD têm 650 MB (0.65 GB), suficiente para guardar uma pequena biblioteca

Os discos modernos têm por volta dos 40 GB. Podem armazenar uma grande biblioteca, cerca de 40.000 livros.

TERABYTE: 1024 gigabytes são um terabyte. Embora discos deste tamanho não existam ainda, algumas

empresas têm sistemas de computadores com muitos discos cujo total de espaçõ em disco é da ordem dos terabytes. Seria suficiente para guardar todos os livros do mundo.

Para dar uma ideia do espaço em disco existente no mundo imaginem: existem 500 milhões de utilizadores. Em média cada disco tem 10 GB. Faz um total aproximado de 5.000.000.000.000.000.000 bytes. Uma pessoa levaria mais de um bilião de anos para ler 10 livros por dia para ler toda essa informação!

Mini teste:

Para testar os conhecimentos adquiridos nesta aula click aqui para fazer um teste online. Atenção que esta **não** será a forma do teste final!

Peter Stallinga, Universidade do Algarve, 13 fevereiro 2002

Aula 4: Introdução à Programação

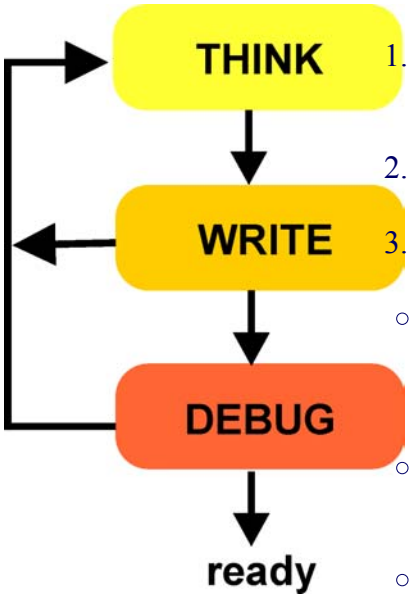
Traduzido por Ana Paula Costa

Qualquer programa não é mais do que um conjunto de instruções para o computador. O computador irá executar um comando a seguir ao outro, em princípio na ordem pela qual os comandos foram escritos (com excepção das instruções chamadas *estruturas de controlo* que serão estudadas mais tarde). O computador fará exactamente aquilo que lhe dissermos para fazer, nada mais, nada menos.

Mais ainda, é preciso dar (escrever) cada instrução com muito cuidado. O computador "compreende" apenas as coisas que lhe ensinamos a compreender.

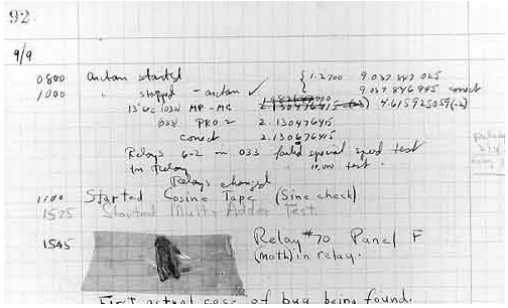
Engenharia de Software

Criar programas consiste sempre dos seguintes passos:



1. Pensar! Estudar e analisar o problema. Recolher informação. Encontrar uma solução possível. Não tocar ainda no teclado! Utilizar lápis e papel.
2. Escrever um programa. Utilizar um qualquer editor para escrever o programa.
3. Eliminar os erros do programa (fazer o "debuggin"). Existem vários tipos de erros:
   - erros de compilação. Erros de escrita do código (Por exemplo quando se escreve prinntf em vez de printf). São erros fáceis de eliminar; o compilador dá uma ajuda apresentado uma mensagem de erro como por exemplo "type-mismatch error in line 34"
   - erros de programação. Por exemplo esquecer de inicializar (atribuir um certo valor) as variáveis. Isto pode também causar os chamados "run-time errors", por exemplo "division by zero error".
   - erros de lógica (por exemplo, não se saber que a Regra de Pitágoras é $a^2 + b^2 = c^2$). O programa irá correr sem gerar erros mas o resultado não será aquele que se esperava.
4. Analisar o resultado. É isto que se pretende? Talvez o programa tenha escrito "A raíz quadrada de 9 é 4", claramente um resultado diferente do pretendido.
5. Se necessário, repetir os passos 3,2 ou mesmo 1.

Demorar mais tempo no ponto 1 significa muitas vezes poupar muito tempo nos outros passos.



The First Computer Bug

Grace Murray Hopper, working in a temporary World War I building at Harvard University on the Mark II computer, found the first computer bug beaten to death in the jaws of a relay. She glued it into the logbook of the computer and thereafter when the machine stops (frequently) they tell Howard Aiken that they are "debugging" the computer. The

very first bug still exists in the National Museum of American History of the Smithsonian Institution. Edison had used the word bug and the concept of debugging previously but this was probably the first verification that the concept applied to computers.(copiado de http://www./firstcomputerbug.html)

A linguagem de programação C

A linguagem C foi inventada nos Laboratórios Bell (Bell Labs) em 1971-9173. Foi uma evolução da linguagem B, que por sua vez foi baseada em BCPL. Em 1983 a linguagem foi standarizada e tornou-se numa versão oficial. É provavelmente a linguagem de programação mais utilizada do mundo.
A evolução do C andou de mãos dadas com a evolução do sistema operativo UNIX que vamos usar nas nossas aulas (na forma de Linux, que é uma variante gráfica do UNIX). De facto, o próprio UNIX foi escrito em C.

Um programa é uma sequência de **instruções**, que informam o computador sobre uma tarefa específica que nós queremos que ele realize.
A maioria das linguagens de programação modernas têm um formato muito legível, próximo do Inglês, tornando fácil a sua leitura e escrita por parte dos humanos. Isto contrasta com as linguagens de programação mais antigas, que eram mais próximas da linguagem que o computador compreende. Ver por exemplo a linguagem assembler (aula 2).

Um programa muito simples em C:

```
#include <stdio.h>

main()
{
  printf("Hello World\n");
}
```

Vamos dar uma espreitadela a este programa.

- Todas as linhas que começam com o símbolo # são instruções para o compilador ou lincador em vez de comandos que serão executados na altura em que o programa estiver a correr ("run-time"). #include <stdio.h> significa que a biblioteca stdio ("standard input-output") tem que ser incluída pelo compilador de forma a que possa compreender o que vem a seguir. Iremos usar o comando printf que faz parte da biblioteca stdio.
- Nós podemos escrever os nossos próprios procedimentos e funções (ver a aula sobre programação modular).
- "main" é uma função especial. A primeira instrução desta função é sempre a primeira instrução a ser executada. Nas primeiras semanas iremos escrever instruções apenas nesta função.
- Assim, a instrução `printf("HellowWorld\n")` é executada primeiro e uma vez que é a única instrução o programa termina. A instrução escreve o texto `Hello World` no écran e muda para a próxima linha (\n).

Palavras Reservadas em ANSI C

auto	float	static

break	for	struct
case	goto	switch
char	if	typedef
continue	int	union
default	long	unsigned
do	register	void
double	return	while
else	short	
extern	sizeof()	

De notar que existe apenas uma função definida na linguagem C, a função `sizeof()`. Todas as outras funções são descritas nas chamadas bibliotecas. Por exemplo, a função printf pode ser encontrada na biblioteca stdio. Assim sendo, para o compilador reconhecer a função printf é necessário colocar a directiva `#include <stdio.h>` no início do código.

Identificadores

Identificadores, como o próprio nome indica, são utilizados para identificar coisas. Podem ser nomes de **funções** e nomes de **variáveis**. Iremos ver isso nas próximas aulas. Como na maioria das linguagens, os nomes dos identificadores em C têm algumas restrições:

- Devem começar por uma letra; "20hours" não é permitido.
- Seguido de qualquer combinação de letras, dígitos ou o caracter "_" ("underscore").
- Espaços não são permitidos, nem caracteres como "(", "{", "[", "%", "#", "?", etc, com excepção de "_". A razão de ser desta restrição tem a ver com o facto destes caracteres serem utilizados para outros fins em C. São chamados **caracteres reservados**.

```
{ } [ ] ( ) - = + / ? < > . , ; : ' " ! @ # $ % ^ & * ~ ` \ |
```

- Identificadores não podem ser iguais a **palavras reservadas** (e funções pré-definidas) do C, tais como "main" ou "int". De notar que identificadores como "main1", "main_" or "Main" são permitidos, embora seja aconselhável evitar este tipo de nomes que podem criar confusão. Nota: Em muitos ambientes de programação, é fácil de perceber quando se está a utilizar uma palavra reservada, porque esta irá mudar de cor quando for escrita.
- É importante escolher bem os identifacadores. Quando uma variável é usada para armazenar uma taxa de interesse, deve-se chamar, por exemplo "interesse" e não "variavel1". Apesar de não ser errado chamar-lhe "variavel1", é muito mais inteligente dar-lhe um nome com mais significado. Isto ajuda as pessoas a compreenderem o programa (e até mesmo o próprio criador do programa quando tem que voltar a trabalhar nele depois de muito tempo).
- O tamanho mínimo dos identificadores é 1, e o tamanho máximo é 255. Deve-se fazer uso desta possibilidade de utilizar nomes longos, mas convém não esquecer que os nomes demasiado longos podem tornar o programa ilegível. É preciso encontrar um meio termo. Qual dos exemplos seguintes é melhor?

```
  r = r + a;
  money = money + interest;
  themoneyintheaccountofpersonwithnameJohnson =
themoneyintheaccountofpersonwithnameJohnson +
thecurrentinterestrateatthetimeofthiswriting;
```

- A linguagem C distingue entre minúsculas e maiúsculas, é "case sensitive", "i" não é igual a "I", etc. Para tornar os programas mais legíveis, deve-se seguir a mesma convenção durante todo o programa. A convenção mais utilizada é minúsculas para variáveis e MAIÚSCULAS para CONSTANTES.

Programação Estruturada

O mais importante em programação é escrever programas claros, lógicos e estruturados.

- Deve-se utilizar **nomes** com significado para as variáveis, procedimentos e funções.
- Utilizar a **indentação**. Comparando os dois programas que se seguem:

```
#include <stdio.h> main() {printf("Hello world!\n")}
```

e

```
#include <stdio.h>

main()
{
  printf("Hello world!\n")
}
```

Ambos os programas fazem exactamente o mesmo, mas o segundo é muito mais legível. A diferença está

  - Em colocar apenas uma instrução por linha.
  - Em usar a indentação. Colocar mais (2) espaços no início da linha sempre que se avança um nível nas estruturas.
  - Em separar blocos de texto (funções e procedimentos) com linhas em branco.
- Evitar o uso da instrução "**goto**". Com estas instruções, o programa rapidamente fica a parecer "esparguete". Enquanto em BASIC (Beginner's All-purpose Symbolic Instruction Code) a utilização da instrução GOTO é inevitável, em qualquer auto-respeitável linguagem, a instrução goto deve ser evitada.
- **Comentário**. Uma vez que o C é praticamente Inglês, o programa deveria explicar-se a si mesmo. Ainda assim, em situações em que a ideia do programa possa não ser clara para o programador, deve-se usar comentários. Em C os comentários são colocados após `//` numa linha única, ou entre `/*` e `*/` para comentários que ocupem várias linhas.
- Utilizar **funções** sempre que isso torne o texto mais organizado. Se em muitos locais diferentes o programa tem que fazer basicamente o mesmo (por exemplo ler uma linha de texto de um ficheiro), deve-se considerar colocar esse código num procedimento ou função (por exemplo FileReadLn()). Isto torna o programa mais legível, mais eficiente e mais curto.

```
            main(l
       ,a,n,d)char**a;{
  for(d=atoi(a[1])/10*80-
 atoi(a[2])/5-596;n="@NKA\
CLCCGZAAQBEAADAFaISADJABBA^\
SNLGAQABDAXIMBAACTBATAHDBAN\
ZcEMMCCCCAAhEIJFAEAAABAfHJE\
TBdFLDAANEfDNBPHdBcBBBEA_AL\
 H E L L O,     W O R L D! "
   [l++-3];)for(;n-->64;)
      putchar(!d+++33^
           l&1);}
```

Olhe, neste exemplo da linguagem C aqui acima. Isto é um exemplo de mal programação. Percebe o que é que o programa faz? Mesmo os especialistas profesionalistas não conseguem determinar o que é que o programa faz. Se quiser saber o que é que o programa faz, clique aqui.
(programa copiado de http://www.ioccc.org/)

Teste Rápido:

Para testar os conhecimentos sobre o que aprendeu nesta aula, prima aqui para fazer um teste on-line. De notar que este **Não** é o formato que será utilizado no teste final!

Peter Stallinga. Universidade do Algarve, 14 outubro 2002

Aula 5: Variáveis

Traduzido por Ana Paula Costa

Tipos de Variáveis

As variáveis armazenam valores e informação. Elas permitem que os programas efectuem cálculos e armazenem os resultados para utilização futura. Imaginemos uma variável como uma caixa que pode conter informação. Quando se deseja saber essa informação, abre-se a caixa e lê-se o valor. No final, volta-se a colocar a informação na caixa, onde fica até ser necessária novamente.

Para facilitar a identificação, e para evitar confusão, cada caixa tem que ter um nome, um identificador (ver aula 4).

O tipo da caixa define o seu tamanho e o tipo de informação que se pode encontrar lá dentro. As variáveis podem ter muitos tamanhos e formas.

As variáveis podem armazenar números, nomes, textos, etc. Nas versões modernas de C existem muitos tipos básicos de variáveis. A implementação exacta dos tipos de variáveis depende do compilador. Para o compilador Borland C temos:

grupo	*tipo de variável*	*intervalo*	*espaço ocupado em memória*
II	unsigned char	0 .. 255	8 bit = 1 byte
II	char	-128 .. 127	8 bit = 1 byte
II	int	–32 768 .. 32 767	16 bit = 2 bytes
II	unsigned int	0 .. 65 535	16 bit = 2 bytes
II	long	–2 147 483 648 .. 2 147 483 647	32 bit = 4 bytes
II	unsigned long	0 .. 4 294 967 295	32 bit = 4 bytes
III	float	–3.4E–38 .. 3.4E38	32 bit = 4 bytes
III	double	–1.7E–324 .. 1.7E308	64 bit = 8 bytes
III	long double	–3.4E–4932 .. 1.1E4932	80 bit = 10 bytes
IV	char	#0 .. #255 ASCII	8 bit = 1 byte

Notas:

- A convenção de escrever expoentes na notação científica: 2.9E−39 significa 2.9 x 10^{-39}, 1.7E308 significa 1.7 x 10^{308}, etc.
- Em C não existe um tipo de variável para informação booleana (bit). Sempre que é necessário utilizar informação booleana, ela é implementada utilizando inteiros, onde "falso" é igual a 0 e todos os outros valores são "verdadeiro".
- char é utilizado (ou pode ser utilizado) tanto para informação ASCII (texto) como para pequenos números inteiros (0..255 ou -128..127).

I **Boolean** é usado para armazenar e manipular informação do tipo verdadeiro- ou-falso ou sim/não. Como se viu na aula 3, isto representa um bit de informação. Em C este tipo de variável não existe. É emulada com inteiros, onde "falso" é igual a 0 e qualquer outro valor é igual a "verdadeiro".

II **int**, **unsigned int**, e **long** e **char** e **unsigned char** todos armazenam valores de números completos. Podem ser usados para coisas que podem ser contadas; número de pessoas numa sala, número de portas de um carro, número de telefonemas que alguém fez, ano lectivo, dia do mês, etc. **Unsigned char** e **int** apenas armazenam números positivos, enquanto **int** e **long** armazenam números positivos e negativos, até ao limite do valor máximo. O **char** ocupa o mínimo de memória, apenas 8 bits, mas o intervalo de valores que pode assumir é por isso muito limitado, apenas 256 valores diferentes. Se é necessário armazenar valores maiores, é preciso utilizar **int**. Se se desejar armazenar números ainda maiores e utilizar valores positivos e negativos utiliza-se **long**.

Nota: dado que um **char** tem 8 bits, pode armazenar 2^8 = 256 números diferentes: 0 .. 255 ou -128..127. Os mesmos cálculos podem ser feitos para as unidades de 16-bits **int** e **unsigned int**: 2^{16} = 65536, números de 0 .. 65535 para **unsigned int**, e -32768 .. 32767 para **int**. Recordem os cálculos da aula 3.

III **Float**, **double** e **long double** são exemplos de variáveis que podem armazenar números de vírgula flutuante ("floating point numbers"), por exemplo 3.1415926535. São usados para coisas não contáveis, como o comprimento de um carro, o intervalo de tempo entre dois acontecimentos, a altura de um edifício, a raíz-quadrada de 3, etc. O tipo mais pequeno é **float**, que ocupa apenas 4 bytes, com o custo de se conseguir uma menor precisão nos cálculos. O melhor é **long double**, com 80 bits (10 bytes), os cálculos terão uma precisão muito elevada, mas demorarão mais tempo e acuparão mais espaço em memória. Um bom meio-termo é usar **double**.

IV O último tipo é usado para armazenar texto. **Char** é utilizado para um único caracter do código

ASCII. Uma vez que char pode ser utilizado para armazenar informação do tipo integer, bem como caracteres ASCII, é preciso ter muito cuidado para não cometer erros.

Mais tarde iremos aprender como definir os nossos próprios tipos de variáveis, por agora vamos ver como se podem usar os tipos básicos em C.

Variáveis: Declaração!

Na maioria das linguagens compiladas modernas, todas as variáveis que se irão usar têm que ser declaradas primeiro. A isto chama-se **declaração**. Para ser mais preciso, declarar significa reservar espaço em memória e associar-lhe um nome, de tal modo que mais tarde se pode utilizar o nome em vez do endereço de memória quando se desejar recuperar a informação.
Em C declaram-se as variáveis escrevendo o nome da variável precedido pelo tipo da variável. Por exemplo

```
int i;
```

O local de declaração de variáveis é no início da função (main), antes da primeira instrução. Por exemplo:

```
main()
{
  int i;
  float f;
  long li;

  instruction1;
  instruction2;
}
```

> Sempre que o programa corre e uma instrução como `int i;` é encontrada, é reservado espaço em memória e o endereço desse espaço é relembrado para referência futura, quando for preciso armazenar informação nessa caixa ou recuperar o valor nela guardado. O espaço é libertado no fim da função (ou programa no caso de `main`).
> *Para os peritos: As variáveis são colocadas na stack e ficam lá até que o scope das variáveis expire.*

Se se desejar definir várias variáveis do mesmo tipo, pode-se fazê-lo numa única linha com as variáveis separadas por vírgulas. Embora não seja proibitivo usar várias linhas para definir diferentes variáveis do mesmo tipo.

```
main()
{
  int i, j, k;
  float f;
  float g;
  int n;

  instruction1;
  instruction2;
}
```

Esquecer de declarar variáveis origina um erro de compilação.

Problemas com variáveis I: Mistura de tipos

É preciso ter sempre muito cuidado quando se realizam cálculos de diferentes tipos de valores.
Considerando o código que se segue:

```
main()
{
  int i, j;
  float f;

  i = 1;
  j = 3;
  f = i/j;
  printf("%f", f);
}
```

Qual será o valor de f no final? Por outras palavras, qual será o resultado do programa?
NÃO 0.3333 como seria de esperar. Em vez disso, o valor de f será 0.0000. Isto deve-se ao facto do cálculo da divisão ser feita com valores inteiros, e 1 dividido por 3 na divisão inteira é 0, que é o valor atribuído a f.
Para evitar esta situação pode-se

* forçar o tipo do valor nos cálculos. A isto chama-se "type casting". Por exemplo:
```
f = (float) i/ (float) j;
```

* Quando os cálculos são efectuados com constantes, pode-se forçar o tipo directamente no valor:
```
f = 1.0 / 3.0;
```
instead of
```
f = 1 / 3;
```

Problemas com variáveis II: Overflow

Considerando o seguinte programa:

```
main()
{
  int i, j, k;

  i = 20000;
  j = 20000;
  k = i + j;
  printf("%d", k);
}
```

Mais uma vez, qual será o resultado do programa? Ingenuamente, podemos pensar que será 40000, mas `k` é do tipo int e os inteiros vão apenas até 32767. Que é feito com o resto? O que aconteçe é que a parte restante ("overflowing part") reentra no limite inferior do intervalo. Neste caso iremos obter -25536. Um overflow pode ocorrer quando se adicionam dois números com intervalo limitado. Um underflow pode acorrer quando se subtrai 2 números.
Isto é mais fácil de demonstrar com uma variável que ocupe apenas um byte (um byte tem 8 bits e um intervalo de 0 a 255 e é chamado `unsigned char` em C):

```
i     130 =    10000010
j     130 =    10000010
i+j  (260)=   100000100
```

O que acontece é que o 9.º bit da equação anterior é ignorado e o resultado será o valor binário `00000100` que é igual a 4 no sistema decimal. Assim, calculando usando um byte, 130 + 130 = 4.

O que pode ser feito para prevenir esta situação?

- Utilizar sempre tipos de variáveis com tamanho suficiente para armazenar todos os resultados possíveis dos cálculos. No exemplo anterior dever-se-ia ter usado (no mínimo) inteiros. No primeiro exemplo dever-se-ia ter usado long int.
- Algumas linguagens permitem verificar isto em "run time". A isto chama-se "range checking", que também verifica "array index out of bounds" como iremos ver na aula sobre arrays.

Problemas com variáveis III: Inicialização

Declarar uma variável não lhe atribui nenhum valor, apenas lhe reserva espaço em memória!

```
main()
{
  int day;

  printf("Today is day %d\n", day);
}
```

No programa de cima, o resultado *poderá* ser

```
Today is day 23741
```

Quando o computador é ligado, normalmente a memória é preenchida com zeros, mas passado algum tempo, depois de muitos programas terem usado a memória e deixado aí algum lixo, o conteúdo de um endereço de memória é imprevissível. Para garantir que se está a trabalhar com valores bem definidos é sempre preciso atribuir um valor a cada variável. Na próxima aula iremos aprender como fazer isso num programa utilizando instruções de atribuição. Por agora é suficiente disser o seguinte: "Nunca se deve assumir que o valor inicial das variáveis é 0".
Nota: Em muitas linguagens de programação é possível atribuir um valor a uma variável no momento da sua declaração. Para se fazer isso em C pode-se usar

```
int day = 0;
```

---

printf

A instruçao `printf` é uma das instruções mais úteis em C e é utilizada para mostrar informação no écran: texto, números, valores de variáveis, etc. Qualquer programa tem uma instrução `printf` algures. Já se mostraram alguns exemplos da utilização de printf, mas até agora ainda não de discutiu o funcionamento desta função.

`printf` faz parte da biblioteca standard-input-output (`stdio.h`). O resultado é formatado, o que significa que se pode especificar exactamente como queremos que a informação surja no écran (quanto espaço deve ocupar no écran, quantos dígitos significativos devem ser mostrados para floats, etc.). O formato é sempre o primeiro argumento da função. O(s) argumento(s) seguinte(s) respeitam à informação a ser mostrada, separada por vírgulas.

```
printf("%d", i);
```

irá mostrar o valor de uma variável do tip integer a que se chamou `i`

```
3
```

para cada informação tem que se especificar o formato:

```
printf("%d %d", i, j);
3 5
```

Outros especificadores de formatos com interesse:

%d	inteiro com sinal

%i	inteiro com sinal
%u	inteiro sem sinal
%x	hexadecimal
%f	float (31.2000)
	por exemplo %10.3f mostra o float com 10 espaços no écran e 3 casas decimais
%e	float com notação científica (3.12E1)
%c	caracter ASCII
%s	string

De notar que também se pode escrever texto directamente no formato:

```
printf("Hello%d World", i);
```

dará como resultado (quando i=3)

```
Hello3 World
```

Existem caracteres especiais que se podem utilizar para controlar a posição do resultado no ecrán. Eles são escritos com \. O mais importante é:

\n	vai para o início da próxima linha

Por exemplo

```
printf("Hello\n%d World", i);
```

dará como resultado (quando i=3)

```
Hello
3 World
```

Teste Rápido:

Para testar os conhecimentos sobre o que aprendeu nesta aula, prima aqui para fazer um teste on-line. De notar que este **Não** é o formato que será utilizado no teste final!

Peter Stallinga. Universidade do Algarve, 22 October 2002

Aula 6: Atribuição, `scanf` e cálculos

Traduzido por Ana Paula Costa

Atribuição

Na aula anterior (aula 5) viu-se como se podem definir variáveis. Agora vamos ver como se pode atribuir um valor a uma variável.

Dar um novo valor a uma variável chama-se **atribuição**. Em C isto é feito com o operador

Do lado esquerdo do operador coloca-se a variável, e do lado direito coloca-se o novo valor ou expressão que irá produzir o valor **do mesmo tipo** que a variável. Exemplos:

```
a = 3.0;
b = 3.0*a;
c = cos(t);
```

Errado (assumindo `a` do tipo float):

```
1.0 = a;
a = TRUE;
a = 1;
```

Na verdade, em C o último exemplo está correcto. O C sabe o que queremos e dá uma ajuda convertendo o inteiro 1 no real 1.0. De qualquer forma, é má prática mistura inteiros com reais e será melhor escrever:
`a = 1.0;`

O símbolo = deve ser distinguido do símbolo matemático =. Nesta fase é interessante fazer-se uma comparação entre o símbolo matemático = e o símbolo de atribuição nas linguagens de programação (= em C). Como exemplo consideremos a seguinte equação matemática

$a = a + 1$

Isto, como todos sabemos, não contem uma solução para a. (Em comparação, outro exemplo: $a = a^2 - 2$, contem uma solução, nomeadamente $a = 2$).
Contudo, nas linguagens de programação deve-se ler a equação de forma diferente:

```
a = a + 1;
```

significa

(o novo valor de a) (será) (o antigo valor de a) (mais 1)

Ou, por outras palavras: primeiro o valor de `a` é carregado da memória, depois é-lhe adicionado 1, e finalmente o resultado é colocado de novo em memória. Isto é fundamentalmente diferente da equação matemática. Na maioria das linguagens modernas o símbolo `=` é utilizado, o que é confuso, especialmente para os programadores novatos. Por essa razão os estudantes são aconselhados a pronunciar o símbolo de atribuição como "torna-se" ou "será" em vez de "é" ou "iguala".

O símbolo = também deve ser distinguido do símbolo de comparação =. Por exemplo `a=b` ? Mais tarde, iremos aprender como fazer comparações (na aula de "if .. then" e "álgebra booleana"), para a qual se utilizará o símbolo `==`.

Para sumariar, o símbolo = é
NÃO parte de uma comparação ("a é igual a b?")
NÃO parte de uma equação ("$a^2 = 2a - 1$")
É uma atribuição ("a transforma-se em b")

Exemplo

O que se segue mostra um exemplo de um programa que utiliza atribuições, variáveis e constantes. O lado direito mostra o valor das variáveis após cada linha

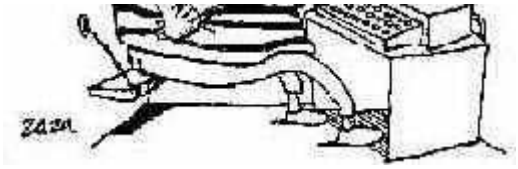
	valor de `a` depois da execução da linha	valor de `b` depois da execução da linha

`main()`
`{`
`  float a;`
`  float b;`
`  float c = 4.3;`
`  a = 1.0;`
`  b = c;`
`  a = a + b + c;`
`}`
	`indefinido`
	`1.0`
	`1.0`
	`9.6`
		`indefinido`
		`indefinido`
		`4.3`
		`4.3`

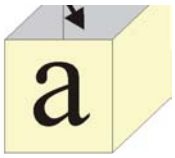
Input formatado: `scanf`

Nas aulas anteriores aprendemos como mostrar os resultados dos cálculos do programa no écran. Com `printf` podemos fazer com que o nosso programa tenha *output*. Em muitos casos, nós também queremos que o nosso programa tenha *input*. O utilizador digita o seu nome, ou introduz números que o programa tem que processar. Ou ainda mais simples, queremos que os utilizadores tenham controlo do programa. Por exemplo, queremos que o utilizador possa parar o programa carregando na tecla escape.

C: com `scanf` podemos obter informação a partir do teclado. Podem-se ler



valores e caracteres do teclado e armazená-los directamente em variáveis especificadas. O formato geral da instrução é



```
scanf("descrição(ões)
 de formato", &var1,
     &var2, ...);
```

De notar o símbolo `&`. Isto significa "o endereço de ...". É preciso fornecer o endereço da(s) variável(veis) para a função `scanf`. Como veremos mais tarde, fornecer o endereço de uma variável é igual a "passar por referência" o que permite que a função possa alterar o valor da variável. De momento, é suficiente dizer que temos que fornecer os endereços das variáveis no `scanf`. Na analogia com as caixas, pode-se ver o fornecer do endereço como dar a caixa à função que depois a preenche com um valor.

As descrições de formato são as mesmas que as da instrução de output `printf`. Por exemplo `%d` para inteiros, etc. De notar que as variáveis a serem lidas não têm que ser do mesmo tipo.

Como por exemplo

```
// lines starting with these // are cooment
// always when we use input or output
// we have to include the standard-I/O library:
#include <stdio.h>

main()
{

// don't forget to declare the variables we will use
  int n1, n2: integer;

  printf("Please enter two numbers separated by a space\n");
  scanf("%d", &n1);
  scanf("%d", &n2);
  printf("n1 = %d  n2 = %d", n1, n2);
}
```

Quando correr, o programa mostrará a mensagem

```
Please, enter two numbers separated by a space
```

O utilizador deve digitar os dois números

128 31<return>

(Atenção às convenções que são utilizadas no texto: coisas que o utilizador introduza (digite) aparecem em *green italics*, e `<return>` significa pressionar a tecla enter).

Depois o programa mostra

```
n1 = 128  n2 = 31
```

Operações, operadores e operandos

Depois de aprender como atribuir valores, podemos agora ver como efectuar cálculos com os nossos programas. As operações mais básicas são a adição, subtracção, multipicação e divisão:

operador	operação

+	adição
–	subtracção
*	multiplicação
/	divisão

Estes quatro *operadores*, quando utilizados em cálculos, necessitam de dois *operandos* e são chamados operadores binários. Em C, um operando é colocado do lado esquerdo e outro do lado direito do operador. Como exemplos:

correcto	*errado*
`3 * a`	`* a`
`a + b`	`3 a +`

Estas expressões resultarão em valores que podem ser atribuídos a variáveis como vimos anteriormente: Como um exemplo:

```
c = 3 * a;
```

De notar novamente, do lado esquerdo do símbolo de atribuição = temos a variável e do lado direito colocamos a nossa expressão que irá resultar num novo valor para a variável.

Em C existem também operadores unitários que necessitam apenas de um operando:

operador	operação
`++`	incremento
`--`	decremento
`!`	negação (álgebra booleana)
`~`	inversão de todos os bits deste inteiro
`-`	negação do número
`&`	endereço de ..

Por exemplo

```
i++;
--k;
```

Incrementa o valor de i e decrementa o valor de k em 1, respectivamente. O operador pode ser colocado antes ou depois do operador. A diferença está no momento em que a variável muda o seu valor, antes da execução do resto da instrução (`++i`) ou depois (`i++`). Por exemplo

`i = 1;`
`j = i++;`
 	
no final
i é igual a 2
j é igual a 1
	`i = 1;`
	`j = ++i;`
	
	no final
	i é igual a 2
	j é igual a 2

Os operadores ! e ~ serão estudados depois na aula sobre álgebra linear.
O operador obtem-endereço será discutido na aula sobre ponteiros.
O operador unário - retorna o número negado (floating point ou inteiro).

Combinações de cálculo e atribuição

Em C existe a peculiar e confusa possibilidade de usar cálculo e atribuição ao mesmo tempo. Por exemplo:

```
c += a;
```

que é equivalente a

```
c = c + a;
```

Qualquer operador binário pode ser utilizado desta forma (`*=`, `/=`, `+=`, `-=`).
Isto pode ser confuso e como tal deve ser evitado. Por outro lado, pode prevenir erros. Por exemplo:

```
matrixelements[i+2*k-offset][n+m-f] = matrixelements[i+2*k-offset][n+m+f] +
matrixelements[i+2*l+offset][n+m-f]
```

É fácil de cometer um erro de escrita aqui. Na verdade existe um erro. Localizou-o?
Mais simples e mais legível seria:

```
matrixelements[i+2*k-offset][n+m-f] += matrixelements[i+2*l+offset][n+m-f]
```

Matemática de Inteiros

Os operadores abordados na secção anterior são usados para cálculos reais (floating point calculations). Para cálculos inteiros (usada para tipos como byte. word, integer, longint, etc.), o operador da divisão funciona de forma um pouco diferente. Vamos imaginar o cálculo de, por exemplo, 7/3. Como se aprendeu na escola primária, isto é igual a **2** com resto de **1** a ser dividido por 3:

$$\frac{7}{3} = 2 + \frac{1}{3}$$

Em C existem dois operadores / e % que reproduzem estes resultados. Exemplo:

expressão	resultado
`7 / 3`	2
`7 % 3`	1

Estes substituem o operador floating-point /. Os outros três operadores (*, +, -) são os mesmos para números inteiros.

Prioridade

No caso de existir mais do que um operador numa expressão, as regras normais da matemática são aplicadas para determinar qual é avaliada primeiro. Os operadores de divisão e multiplicação têm a precedência mais alta. Assim, quando se escreve

```
a = 1 + 3 * 2;
```

O resultado será 7.
Se se pretender alterar a ordem pela qual os operadores são avaliados, colocam-se parentesis (e). Assim, por exemplo

```
a = (1 + 3) * 2;
```

irá resultar em 8. Colocar parentesis não custa!

```
  a = (1 + 3) - (4 + 5);
```

Exemplos

```
main()
 /* an example floating point calculation */
{
  double x, y, sum, diff, divis;

  printf("Give the value of the first variable x:\n");
  scanf("%f", &x);
  printf("Give the value of the second variable y:\n");
  scanf("%f", &y);
  sum = x + y;
  printf("The sum of %f and %f is %f\n", x, y, sum);
  diff = x - y;
  printf("The difference between %f and %f is %f\n", x, y, diff);
  divis = x / y;
  printf("%f divided by %f is %f\n", x, y, divis);
}
```

irá produzir ao correr:

```
Give the value of the first variable x:
3.4
Give the value of the second variable y:
1.8
The sum of 3.4000 and 1.8000 is 5.2000
The difference between 3.4000 and 1.8000 is 1.6000
3.4000 divided by 1.8000 is 1.8889
```

```
main()
 /* an example integer calculation */
 /* note the different format specifiers in printf and scanf */
{
  int x, y, sum, diff, divis, divr;

  printf("Give the value of the first variable x:\n");
  scanf("%d", &x);
  printf("Give the value of the second variable y:\n");
  scanf("%d", &y);
  sum = x + y;
  printf("The sum of %d and %d is %d\n", x, y, sum);
  diff = x - y;
  printf("The difference between %d and %d is %d\n", x, y, diff);
  divis = x / y;
  divr = x % y;
  printf("%f divided by %d is %d plus %f/%d\n", x, y, divis, divr, y);
}
```

irá produzir ao correr:

```
Give the value of the first variable x:
13
```

```
Give the value of the second variable y:
5
The sum of 13 and 5 is 18
The difference between 13 and 5 is 8
13 divided by 5 is 2 plus 3/5
```

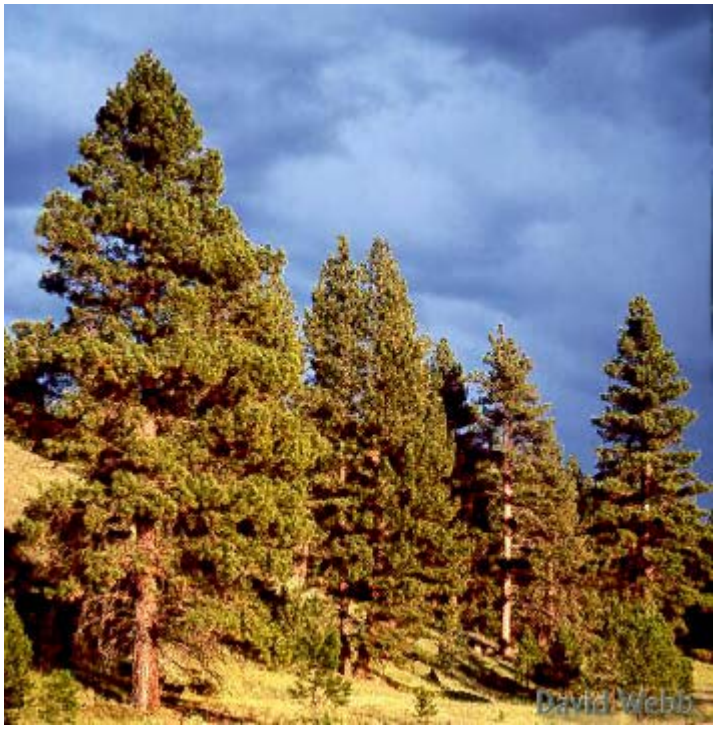
Teste Rápido:

Para testar os conhecimentos sobre o que aprendeu nesta aula, prima aqui para fazer um teste on-line. De notar que este **Não** é o formato que será utilizado no teste final!

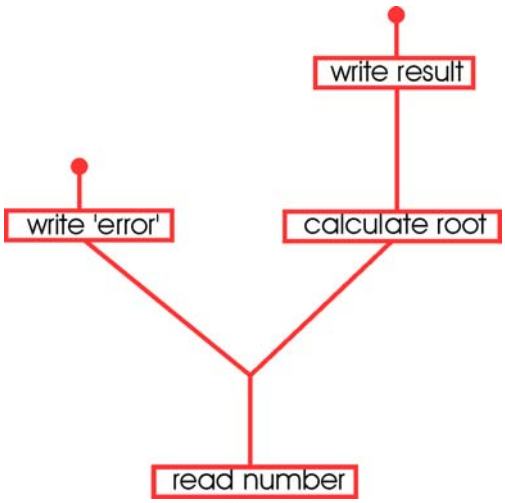
Peter Stallinga. Universidade do Algarve, 10 October 2002

Aula 7: Selecção/Ramificação I (`if` ... , `if` ... `else` ...)

Traduzido por Ana Paula Costa



Até agora, todas as instruções que se colocaram num programa foram executadas. Mais ainda, elas foram executadas exactamente na ordem em que foram colocadas. A primeira linha do programa foi executada primeiro, depois a segunda, depois a terceira, e por aí fora. Nem sempre é assim. Com a *ramificação* (um *ramo* faz parte de uma árvore) podemos controlar o fluxo de um programa.



Vamos imaginar um programa que recebe um número e calcula o raíz quadrada desse número. Calcular a raíz quadrada de um número negativo não faz sentido (a não ser que se esteja a trabalhar com número complexos, claro), e então queremos que o programa gere um erro e páre quando o utilizador introduzir um número negativo. Deverá surgir no écran uma mensagem como:

```
Negative numbers are not allowed!
```

Obviamente, nem sempre queremos que esta mensagem surja no écran; no caso do utilizador introduzir um número positivo, queremos que seja calculada a sua raíz quadrada e que o resultado apareça no écran:

```
The square-root of 5 is 2.23607
```

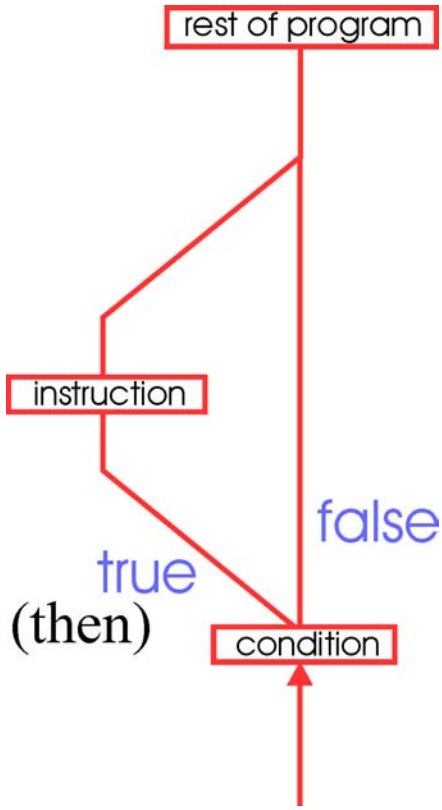
É preciso ter uma forma de verificar o número e dependendo do resultado, executar partes do programa.

---

if ...

A maneira mais simples de ter o controlo sobre as instruções que serão executadas é utilizando a instrução if ... A sintaxe completa da instrução é

```
if condição
  instrução;
```



Em `condição` iremos colocar a nossa condição e em `instrução` colocamos a(s) nossa(s) instrução(ões) que serão executadas se e só se a condição for verdadeira.
A `condição` é uma expressão que irá resultar num valor do tipo booleano (ver aula 4). Pode ser uma variável, por exemplo, sendo `b` declarado como `int`, o seguinte está correcto:

```
if b instrução;
```

Por outro lado, são mais comuns condições com expressões que comparam variáveis, tais como:

```
if (x == y) instrução;
if (x < y) instrução;
```

comparação	significado
`(a == b)`	`a` igual a `b`
`(a != b)`	`a` diferente de `b`
`(a < b)`	`a` menor que `b`
`(a > b)`	`a` maior que `b`
`(a <= b)`	`a` menor ou igual a `b`
`(a >= b)`	`a` maior ou igual a `b`

Não esquecer que se se pretender executar mais do que uma instrução, elas devem ser agrupadas entre { ... } , de maneira a serem interpretadas pela instrução if como uma só.

```
if (a == b)
  {
    instrução1;
    instrução2;
  }
```

Neste caso, a `instrução1` e a `instrução2` serão executadas quando a for igual a b.

A execução normal do programa irá prosseguir após o bloco de instruções. No exemplo seguinte, a `instrução3` e a `instrução4` serão executadas independentemente da condição (a = b).

```
if (a == b)
  {
    instrução1;
```

Ao ser executado:

(a igual a b)	(a diferente de b)

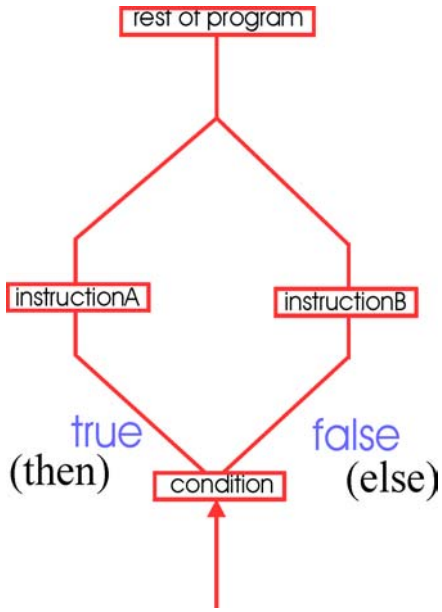
```
    instrução2;
  }
instrução3;
instrução4;
```

instrução1 instrução2 instrução3 instrução4	 instrução3 instrução4

De notar que a analogia com a ramificação das árvores termina aqui. Numa árvore, os ramos nunca se voltam a encontrar novamente; uma vez num ramo, não é possível voltar ao tronco principal.

`if ... else ...`

Se também se quiser que o programa faça coisas no caso da condição não ser verdadeira deve-se utilizar a instrução `if ... else`. A forma geral desta instrução é




```
if condição
  instruçãoA;
else
  instruçãoB;
```

exemplo:

```
if (a==b)
  {
    instrução1;
    instrução2;
  }
else
  {
    instrução3;
    instrução4;
  }
instrução5;
instrução6;
```

Ao ser executado:

(a igual a b)	(a diferente de b)
instrução1 instrução2 instrução5 instrução6	instrução3 instrução4 instrução5 instrução6

Aqui está o programa completo que demonstra o uso da estrutura de selecção para calcular a raíz quadrada de um número:

```
#include <stdio.h>

#include <math.h>

main()
{
   double x;
   double root;

  printf("Give a number");
  scanf("%f", &x);
  if (x<0)
    printf("Negative numbers are not allowed!\n");
  else
    {
      root = sqrt(x);
      printf("The square-root of %0.4f is %0.4f\n", x, root);
```

```
    }
  printf("Have a nice day\n");
}
```

Correndo o programa; dois exemplos:

```
Give a number
3.68
The square-root of 3.6800 is 1.9183
Have a nice day
```

```
Give a number
-3.68
Negative numbers are not allowed!
Have a nice day
```

Aula 8: ... de raízes e ramos 

Teste Rápido:

Para testar os conhecimentos sobre o que aprendeu nesta aula, prima aqui para fazer um teste on-line. De notar que este **Não** é o formato que será utilizado no teste final!

Peter Stallinga. Universidade do Algarve, 10 October 2002

Aula 8: Selecção/Ramificação II / Álgebra Booleana

Traduzido por Ana Paula Costa

Álgebra Booleana

George Boole (1815-1864)

English mathematician. His work "The mathematical analysis of logic" (1847) established the basis of modern mathematical logic, and his boolean algebra can be used in designing computers.
Boole's system is essentially two-valued. This can be symbolized by

0 or 1	*"binary representation"*
TRUE or FALSE	*"truth representation"*
0 V or 5 V	*"TTL electronics (transistor-transistor logic)"*
0 pC or 1 pC (pC = pico-Coulomb)	*"the charge in a condensator, the elementary memory unit in (dynamic) RAM"*

Na aula anterior (aula 7) vimos como se pode controlar o fluxo de um programa utilizando as instruções de selecção `if ...` e `if ... else ...`. Usamos condições como (x<1.0). Imaginemos agora que queremos calcular a raíz quadrada de (x^2–4). Claramente, não existe resposta para *x* entre –2 e +2. É importante verificar se *x* se encontra neste intervalo ou não. Podemos resolver a situação assim:

```
if (x<2)
  if (x>-2)
    printf("Error");
```

Mas seria muito melhor se pudéssemos fazê-lo com uma única condição. E tal é possível, vamos ver como:

```
if ((x<2) && (x>-2))
  printf("Error");
```

o que significa que ambas as condições (*x*<2 e *x*>–2), devem ser verdadeiras para que a condição completa seja verdadeira.
Este é um exemplo de um cálculo Booleano

```
condição3 = condição1 && condição2;
if condição3
  instrução;
```

operadores lógicos em C

& &	AND (E)
| |	OR (Ou)

Um outro operador lógico importante é o XOR, embora não seja implementado nos cálculos Booleanos em C. `a XOR b` significa "a ou b verdadeiros, mas não ambos!". Iremos utilizar o XOR mais tarde nas operações lógicas com inteiros. Outros operadores não implementados incluem o NAND e o NOR que são importantes apenas ao nível electrónico, não sendo muito úteis ao nível da programação.
Finalmente, existe um negador booleano !. Significa obter o oposto; Se `a` é falso (0), `!a` é verdeiro (1), e vice versa. Enquanto `&&` e `||` necessitam de dois operandos (por exemplo `a || b`), ! necessita apenas de um (`!a`).
Com estes operadores podemos calcular todas as condições possíveis que precisemos.

Aqui estão as tabelas completas ("tabelas de verdade") dos cálculos para os quatro operadores utilizados nas linguagens de programação modernas.

Operador Booleano	C
AND (E)	&&
OR (Ou)	||
XOR (Ou exclusivo)	
NOT (Negação)	!

AND (&&)

a	b	a AND b
true	**true**	**true**
true	false	false
false	**true**	false
false	false	false

OR (||)

a	b	a OR b
true	**true**	**true**
true	false	**true**
false	**true**	**true**
false	false	false

XOR

a	b	a XOR b
true	**true**	false
true	false	**true**
false	**true**	**true**
false	false	false

NOT (!)

a	NOT a
true	false
false	**true**

Exemplos:

```
a = 1;
b = -2;
c = 3;
d = 3;

if (a>0)
  printf("TRUE");
else
  printf("FALSE");
```

```
(a>0)                   TRUE
(b>0)                   FALSE
(a>0) && (b>0)          FALSE
(a>0) || (b>0)          TRUE
!(a>0)                  FALSE
(! (a>0)) || (b>0)      FALSE
(2==b) || (!(2==b))     TRUE
```

Álgebra Booleana para inteiros

Também se pode aplicar a álgebra booleana a números completos (char, int, long int, ver aula 5). Embora

não fizesse parte da ideia original de Boole, podemos facilmente realizar o mesmo tipo de cálculos com números, desde que seja feito "um bit de cada vez" e se utilize a representação "1 = verdeiro" e "0 = falso". Para evitar confusões, os símbolos para operações com números são diferentes dos que foram dados anteriormente, nomeadamente "&" para "AND", "|" para "OR", "^" para "XOR" e "~" para "NOT".

operation	C
AND	&
OR	|
XOR	^
NOT	~

Com esta convenção, as tabelas de verdade para a álgebra booleana bit-a-bit (bit-wise) ficam assim:

AND (&)

a	b	a AND b
1	**1**	**1**
1	0	0
0	**1**	0
0	0	0

OR (|)

a	b	a OR b
1	**1**	**1**
1	0	**1**
0	**1**	**1**
0	0	0

XOR (^)

a	b	a XOR b
1	**1**	0
1	0	**1**
0	**1**	**1**
0	0	0

NOT (~)

a	NOT a
1	0
0	**1**

```
 49 =  0 0 1 1 0 0 0 1
 24 =  0 0 0 1 1 0 0 0
 49 |
 24 =  0 0 1 1 1 0 0 1  = 57
 49 &
 24 =  0 0 0 1 0 0 0 0  = 16
 49 ^
 24 =  0 0 1 0 1 0 0 1  = 41
~49 =  1 1 0 0 1 1 1 0  = 206
~24 =  1 1 1 0 0 1 1 1  = 231
```

Como exemplo, vamos imaginar que queremos calcular 49 OR 24.
Primeiro, temos que converter estes números para o sistema binário (ver aula 3):
49 = 1*32 + 1*16 + 0*8 + 0*4 + 0*2 + 1*1 = 110001
24 = 0*32 + 1*16 + 1*8 + 0*4 + 0*2 + 0*1 = 011000
Depois, fazemos os cálculos bit-a-bit com as convenções da tabela de verdade de cima (1 OR 1 = 1, 1 OR 0 = 1, 0 OR 1 = 1, 0 OR 0 = 0), o que dará 111001
Finalmente, convertemos o resultado para o sistema decimal
111001 = 1*32 + 1*16 + 1*8 + 0*4 + 0*2 + 1*1 = 57

Na tabela ao lado, são exemplicadas também as outras operações com os números 49 e 24.

De notar que o operador ~ (bit-wise inverter) depende do tamanho da variável:
Para um byte (unsigned char) ~49 = ~(00110001) = (11001110) = 128 + 64 + 8 + 4 + 2 = 206, enquanto para um unsigned int ~49 = ~(0000000000110001) = (1111111111001110) = 65486.

Selecção/Ramificação Múltipla: `switch`

Na aula anterior (ver aula 7) vimos como utilizar a instrução `if ...` para seleccionar entre duas possíveis partes de um programa. Em alguns casos, podemos ter mais do que dois caminhos possíveis para o programa continuar. Para estas situações existe a instrução `switch` .
Vamos imaginar o programa que se segue que pergunta que ano o aluno frequenta:

```
main()
 // outputs the lectures to follow on basis of year
{
```

```
    int ano;

    printf("From what year are you?\n");
    scanf("%d", &ano);
    if (ano==1)
      printf("primeiro ano: MAT-1, CALC-1\n");
    else
      if (ano==2)
        printf("segundo ano: INF, LIN-ALG\n");
      else
        if (ano==3)
          printf("terceiro ano: ELEC, FYS\n");
        else
          if (ano==4)
            printf("quarto ano: QUI, MAT-2\n");
          else
            if (ano==5)
              printf("quinto ano: PROJECTO\n");
            else
              printf(">5: AINDA NAO ACABOU?\n");
}
```

(De notar a estrutura do programa, com indentações. Reparar também na ausência de ; antes de todos os `else`).
Este programa irá correr sem qualquer problema

```
From what year are you?
 1
primeiro ano: MAT-1, CALC-1

From what year are you?
 4
quarto ano: QUI, MAT-2
```

Apesar disso, a sua estrutura não é muito legível. Para tornar o programa melhor (mais legível) podemos utilizar a instrução `switch`. Enquanto em "`if (condição) instrução1;`" a condição é forçosamente do tipo boolena (verdeiro ou falso), com o `switch` podemos utilizar qualquer um dos tipos de variáveis que têm valores discretos (em contraste com as variáveis floating point que não têm valores discretos).

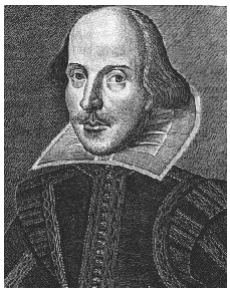
```
switch (expressão)
{
  case valor1: instrução1;
       break;
  case valor2: instrução2;
       break;
         |
  case valorN: instruçãoN;
       break;
  default: instruçãoE;
}
```

A `expressão` tem que resultar num valor de um tipo contável (por exemplo: `int`, `long int`, mas também `char`. Não pode ser, por exemplo, `float`). Tal pode ser uma simples variável ou um cálculo que resulte num valor. Os valores `valor1` a `valorN` têm que ser do mesmo tipo, mas não podem conter expressões ou

variáveis; têm que ser do tipo constant (constante).
A palavra reservada "default" significa que será executada esta instrução se a `expressão` não resultar em nenhum dos valores `valor1`.. `valorN`.
A instrução `break` força o programa a saltar para o fim do ciclo (loop) (um ciclo de switch neste caso).
Como exemplo, segue-se o programa anterior reescristo para utilizar a instrução `switch`:

```
main()
 // same program as above, but with switch statement
{
  int ano;

  printf("From what year are you?\n");
  scanf("%d", &ano);
  swicth (ano)
   {
    case 1: printf("primeiro ano: MAT-1, CALC-1\n");
            break;
    case 2: printf("segundo ano: INF, LIN-ALG\n");
            break;
    case 3: printf("terceiro ano: ELEC, FYS\n");
            break;
    case 4: printf("quarto ano: QUI, MAT-2\n");
            break;
    case 5: printf("quinto ano: PROJECTO\n")
            break;
    default: printf(">5: AINDA NAO ACABOU?\n");
   }
}
```

Este programa terá o mesmo output que o programa anterior, mas é muito mais legível.



Aula 8: `(2*B) || (! (2*B))`

Teste Rápido:
Para testar os conhecimentos sobre o que aprendeu nesta aula, prima aqui para fazer um teste on-line. De notar que este **Não** é o formato que será utilizado no teste final!

Peter Stallinga. Universidade do Algarve, 28 October 2002

Aula 9: Ciclos I: `for`

Traduzido por Ana Paula Costa

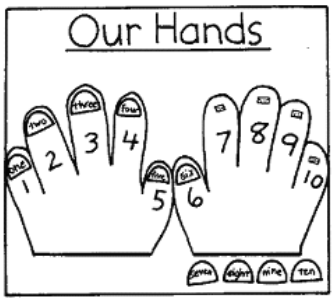
Os ciclos servem para repetir parte do código um certo número de vezes. Em C existem três tipos de ciclos,

- for
- while
- do ... while

Não existe diferença entre os dois primeiros tipos de ciclos, apenas uma questão de legibilidade do programa. O ciclo do ... while difere dos outros dois pelo facto de a condição de saída do ciclo ser verificada no final, enquanto nos outros dois (for e while) é verificada no início do ciclo.
A aula de hoje é sobre o ciclo for.

Ciclo for

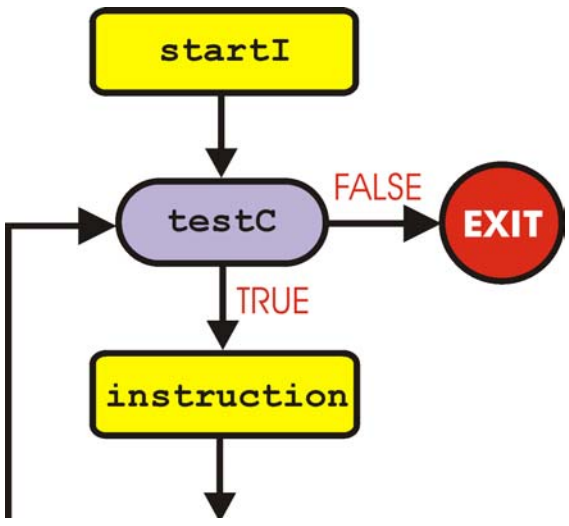
O ciclo mais comum em C é o ciclo for. Este ciclo, em princípio, é utilizado para executar coisas um número pré-definido de vezes de uma forma **contável**. Isto contrasta com os ciclos que são executados enquanto uma determinada condição é verdeira, que iremos aprender na próxima aula.



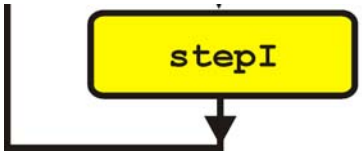
A estrutura geral do ciclo for é

```
for (inícioI; testeC; passoI)
  instrução;
```

com `inícioI` e `passoI` a serem instruções gerais e `testeC` a ser uma condição Booleana. As instruções `instrução` e `passoI` serão repetidas até que a condição de teste resulte em "falso".



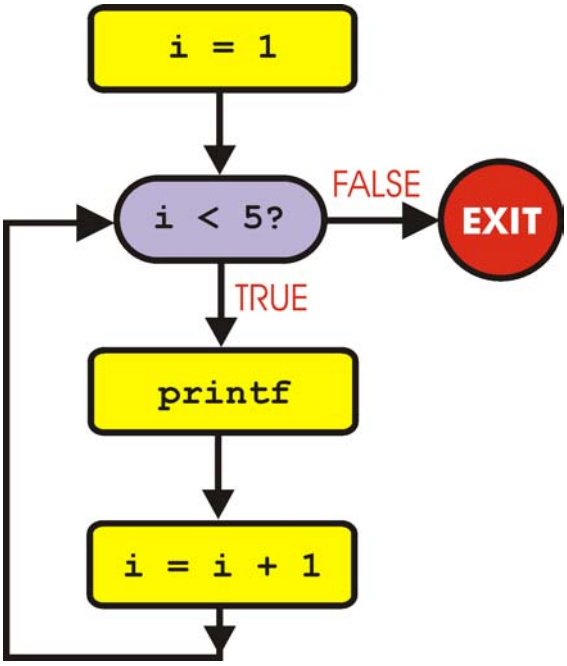
A `instrução` é repetida um número de vezes, determinado pelas instruções de controlo `inícioI` e `passoI` e a condição de teste `testeC`. As instruções de controlo `inícioI` e `passoI` podem ser qualquer instrução ou combinação de instruções (separadas por vírgulas).
A instrução `inícioI` é executada apenas no início do ciclo e é executada uma única vez. Imediatamente depois disso, a condição `testeC` é avaliada. Se o seu resultado é "falso" (0), o ciclo termina de imediato. Assim, é possível com o ciclo for que a `instrução` não seja executada nem sequer uma vez (ao contrário do ciclo `do .. while` como veremos na próxima aula). Se o resultado da condição é "verdadeiro" (!=0), as instruções `instrução` e `passoI` são executadas.



Uma vez que o ciclo for está desenhado para fazer coisas de uma forma **contável**, é aconselhável utilizar variáveis de controlo do tipo inteiro, como é demonstrado no próximo exemplo. É preciso não esquecer de declarar a variável (ver aula 5).

código do programa	*output*

`main()`
`// for-loop example`
`{`
`  int i;`
 	
`  for (i=1; i<5; i++)`
`    printf("Ola\n");`
`}`
	`Ola`
	`Ola`
	`Ola`
	`Ola`



Este programa faz o seguinte
1) atribui 1 a `i`
2) verifica se `i` é menor do que 5
3) se não é: SAI DO CICLO (EXIT LOOP) imediatemente. Caso contrário
4) executa `printf("Ola\n");`
5) adiciona 1 a `i`
6) vai para o passo 2)

Instruções Múltiplas

Tal como na estrutura `if ... else ...` , também podemos agrupar várias instruções com `{..}` nos ciclos:

código do programa	*output*

`for (i = 1; i<5; i++)`
`{`
`  printf("Ola\n");`
`  printf("It is a nice day\n");`
`}`
	`Ola`
	`It is a nice day`
	`Ola`
	`It is a nice day`
	`Ola`
	`It is a nice day`
	`Ola`
	`It is a nice day`

Comparar com

código do programa	*output*
`for (i = 1; i<5; i++)`	`Ola`

```
printf("Ola\n");
printf("It is a nice day\n");
```

```
Ola
Ola
Ola
It is a nice day
```

Utilização da variável do ciclo

Dentro do ciclo, a variável que controla o ciclo pode ser utilizada, mas **não é nada aconselhável** fazê-lo.

Bom código:

código do programa

```
for (i=1; i<5; i++)
  printf("%d Ola\n", i);
```

output

```
1 Ola
2 Ola
3 Ola
4 Ola
```

Mau código:

código do programa

```
for (i=1; i<5; i++)
  {
    printf("%d Ola\n", i);
     i = i + 1;
  }
```

output

```
1 Ola
3 Ola
```

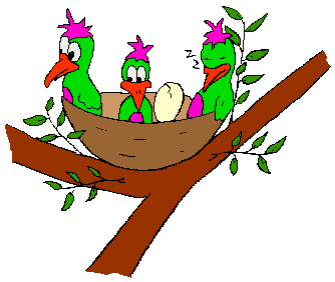
O programa da direita é um exemplo de mau código. Este estilo de programação, apesar de às vezes poupar espaço e talvez tempo de execução, torna o programa mal estruturado e muito difícil de compreender por outros! Se se pretender algo como o que está representado no programa da direita, o melhor é utilizar outros tipos de ciclos, como `while` ou `do-while`, ou , melhor ainda, utilizar algo parecido com o programa de baixo.

código do programa

```
for (i=1; i<5; i++)
  printf("%d Ola", 2*i-1);
```

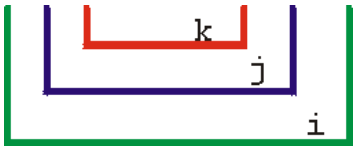
output

```
1 Ola
3 Ola
```



Ciclos encadeados

Os ciclos for (bem como os outros ciclos) podem ser encadeados, o que significa que podem ser colocados uns dentro dos outros. Podemos criar ciclos duplos, ou ciclos triplos (como o da figura de baixo à esquerda) ou de qualquer outro nível. Eis alguns exemplos:



código do programa

```
main()
// three nested loops
{
 int i, j, k;

 for (i=1; i<=2; i++)
   for (j=1; j<=2; j++)
     for (k=1; k<=2; k++)
       printf("i=%d j=%d k=%d\n",
i, j, k);
}
```

output

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

código do programa

```
main()
// three nested loops
{
 int i, j, k;

 for (i=1; i<=2; i++)
   for (j=1; j<=2; j++)
     {
       for (k=1; k<=2; k++)
         printf("i=%d j=%d k=%d\n",
i, j, k);
       for (k=1; k<=2; k++)
         printf("i=%d j=%d k=%d\n",
i, j, k);
     }
}
```

output

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

Um exemplo. Fibonacci.

Os números de Fibonacci são números que seguem as seguintes regras:

$F_1 = 1$

$F_2 = 1$

$F_n = F_{n-1} + F_{n-2}$

O programa a seguir implementa os números de Fibonacci. É pedido ao utilizador para dar o número de números de Fibonacci a calcular (n), e depois, num ciclo for, o próximo número de Fibonacci é calculado, até se calcularem os n números de Fibonacci.

```
main()
/* program to calculate the first n Fibonacci numbers
   using the algorithm
   F(1) = 1, F(2) = 1 and F(i) = F(i-1) + F(i-2) */
{
  int i, n, fi, fi1, fi2;

  // first ask the user for the number of Fibonacci numbers to calculate
  printf("How many Fibonacci numbers do you want to see?\n");
  scanf("%d", &n);
```

```
  fi1 = 1;                // variables to store F(i-1) and F(i-2)
  fi2 = 1;                // initialize them to their starting value
  printf("1 1 ");         // print the first 2 Fibonacci numbers
  for (i=3; i<=n; i++)    // print the rest
    {
      fi = fi1 + fi2;     // calculate the next number
      fi2 = fi1;          // calculate the new F(i-1) and F(i-2)
      fi1 = fi;
      printf("%d ", fi); // print the result
    }
}
```

Mais tarde iremos aprender uma forma muito mais elegante de efectuar estes cálculos, utilizando programação recursiva.

Teste Rápido:

Para testar os conhecimentos sobre o que aprendeu nesta aula, prima aqui para fazer um teste on-line. De notar que este **Não** é o formato que será utilizado no teste final!

Peter Stallinga. Universidade do Algarve, 30 October 2002

Aula 10: Ciclos II: `while ...` e `do ... while`

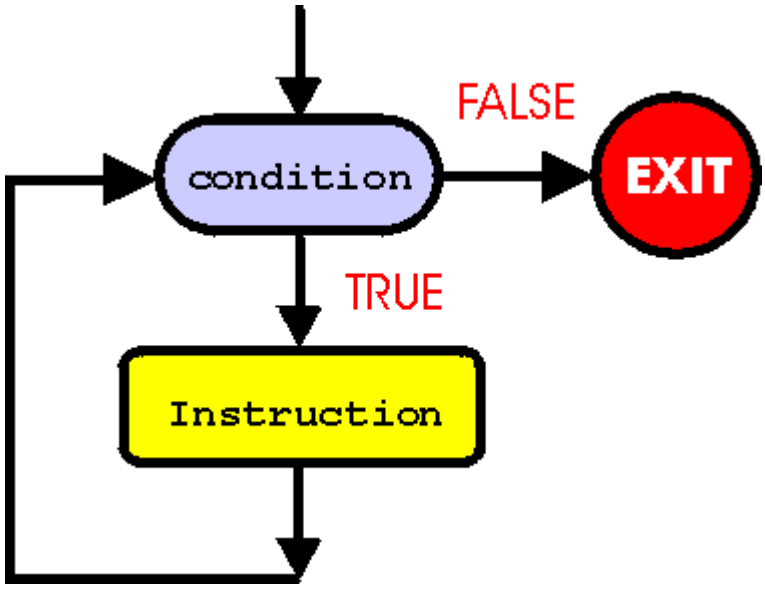
Traduzido por Ana Paula Costa

`while ...`

Os ciclos que iremos aprender nesta aula, while e do-while, são utilizados para repetir instruções um número indeterminado de vezes. Normalmente, estes ciclos são utilizados nas situações em que não se sabe exactamente quando o ciclo irá terminar, por exemplo porque a variável de controlo muda dentro do ciclo (o que é desaconselhável no ciclo for). Também quando se pretende fazer um ciclo utilizando uma variável do tipo float para controlo temos que utilizar os ciclos while e/ou do-while.
Como iremos ver, a diferença entre o ciclo while e o ciclo do-while está no momento em que a condição é testada.

O formato geral do ciclo while é

```
while (condição)
  instrução;
```



A instrução é repedita enquanto a condição é verdadeira. A `condição` é qualquer condição que resulte num valor booleano (Verdeiro ou Falso), como foi discutido na aula 8. Pode ser uma comparação, ou qualquer outra coisa (ainda existe informação no ficheiro?, o utilizador premiu uma tecla?, etc.).
De notar que no ciclo while não é atribuído nenhum valor de início a nenhuma variável, ao contrário do que acontece no ciclo for. Assim sendo, temos que ser nós a fazê-lo.
Exemplo:

código do programa	*output*

```
main()
// while example
{
  float x;

  x = 0.0;
  while (x<=1.0)
```

```
0.0
0.1
0.2
0.3
0.4
0.5
0.6
```

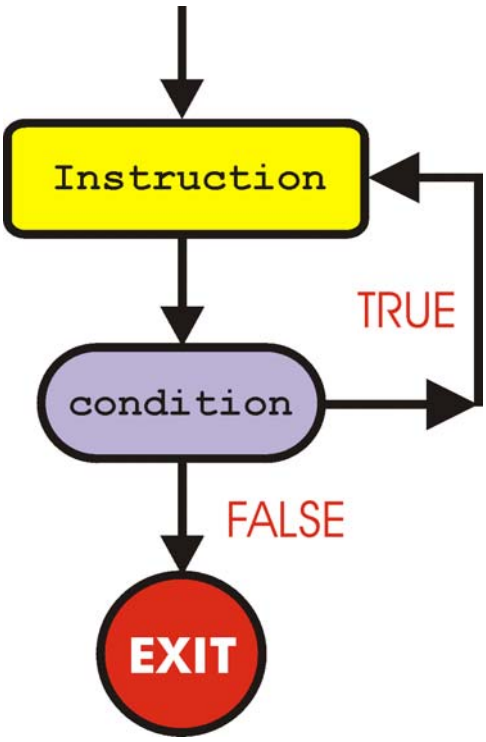
```
    {
      printf("%0.1f\n", x);
      x = x + 0.1;
    }
}
```

```
0.7
0.8
0.9
1.0
```

do ... while

O ciclo do-while é muito semelhante ao ciclo while. A única diferença é que agora a condição é verificada depois das instruções terem sido repetidas. O formato geral é

```
do
  instrução;
while (condição);
```



Uma diferença muito importante entre o while e o do-while é que no while a condição é verificada no **início** do ciclo, enquanto no do-while é verificada no **fim**. Assim, **as instruções no do-while são executadas pelo menos uma vez.**

Vejam-se os programas seguintes (também incluem um exemplo do ciclo for). Apenas o código do ciclo do-while produz output.

código do programa

```
x = 100;
do
  {
    printf("Ajax\n");
    x = x + 1;
  }
while (x<=10);
```

output

```
Ajax
```

código do programa

```
x = 100;
while (x<=10)
  {
    printf("Ajax\n");
    x = x + 1;
  }
```

output

código do programa

```
for (i=100; i<=10; i++)
  printf("Ajax\n");
```

output

`for`, `while` e `do-while` comparados

Efectivamente, os ciclos for e while são exactamente o mesmo. Both have a condition that is checked in the beginning of the loop and a step instruction. Ambos têm a condição que é verificada no início do ciclo e têm uma instrução de passo. A única diferença é que o ciclo for também inclui uma instrução de começo. Se precedemos o ciclo while com uma instrução de começo, os dois ciclos serão iguais.

for	while

`for (inícioI; testeC; passoI)`
`  instrução;`
	`inícioI;`
	`while (testeC)`
	`  {`
	`  instrução;`
	`  passoI;`
	`  }`

`for (i=1; i<=10; i++)`
`  printf("%d", i);`
	`i=1;`
	`while (i<=10)`
	`  {`
	`  printf("%d", i);`
	`  i++;`
	`  }`

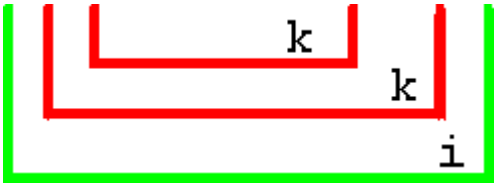
Apenas por uma questão de legibilidade do programa se utiliza o ciclo for para se repetir instruções um número contável de vezes. Em todas as outras situações utiliza-se o while e do-while. No ciclo do-while as instruções são executadas pelo menos uma vez. Deve-se ter estas considerações sempre em conta quando se vai decidir qual o tipo de ciclo a utilizar.

Encadeamento II

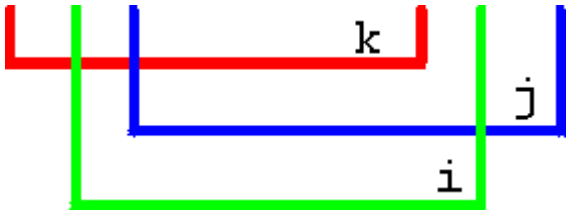
Agora que já conhecemos todos os tipos de ciclos, vamos ver as regras de bom comportamento relacionadas com o encadeamento:

- Cada ciclo (for) deve utilizar a sua própria variável de controlo.
- O ciclo interior deve começar e acabar completamente dentro do ciclo exterior.

Exemplos de mau código:




```
for (i=1; i<=10; i++)
  {
    for (i=1; i<=20; i++)
      printf("%d", i);
  }
```




```
for  (i=1; i<=10; i++)
  {
    x = i;
    do
      x = x+0.1;
```

	`  }` `while (x<20);`
Dois ciclos com a mesma variável de controlo `i`.	Dois ciclos mal aninhados. Uma boa indentação do programa evita sempre este tipo de erros.

Teste Rápido:

Para testar os conhecimentos sobre o que aprendeu nesta aula, prima aqui para fazer um teste on-line. De notar que este **Não** é o formato que será utilizado no teste final!

Peter Stallinga. Universidade do Algarve, 4 November 2002

Aula 11: Programação Modular I

Funções

Traduzido por Ana Paula Costa

Até agora estivemos apenas a utilizar coisas que foram fornecidas pelo C. Aprendemos como escrever ciclos (`for`, `while`, `do-while`), como obter input e output (`scanf`, e `printf`), como controlar o fluxo de um programa (`if`, `if-else`, `switch`), como declarar variáveis, como atribuir valores a constantes (=) e como calcular álgebra Booleana, mas nunca inventamos nada NOVO. Com **funções** podemos fazer exactamente isso.
Já conhecemos algumas funções que são standard em C, nomeadamente `printf` e `scanf`. Agora vamos definir as nossas próprias funções.

Funções são pequenos subprogramas ou **módulos** do programa principal. Cada um destes módulos executa uma determinada tarefa. Isto ajuda a organizar o programa e a torná-lo mais lógico e pode também aumentar a eficiência de programação evitando repetições de partes do programa. Mais ainda, as funções permitem copiar facilmente partes do código para outros programas.

---

input/output

Módulos (funções) podem ter input e/ou output. Neste caso, input(entrada) significa aceitar parâmetros, enquanto output(saída) significa retornar um valor. De notar que, independentemente do facto da função aceitar parâmetros de entrada ou não, a função terá sempre parêntesis, tanto na declaração como na chamada da função, como veremos mais à frente. Isto serve para distinguir funções de variáveis.

exemplos	sem input	com input
sem output	`abort()`	`printf()`
com output	`rand()`	`sqrt()`

`abort()` pára o programa, `rand()` retorna um valor aleatório, `sqrt()` retorna a raíz quadrada do argumento, `printf()` mostra o argumento no écran.

Funções são como programas dentro de programas. Elas

- têm que ter um nome. As regras que se aplicam para nomear identificadores, aplicam-se para nomear funções. Ver aula 5.
- podem ter variáveis. Estas têm que ser declaradas na função e estão acessíveis apenas para a função.

- têm que ter uma combinação `{` e `}`a indicar o início e o fim da função.
- (podem) ter instruções.

Um protótipo da declaração de uma função:

```
tipo nomedafunção(tipo <parâmetros>)
{
  tipo <variáveis>;

  instruções;
}
```

---

output (tipo, void, return)

O **tipo** de valor que a função irá retornar tem que ser especificado na declaração da função. Por exemplo

```
int countnumber();
```

significa que a função irá retornar um valor do tipo inteiro.

Algures dentro da função **temos** que retornar um valor para o programa que chamou a função. Para fazer isso utilizamos a instrução **return**

```
return <valor>;
```

por exemplo

```
return 3;
```

Quando queremos que a função não retorne nada, podemos especificar isso com a palavra **void**:

```
void showresult();
```

não irá retornar nada. Assim, funções do tipo void não precisam da instrução return.

Nota: em alguns compiladores a declaração do tipo do valor a retornar é opcional. Em alguns compiladores, a ausência da declaração do tipo significa que a função é do tipo `int`, enquanto em outros significa `void`. Para aumentar ainda mais a confusão, omitir a instrução return em alguns compiladores é um erro, enquanto noutros não é.
Assim sendo: o melhor é seguir as convenções standard do ANSI C (em cima) e escrever programas independentes do compilador!

---

main()

De facto, `main` não é mais do que uma função normal, com a única diferença de o programa começar sempre pela primeira instrução desta função. De resto, `main` segue todas as regras de funções descritas antes. Tecnicamente falando, ou temos que especificar o tipo como void ou retornar algo no fim do programa.

```
void main()
{
 ...
}
```

```
int main()
{
 ...
 return 0;
}
```

---

posicionamento/localização

O local para declarar as nossas próprias funções é **antes** da função `main`. Se forem declaradas depois da função main não poderemos utilizar as funções dentro de main porque o compilador não terá conhecimento da sua existência quando começa a compilar a função main. Uma função só pode chamar outras funções que tenham sido declaradas antes (de alguma forma).

chamada

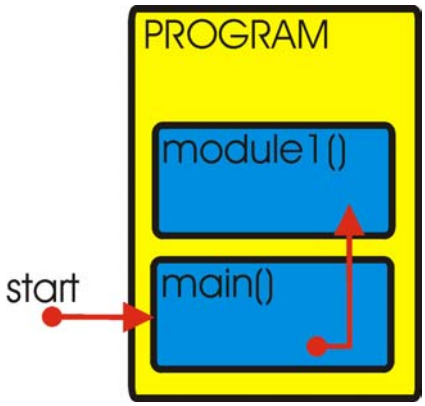
Após declarar uma nova função, podemos utilizá-la no programa principal. A isto designa-se **chamada** da função. Para fazer a chamada da função escreve-se o nome da função. Eis um exemplo completo de um programa com um único módulo (função) sem parâmetros de entrada ou saída (void):

código do programa

```
#include <stdio.h>

void module1()
{
  int y;

  printf("Now I am entering Procedure Module1\n");
  printf("Give a value for y\n");
  scanf("%d", &y);
  printf("%d", 3*y);
// Type of function is void.
// No need to return anything.
}

void main()
{
// The program starts at the first instruction
// of function main:
  printf("Starting the program\n");
// now our function will be called:
  module1();
}
```

output

```
Starting the program
Now I am entering Procedure Module1
Give a value for y:
 4
12
```



Um programa a chamar uma das suas funções

funções a chamar funções

As funções também podem ser chamadas por outras funções. Vamos ver o programa que se segue e o resultado que gera ao correr:

código do programa

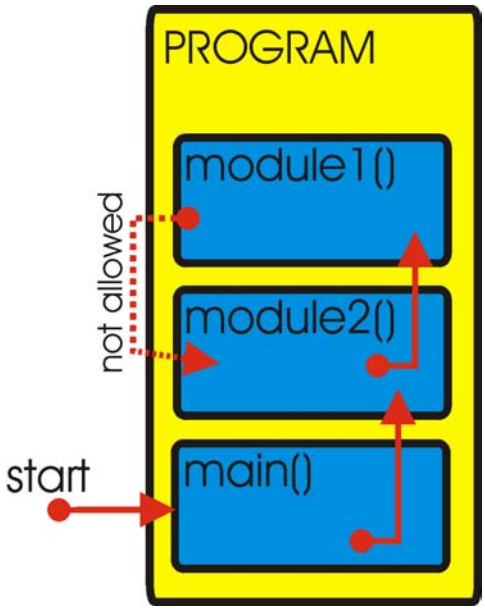
```
# include <stdio.h>

void module1()
{
  printf("    inside module 1\n");
  printf("    Hello World\n");
```

output

```
starting the program
calling module 2
  inside module 2
  calling module 1
```

```
  printf("    leaving module 1\n");
}


void module2()
{
  printf("  inside module 2\n");
  printf("  calling module 1\n");
  module1();
  printf("  back in module 2\n");
  printf("  leaving module 2\n");
}

void main()
{
  printf("starting the program\n");
  printf("calling module 2\n");
  module2();
  printf("back in main\n");
  printf("ending program");
}
```

```
    inside module 1
    Hello World
    leaving module 1
  back in module 2
  leaving module 2
back in main
ending program
```



Um programa a chamar uma das suas funções
que, por seu lado, está a chamar uma outra função

De notar que chamar funções que só são declaradas depois, como seria no exemplo anterior uma chamada de `module2()` feita em `module1()`, não são permitidas em circunstâncias normais. main pode chamar `module1()` e `module2()`, `module2()` pode chamar `module1()` e `module1()` não pode chamar nada.

Teste Rápido:

Para testar os conhecimentos sobre o que aprendeu nesta aula, prima aqui para fazer um teste on-line. De notar que este **Não** é o formato que será utilizado no teste final!

Peter Stallinga. Universidade do Algarve, 5 November 2002

Aula 12: Programação Modular II:

Funções com entrada e saída

Traduzido por Ana Paula Costa

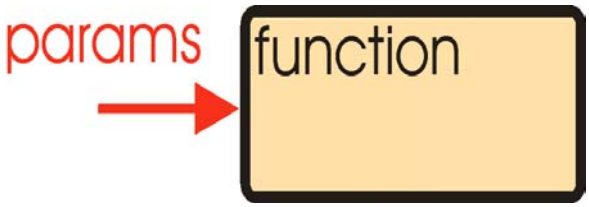
Na aula anterior (aula 11) vimos funções que não aceitam parâmetros e não retornam valores. Estas eram funções simples. Agora vamos ver funções que aceitam parâmetros de entrada e funções que produzem valores de retorno.

Parâmetros para passar para as funções

Nós podemos passar parâmetros para as funções. As funções podem depois trabalhar com estes parâmetros. Dentro da função, os parâmetros funcionam como variáveis normais. Em C, os parâmetros que a função espera são colocados depois do nome da função entre parêntesis. Para uma função que não tem saída (ou por outras palavras, que retorna o tipo void):

Uma funcção com entrada(input) (mas sem saída(output)):

```
 void
nomedafunção(listadeparâmetros)
{
 <declaração de variáveis
locais>

  instruções;
}
```



As variáveis na lista de parâmetros são declaradas da mesma maneira que as variáveis normais de um programa ou função, nomeadamente temos que especificar o tipo de cada parâmetro. Dentro da função podemos utilizar o parâmetro como se fosse uma variável normal. Podemos efectuar cálculos com ele, utilizá-lo em condições, e até mesmo alterar o seu valor. Não necessita de ser inicializado, porque a inicialização vem do programa que faz a chamada à função.

Como um exemplo, o programa que se segue irá calcular e mostrar o quadrado de uma variável `x`: De notar a forma como o parâmetro `r` é declarado e utilizado.

código do programa

```
#include <stdio.h>

void write_square(float r)
{
  float y;

  y = r*r;
  printf("The square of %f is %f", r, y);
}
```

output

```
The square of 4.0 is 16.0
The square of 3.0 is 9.0
```

```
void main()
{
  float x;

  x = 4;
  write_square(x);
  write_square(3.0);
}
```

Como se pode ver no programa de cima, as funções com parâmetros podem ser chamadas com uma variável, como é o caso em `write_square(x)`, ou com uma constante, como em `write_square(3.0)`.

Outro exemplo, que utiliza dois parâmetros:

código do programa | *output*

```
#include <stdio.h>

void write_sum(int i1, i2)
/* write the sum of i1 and i2 */
{
 int j;

  j = i1+i2;
  printf("The sum of %d  and %d is %d',
    i1, i2, j);
}

void main()
{
  int x, y;

  x = 4;
  y = 5;
  write_sum(x, y);
  write_sum(3, 4);
}
```

```
The sum of 4 and 5 is 9
The sum of 3 and 4 is 7
```

De notar que temos que passar para a função o tipo de informação que ela espera. Neste caso, a função espera por dois inteiros, e então nós devemos passar-lhe dois inteiros (x e y).
Por fim, um exemplo com uma lista de parâmetros de vários tipos. As variáveis a serem declaradas na lista de parâmetros são separadas por uma vírgula (,).

código do programa | *output*

```
#include <stdio.h>

void write_N_times(float r, int n);
  /* Will write n times the real r */
{
  int i;

  for (i= 1; i<=n; i++)
    printf("%10.3f\n", r);
}

void main()
{
```

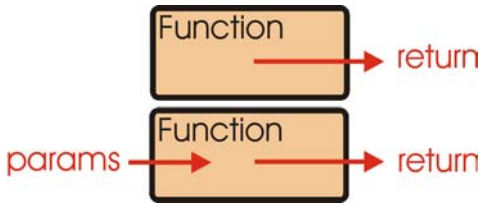
```
3.000
3.000
3.000
3.000
```

```
  write_N_times(3.0, 4);
}
```

Funções com saída (output)

As funções podem retornar um valor de saída. O **tipo** do valor de retorno tem que ser especificado no momento da declaração da função, antes do nome da função.

```
 tipo
nomedafunção(listadeparâmetros)
 {
  <listadevariáveis>

    instruções;
 }
```



Funções com saída e com parâmetros de entrada

Algures nas instruções temos que especificar o **valor de retorno**. Fazemos isso com a palavra reservada `return`

```
return (valor);
```

Obviamente, o valor tem que ser do mesmo tipo que o tipo da declaração da função.
De notar que a instrução return sai imediatamente da função e as instruções que se encontrem depois de return não serão executadas.

```
double square(double r)
  /* will return the square of of the parameter r */
{
  r = r*r;
  return (r);
}
```

No local onde a função será chamada, podemos atribuir este valor a uma variável (do mesmo tipo do valor retornado pela função!), por exemplo

```
y = square(3.0);
```

utilizá-lo como parte de uma expressão, por exemplo

```
y = 4.0 * square(3.0) + 1.0;
```

ou utilizá-lo noutra função, por exemplo

```
printf("%f", square(3.0));
```

Um exemplo completo:

código do programa	*output*

```
/* example with parameters */                    The square of 4.0 is 16.0
#include <stdio.h>                               The square of 3.0 is 9.0

double square(double r)
  (* will return the square of of the parameter r *)
{
  r = r*r;
  return(r);
}

void main()
{
  double x, y;
  x = 4.0;
  y = square(x);
  printf("The square of %f is %f\n", x, y);
  printf("The square of %f is %f\n", 3.0, square(3.0));
}
```

O que está a acontecer com a instrução `y = square(x)` é o seguinte:

1. O valor da expressão dentro de parêntesis é calculado. Neste caso é simples. É o valor de `x`, nomeadamente 4.0;
2. Este valor (4.0) é passado para a função `square()`. Dentro da função:
3. É criada uma variável temporária com nome `r`.
4. O valor passado para a função é atribuído a essa variável `r`. Efectivamente aconteceu uma instrução `r=4.0`.
5. `r=r*r;` O novo valor de `r` é calculado. r agora tem o valor 16.0.
6. `return(r);` O valor da expressão dentro de parêntesis (16) é passado de volta à instrução que fez a chamada (`y=square(x);`); onde poderá ser utilizado.
7. Neste caso, o valor retornado (square(4.0) que é 16.0) será atribuído a `y`. A instrução `y=square(x);` efectivamente transforma-se em `y=16.0;`
8. A linha seguinte (`printf`...) mostra os valores das variáveis x e y. (**De notar que o valor de `x` se manteve inalterado** e continua a ser 4.0. Mais tarde, na aula sobre "passagem por valor / passagem por referência" iremos ver que não é necessariamente assim.)

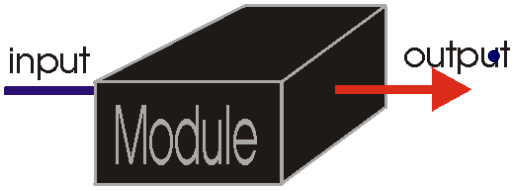
De notar que em C não temos que utilizar o valor que é retornado pela função. Podemos, por exemplo escrever na nossa função `main()`

```
square(3.0);
```

Esta construção é muito confusa e deve ser evitada quando possível em programas estruturados; um valor retornado por uma função deve, em princípio, ser utilizado na parte que fez a chamada.

Porquê?

Agora a grande questão é "porquê?". Porquê escrever funções se podemos fazer o mesmo com linhas de instruções normais? De facto, as primeiras linguagens (por exemplo o BASIC) não tinham a possibilidade de escrever funções e mesmo assim podíamos escrever programas para resolver qualquer problema. Existem no entanto três razões importantes que justificam a utilização de módulos.



Com os módulos, devido ao facto de serem como **caixas negras**, podemos distribuir as nossas tarefas de programação por várias pessoas, ou grupos de pessoas, sem ter que haver muita comunicação entre essas pessoas. Podemos dizer a alguém que necessitamos de uma função que inverta uma matriz sem termos

que dizer como queremos que o faça. Temos que especificar apenas o tipo de parâmetros a passar para a função.
- Pela mesma razão, podemos facilmente copiar partes de outros programas ou bibliotecas para servir os nossos propósitos (desde que se utilizem apenas parâmetros e variáveis locais, como iremos ver mais tarde). No caso ideal, nós apenas temos que fazer o "link" dessas funções que iremos utilizar, com o nosso programa, sem saber exactamente como elas funcionam. (Sabendo, claro, o que elas irão fazer e como as chamar). Uma consequência interessante disto é que podemos distribuir as nossas funções de uma forma pré-compilada, de forma a que o código se mantenha seguro na nossa posse.
- Com as funções o programa fica mais pequeno, mais eficiente e mais legível, evitando-se repetições de código e organizando-se o programa de forma mais lógica.

Teste Rápido:

Para testar os conhecimentos sobre o que aprendeu nesta aula, prima aqui para fazer um teste on-line. De notar que este **Não** é o formato que será utilizado no teste final!

Peter Stallinga. Universidade do Algarve, 8 November 2002

Aula 13: Arrays

Traduzido por Ana Paula Costa

Array

Imaginemos que queremos escrever um programa para calcular a média de 10 números. Com os conhecimentos que adquirimos até agora, podíamos fazê-lo definindo 10 variáveis diferentes, por exemplo

```
float a1, a2, a3, a4, a5, a6, a7, a8, a9, a10;
float average;
```

e no código do programa:

```
average = (a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8 + a9 + a10) / 10.0;
```

Espero que todos estejam de acordo que isto não é uma solução muito elegante e é até incómoda. E podia ser ainda pior: vamos imaginar que queremos que o utilizador seleccione quantos números se devem usar no cálculo da média:

```
int n;

scanf("%d", &n);
switch (n)
 {
   case 1: average = a1;
           break;
   case 2: average = (a1 + a2) / 2.0;
           break;
   case 3: average = (a1 + a2 + a3) / 3.0;
           break;
         |
   case 10: average = (a1 + a2 + a3 + a4 + a5 + a6 + a7 + a8 + a9 + a10) / 10.0;
 }
```

É para estes casos que existem os arrays. Um array permite definir um conjunto de variáveis do mesmo tipo com uma forma fácil de acesso, nomeadamente com um índice. Tal como na matemática, a_i é o i-éssimo elemento de um vector ou série a, `a[i]` fornece o i-éssimo elemento do array `a`.

Um array é um conjunto de variáveis indexadasdo mesmo tipo.

Declaração de um array

Para declarar um array utilizamos a seguinte sintaxe:

tipo nome[numelementos] ;

nome é o identificador do array, como um nome para as outras variáveis.
numelementos define o número de elementos do array. Os elementos vão de 0 a numelementos-1. Isto pode ser um pouco confuso para os programadores iniciantes, que estão acostumados a índices de 1 a n.
tipo é qualquer tipo de variável, por exemplo float ou int.

Exemplos:

```
float account[100];
```

Este array poderá ser usado para armazenar informação de 100 contas bancárias.

```
long int prime[10];
```

Poderá ser utilizado para armazenar os 10 primeiros números primos.

```
int propinas[2000];
```

Poderá ser utilizado para armazenar informação sobre o pagamento de propinas dos alunos com números entre 0 e 1999.

Utilização de um array

Dentro de um programa podemos utilizar os elementos de um array.

nome[índice]

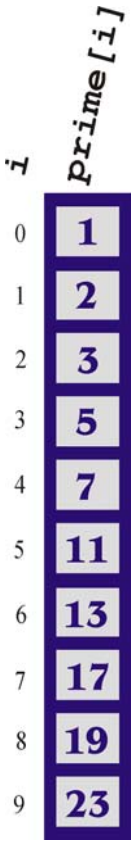
`nome[índice]` irá retornar o valor do elemento com número `índice` do array `nome`. Este será um valor do tipo descrito na declaração do array. Exemplos de arrays declarados na secção anterior:

```
account[20]
```

é o valor - do tipo float - do 20.º elemento do array com nome `account`.

```
prime[8]
```

é o valor - do tipo long int - do 8.º elemento do array `prime`. A figura da direita pode representar esse array, assim o elemento número 8 será igual a 19. Naturalmente, o nosso programa tem que preencher o array de alguma forma antes que venha a conter os números primos.




```
propinas[1055]
```

é 1 ou 0 (tipo int).O elemento 1055 do array `propinas`. O estudante 1055 pagou as suas propinas? Provavelmente a administração da universidade terá algures num computador um array com essa informação.

Podemos também utilizar uma variável para o índice que indica um elemento único do array. Naturalmente, esta variável tem que ser do tipo inteiro, porque o índice é algo contável. O índice 3.4981 não faz qualquer sentido. O índice 3 já faz sentido, irá aceder o terceiro elemento do array. O código seguinte irá mostrar o array completo de 20 contas bancárias:

```
for (i=0; i<20; i++)
  printf("%f\n",account[i]);
```

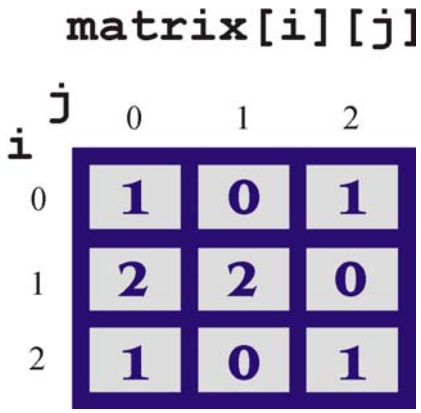
Arrays múltiplos

Tal como na matemática, onde temos vectores (tensores de 1 dimensão) e matrizes (tensores de 2 dimensões), podemos ter arrays de uma dimensão ou 2 dimensões, ou até mais. Podemos especificar isto da seguinte forma, por exemplo um array duplo (um array de duas dimensões):

```
tipo nome[numíndice1][numíndice2];
```

A utilização de um array duplo é similar à utilização de um array simples. Separamos os índices com uma vírgula, ou colocando-os em parêntesis rectos separados:

```
nome[índice1][índice2]
```

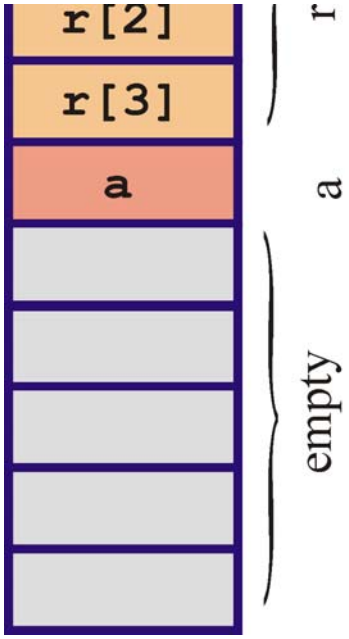


Por exemplo: Para escrever a matriz da figura da esquerda podemos fazer o programa seguinte. De notar que o array consiste de 9 (3x3) elementos do tipo inteiro.

```
void main()
{
  int  matrix[3][3];

  matrix[0][0] = 1;
  matrix[0][1] = 0;
  matrix[0][2] = 1;
  matrix[1][0] = 2;
  matrix[1][1] = 2;
  matrix[1][2] = 0;
  matrix[2][0] = 1;
  matrix[2][1] = 0;
  matrix[2][2] = 1;
  for (i=0; i<3; i++)
    {
      for (j=0; j<3; j++)
        printf("%d ", matrix[i][j]);
      printf("\n");
    }
}
```

Cuidado

Quando estamos a utilizar um array temos que ter cuidado para não usar um índice fora do limite. Isto significa que temos sempre que usar um índice que seja menor do que o número total de elementos do array. Se utilizarmos um índice demasiado grande, os resultados do nosso programa podem ser muito estranhos. Um exemplo é a melhor forma de ilustrar esta situação. O



programa que se segue define um array `r` de 4 inteiros que vão de `r[0]` a `r[3]`, e um inteiro normal `a`. A figura da esquerda mostrar como as variáveis poderiam estar colocadas na memória. O que irá acontecer se o programa atribui um valor a `r[4]`? Se `r[4]` existisse, teria ocupado o lugar que está agora a ser utilizado por `a`, e uma atribuição a `r[4]` iria colocar um valor na caixa que é agora ocupada por `a`. A maioria das linguagens não se preocupam e colocam o valor de `r[4]` de qualquer forma, e assim o valor de `a` será eliminado (substituido por o de `r[4]`).

```
main()
{
  int a;
  int r[4];

  a = 0;
  printf("a=%d\n", a);
  r[4] = 1;
  printf("a=%d\n", a);
}
```

O output do programa será provavelmente

```
a=0
a=1
```

Algumas linguagens de programação podem fazer a verificação do índice na altura em que o programa está a correr (em "run time"). Isto é chamado de "range-checking" e quando o programa tenta utilizar um índice errado, uma mensagem de erro do tipo "range-check error" ou "array index out of bounds" será mostrada. A desvantagem de fazer isto é que o programa fica mais lento e o programa compilado irá ocupar mais espaço em memória ou em disco.

Nota: isto depende da implementação exacta da linguagem/compilador. Em algumas linguagens, irá escrever por cima da variável declarada *antes* do array como no exemplo anterior, enquanto em outras linguagens irá escrever por cima da variável *depois* do array. Podemos ficar a saber declarando as variáveis

```
int a;
int r[4];
int b;
```

e ver se `r[4]` escreve por cima de `a` ou `b`.

"Hurray! *(Viva!) Eu sei tudo sobre arrays."*

Teste Rápido:

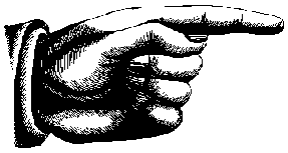
Para testar os conhecimentos sobre o que aprendeu nesta aula, prima aqui para fazer um teste on-line. De notar que este **Não** é o formato que será utilizado no teste final!

Peter Stallinga. Universidade do Algarve, 12 November 2002

Aula 14: Ponteiros

Traduzido por Ana Paula Costa

Ponteiro



Um ponteiro é um tipo especial de variável. Por si só não contém informação útil. É apenas um endereço para uma localização que (talvez) contém informação útil.

> Um ponteiro contém o endereço de uma localização em memória

Isto é como manter o endereço de um apartamento na agenda. É apenas um endereço e nada mais.

Declaração

Para declarar um ponteiro podemos utilizar a seguinte sintaxe:

```
tipo *nome;
```

Sendo `nome` o nome da variável ponteiro e `tipo` o tipo de informação a que o ponteiro está a apontar. Isto pode ser qualquer tipo dos que já conhecemos, desde simples reais e inteiros, a tipos mais complicados que iremos aprender mais tarde.
Exemplos:

```
int *p;
```

Agora `p` é um ponteiro que aponta para informação do tipo int.

```
float *f;
```

Agora `f` é um ponteiro que aponta para informação do tipo float. O valor de `f` em si não é do tipo float, porque `f` é um ponteiro e o seu valor é um endereço. Apenas o *conteúdo* do endereço de memória para o qual `f` está a apontar contém informação do tipo float.

Operações com ponteiros

Existem duas operações com ponteiros em C.

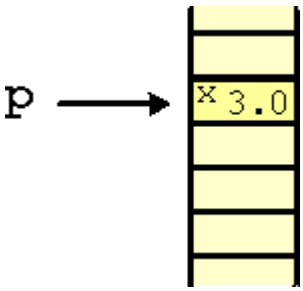
> **`&x`** retorna o endereço da (aponta para a) variável **`x`**

> **`*p`** é o valor que está a ser apontado por **`p`**

(conteúdo do endereço p)

Vamos declarar uma variável do tipo float, x, e um ponteiro para floats, p:

```
float x;
float *p;
```



Na figura da esquerda, `x` é uma variável do tipo float que tem o valor de 3.0. Como queremos que o ponteiro `p` fique a apontar para a variável x, podemos escrever

```
p = &x;
```

O valor de `p` é agora um endereço de memória, nomeadamente o endereço que contém a variável `x`. Para mostrar o conteúdo da localização de memória a que o ponteiro p está a apontar, podemos fazer

```
printf("%f", x);
```

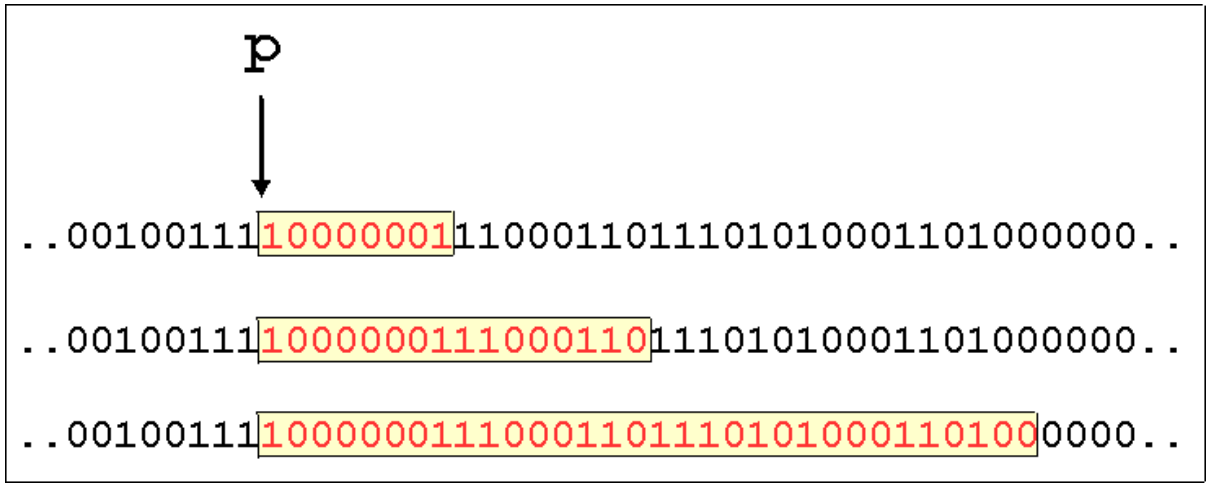
ou através do ponteiro

```
printf("%f", *p);
```

Especificação do tipo

Porque razão é tão importante que se especifique o tipo para o qual o ponteiro aponta? Isto consegue-se mostrar melhor com um exemplo.

Vamos imaginar a seguinte situação: A figura em baixo mostra parte da memória e o seu conteúdo. Um ponteiro `p` aponta para uma localização nesta memória. Se `p` está a apontar para um byte(`unsigned char`) como representado na linha de cima, o conteúdo do endereço `p` será `*p` =129 (binary: 10000001), enquanto com `p` a apontar para o mesmo local da memória, mas a apontar para um `unsigned int *p` = 25473 (binary: 0110001110000001), ou `p` a apontar para um `long int` (4 bytes, 32 bits) irá dar 743924609 (binary: 00101100010101110110001110000001). (De notar que nos computadores baseados no processador Intel, os números são armazenados com o seu bit de valor menos significativo [LSB=least significant bit] primeiro).



Assim sendo, temos que especificar o tipo a que o ponteiro irá apontar.

Exemplo

Vamos agora ver um exemplo para verificar se está tudo sobre controlo. Vamos criar um ponteiro do tipo

apontador para uma word (**unsigned int**), e deixá-lo apontar para uma word.

```
void main()
{
    /* declare a word (unsigned int) and  a pointer to word: */
  unsigned int *wordptr;
  unsigned int w;

    /* assign a value to the word */
  w = 25473;

    /* let a 'pointer to word' point to our word */
  wordptr = &w;
    /* show the contents of the memory wordptr points to */
  printf("%d", *wordptr);
}
```

output:

```
 25473
```

Vamos ver agora um exemplo mais complicado. Vamos criar um ponteiro do tipo apontador para um byte (**unsigned char**), e apontá-lo para a nossa word **unsigned int**:

```
void main()
{
    /* declare a word (unsigned int) and  a pointer to word: */
  unsigned char *charptr;
  unsigned int w;

    /* assign a value to the word */
  w = 25473;

    /* let a 'pointer to char' point to our word */
  charptr = &w;
    /* show the contents of the memory wordptr points to */
  printf("%d", *charptr);
}
```

output:

```
 129
```

(output for Borland C++ version 3.1 for MS-DOS. On other computers or versions the output might be different)
Isto mostra como temos que ser cuidadosos com o que o nosso ponteiro aponta. O valor depende do tipo!

Pointers and arrays in C

In C, all arrays are pointers. What this means is that when we declare an array

```
  int a[10];
```

this will
- reserve 10x2 bytes in memory
- assign the address of this memory to `a`

Therefore,
`a` is of type 'pointer to int' and the value of a is an address.
`*a` is the contents of address `a`. In this address resides the first element of `a`, namely `a[0]`.

Therefore, the two forms, `*a` and `a[0]` are completely interchangeable and any of the two can be used at any time. We will see this later, when we discuss strings (arrays of `char`).
We have to remember this when we pass information to a function; when we pass an array, we pass the **address** of the array, rather than all the elements of the array. With arrays in C we always use the technique of passing by reference (see lecture 15).

Initialization

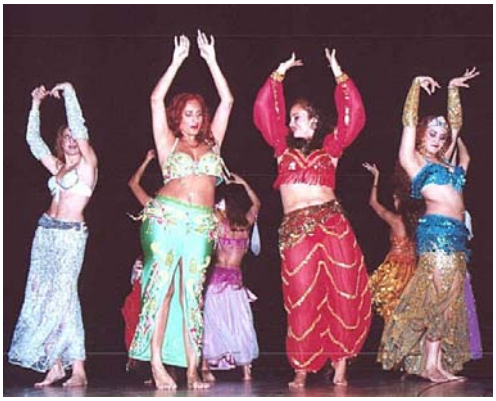
Initialization of a variable is even more important for pointers. Without initialization, a pointer points to a random part of memory, where important programs might be running. These programs and the computer can crash when we write in this place. The following program might crash the computer. It defines a pointer and doesn't assign an address to it. The value of the pointer (the address) is therefore unpredictable. The program then writes a value in this random address.

```
void main()
{
  int *p;

  *p = 0;
}
```

Porquê?

Porquê utilizar ponteiros? Existem várias razões para se usarem ponteiros em vez de variáveis normais. As mais importantes são

- Velocidade



- Flexibilidade

Velocidade: Imaginemos que queremos escrever um programa que move uma série de informação. Na forma convencional, isto iria significar copiar muitos bytes de uma parte da memória para outra parte. Pensemos por exemplo na ordenação de um array.
(Um ponteiro são apenas 4 bytes nos computadores Intel).

Flexibilidade: Se, no início do programa nós ainda não sabemos quantas variáveis iremos precisar teríamos que reservar espaço para todas as eventualidades possíveis. Se queremos escrever um programa que calcule os primeiros N números primos, com N dado pelo utilizador, teríamos que criar um array de tamanho máximo para garantir que podemos dar resposta a qualquer pedido do utilizador. Isto iria ocupar completamente a memória do computador. Mais nada poderia correr a partir daí. Muito melhor seria se pudéssemos declarar o array (as variáveis) dinamicamente de forma a que fosse utilizada apenas a

memória realmente necessária. Com ponteiros isto consegue-se facilmente.

Os ponteiros e a ideia da criação dinâmica de variáveis também são a base da programação orientada a objectos ("object-oriented programming"), que é o tipo de programação mais moderna e mais utilizada hoje em dia. A programação orientada a objectos está fora do âmbito desta aula.

Teste Rápido:

Para testar os conhecimentos sobre o que aprendeu nesta aula, prima aqui para fazer um teste on-line. De notar que este **Não** é o formato que será utilizado no teste final!

Peter Stallinga. Universidade do Algarve, 13 November 2002

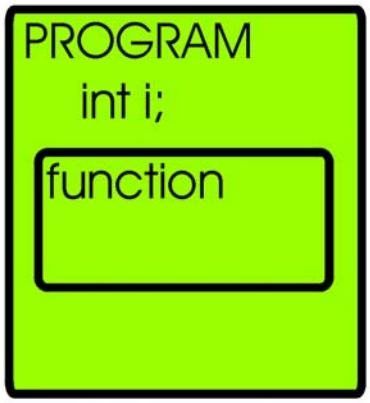
Aula 15: sobre variáveis e funções

Traduzido por Ana Paula Costa

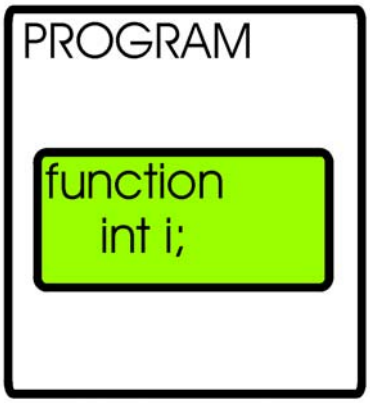
Nesta aula:

- Variáveis globais e locais
- Passagem por valor, passagem por referência

Alcance ("scope") das variáveis: Globais ou locais

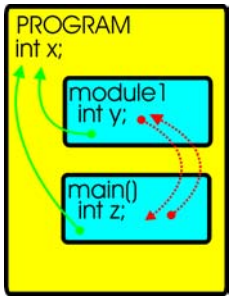


Variáveis **globais** são variáveis que podem ser utilizadas em qualquer parte de um programa.



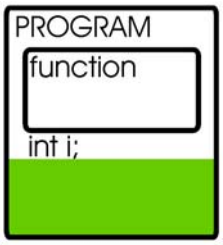
Variáveis **locais** estão apenas definidas dentro da função onde são declaradas.

Com esta nova informação, torna-se muito mais claro saber quando se pode utilizar cada variável. Dentro das funções podemos usar as variáveis globais **e** as locais. A função main não pode usar a variável da função modulo1, `int y`. Modulo1 não pode usar a variável da função main, `int z`. Contudo, as duas funções podem usar a variável global `x`.



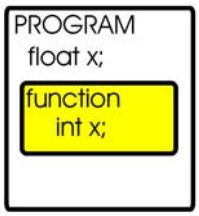
Aviso: Deve-se evitar a utilização de variáveis globais em funções tanto quando possível. O motivo é simples. Se quisermos copiar a função para outro programa será mais difícil, porque o novo programa provavelmente não terá as mesmas variáveis globais. Utilizando apenas variáveis locais nas funções é muito melhor. Se queremos utilizar as variáveis globais, elas devem ser passadas como parâmetros para a função. Idealmente, uma função é uma unidade independente.

Mais uma regra, tipicamente para linguagens como o C (*compiladores "single-pass"*):
As variáveis só podem ser utilizadas DEPOIS de serem declaradas no programa, assim, se colocamos a declaração de uma variável depois de uma função, essa função não poderá usar a variável, mesmo se se tratar de uma variável global.
Vamos ver alguns exemplos. Comecemos pelo programa da aula 13:




```
#include <stdio.h>

 /* declare global variables x and y */
double x, y;

double square(double r)
{
 /* declare local variable localr */
  double localr;

 /* using the local variable localr */
  localr = r*r;
 /* using the global variable x */
  x = localr;
  return(x);
}

void main()
{
  x = 4.0;
  y = square(x);
}
```

Prioridade



A variável x é uma variável local e global. Dentro da função, será usada a variável local.

Quando existem variáveis globais e variáveis locais com o mesmo nome, a variável local tem prioridade mais elevada e será portanto utilizada dentro da função. De qualquer forma isto é confuso, e então **devemos sempre evitar utilizar o mesmo identificador novamente**!

Algumas linguagens não fazem distinção entre variáveis locais e globais (por exemplo BASIC). Isto significa que não podemos usar o mesmo nome para variáveis duas vezes.

Passagem por valor ou passagem por referência

Quando passamos parâmetros para as funções, podemos fazê-lo de duas formas diferentes: passagem por valor ou passagem por referência.
Passagem por valor:
Até agora vimos apenas este tipo de passagem. Desta forma, apenas um valor é passado para a função. Qualquer coisa que se faça com esse valor na função não terá nenhum efeito no valor original da variável

que foi usada na chamada da função. Para ficar bem claro, eis um exemplo. Vamos assumir que temos uma função que escreve o quadrado de um parâmetro p. Para calcular o quadrado atribuímos um novo valor a p, (p*p). O valor de p irá ser alterado dentro da função

```
void write_square(double p)
{
  p = p*p;
  printf("%f", p);
}
```

Quando chamamos a função com a variável `x`, o valor desta variável não irá ser alterado com a chamada da função. No programa principal:

```
void main()
{
  double x;

  x = 2.0;
  write_square(x);
  printf("%f", x);
}
```

Após correr a função, o valor de x não foi alterado. O resultado completo do programa de cima será

```
4.0
2.0
```

Passagem por referência:

Por outro lado, se queremos alterar o valor da variável utilizada para chamar a função, podemos fazê-lo passando o endereço da variável para a função. Todas as alterações ao conteúdo deste endereço são permanentes.

```
void write_square(double *p)
 /* p is a pointer-to-double */
{
    /* change the contents of address p: */
  *p = *p * *p;
    /* show the contents of address p: */
  printf("%f", *p);
}

void main()
{
  double x;

  x = 2.0;
  /* now we have to pass the address (&) of the variable x: */
  write_square(&x);
  printf("%f", x);
}
```

Se se correr o programa, o resultado será

```
4.0
4.0
```

porque o valor de `x` foi alterado simultaneamente com o conteúdo do endereço p.

Passagem por valor *Passagem por referência*

Na alanogia das caixas a representar variáveis: passagem por referência é entregar a caixa (variável) para a funçao que poderá então utilizar e alterar o valor dentro da caixa, enquanto passagem por valor é equivalente a abrir a caixa, copiar o valor lá contido e entregar apenas esse valor à função. Obviamente, que assim o valor original fica na caixa.
Um outro exemplo: Se eu disser quanto tenho na minha conta bancária, podem utilizar esse valor para calcular quanto representa em dólares, ou posso dar-lhes o direito de alterarem a quantia da minha conta bancária, e neste caso, a quantia irá provavelmente ser alterada.

`scanf()` revisitado

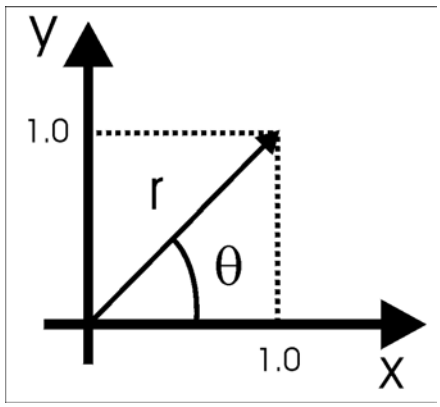
Com estes conceitos bem presentes, vamos ver de novo a função `scanf()` que recebe input do utilizador. De recordar que temos sempre que usar a forma

```
scanf("%d", &i);
```

Ou, por outras palavras, temos sempre que fornecer à função `scanf` o endereço (&) da variável (`i`). Agora já faz sentido. Nós queremos alterar o valor de `i` com a função `scanf`. Portanto, temos que utilizar a técnica de passagem por referência. Assim sendo, temos que fornecer a `scanf` o endereço de `i`, em vez do valor de `i`.

Example: Carthesian to polar coordinates

As we know, functions can only return a single value. What if we want to return more than one value to the calling part of the program? Take for example the conversion from Carthesian to polar coordinates. As input we have a coordinate (x, y) and as output we have a Carthesian coordinate (r, θ). These are two values, one value for the radius (r) and one for the angle (θ). The following solution shows how we can pass this information without using global variables:




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

void convert_to_polar(float x, float y, float *r, float
*theta)
/*********************************************************
 * function to convert carthesian coordinate (x,y) to    *
 * polar coordinate (r,theta)                            *
 * parameters:                                           *
 *    x,y: floats, passed by value                       *
 *    r, theta: (pointers to) floats, passed by reference *
 *                  (changes are permanent)              *
 *********************************************************/
{
  *r = sqrt(x*x + y*y);
  *theta = atan(y/x);
}

void main()
{
  float rl, thetal;

  convert_to_polar(1.0, 1.0, &r1, &theta1);
  printf("polar coordinate is (%f, %f)\n", rl, thetal)
```

```
}
```

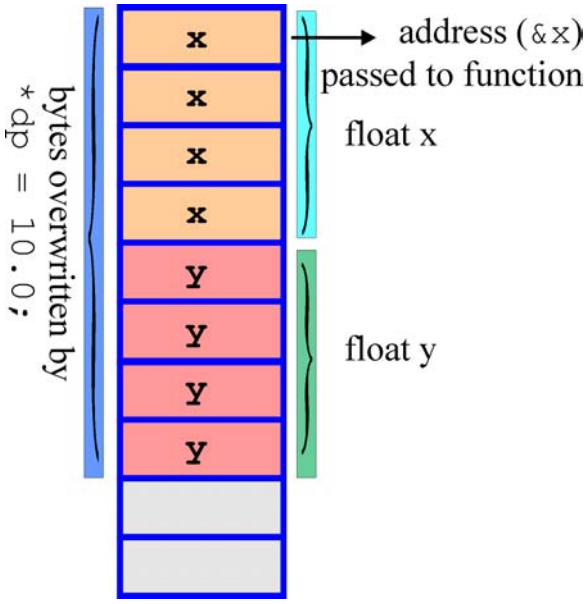
output:

```
polar coordinate is (1.4142, 0.78539)
```

Type mixing

When passing information to functions we have to take even more care that we don't do any mixing of types, especially when we use pointers. As a (bad) example:

```
void double10(double *dp)
{
  *dp = 10.0;
}

void main()
{
  float y=10.0;
  float x=10.0;

  double10(&x);
  printf("%f %f\n", x, y);
}
```



A double occupies 8 bytes. Assigning a value to the contents of a pointer-to-double will therefore write in 8 consecutive bytes of memory. The pointer passed to the function is of type pointer-to-float, however. We have (by declaring the variable `x` of type `float`) only reserved 4 bytes of space in memory. The instruction `*dp = 10;` will now write in these 4 bytes **and** the 4 bytes next to it (that don't belong to `x` but to `y`).

The output of the program above (Borland C++ 3.01 for MS-DOS):

```
0.000000 2.562500
```

Always avoid mixing of type:

Give to the function what the function wants

Teste Rápido:

Para testar os conhecimentos sobre o que aprendeu nesta aula, prima aqui para fazer um teste on-line. De notar que este **Não** é o formato que será utilizado no teste final!

Peter Stallinga. Universidade do Algarve, 19 November 2002

Aula 16: Programação Recursiva

Traduzido por Ana Paula Costa

Recursividade

Uma função é recursiva se é definida em termos de si mesma (i.e.,se se chama a si própria)

Exemplo 1: Factorial

O exemplo clássico é o cálculo da função factorial *n*! Nós podemos implementar a função com um ciclo, tal como foi descrito nas aulas 11 e 12:

```
int factorial(int n)
 /* returns n! */
{
  int i, result;

  result = 1;         /* initialize the variable */
  for (i=1; i<=n; i++)
    result = i*result;
  return(result);
}
```

Esta função de facto irá retornar o factorial do argumento, por exemplo `factorial(5)` = 120.

Uma solução muito mais interessante é através da definição da função factorial em termos de si mesma, tal como aprendemos na escola,

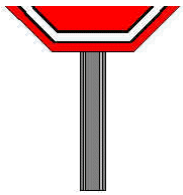
$n! = n*(n-1)!$

Vamos fazer exactamente isso:

```
int factorial(int n)
 /* should return n! */
{
    /* the value to be returned is expressed in terms of itself: */
  factorial = n*factorial(n-1);
}
```

Esta função está quase correcta. O único problema é que nunca irá parar de se chamar a si própria. Por exemplo, podemos chamá-la com `factorial(4)`, que irá depois tentar calcular `4*factorial(3)` e assim chamar `factorial(3)`. `factorial(3)` irá tentar calcular `3*factorial(2)`, que irá chamar `factorial(2)`, ... que irá chamar

`factorial(1)` ... que irá chamar `factorial(0)` ... que irá chamar `factorial(-1)` ... que irá chamar `factorial(-2)` ... e por aí fora. O programa nunca irá retornar nada e o computador irá encravar, gerando provavelmente um erro do tipo "stack overflow". Claramente, temos que construir a função de forma a que em determinada altura ela páre de se chamar a si própria. Relembrar que na forma matemática de definir uma função em termos de si mesma temos sempre que incluir uma forma de paragem, para a função factorial é



$1! = 1$

Vamos colocar a forma de paragem na nossa função:

```
int factorial(int n)
 /* returns n! */
{
  if (n==1)
    return(1);
  else
    return(n*Factorial(n-1));
}
```

A ideia que importa reter é que se deve incluir na função a possibilidade de ela obter um resultado e terminar o cálculo recursivo. Tal como temos que dar a um ciclo a possibilidade de terminar, temos que dar essa possibilidade também às funções recursivas. Caso contrário, o programa continuará para sempre (ou irá encravar).

Exemplo 2: Fibonacci

Recordando as aulas práticas, a definição do número de Fibonacci é feita em termos de si mesmo:
$f_n = f_{n-2} + f_{n-1}$
com as condições de paragem
$f_1 = 1$
$f_2 = 1$
Por exemplo,
$f_3 = 1 + 1 = 2$
$f_4 = 1 + 2 = 3$
$f_5 = 2 + 3 = 5$
$f_6 = 3 + 5 = 8$
$f_7 = 5 + 8 = 13$

Podemos implementar isto numa função, reparar na condição de paragem:

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

Variáveis

As variáveis que são declaradas dentro de funções recursivas são todas locais. Mais ainda, sempre que a

função é chamada, uma nova instância da variável é criada e existe até que a função termine. (*Em C podemos prevenir esta situação com a palavra "static" antes da declaração de variáveis*). Cada uma destas variáveis, embora tenham o mesmo nome, irão apontar para um espaço diferente na memória. Para exemplificar, vamos criar uma variável local na nossa função factorial:

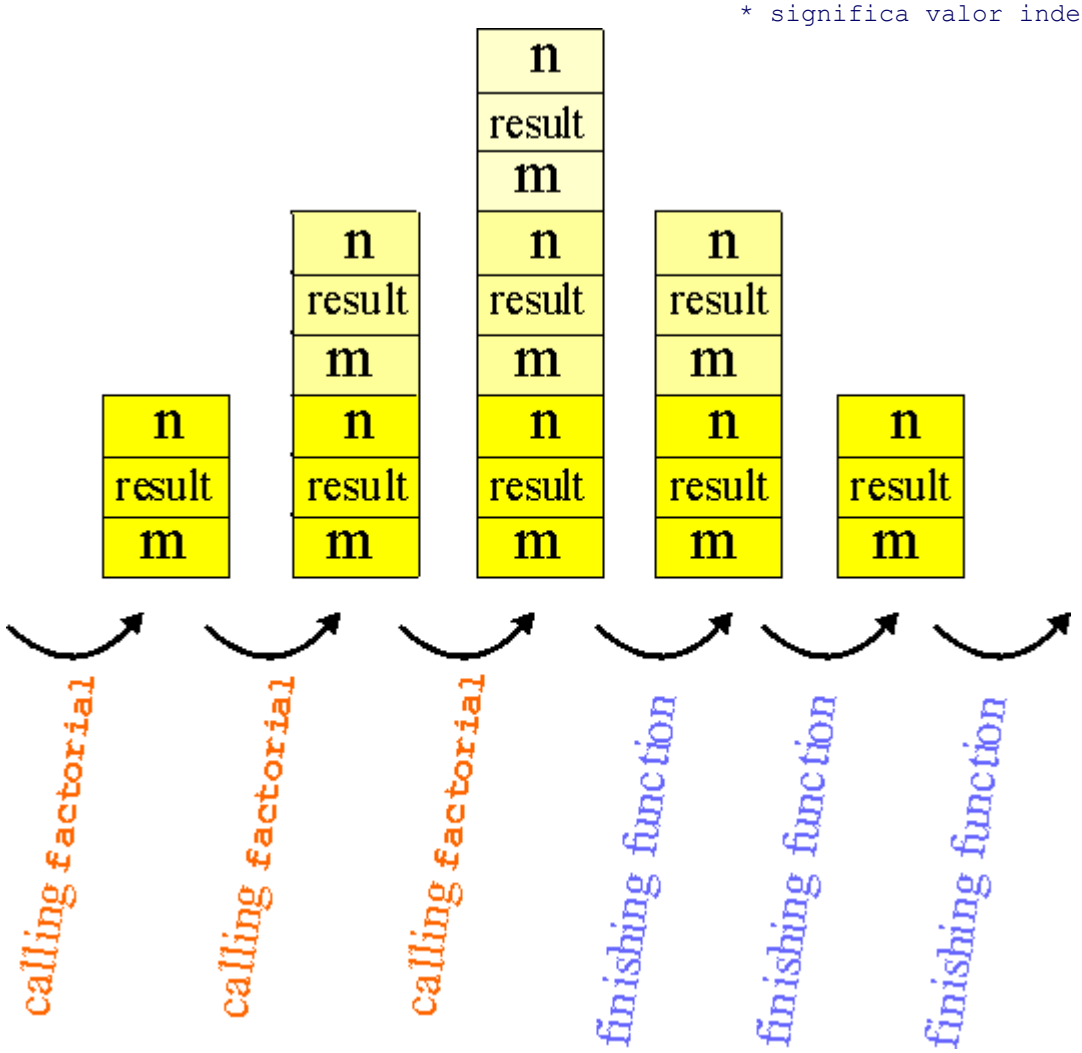
```
int factorial(int n)
{
  int m, result;

  m = 2*n;
  printf("%d ", m);
  if (n==1) then
    result = 1;
  else
    result = n*factorial(n-1);
  printf("%d ", m);
  return(result);
}
```

Quando chamamos a função com argumento 3, irá acontecer o seguinte:

instrução	variáveis
factorial(3) é chamada	
são criadas as variáveis n, m e result	m=* r=* n=*
3 é atribuído a n (da chamada da função) 2*3 é atribuído a m	m=6 r=* n=3
factorial(2) é chamada	m=6 r=* n=3
são criadas as variáveis n, m e result (novas; diferentes das anteriores!)	m=* r=* n=* m=6 r=* n=3
2 é atribuído a n (da chamada da função) 2*2 é atribuído a m	m=4 r=* n=2 m=6 r=* n=3
factorial(1) é chamada	m=4 r=* n=2 m=6 r=* n=3
são criadas as variáveis n, m e result (diferentes das de cima)	m=* r=* n=* m=4 r=* n=2 m=6 r=* n=3
1 é atribuído a n (da chamada da função) 2*1 é atribuído a m	m=2 r=* n=1 m=4 r=* n=2 m=6 r=* n=3
... chegámos à condição de paragem e result = 1;	m=2 r=1 n=1 m=4 r=* n=2 m=6 r=* n=3
é escrito o valor de m: 2	m=2 r=1 n=1 m=4 r=* n=2 m=6 r=* n=3
o valor de result (1) é retornado para a instrução de chamada factorial(1) termina, os últimos n, m e result são destruídos	m=4 r=* n=2 m=6 r=* n=3
result é calculado: n*factorial(n-1) fica n*'valor retornado' -> 2*1 = 2	m=4 r=2 n=2 m=6 r=* n=3
é escrito o valor de m: 4	m=4 r=2 n=2 m=6 r=* n=3
o valor de result (2) é retornado para a instrução de chamada factorial(2) termina, os últimos n, m e result são destruídos	m=6 r=* n=3
result é calculado: n*factorial(n-1) fica n*'valor retornado' -> 3*2 = 6	m=6 r=6 n=3
é escrito o valor de m: 6	m=6 r=6 n=3

o valor de result (6) é retornado para a instrução de chamada factorial(3) termina, os últimos n, m e result são destruídos	
o valor retornado por factorial(3) é 6	

r significa variável 'result'
* significa valor indeterminado



O que aprendemos daqui é que as variáveis não são objectos estáticos na memória, que apontam para certos endereços, mas em vez disso, as variáveis são criadas na altura em que corremos o programa. Cada vez que entramos na função, novas variáveis são criadas e irão existir até que se saia da função. Mais ainda, como se pode ver no exemplo em cima, as últimas variáveis criadas são as usadas localmente. (Isto aplica-se apenas a linguagens que têm variáveis dinâmicas, como o C ou PASCAL).

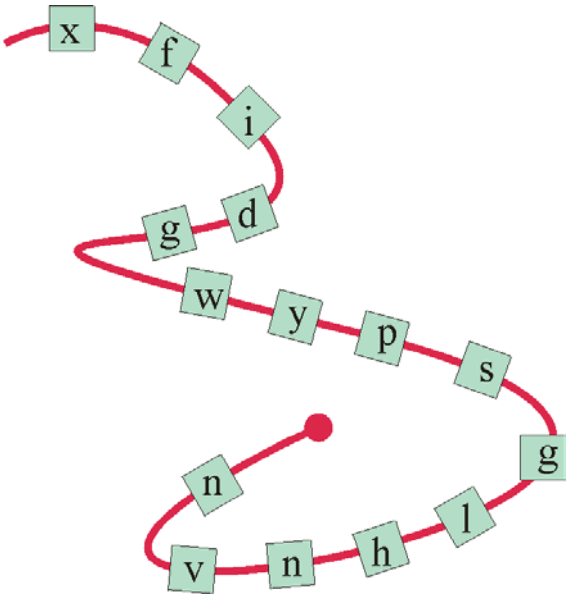
Teste Rápido:

Para testar os conhecimentos sobre o que aprendeu nesta aula, prima aqui para fazer um teste on-line. De notar que este **Não** é o formato que será utilizado no teste final!

Peter Stallinga. Universidade do Algarve, 22 November 2002

Aula 17: strings (cadeias de caracteres)

Traduzido por Ana Paula Costa



string

Uma string é um array de `char`

Vamos ver as implicações desta simples definição.

Para exemplificar: declarando uma string de 100 caracteres:

```
char n[100];
```

Ingenuamente poderíamos pensar que atribuir um valor a esta variável poderia ser feito com

```
n = "Benfica";
```

mas não podemos esquecer que `n` é um ponteiro para char (ver aula 15) e o lado direito do = não é (é uma constante do tipo string). A forma de se fazer a atribuição seria atribuir um valor a cada elemento do array:

```
n[0] = 'B';
n[1] = 'e';
n[2] = 'n';
n[3] = 'f';
n[4] = 'i';
n[5] = 'c';
n[6] = 'a';
n[7] = '\0';
```

Do lado esquerdo de cada = está um elemento do array (char) e do lado direito está uma constante do tipo char. Desta forma as atribuições estão correctas. (De notar no caracter `'\0'` no final, que significa fim-da-string.)
No entanto, esta forma de trabalhar não é muito prática. Por esta razão, e em geral para facilitar o trabalho com strings, existem muitas instruções para ajudar na manipulação de strings. Essas instruções estão definidas na biblioteca `string.h` e quando as queremos utilizar temos que fazer o "include" da biblioteca no início do programa com

```
#include <string.h>
```

string.h

As seguintes funções de `string.h` são interessantes:

`strcpy`	copia o conteúdo de uma string para outra
`strcat`	acrescenta uma string a outra
`strcmp`	compara duas strings
`strlen`	retorna o comprimento (número de caracteres, excluindo \0) de uma string
`strstr`	procura pela posição de uma string numa outra string

strcpy

diminuitivo de "**str**ing **cpy**". copia o conteúdo de uma string para outra.

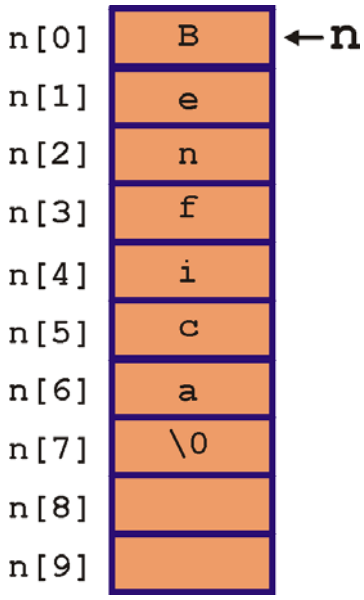
definição da função: `int *strcpy(char *s1, const char* s2)`
copia o conteúdo de `s2` para `s1`.

É uma função que tem dois parâmetros, duas strings (array de char, pointer para char, portanto) `s1` e `s2`, e copia o conteúdo de `s2` para `s1`. As alterações em `s1` são permanentes. A função retorna um `int` que indica o sucesso da operação. Para já podemos ignorar este resultado, para não complicar mais. Iremos ignorar também a palavra `const` da declaração do segundo parâmetro.
Um exemplo:

```
  char n[10];
  strcpy(n, "Benfica");
  printf("%s\n", n);
```

resultado

```
  Benfica
```



Efectivamente, o que a função `strcpy` está a fazer é copiar os caracteres da string s2 para s1 até que seja encontrado o caracter fim-da-string:

```
do
{
  *s1 = *s2;          // copy a character; copy contents of
                      //   address s2 to address s1
  s1++;               // make pointer s1 point to next character
  s2++;               // make pointer s2 point to next character
}
while (*s2 != '\0');  // repeat until end-of-string encountered
```

strcat

diminuitivo de "**str**ing con**cat**enate". adiciona uma string a outra string

definição da função: `char *strcat(char *s1, const char *s2)`
acrescenta `s2` ao final de `s1`.

Dois parâmetros, `s1` e `s2` são ponteiros para char (arrays de char). Mais uma vez, como em cima, iremos ignorar o valor (ponteiro char) que a função retorna.
Exemplo:

```
  char n[30];
  strcpy(n, "Benfica");
  strcat(n, " o glorioso");
  printf("%s\n", n);
```

resultado

```
  Benfica o glorioso
```

Efectivamente, a função `strcat` é equivalente ao seguinte código:

```
while (*s1 != '\0')  // look for
  s1++;              //   end-of-string
strcpy(s1, s2);      // now copy the string s2 at place of the new
                     //   address s1 that points to \0
```

strcmp

diminuitivo de "**str**ing **c**o**mp**are". compara duas strings

definição da função: `int strcmp(const char *s1, const char *s2)`
compara `s1` com `s2`. Se iguais retorna 0, se alfabeticamente `s1<s2` retorna um número negativo, ou retorna um número positivo se `s1>s2`.

Exemplo:

```
  char n[30], m[30];
  int cmp;

  strcpy(n, "Benfica");
  strcpy(m, "Sporting");
  cmp = strcmp(n, m);
  printf("%s\n", n);
  if (cmp==0)
    printf("strings are equal");
  else
    if (cmp>0)
      printf("%s is after %s", n, m);
    else
      printf("%s is before %s", n, m);
```

resultado

```
  Benfica is before Sporting
```

A função `strcmp` poderia ser implementada com algo semelhante ao que se segue:

```
  // possible implementation of strcmp
while ((*s1==*s2) && (*s1!='\0'))  // still equal and not end-of-string?
{
  s1++;    // move pointer s1 one place
  s2++;    // move pointer s2 one place
}
return((int) *s1 - *s2);  // return difference between contents of s1 and s2
```

strlen

diminuitivo de "**str**ing **len**gth". Retorna o número de caracteres da string (sem contar com \0)

definição da função: `int strlen(const char *s1)`
Retorna o número de caracteres da string `s1` (sem contar com \0)

Exemplo:

```
  char n[30];
  strcpy(n, "Benfica");
  printf("%s has %d characters", n, strlen(n));
```

resultado

```
  Benfica has 7 characters
```

A função `strlen` poderia ser implementada com algo semelhante a:

```
cnt=0;
while (*s1 != '\0')
{
  cnt++;
```

```
  s1++;
}
return(cnt);
```

strstr

diminuitivo de "**str**ing **str**ing". Retorna a posição de uma string numa outra string.

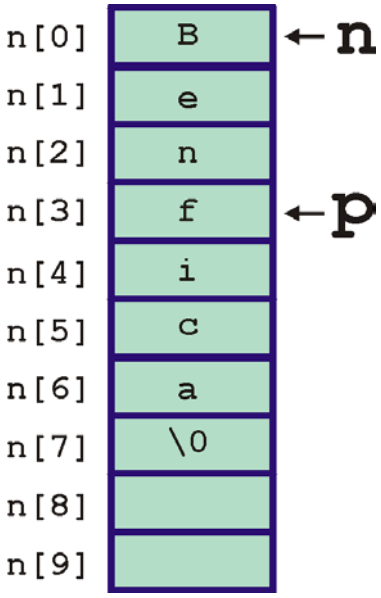
definição da função: `char *strstr(const char *s1, const char *s2)`
Retorna um ponteiro (array de char) para a posição do texto `s2` no texto `s1`.

Exemplo:

```
  char n[30];
  char *p;
  strcpy(n, "Benfica");

  p = strstr(n, "fic");
  printf("%s", p);
```

resultado

```
  fica
```

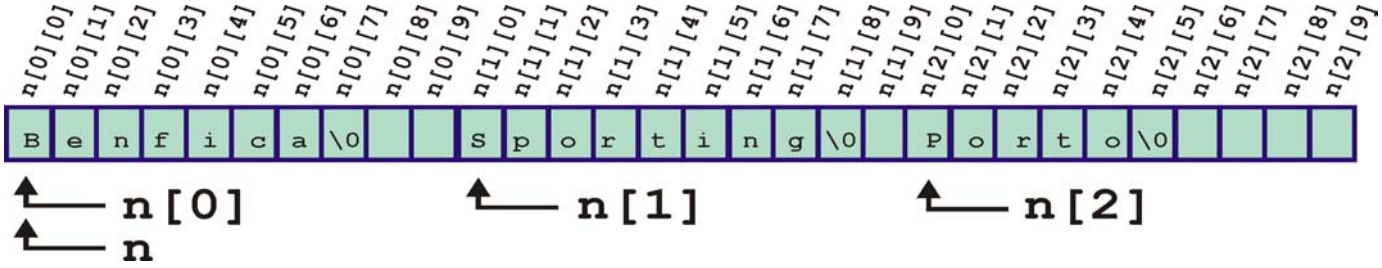


Arrays de strings

Arrays de strings são arrays de arrays de caracteres:
Por exemplo, se quisermos armazenar os nomes de três equipas de futebol portuguesas, podemos fazê-lo com

```
  char n[3][10];
```

A figura em baixo mostra como os nomes podem ser armazenados neste array de strings.




```
for (i=0; i<3; i++)
  printf("%s\n", n[i]);
```

resultado:

```
 Benfica
 Sporting
 Porto
```

Teste Rápido:

Para testar os conhecimentos sobre o que aprendeu nesta aula, prima aqui para fazer um teste on-line. De notar que este **Não** é o formato que será utilizado no teste final!

Peter Stallinga. Universidade do Algarve, 29 November 2002

Aula 18: Estruturas

Traduzido por Ana Paula Costa

Estrutura (`struct`)

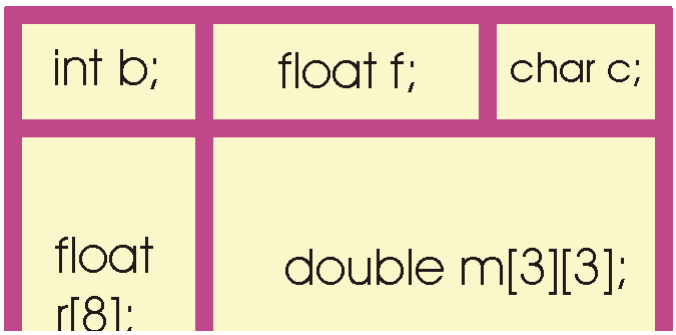
Na aula 13 aprendemos que um array pode armazenar variáveis do mesmo tipo de uma forma bem ordenada e indexada, como um arquivo em que cada gaveta contém o mesmo tipo de informação. Se quisermos agrupar variáveis que não são do mesmo tipo, podemos fazê-lo com uma **estrutura (struct)**.

Os três arquivos armazenando coisas
***do mesmo tipo**, tal como os **arrays**. O da esquerda podia*
conter "bytes", o do meio
"inteiros" e o da direita "reais".

O arquivo do centro é usado para armazenar coisas de vários tipos. Da mesma forma,
*uma **estrutura** é usada para armazenar variáveis*
*de **tipos diferentes**, inteiros, reais, ou qualquer outro, todas juntas na mesma caixa.*

Uma estrutura é um conjunto de variáveis de tipos distintos.

Declaração de uma estrutura



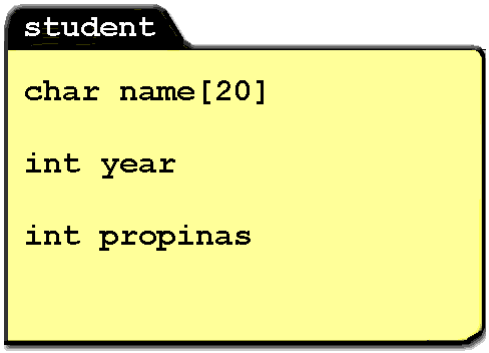
Cada variável dentro de uma estrutura é chamada campo. Na figura ao lado temos 5 campos: um int (b), um float (f), um char (c), um array unidimensional de floats (r) e um array de duas dimensões de doubles (m).

Para declarar uma estrutura fazemos o seguinte:

```
struct {
    tipo1 item1;
    tipo2 item2;
       |
    tipoN itemN;
} nome;
```

com
nome: o nome da variável que suporta a estrutura.
item1..itemN: o nome dos vários campos da estrutura. Estes têm as mesmas regras dos outros identificadores para variáveis, funções, etc. De notar que podemos colocar tantos campos na estrutura quantos desejarmos, com qualquer combinação de tipos.
tipo1..tipoN: o tipo dos campos da estrutura. Pode ser qualquer tipo de variável que conhecemos, incluindo estruturas!

Por exemplo, a definição de uma estrutura contendo a informação de um estudante pode ter campos para o nome, ano, e propinas pagas:




```
struct {
  char name[20];
  int year;
  int propinas;
} student;
```

Esta estrutura pode conter apenas a informação de um único estudante. Mais tarde iremos ver como criar arrays de estruturas.

Utilização de uma estrutura

Para aceder a uma estrutura utilizamos o formato

```
nome.campo
```

Por exemplo, para atribuir valores à estrutura `estudante` podemos fazer o seguinte:

```
strcpy(student.name, "Peter Stallinga");
student.year = 2002;
student.propinas = 1;
```

para recordar o funcionamento da função `strcpy`, ver a aula sobre strings.
Outro exemplo:

```
struct {
    float x;
    float y;
} coordinate;

coordinate.x = 1.0;
coordinate.y = 0.0;
```

Isto não é propriamente um conjunto de variáveis de diferentes tipos, portanto também podíamos utilizar um array:

```
float coordinate[2];

coordinate[0] = 1.0;
coordinate[1] = 0.0;
```

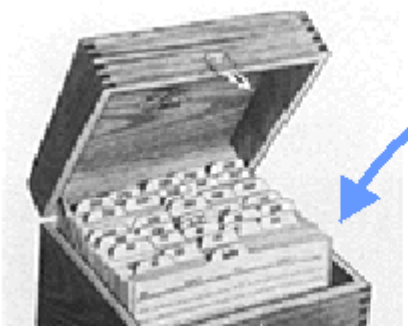
mas, a primeira versão, com a estrutura, é mais lógica.
Ainda outro exemplo:

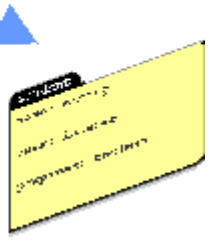
```
struct {
  char rua[20];
  int numero;
  int andar;
  char porta;
} address;

strcpy(address.rua, "Rua Santo Antonio");
address.numero = 34;
address.andar = 3;
address.porta = 'E';

printf("%s %d", address.rua, address.numero);
printf("%d %c", address.andar, address.porta);
```

```
Rua Santo Antonio 34
3 E
```

Arrays de estruturas, estruturas de arrays

Os exemplos anteriores utilizam estruturas simples. Contudo, com a informação da aula sobre arrays (aula 13), sabemos como construir um array que pode armazenar informação sobre muitos objectos iguais do mesmo tipo, mesmo estruturas. Vamos criar um array de 2000 estudantes:
(Questão: Quantos bytes ocupa esta variável em memória? Resposta no final)





```
struct {
  char name[20];
  int year;
  int propinas;
} students[2000];
```

```
i = 1055;
strcpy(students[i].name, "Peter Stallinga");
students[i].year = 2002;
students[i].propinas = 1;
```

De notar a estrutura de arrays de estruturas. `students` é um array de estruturas, portanto, `students[i]` é uma destas estruturas e se queremos atribuir algum valor a um campo utilizamos o ponto e o nome do campo, assim `students[i].year` é um inteiro que contém o ano do estudante número i.
A sintaxe `students.year[i]` estaria errada (podíamos utilizar desta forma se tivéssemos uma única estrutura `students` contendo um campo `year` que fosse um array)
Igualmente errado seria: `students.[i]year`, que não faz qualquer sentido. Cuidado com o sítio onde se colocam os pontos.

Mais tarde iremos ver que podemos facilmente declarar variáveis do tipo array de estruturas (ver aula 19).

Teste Rápido:
Para testar os conhecimentos sobre o que aprendeu nesta aula, prima aqui para fazer um teste on-line. De notar que este **Não** é o formato que será utilizado no teste final!
Resposta à questão no texto: 2000 x (20+2+2) = 48000 bytes.

Peter Stallinga. Universidade do Algarve, 28 November 2002

Aula 19: Definindo novos tipos e estruturas

Traduzido por Ana Paula Costa

Tipo

Por vezes é útil poder definir um novo tipo de variável que possa ser depois utilizado no programa. É útil para tornar o código mais legível, ou para evitar ter que se forçar muitas vezes o tipo. A definição de novos tipos de variáveis pode ser feito com a palavra typedef.

```
 typedef descrição
nomedotipo;
```

com `nomedotipo` o nome que queremos dar ao novo tipo e `decrição` qualquer um dos tipos de variável que aprendemos até agora, incluindo arrays, ponteiros e todos os tipos de variável simples.

Exemplos:

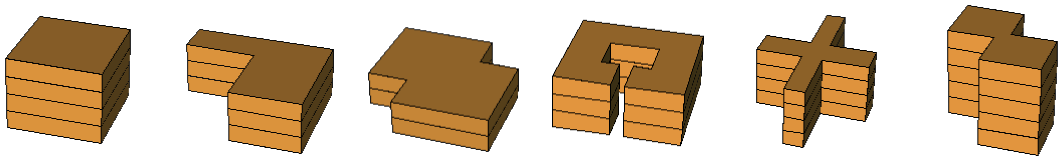
```
typedef float real;
```

Isto pode ser útil para pessoas que estão acostumadas a programar em PASCAL. Após se escrever a linha de cima, podemos usar variáveis do tipo real, exactamente como em PASCAL. Variáveis do 'tipo' real serão traduzidas pelo compilador em variáveis do tipo float.

```
typedef float floatarray[10];
```

Define um novo tipo de variável. O novo tipo é chamado floatarray e uma variável deste tipo será igual a um array de 10 floats.

De notar que a definição de um novo tipo **não cria uma variável!** Não reserva espaço em memória e não atribui um nome a uma variável. É apenas uma **descrição** de um tipo que será depois utilizado para declarar uma variável.



... definindo novas caixas para variáveis.

Utilizando um novo tipo

Após definir um tipo, podemos declarar variáveis desse tipo:

```
nomedotipo nomevar;
```

com `nomevar` o nome da nova variável e `nomedotipo` o tipo dessa variável, como foi descrito antes.
Depois da declaração, podemos utilizar a variável tal como se tivesse sido declarada da forma normal.

Exemplos:
depois da declaração do novo tipo `typedef float real;` podemos usar

```
real x;
```

Isto já se parece mais com PASCAL (*em PASCAL seria "`Var x: real`"*)

```
floatarray ra;
```

E no código podemos utilizar este array:

```
ra[1] = 2.68;
```

Isto é equivalente a

```
float ra[10];
ra[1] = 2.68;
```

Mais exemplos

```
/* example with typedef */

/* definig a new type of variable: */
typedef int ra[6];
/* declare two global variables: */
ra x;
int y[7];

int AreEqual(ra r)
  /* Note that the definition can also be used for parameters */
{
  if (r[0]==r[1])
    return(1);
  else
    return(0);
}

void main()
{
  x[1] = 1;
  x[2] = 0;
  if AreEqual(x)
    printf("First two elements are equal");
  else
    printf("First two elements are different");
  y[1] = 1;
  y[2] = 0;
   /* The following instruction is bad code, because the type of value we pass to
      the function is different than the type of value the function expects: *\
  AreEqual(y);
}
```

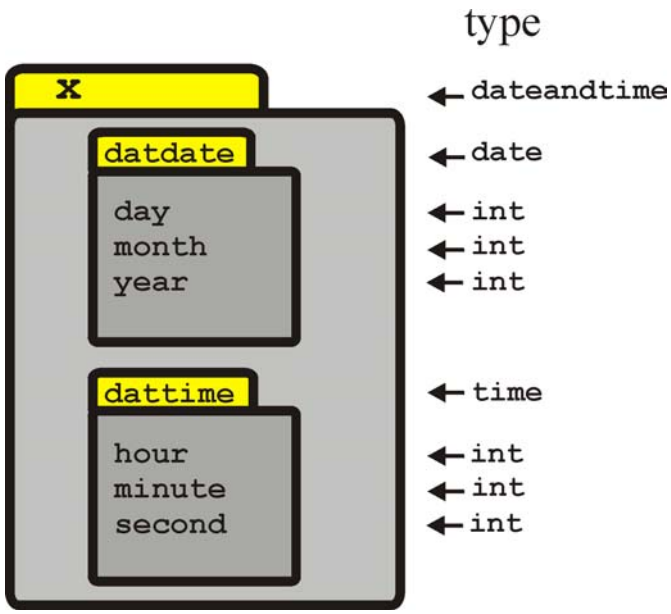
```
#include <stdio.h>
```

```
typedef struct {
  hour, minute, second: integer;
} time;

void showtime(time t);
  /* Will show the time in format h:m:s */
{
  printf("%d:%d.%d\n", t.hour, t.minute, t.second);
}

void main()
{
 time atime;

 atime.hour = 23;
 atime.minute = 16;
 atime.second = 9;
 showtime(atime);
}
```

---

estrutura de estruturas:

```
typedef struct {
   int day, month, year;
} date;

typedef struct {
   int hour, minute, second;
} time;

typedef struct {
    time dattime;
    date datdate;
} dateandtime;

dateandtime x;

x.dattime.hour = 1;
```



`x` é uma variável do tipo `dateandtime` que é uma estrutura que contém dois campos. Um campo é `dattime` que é do tipo `time`. O outro campo é do tipo `date`. Um dos campos da estrutura time é `hour` que é um `int`.

Teste Rápido:

Para testar os conhecimentos sobre o que aprendeu nesta aula, prima aqui para fazer um teste on-line. De notar que este **Não** é o formato que será utilizado no teste final!

Peter Stallinga. Universidade do Algarve, 30 November 2002

Aula 20: Ficheiros

Traduzido por Ana Paula Costa

Saída para ficheiros

Hoje vamos aprender como ler de ficheiros e como escrever para ficheiros. Como exemplo vamos apenas aprender como ler e escrever ficheiros de texto, ficheiros em que a informação é armazenada em formato ASCII. Estes ficheiros diferem do formato binário porque também podem ser legíveis para os humanos. Quando escrevemos os nossos programas nas aulas práticas temos utilizado ficheiros de texto.
Estes ficheiros podem ser colocados em disquetes, discos rígidos, ou até mesmo CD-ROMs (caso em que podem - claro - ser apenas lidos e não escritos).

Palavras chave

As seguintes palavras chave estão relacionadas com o acesso a ficheiros:

`FILE`	`feof()`
`fopen()`	`fputs()`
`fclose()`	`fgets()`
`fscanf()`	`fputc()`
`fprintf()`	`fgetc()`

Elas estão definidas na biblioteca <stdio.h>, que deve portanto ser incluída no início do programa.

Declarar uma variável para acesso a um ficheiro:

Antes de se poder abrir um ficheiro e ter acesso a ele (seja para ler ou para escrever) temos que declarar um "handle" para esse ficheiro. O "handle" é uma variável que armazena informação sobre o estado do ficheiro. Em C podemos declarar um ficheiro de texto da seguinte forma:

```
FILE *filehandle;
```

Com `filehandle` a variável que irá armazenar o ponteiro para a informação. Isto **NÃO** é igual ao nome do ficheiro, como iremos ver. Esta declaração deve ser feita no mesmo local das outras variáveis.
Exemplo:

```
FILE *f;
```

Isto cria `f` um ponteiro para um handle de ficheiro.
Só a título de curiosidade, não é preciso saber isto, o tipo de variável FILE está definido como

```
typedef struct{
  short          level;
  unsigned       flags;
```

```
  char           fd;
  unsigned char  hold;
  short          bsize;
  unsigned char *buffer, *curp;
  unsigned       istemp;
  short          token;
} FILE;
```

Para mais informação, ver a ajuda do compilador.

Abrir um ficheiro

No programa abrimos o ficheiro com a função fopen() pertencente a <stdio.h>

`fopen()`

diminutivo de "**f**ile **open**". Abre um ficheiro.

definição da função: **`FILE *fopen(const char *filename, const char *mode)`**
abre um ficheiro. Retorna 0 se não for bem sucedida.

`fopen()` retorna um ponteiro para um FILE (ao qual podemos atribuir a nossa variável do tipo ponteiro-FILE). `fopen()` recebe dois parâmetros, ambos strings (pointeiros para char). O primeiro é o nome do ficheiro e o segundo é a forma como deve ser aberto, ("r") para leitura, ("w") para escrita, ou ("a") para acrescentar ao final do ficheiro.
Exemplos:
Para abrir um ficheiro chamado "OLA.TXT" para leitura usamos

```
FILE *f;
f = fopen("OLA.TXT", "r");
```

Para abrir um ficheiro chamado "output.asc" para escrita usamos

```
FILE *f;
f = fopen("output.asc", "w");
```

Ler e Escrever

Ler de um ficheiro e escrever para um ficheiro é feito exactamente da mesma forma do que ler e escrever para o ecrãn, com a única diferença que se utiliza a função `fscanf()` em vez de `scanf()` para ler e `fprintf()` em vez de `printf()` para escrever. O primeiro parâmetro destas funções tem que ser o handle do ficheiro (por exemplo `f`)

`fscanf()`

diminutivo de "**f**ile **scan** **f**ornatted". input a partir do ficheiro

definição da função: **`int fscanf(FILE *stream, const char *format, ....)`**
obtém input formatado do ficheiro

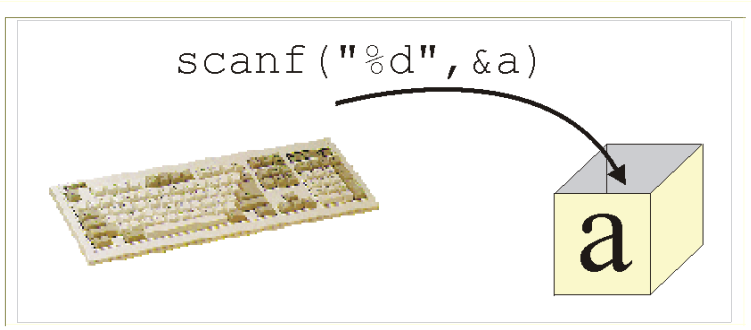
`fprintf()`

diminutivo de "**f**ile **print** **f**ormatted". Output formatado para o ficheiro

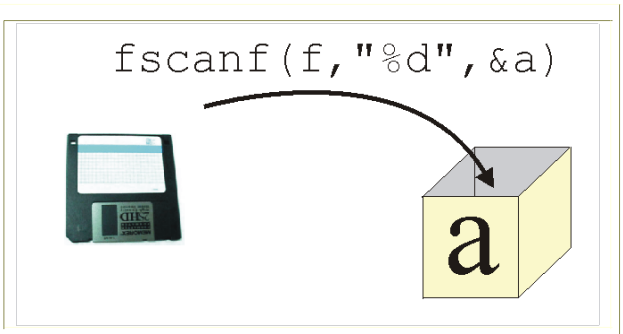
definição da função: **`int fprintf(FILE *stream, const char *format, ...)`**
o resultado vai para um ficheiro . Retorna o número de caracteres escritos, EOF se falhar.

De notar que só podemos usar instruções `fscanf` para ficheiros que tenham sido previamente abertos para leitura ("r") e `fprintf` é para ser utilizada apenas com ficheiros que tenham sido abertos para escrita ("w"). Exemplos:

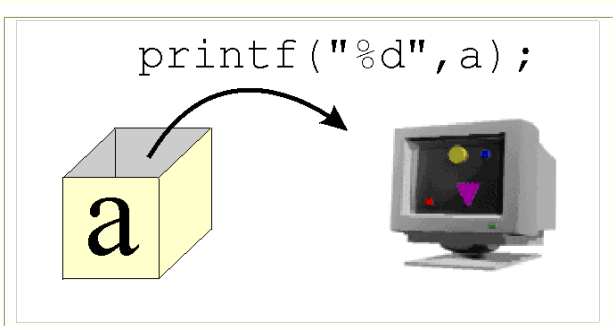
```
fprintf(f, "%f", r);
fscanf(f, "%d", opcao);
```



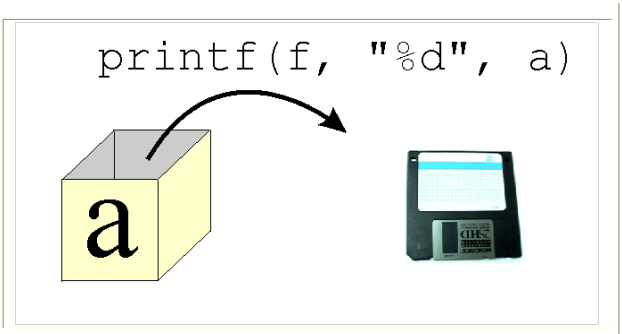
Lê uma variável com o teclado



Lê uma variável do ficheiro



Escreve uma variável para o ecrãn



Escreve uma variável para o ficheiro

Fechar o ficheiro

Quando já não necessitamos do ficheiro temos que fechá-lo. Especialmente para ficheiro de output. Se nos esquecemos de fechar o ficheiro antes de terminar o programa, provavelmente haverá informação que não será escrita no ficheiro (o "buffer" não será esvaziado). Para fechar o ficheiro utilizamos `fclose()`.

`fclose()`

diminutivo de "**f**ile **close**". Fecha um ficheiro.

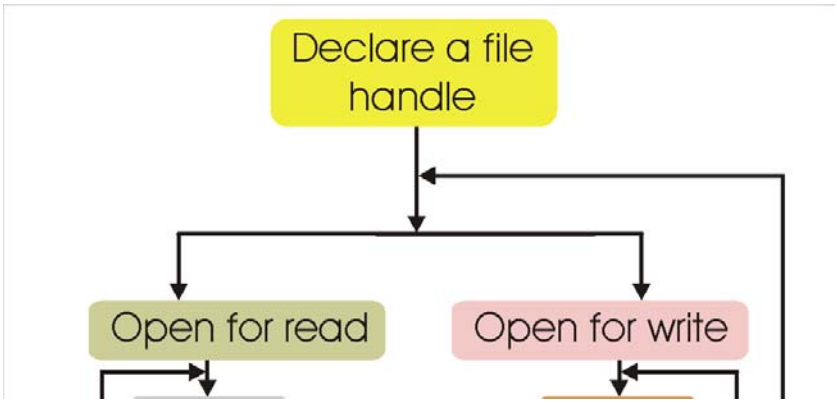
definição da função: **`int fclose(FILE *stream)`**
fecha um ficheiro. Retorna 0 se tiver sucesso.

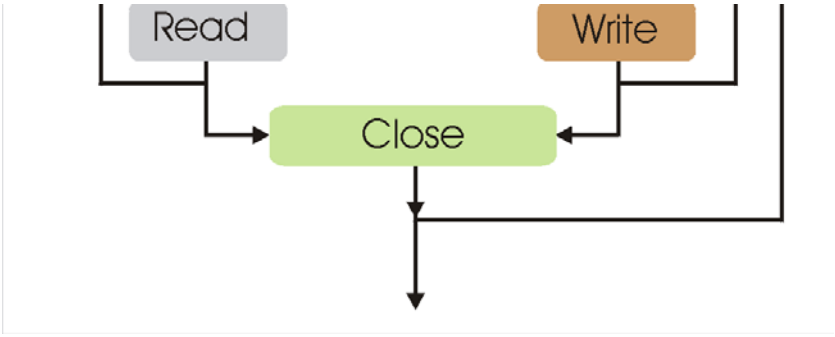
por exemplo:

```
fclose(f);
```

Sumário

Entrada e saída de/para ficheiros consiste dos seguintes passos:





Testar Fim-do-ficheiro (End-of-file)

A seguinte instrução pode ser útil

feof(filehandle): Retorna verdadeiro se estamos a ler o fim do ficheiro.
eof significa end-of-file (fim-do-ficheiro)

Exemplo:

```
while (!eof(f))
{
  fscanf("%s", s);
}
```

que irá ler do ficheiro até que seja encontrado o fim do ficheiro.

Exemplos

código	*ecrãn*	*ficheiro TEST.TXT* ***depois*** *de correr o programa*

`/* With File Output */`
`#include <stdio.h>`
 	
`FILE *f;`
`char s[100];`
`int i;`
 	
`void main()`
`{`
`  printf("Name of File:");`
`  scanf("%s", s);`
`  f = fopen(s, "w");`
`  for (i=1; i<=10; i++)`
`    fprintf(f, "%d Hello", i);`
`  fclose(f);`
`}`
	`Name of File:`
	`TEST.TXT`
		`1 Hello`
		`2 Hello`
		`3 Hello`
		`4 Hello`
		`5 Hello`
		`6 Hello`
		`7 Hello`
		`8 Hello`
		`9 Hello`
		`10 Hello`

código	*ecrãn*	*ficheiro TEST.TXT* ***antes*** *de correr o programa*

`/* With File Input */`
`#include <stdio.h>`
 	
`FILE *f;`
`char s[100];`
`int i;`
 	
`void main()`
	`Name of File:`
	`TEST.TXT`
	`1`
	`Hello`
	`2`
	`Hello`
	`3`
		`1 Hello`
		`2 Hello`
		`3 Hello`
		`4 Hello`
		`5 Hello`
		`6 Hello`
		`7 Hello`

```
{                          Hello          8 Hello
  printf("Name of File:"); 4              9 Hello
  scanf("%s", s);          Hello          10 Hello
  f = fopen(s, "r");       5
  while (!feof(f))         Hello
   {                       6
     fscanf(f, "%s", s);   Hello
     printf("%s\n", s);    7
   }                       Hello
  fclose(f);               8
}                          Hello
                           9
                           Hello
                           10
                           Hello
```

O resultado provavelmente não é aquele que gostaríamos de ter obtido. Talvez devêssemos utilizar a função `fgets()` para ler as strings. Ver em baixo.

Outras funções para ficheiros

Outras funções para ficheiros bastante úteis são

`fgets()`

diminutivo de "**f**ile **get** **s**tring". Lê uma string do ficheiro.

definição da função: **`char *fgets(char s1, int n, FILE *stream)`**
lê uma string do ficheiro até eol (end of line), eof (end of file) ou o máximo de n-1 caracteres serem lidos

`fgetc()`

diminutivo de "**f**ile **get** **c**har". Lê um caracter do ficheiro.

definição da função: **`int fgetc(FILE *stream)`**
lê um único caracter do ficheiro. Retorna o valor do caracter se bem sucedida.

`fputs()`

diminutivo de "**f**ile **put** **s**tring". Escreve uma string para o ficheiro.

definição da função: **`int fputs(const char s1, FILE *stream)`**
escreve uma string para o ficheiro. Retorna 0 se bem sucedida.

`fputc()`

diminutivo de "**f**ile **pu** **c**har". Escreve um caracter para o ficheiro.

definição da função: **`int fputc(char *s1, FILE *stream)`**
escreve um único caracter para o ficheiro. Retorna o valor do caracter se bem sucedida.

Teste Rápido:
Para testar os conhecimentos sobre o que aprendeu nesta aula, prima aqui para fazer um teste on-line. De notar que este **Não** é o formato que será utilizado no teste final!

Peter Stallinga. Universidade do Algarve, 17 December 2002

Mini Teste 2: Computadores

---

1. O primeiro computador foi inventido por

- Bill Gates for Microsoft
- Blaise Pascal
- Charles Babbage
- IBM

---

2. O computador nos temos em casa é do tipo

- Supercomputer
- Mainframe
- Minicomputer
- Microcomputer
- Micro processor

---

3. Indique para cada peça de *hardware* a sua função

	input	*output*	armazemento	*processing*
Rato	○	○	○	○
Teclado	○	○	○	○
Memória	○	○	○	○
Monitor	○	○	○	○
Impressora	○	○	○	○
CPU	○	○	○	○

---

4. Para traduzir um programa em C usamos um

- Compilador
- Dicionário
- Sistema operativo
- Disco rígido

Mini Teste 3: Unidades de Informação / Memória

1. A unidade de informação mais pequena é

- um bit
- um byte
- um nibble
- um integer

2. A mais pequena unidade de informação ainda endereçavel é

- o bit
- o byte
- o nibble
- o integer

3. 1101 no sitema binário é (no sistema decimal) igual a

- 1101
- 15
- 13
- D

4. 2A no sistema hexadecimal é (no sistema decimal) igual a

- 2A
- 42
- 20
- 0

5. A maneira mais popular para representar texto é

- binário
- hexadecimal
- decimal
- ASCII

6. Quanto informação cabe aproximademente num standard disquete?

- 1 byte: um caracter ASCII
- 1 kilobyte (1 kB): um quarto duma página A4 em formato ASCII
- 1 megabyte (1 MB): um livro em formato ASCII
- 1 gigabyte (1 GB): uma pequena biblioteca em formato ASCII

Mini Test 4: Introdução a C

1. Quais dos seguintes *identifiers* são válidos

	válido	inválido	explicação
birthday8	○	○	
1shot	○	○	
hot?	○	○	
OLD_TIME	○	○	
down.to.earth	○	○	

2. Em C commentário escrevemos

- ○ depois "REM"
- ○ entre "{" e "}"
- ○ depois "//"
- ○ depois "comment"

Mini Teste 5: Variáveis

1. O uso de `\n` em `printf`

- é usado para output à impressora.
- significa o fim-do-texto (end-of-string).
- põe o cursor no início da linha a seguir.
- é usado para especificar o formato `int`.

2. Para armazenar números inteiros usamos variáveis do tipo

- `pointer`
- `int`, or `long int`
- `float` or `double`
- `boolean`

3. A gama de um int é

- 0 .. 255
- 0 .. 65535
- -32768 .. 32767
- -2147483648 .. 2147483647

4. Declarar uma variável significa

- Reservar espaço na memória e associar um nome ao aquelo espaço.
- Atribuir um nome e um valor
- Inicilizar uma variável
- Mostrar o seu valor no ecrãn

5. Variáveis (em C)

- são inicializado com 0 no começo do programa
- saõ declarado pelo seu primeiro uso
- devem ser atribuido um valor na altura de declaração
- têm valores inprevisíveis no começo do programa

6. Para calculos do tipo floating point com a mais alta precisão usamos variáveis do tipo

- boolean
- float
- double
- long double

Mini teste 6: Atribuição, Input e Cálculo

1. Quando queremos atribuir o valor 8.3 à variável r fazêmo-lo:

- `r = 8.3;`
- `r := 8.3;`
- `r == 8.3;.`
- `8.3 -> r;`

2. Depois da atribuição da questão 1, qual das linhas seguintes de C produzirá
`8.3000`

- `printf(8.3000);`
- `printf(6*r,4);`
- `printf(r:6:4);`
- `printf("%6.4f", r);`

3. O que está errado no programa a seguir?

```
main()
{
  float x;
  double c = 1.0;

  x*x = 2*c;
}
```

- Uma constante não pode mudar do valor
- O lado esquerde de `=` só pode conter uma só variável
- A variável `x` não está definida bem
- O lado direito de `=` não pode conter expresssões

4. Qual é a saída do próximo programa?

```
main()
{
  double x;
  double C = 1.0;

  x = C + 1.0;
  x = 2;
  x = x + 3.0;
  printf("%4.1f", x);
}
```

- `7.0`
- `5.0`
- `3.0`
- `1.0`

5. Qual é o resultado de "33 / 2"?

- 1.5
- 16
- 16.5
- 1

6. Qual é o resultado de "33 % 2"?

- 1.5
- 16
- 16.5
- 1

7. Quais são as prioridades de +, * e (..) ?

- primeiro +, depois *, depois (..)
- primeiro *, depois +, depois (..)
- primeiro (..), depois *, depois +
- primeiro (..), depois +, depois *

8. Qual é o resultado deste expressão?
`"1.0 + 2.0 * 3.0 - 6.0 / 2.0"?`

- 4.0
- 1.5
- 14
- 9.0

Quick Test 7: if ... else ...

1. Qual comparação é inválido em C:

- `(a = b)`
- `(a != b)`
- `(a == b)`
- `(a > b)`

2. O sintaxe para bifurcação simples é

-
```
if (condition) then
    instruction;
```
-
```
case condition
    instruction;
```
-
```
if (condition)
    instruction;
```
-
```
case condition of
    instruction;
```

3. Qual será o resultado deste programa?

```
main()
{
  int a, b, c, d;

  a = 5; b = 3; c = 99; d = 5;
  if (a>6) printf("A");
  if (a>b) printf("B");
  if (b==c)
    {
      printf("C");
      printf("D");
    }
  if (b!=c) printf("E"); else printf("F");
  if (a>=c) printf("G"); else printf("H");
  if (a<=d)
    {
      printf("I");
      printf("J");
    }
}
```

verifique não sei

4. Qual será o resultado deste programa?

```
main()
{
  int a=1;

  if (a>0)
    printf("The value of a ");
    printf("is larger than zero");
  else
    printf("The value of a ");
    printf("is less than zero");
}
```

- `The value of a is larger than zero`
- `The value of a is less than zero`
- A estrutura do programa está mal. Temos de juntar as instruções com {..}
- Depende do valor de `a`.

Mini Teste 8: Álgebra Booleana / `switch`

1. Qual será o output do programa?

```
main()
{
  double a, b;
  double c = 10.0;

  a = 9.0;  b = 2.0*c;
  if ((a>0) || (b>0))
    printf("Fixe!");
  else
    printf(" Uma pena");
}
```

- Fixe!
- Fixe! Uma pena
- Uma pena
- O programa não gera output.

2. O que está mal no programa a seguir

```
main()
{
  double a;
  double c = 2;

  a = 3.0;
  switch (a+1.0)
   {
    case 1: printf("Fixe!\n");
        break;
    case c:
         printf("Cool!\n");
         printf("Ingles");
        break;
    case 3: printf("Super!");
        break;
    default: printf("Language?\n");
   }
}
```

- a) `switch` não pode conter expressões (`a+1.0`)
- b) `switch` não funciona com expressões do tipo float (`a+1.0`)
- c) Na estrutura `switch` não podemos usar variáveis (`case c:`)
- d) Ambas b) e c)

3a. O que é o resultado do cálculo Booleano `(41 | 35)`?

Check | Help | More help | I give up

3b. O que é o resultado do cálculo Booleano `(41 & 6)`?

Check | Help | More help | I give up

3c. O que é o resultado do cálculo Booleano `(41 && 6)`?

4.

```
(3*4 + 12/6*i - j*2)
```

é um exemplo de

- uma expressão
- uma condição
- uma atribuição
- uma operação

Check Help More help I give up

Mini Teste 9/10: Ciclos

1. Em qual tipo de ciclos a instrução é executada no mínimo uma vez?

- for
- while
- do-while
- não existe

2. Quero escrever um programa que vai pedir ao utilizador um número e o programa deve mostrar todos os números primos até este número. Melhor seria usar um ciclo do tipo

- for
- while
- do-while
- Outra estrutura

3. Quais são as duas regras do *nesting* dos ciclos?

1:
2:

Ajuda Resposta correcta

4. A diferença entre os ciclos while e do-while é

- while é para ciclos com variáveis inteiros, do-while é para ciclos com variáveis do tipo floating point.
- do-while é para ciclos com variáveis inteiros, while é para ciclos com variáveis do tipo floating point.
- No ciclo do-while a condição é verificada no início, no ciclo while no fim.
- No ciclo while a condição é verificada no início, no ciclo do-while no fim.

5. O que está mal no código a seguir?

```
x = 0.0;
while (x<10.0)
  {
    y = x*x;
    z = x*y;
    printf("The square of %f is %f", x,
y);
    printf("The cube of %f is %f", x, z);
  }
```

- O ciclo nunca vai acabar
- Temos de usar um ciclo do tipo do-while.
- Temos de usar um ciclo do tipo for.
- A condição não pode conter variáveis do tipo floating point.

6. Queremos escrever um programa que vai pedir ao utilizador de escolher um tipo de cálculo ou sair do programa (1=addicionar, 2=subtrair, 0=sair). O programa deve continuar fazer isto até sempre (excepto, claremente, quando o utilizador escolhe 0). Neste caso, o melhor ciclo seria

- for
- while
- do-while
- Outra estrutura

Mini Teste 11,12: Programação Modular

1. Que palavra de C indica que a função não retorna nada?

Check Correct Answer

2. Que são as vantagens de usar módulos?

1:
2:

Correct Answer

3. Que será o resultado do programa seguinte?

```
#include<stdio.h>
void write_N(float r, int n)
{
  printf("%f" ,r);
}

void main()
{
  float x;
  x = 10.0;
}
```

- 10.0
- r
- x
- Este programa não gera output; esquecemos de chamar a função!

4. Porquê não precisamos de inicializar parâmetros?

- Um parâmetro não pode mudar o seu valor.
- A inicialização vem da chamada da função.
- Parâmetros são automaticamente inicializado com 0.
- Não é verdade! Parâmetros são como variáveis normais e têm de ser inicializado no começõ da função.

Mini Teste 14: Ponteiros

1. Como declarar um apontador para int?

- `int *a;`
- `int a*;`
- `int &a;`
- `int a&;`

2. Como atribuir o endereço da variável `x` a `p`?

- Depende do tipo de `x`.
- `p = *x;`
- `p = &x;`
- `p = ^x;`

3. Assume `b` é uma variável do tipo "apontador para int";
Como pôr o valor de 0 em endereço `b`?

Verifique Não sei

4. O que acontece se esquecemos de inicializar um apontador?

- O resultado será 0.
- O compilador vai nos avisar.
- O programa vai crashar.
- Um apontador é inicializado automaticamente.

5. O que está mal no código a seguir?

```
double *p;
int i;
p = &i;
*p = 10.0;
```

- Trocámos `*` com `&`.
- `*p = 10.0;` vai sobreescrever outras variáveis ou código.
- `p = &i;` vai gerar um erro.
- Nada! Tudo está bom.

6. Que são as vantagens de ponteiros?

1:
2:

Mostra

Mini Teste 15: Âmbito das variáveis, passagem por valor e passagem por referência

1. Qual o âmbito de cada objecto no programa a seguir?

```
float a;

void proc1(float b)
{
  float c;
  int d = 10;

  c = b+ (float) d;
  printf("%f", c);
}

float proc2(float *e)
{
  float f = 20.0;

  return(*e+f);
}

float g;

void main()
{
  float h;
```

2. Considere o programa abaixo

```
int x;

void show(int *a)
{
  printf("%d ", *a);
  *a = *a + 1;
}

void main()
{
  x = 0;
  printf("%d ", x);
  show(&x);
  printf("%d ", x);
}
```

O procedimento usa a técnica de

- ○ Passagem por valor
- ○ Passagem por referência

e, por isso o *output* será

Correct Answer

```
  a = 10.0;
  proc1(a);
  proc2(&a);
}
```

	local	global	parâmetro	nenhum
a	○	○	○	○
b	○	○	○	○
c	○	○	○	○
d	○	○	○	○
e	○	○	○	○
f	○	○	○	○
g	○	○	○	○
h	○	○	○	○

3. Qual será o output do programa a seguir?

```
/* double names */
int x;

void show()
{
  int x;

  x = 1;
  x = x*x;
  printf("%d ", x);
}

void main()
{
  x = 0;
  show;
  printf("%d ", x);
}
```

- ○ Não é permitido usar o mesmo nome para uma variável duas vezes!
- ○ 0 0
- ○ 1 0
- ○ 0 1
- ○ 1 1

Mini Teste 16: Programação recursiva

Considera o programa a seguir:

```
#include <stdio.h>
float a;

float XfuncN(float x, int n)
{
  float c;

  c = 0.0;
  if (n==0)
    return(1.0)
  else
    return(x*XfuncN(x, n-1));
}

void main()
{
  printf("%0.1f", XfuncN(3.0, 3));
}
```

1. Qual será o *output* do programa?

Verifica Não sei

2. Quantas cópias da variável local c existem no máximo?

Verifica Não sei

Mini teste 16 extra: Programação Recursiva

Considere o programa a seguir

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

int func(char *p)
{
  int c;

  c = 0;
  if (*p == '\0')
    return(0);
  else
   {
     p++;
     return(1 + func(p));
   }
}

void main()
{
  char a[20];

  strcpy(a, "Ola");
  printf("%d", func(a));
}
```

1. Qual será o output do programa?

Check Correct Answer

2. Quantas cópias da variávl `c` existem no máximo?

Check Correct Answer

Mini Teste 18: Arrays e structs

1. Qual a diferença entre um array e um struct?

- Um array é só para armazenar coisas contáveis, com struct é possível armazenar tudo.
- Um struct é só para armazenar coisas contáveis, com arrays é possível armazenar tudo.
- Um struct é só para armazenar coisas contáveis, com arrays é possível armazenar tudo.
- Struct são para combinar variáveis de tipos diferentes, arrays são para armazenar variáveis do mesmo tipo.

```
float maximum(float a, b)
{
  float max;

  if (a>b)
    max = a;
  else
    max = b;
 .....
}
```

2. Agora, como deixar a função retornar o valor de max à instrução que chamou este função?

- Nada é automaticamente.
- `return (max);`
- `maximum = max;`
- Esta função não gera output e por isso não vai retornar nada!

```
struct {
  struct {
    float z[10];
    int i[3];
  } x;
  struct {
    float r;
    double p;
  } y;
} a[10];
```

3. Como atribuir um valor de 0 ao (primeiro) `i` do array?

Verifica Não sei

4. Queremos construir um base de dados para armazenar a informação de 1000 alunos. Melhor seria fazer isto com uma variável

-
```
struct {
          int number;
          char name[20];
          int year;
   } a;
```
-
```
struct {
          int number;
          char name[20];
          int year;
   } a[1000];
```
-
```
struct {
       int number[1000];
       char name[1000][20];
       int year[1000];
  } a;
```
-
```
struct {
       int number[1000];
       char name[1000][20];
       int year[1000];
  } a[1000];
```

Mini Teste 19: typedef

1. O que é que o `typedef` faz?

- Escreve texto no ecrã.
- Define um novo tipo da variável.
- Faz combinações de arrays e records.
- Declara variáveis dos tipos mistos.

```
typedef float b;
 ......
b = 3.1;
```

2. Porquê o código acima não funciona?

- Temos de usar '`typedef b float`' em vez.
- `float` já está definido.
- A sintaxe está mal, em vez temos de usar '`type`'.
- `typedef` só faz uma especificação de um tipo de variável para declarar depois.

```
typedef struct {
   struct {
      float x[10];
      int y[3];
    } ri;
    struct {
      float v;
      double w;
    } rd;
  end;
} mystructs[10];

mystructs b;
```

3. Como atribuir um valor de 0 ao ('primeiro') `y` do programa?

Check Correct Answer

4. Qual será o output do seguinte código?

```
typedef float reals[10];

void WriteIt(reals r)
{
  printf("%f\n", r[1]);
}

void main()
{
  int x[10];

  x[1] = 3;
  WriteIt(x);
}
```

- Inprevisível. Esquecemos de inicilizar o array `r`!
- `3.0`
- Nada; fizemos uma mistura de tipos em chamar a função.
- `3`

Programação I

Aula Prática 1 (parte A)

Sumário

Noções sobre o Linux

O Linux é um sistema operativo. Um sistema operativo é um conjunto de programas que gerem os recursos do computador, permitindo-nos trabalhar com ele.

O Linux é um sistema operativo multi-utilizador e multi-tarefa (significa que várias pessoas podem usar o mesmo computador ao mesmo tempo e correndo programas diferentes).

O Linux tem um mecanismo de segurança que impede os utilizadores normais de danificarem ficheiros que são essenciais para o bom funcionamento do sistema. Portanto, não tenham medo de experimentar. O computador "não morde", e o pior que pode acontecer é perderem os vossos ficheiros pessoais.

Cada utilizador tem um nome (*login name*) e uma password, que o identifica no sistema. Tem também uma área de trabalho só dele, a que se chama *home*, aqui encontram-se todos os ficheiros que lhe pertencem.

O Linux é sensível às maiúsculas e minúsculas.

Comandos Básicos do Linux

Grande parte dos comandos são abreviaturas de palavras inglesas.

Comandos de uso geral

login - identifica os utilizadores que entram no sistema.

```
Login:  nome do utilizador
Password: ***********
```

exit - para terminar a sessão de trabalho.

```
$ exit
```

yppasswd - para alterar a password.

```
$ yppasswd
Password Actual:
Password Nova:
Password Nova (confirmação):
```

Nota: É importante não esquecer a password.

man - para pedir ajuda sobre um comando (**man** de **man**ual).

```
$ man nome_comando
```

Para sair da ajuda, tecla q (quit).

Comandos sobre directorias

As directorias servem para agrupar os ficheiros por temas, permitindo uma melhor organização da informação.

pwd - para indicar a directoria corrente (**pwd** de **p**rint **w**orking **d**irectory).

```
$ pwd
```

ls - para ver o conteúdo das directorias (**ls** de **ls**t).

```
$ ls nome_directoria    - mostra os ficheiros e outras directorias
                          no interior da directoria indicada.

$ ls -F nome_directoria - coloca / à frente das directorias,
                          e * à frente dos ficheiros executáveis.

$ ls -l nome_directoria - mostra mais informação sobre cada
                          ficheiro (permissões, links, dono, grupo,
                          tamanho, data, hora, nome do ficheiro).
```

cd - para mover entre directorias (**cd** de **c**hange **d**irectory).

```
$ cd /                  - vai para a raíz.

$ cd ..                 - sobe um nível, vai para a directoria a cima.

$ cd .                  - a própria directoria.

$ cd nome_directoria    - vai para a directoria indicada.

$ cd                    - vai para a home.
```

mkdir - para criar novas directorias (**mkdir** de **m**a**k**e **dir**ectory).

```
$ mkdir nome_directoria
```

rmdir - para apagar directorias (**rmdir** de **r**e**m**ove **dir**ectory).

```
$ rmdir nome_directoria         - apaga apenas se a directoria
                                 estiver vazia.
```

nome_directoria: Caminho desde a raíz (/) até á directoria desejada. Para cada novo nível na árvore coloca-se uma nova /.

Comandos sobre ficheiros

cp - para copiar ficheiros (**cp** de **c**o**p**y).

```
$ cp nome_directoria_origem/nome_ficheiros nome_directoria_destino
```

mv - para mover ficheiros (**mv** de **m**o**v**e).

```
$ mv nome_dirertoria_origem/nome_ficheiros nome_direrctoria_destino/nome_ficheiros
```

Cuidado porque não pede confirmação e pode-se escrever por cima de dados importantes.

rm - para apagar ficheiros (**rm** de **rem**ove).

```
$ rm nome_directoria/nome_ficheiros     - apaga os ficheiros da directoria indicada.

$ rm nome_directoria -r                 - apaga a directoria mesmo sem estar vazia.
```

more - para ver o conteúdo de ficheiros.

```
$ more nome_directoria/nome_ficheiros
```

Mostra no écran o contéudo do(s) ficheiro(s).

```
Comandos essenciais do more:
        Space - ir para a próxima página
        b - ir para a página anterior
        q - quit (sair)
```

Metacaracteres (* e ?)

Os metacaracteres podem ser utilizados com qualquer um dos comandos de ficheiros, e facilitam bastante quando se pretende fazer a mesma operação sobre ficheiros que têm algo em comum no seu nome.

O * representa uma cadeia de caracteres.

O ? representa um caracter.

Permissões de Ficheiros

O Linux tem um mecanismo que nos permite proteger os nossos ficheiros dos outros utilizadores. A esse mecanismo chama-se Permissões de ficheiros.

```
Existem 3 tipos de Permissões:
        r - Leitura (read), pode ler o ficheiro;
        w - Escrita (write), pode escrever no ficheiro;
        x - Execução (execute), pode executar o ficheiro;


Podem ser aplicadas a ficheiros e directorias.


Existem 3 classes de utilizadores, cada classe com o seu tipo de permissões:
        O dono do ficheiro,
        O grupo a que o dono pertence,
        Os outros utilizadores.
```

chmod - para mudar as permissões, só o dono o pode fazer.

```
        $ chmod { a, u, g, o } { +, - } { r, w, x } nome_ficheiros


Significado:

        { a, u, g, o }  - para definir que classes de utilizadores vai afectar.
                                a - all; u - user (dono); g - group; o - other;

        { + , - }       - para definir se atribui (+) ou retira (-) a permissão.

        { r, w, x }     - para definir os tipos de permissões a mudar.
                                r - leitura; w - escrita; x - execução;
```

Para ver as permissões existentes:

```
        $ ls -l
```

```
Surge:
        -  rwx  r--  r--  user  group  tamanho  data  nome_ficheiro

Significado:

        (-)       - tipo do ficheiro

        (rwx)     - permissão do dono (user)
                    Neste caso, pode ler, escrever e executar o ficheiro.

        (r--)     - permissão do grupo
                    Neste caso, os utilizadores do grupo do dono só podem ler.

        (r--)     - permissão dos outros utilizadores
                    Neste caso, igual ao do grupo

        (user)    - quem é o dono

        (group)   - a que grupo pertence

        (tamanho) - tamanho do ficheiro

        (data)    - data do ficheiro
```

Nota: As permissões de um ficheiro dependem também das permissões da directoria em que se encontra.

O Ambiente de Trabalho

Para criar um programa numa linguagem de programação qualquer, existem quatro passos que têm sempre que acontecer:

1. Pensar no problema, de preferência com papel e lápis.
2. Escrever o programa, para isso é necessário um editor de texto.
3. Verificar se o código que se escreveu não tem erros, para isso é necessário um compilador da linguagem de programação usada.
4. Quando o programa estiver livre de erros, criar o ficheiro executável e correr o programa.

Em Linux, como se podem realizar os três últimos passos?

2 - Editar ficheiros

Existem muitos editores de texto em Linux. O vi, o joe, o emacs, o gedit, são apenas alguns deles.

Podem usar qualquer um deles, e até podem gravar os ficheiros numa disquete e continuar a trabalhar com eles em casa.

3 e 4 - Compilar e executar programas

Para compilar/executar um programa devem chamar o compilador gcc na linha de comandos:

```
        Compilar e tornar executável:

                $ gcc -o nome_ficheiro_executável nome_ficheiro.c

        Correr o programa:

                $ nome_ficheiro_executável
```

Programação I

Aula Prática 1 (parte B)

Sumário

- Exemplos práticos dos comandos básicos do Linux

Exemplos práticos dos comandos básicos do linux

1. Comandos de uso geral

Comando **man**

```
$ man yppasswd
```

2. Comandos sobre directorias

Comando **pwd**

```
$ pwd
```

Comando **ls**

```
$ ls
$ ls /etc
$ ls -F /etc
$ ls -l /etc/X11
```

Comando **cd**

```
$ cd /home
$ ls            (para ver os utilizadores existentes)
$ cd ../etc
$ cd            (para voltar para a home, independentemente de onde se encontre)
$ pwd
```

Comando **mkdir**

```
$ mkdir src
$ mkdir temp
$ ls
$ cd src
$ ls
```

```
$ mkdir prog1
$ ls
```

Comando **rmdir**

```
$ cd
$ rmdir temp
$ rmdir src
$ ls
```

Comando **cp**

```
$ cp /etc/fstab .
$ ls
$ mkdir temp
$ cp /etc/X11/XF86Config temp
$ ls temp
```

Comando **mv**

```
$ mv fstab temp
$ cd temp
$ ls
$ ls ~
```

Comando **rm**

```
$ rm fstab
$ ls
$ cd ..
$ rm temp -r
$ ls
```

Comando **more**

```
$ more /etc/fstab
```

3. Metacaracteres

```
$ ls /etc/*.conf
$ ls /etc/sys*.conf
$ ls /etc/*
$ ls ~/src/prog?.c
```

4. Permissões de ficheiros

Comando **chmod**

```
$ ls -l /home/apcosta

$ cp /home/apcosta/teste.c .
```

```
Fazer na nossa área:    $ chmod a+r teste.c

Repetir:

$ ls -l /home/apcosta

$ cp /home/apcosta/teste.c .
```

5. Ambiente de Trabalho

1. Abrir o editor de texto.
2. Escrever um texto livre.
3. Guardar num ficheiro.
4. Abrir um novo ficheiro.
5. Escrever o programa Hello World.

```
#include <stdio.h>

main()
{
   printf("Hello World\n");
}
```

6. Guardar num ficheiro hello.c.
7. Compilar e correr via linha de comandos.

Programação I

Aula Prática 1 (parte C)

Sumário

A Internet

A Internet é uma rede mundial de computadores que estão espalhados pelo mundo inteiro. Cada computador ligado à Internet utiliza software próprio para poder disponibilizar e/ou aceder a informação. A Internet é um meio através do qual se pode aceder a informação disponível em documentos ou ficheiros que estão contidos noutros computadores. A Internet pode ser comparada com uma infra-estrutura para suportar uma espécie de biblioteca gigantesca.

Os computadores ligados à Internet podem aceder aos seguintes serviços:

- Correio electrónico (e-mail).

  Permite receber e enviar mensagens para outras pessoas em qualquer ponto do mundo.
- Telnet ou login remoto.

  Permite que utilizes o teu computador para te ligares a um computador remoto (possivelmente noutro país) e usá-lo como se estivesses lá.
- FTP (File Transfer Protocol).

  Permite que o teu computador possa transferir ficheiros de/para outros computadores.
- World Wide Web (WWW ou "Web").

  Quando se ligam à Internet utilizando o Netscape ou outro browser qualquer (ex: Windows Explorer, Mozilla), estam a ver documentos na WWW. A WWW está baseada na noção de Hipertexto. Trata-se de um sistema que tem links, uma espécie de apontadores para outros documentos na Web. Cada documento é identificado por um URL (Uniform Resource Locator) que não é mais do que um endereço único para um documento da Web.

  Exemplos de URLs:

  - http://www.cnn.com é o endereço do canal Americano CNN.
  - http://www.publico.pt é o endereço do jornal Publico.
  - http://www.ualg.pt é o endereço da Universidade do Algarve.
  - http://w3.ualg.pt/~flobo/p1 é o endereço da cadeira de Programação I.

  Sempre que quiserem encontrar coisas na Internet, aconselho a utilizarem o Google, cujo URL é http://www.google.com. O Google permite que escrevam palavras e devolve um conjunto de URLs com informação relacionada com essas palavras. Experimentem.

  Os documentos na Web estão feitos utilizando uma linguagem chamada HTML (HyperText Markup Language). O HTML tem um conjunto de comandos (tags) que permitem formatar texto, incluir imagens, e especificar links para outros documentos. Se quiserem ver o HTML que gerou o

documento que estam a ver neste preciso momento, é só ir à opção View e dentro dessa opção escolher Page Source.

Programação 1

Aula prática 2

Sumário

- O compilador **gcc**
- Um exemplo passo a passo.
- Programas introdutórios em linguagem C.

O compilador

Nas nossas aulas vamos utilizar o compilador de linha de comando **gcc**. O **gcc** da GNU é um dos compiladores de C mais avançados e mais versáteis que existe no mercado e, ainda por cima, é software de domínio livre. O que significa ser software de domínio livre? Significa que temos liberdade para:

1. Executar o software, qualquer que seja o nosso propósito
2. Estudar o modo como o software funciona e adaptá-lo às nossas necessidades
3. Distribuir cópias do software
4. Melhorar o software e distribuir esses melhoramentos para benefício da comunidade

O **gcc** suporta os standards modernos da linguagem C (como p/ex. o ANSI C) ao mesmo tempo que mantém a compatibilidade com os compiladores e estilos mais antigos. É também um compilador de C++, uma linguagem de programação orientada a objectos que poderás aprender mais tarde noutra cadeira do curso.

Podes ver aqui mais informações sobre este compilador.

Um exemplo passo a passo

Vamos então seguir, em quatro passos, um pequeno exemplo para criação do nosso primeiro programa.

1. Definir um espaço para o programa
2. Criar o programa
3. Compilar o programa
4. Executar o programa

Definir um espaço para o programa

É mais fácil mantermos o nosso espaço em disco organizado se criarmos uma directoria para cada programa no qual estejamos a trabalhar (excepção feita para programas ou projectos cujo código fonte esteja dividido em vários ficheiros). Neste caso vamos criar uma directoria chamada **prog1** para guardar o nosso primeiro programa.

```
mkdir prog1
cd prog1
```

Criar o programa

Um programa começa por ser um ficheiro de texto. Utiliza um editor de texto do teu agrado (p/ex. o **emacs**, o **gedit**, o **vim** ou o **joe**) para escrever esse texto a que é comum chamar-se código fonte. Consoante a escolha podes dar um dos seguintes comandos:

emacs prog1.c ou **gedit prog1.c** ou **vim prog1.c** ou **joe prog1.c**

e escrever o programa que se segue:

```
#include <stdio.h>

main()
{
   printf("Este é o meu primeiro programa\n");
}
```

Compilar o programa

O compilador pega no código fonte e converte-o num programa executável. Para compilares o teu código fonte usando o compilador gcc executa o comando:

gcc -o prog1 prog1.c

O parâmetro **-o** diz ao compilador que o ficheiro executável deverá chamar-se prog1, enquanto o prog1.c no final do comando indica em que ficheiro se encontra o código fonte.

Executar o programa

Para correr o programa faz

prog1 ou então **./prog1**

e a mensagem "Este é o meu primeiro programa" vai aparecer no ecrã.

Programas introdutórios em linguagem C

Agora podes fazer os programas que se seguem.

Programação 1

Aula prática 3

---

1. `printf`

Qual será o output do programa a seguir?

```
#include <stdio.h>

main()
{
   printf("%d %c %f\n", 65, 65, 65);
}
```

Resposta: .

Agora verifique.

---

2. Tamanho das variáveis

Em C existe uma instrução (`sizeof`) que vai-nos dizer o espaço que uma variável ocupa em memória. Hoje vamos usar esta instrução para determinar o tamanho de todas as variáveis que nós conhecemos (veja aula 5).
A instrução `sizeof` é um exemplo duma função. Ainde não sabemos o que é uma função, mas vamos dar aqui o uso geral de SizeOf:

```
sizeof(nome)
```

retorna o tamanho da variável `nome`. Para mostrar o resultado, vamos usar `printf` (aula 5).

```
printf("%d\n",sizeof(nome));
```

Escreve um programa que

- declare variáveis de todos os tipos de variáveis nós conhecemos.
- mostre o tamanho de cada variável no ecrãn, por exemplo:

```
int: 2 bytes
float: 4 bytes
```

---

3. Inicialização

Muda o programa do exercício 2 acima de forma que o programa mostre o valor da cada variável, em vez de mostrar o seu tamanho. As variáveis têm todas valor igual a 0?
Agora faz uma inicialização de cada variável e experimenta outra vez.
Nunca se deve assumir que o valor duma variável é 0.

---

4. Cálculos

Qual será o output do programa a seguir?

```
#include <stdio.h>

main()
{
   unsigned char i, j;
   i = 200;
   j = 2*i;
   printf("2*%d=%d\n", i, j);
}
```

Resposta: .

Agora verifica. Elimina o erro.

---

5. `if` 1:

Faz um programa que calcule o máximo de 2 números reias. Os 2 números devem ser introduzidos pelo utilizador.

Ao ser executado, o programa vai fazer o seguinte (o que aparece em **bold** é escrito pelo utilizador):

```
Introduza um número:
3
 Introduza outro número:
8
 O máximo é 8.
```

---

6. `if` 2: nota final

Imagina que a uma dada cadeira, a nota de frequência é obtida da seguinte forma: o trabalho tem peso 25% e o teste tem peso 75%. Faz um programa que calcula a vossa nota de frequência a essa cadeira. Se a nota de frequência for inferior a 9.5, o computador deve imprimir a mensagem: "tem de ir a exame". Caso contrário, imprime a mensagem: "Passaste com x valores".

Ao ser executado, o programa vai fazer o seguinte:

```
Nota do trabalho:
18
 Nota do teste:
13
 Passaste com 14.25 valores.
```

---

5. `if` 3: estruturas mais complicadas.

Faz um programa que pede ao utilizador 3 números e escrevo-os no ecrã por ordem crescente:

Ao ser executado, o programa vai fazer o seguinte:

```
Diz um número:
18
 Diz outro número:
-4
 Diz outro número:
5
 Ordem crescente: -4 5 18
```

---

6. `if` 4:

Faz um programa que calcula o valor absoluto (módulo) de um número.

Ao ser executado, o programa vai fazer o seguinte:

```
Diz um número:
-1
 O valor absoluto de -1 é 1.
```

Faz um programa que, ao receber o valor da largura e do comprimento de uma figura geométrica, detecta se esta é um quadrado ou um rectângulo.

Ao ser executado, o programa vai fazer o seguinte:

```
Diz a largura:
7
 Diz o comprimento:
5
 A figura é um rectângulo.
```

Faz um programa em C que calcule o salário de um empregado baseado no n.º de horas que trabalhou, e no seu salário por hora (as horas extraordinarias (> 40) são pagas a dobrar).

RESPOSTAS

PROGRAMAÇÃO IMPERATIVA TP3

2.

```
#include <stdio.h>

void main()
{
  char caracter;
  int inteiro;
  float flutuante;
  double duplo;
  long int inteiro_comprido;
  char array1[10];

  printf("\n Tamanho dum caracter,: %i", sizeof(caracter));
  printf("\n Tamanho dum caracter,: %i", sizeof(char));
  printf("\n Tamanho dum inteiro,: %i", sizeof(inteiro));
  printf("\n Tamanho dum inteiro,: %i", sizeof(int));
  printf("\n Tamanho dum real flutuante ,: %i", sizeof(flutuante));
  printf("\n Tamanho dum real flutuante ,: %i", sizeof(float));
  printf("\n Tamanho dum duplo,: %i", sizeof(duplo));
  printf("\n Tam. dum inteiro comprido,%i", sizeof(inteiro_comprido));
  printf("\n Tam. de arranjo com 1o pos.,: %i", sizeof(array1));
}
```

3.

```
#include <stdio.h>

void main()
{
  char caracter = 'z';
  int inteiro = 100;
  float flutuante = 12.345;
  double duplo = 345678.32345;
  long int inteiro_comprido = 12345678;
  char array1[10] = "quebonito";

  printf("\n O valor da variável caracter,: %c",caracter);
  printf("\n O valor da variável inteiro,: %i",inteiro);
  printf("\n O valor da variável flutuante,: %f",flutuante);
  printf("\n O valor da variável duplo,: %f",duplo);
  printf("\n O valor da var. inteiro_comprido,%li",inteiro_comprido);
  printf("\n O conteúdo da variável array1,: %s\n", array1);
}
```

4.

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

```
void main()
{
  float i, j;/*correção */

  i = 16385.0;
  j = 2.0 * i;
  printf("\n2 * %.0f = %.0f\n", i, j);/*correção */
}
```

5.

```
#include <stdio.h>

void main()
{
  float n1, n2;

  printf("\nEste programa calcula o maior de dois números reais \n");
  printf("\nFaz favor de entrar o primeiro número: ");
  scanf("%f", &n1);
  printf("\nFaz favor de entrar o segundo número: ");
  scanf("%f", &n2);

  /*-------agora determinamos qual número entrado , maior ----------*/
  if (n1 > n2)
    printf("\n\nO %.2f , maior do que o: %.2f", n1, n2);
  else
   {
    if (n1 < n2)
      printf("\n\nO %.2f , maior do que o: %.2f", n2, n1);
    else
      printf("\n\nO número %.2f , igual ao número %.2f", n1, n2);
   }
}
```

6.

```
#include <stdio.h>

void main()
{
  float teste, trabalho, frequencia;
  float peso_teste = 0.75, peso_trab = 0.25;

  teste = 0.0; trabalho = 0.0;

  printf("\nEste programa calcula a nota de frequência numa cadeira \n");
  printf("\nA nota do teste tem peso 75% e do trabalho , 25%\n");
  printf("\nFaz favor de entrar a nota do teste: ");
  scanf("%f", &teste);
  printf("\nFaz favor de entrar a nota do trabalho: ");
  scanf("%f", &trabalho);

  /*-------agora determinamos a nota de frequência: */

  frequencia = (teste * peso_teste) + (trabalho * peso_trab);
```

```
  /*agora testamos se a nota obtida de frequência dá para passar: */

  if (frequencia >= 9.5)
    printf("\n\ Aprovaste com %.2f valores", frequencia);
  else
    printf("\n\n A sua nota ,: %.2f; tem que ir a exame", frequencia);
}
```

o que teria acontecido se no primeiro 'if' escrevíamos:
"if (frequencia > 9.5)"?
(resp.: alunos bater a porta do prof.)

7.

```
#include <stdio.h>

void main()
{
  int n1, n2, n3;

  printf("\nEste programa recebe três inteiros DISTINTOS e vai escrever-lhes na\
   ordem crescente. \n");
  printf("\nFaz favor de entrar o primeiro número: "); scanf("%d", &n1);
  printf("\nFaz favor de entrar o segundo número:"); scanf("%d", &n2);
  printf("\nFaz favor de entrar o terceiro número: "); scanf("%d", &n3);

  /*-------agora temos que comparar para escrever correctamente:
    Quais são todas as hipóteses possíveis?.
    Calcular primeiro aquilo com papel e lápis.
    Como quase sempre, á  muitas maneiras de escrever um programa.
    Esta que vai a seguir , é só uma delas.
    Aqui não vamos utilizar a conjunção AND (e) que em C escreve-se '&&'
    o número de ordem serve para saber qual opção utiliza o pgm. */

  if (n1 < n2)
   {
    if (n2 < n3)
      printf("\n\ ordem crescente(1): %d%d%d",n1,n2,n3);
    else
      if (n1 < n3)
        printf("\n\ ordem crescente(2): %d%d%d",n1,n3,n2);
      else
        printf("\n\ ordem crescente(3): %d%d%d",n3,n1,n2);
   }
  else
   {
    if (n3 > n1)
      printf("\n\ ordem crescente(4): %d%d%d",n2,n1,n3);
    else
      if (n3 < n2)
        printf("\n\ ordem crescente(5): %d%d%d",n3,n2,n1);
      else
        if (n3 < n1)
          if (n3 > n2)
            printf("\n\ ordem crescente(6): %d%d%d", n2,n3,n1);
   }
```

```
}
```

8.

```
#include <stdio.h>

void main()
{
  int n;

  printf("\n Esta rotina calcula o valor absoluto de x -> |x|. \n");
  printf("\n Faz favor de entrar o número: "); scanf("%d", &n);

  // Como se calcula o valor absoluto dum número?

  if (n < 0)
    printf("\n |%d| = %d", n, (n * -1));
  else
    printf("\n |%d| = %d", n, n);
}
```

9.

```
#include <stdio.h>

void main()
{
  float largura, comprimento;

  clrscr();
  printf("\n Este programa, dados largura e comprimento numa figura geométrica\
         , determina se a dita figura é um quadrado ou um rectângulo. \n");
  printf("\n Indicar a largura: "); scanf("%f", &largura);
  printf("\n Indicar o comprimento : "); scanf("%f", &comprimento);

  if (comprimento == largura)
    printf("\n A sua figura é um quadrado de lado = %.2f", comprimento);
  else
    printf("\n A sua figura é um rectângulo de %.2f x %.2f",
           largura, comprimento);
}
```

10.

```
#include <stdio.h>

void main()
{
  float horas, horas_extra, salario, total;

  clrscr();
  printf("\nEste programa calcula o salário dum operário, baseado no número\
          de horas trabalhadas por semana e o valor horário do salário. As horas\
          extraordinárias (> 40 hrs/semana) são pagas no dobro do salário normal.\
```

```
            Todos estes valores são em Euros (€). \n");

  printf("\n Indicar o salário normal por hora: ");
  scanf("%f", &salario);
  printf("\n Indicar o número de horas trabalhadas na semana: ");
  scanf("%f", &horas);

  /* fazer sempre primeiro com papel e lápis o algoritmo do problema:
     1) Determinar se existirem horas extraordinárias
     2) Se houver, as horas normais ficam reduzidas a 40. As horas extra a diferença.
     3) Calcular o salário total = salário das 40 horas + salário das extra.*/

  if (horas > 40)
   {
    horas_extra = horas - 40;
    horas = 40;
   }
  else
    horas_extra = 0.0;

  /*agora calculamos o total ganho durante uma semana */

  total = (horas * salario) + (horas_extra * (salario * 2));

  printf("\n Salário 40 horas  : %8.2f euros",(horas * salario));
  printf("\n Salário Horas Extra: %8.2f euros",(horas_extra * salario) * 2);
  printf("\n-----------");
  printf("\n total%8.2f euros\n", total);
}
```

Programação 1

Aula prática 3

---

1. A função printf(..)

Qual será o output (saída) do programa a seguir?

```
#include <stdio.h>

main()
{  int i = 67;
   printf("%d %c %f\n", i, i, i);
}
```

Resposta: .

Verifique.

---

2. Variáveis e a sua capacidade de armazenamento

Em C, igual que noutras linguagens de programação, cada variável está desenhada para guardar tipos de valores específicos.

2.1) Escreve um programa que
declara as seguintes variáveis cujos nomes devem ser:
i = inteiro, c = caracter, f = ponto fluctuante, uc = caractere sem sinal, d = double.

Utilize os modificadores de tipo "long" e "unsigned" para responder as seguintes perguntas:

a) Numa variável de tipo inteiro , pode armazenar o número 32701? e o número 32823? Imprima o que resulta declarando uma variavelde tipo inteiro para armazenar cada valor.
b) Qué resultado obteve? Porqué tem um valor negativo quando imprimiu o segundo número? Intente este problema resolver olhando para os apontamentos do capítulo 3.
c) Como devería modificar o tipo da variável inteira para poder armazenar este último número?

2.2)
Crie duas variáveis numéricas (n1 e n2) dum tipo apropriado e armazene nelas os seguintes valores: 4.5678 e 3.21
Declare igualmente a variável n3 do mesmo tipo para armazenar o resultado da seguinte operação:
Execute a divisão onde n1 seja o numerador e n2 seja o denominador. Guardaro valor numa variável n3.
Olho com o tipo que escoger para ela!
Que resultado obtem se imprimir?
Que acontecia se n3 fose declarada como de tipo inteiro? Imprima e compare. Corrigir se necessário.

---

3. Cálculos

Qual será o output do programa a seguir?

```
#include <stdio.h>

main()
{
   unsigned int a, b;
   a = 2500;
   b = a * a;
   printf("%d*%d=%d\n", a, a, b);
}
```

Resposta: .

Agora verifica. Elimina o erro.

---

4. Utilizando a instrução `if` : nota final

Imagina que a uma dada cadeira, a nota de frequência é obtida da seguinte forma: um trabalho com peso 30% e um teste com peso 70%. Fazer um programa que calcula a vossa nota de frequência dessa cadeira. Se a nota de frequência for inferior a 9.5, o computador deve imprimir a mensagem: "tem de ir a exame". Caso contrário, imprime a mensagem: "Passaste com x valores".

Ao ser executado, o programa vai fazer o seguinte:

```
Nota do trabalho:
15
 Nota do teste:
10.5
Logo imprimirá "Passaste com 11.9 valores"
```

(Repare que o resultado debe ser mostrado com apenas um valor decimal.)

---

5. `if` 2: estruturas mais complicadas.

5a) Faz um programa que pede ao utilizador 3 números e escrevo-os no ecrã na ordem **decrescente:**

Ao ser executado, o programa vai fazer o seguinte:

```
Diz um número:
18
 Diz outro número:
-4
 Diz outro número:
5
 Ordem decrescente: 18  5  -4
```

---

6. Mais cálculos

Fazer um programa que calcule a **pendente** entre dois pontos ($x1$, $y1$) e ($x2$, $y2$) no espaço, conforme a conhecida fórmula que diz que o resultado corresponde a diferença entre as coordenadas $y2$ e $y1$ dividido pela diferença entre as coordenadas $x2$ e $x1$. Chamaremos m a pendente obtida. Imprima o valor da pendente calculada.

(Em geral a pendente representa uma quantidade que da a inclinação duma curva o linha com respecto a uma outra curva o linha.)
Dica: Calcule a pendente de dois pontos conhecidos por você. Introduzindo estes valores o programa pode ser testado como correcto quando produzir os mesmos resultados seus feitos a mão. Isto sempre ajuda.

Programação 1

Aula prática 3. Respostas

---

1. A função `printf(..)`

Qual será o output (saída) do programa a seguir?

```
#include <stdio.h>

main()
{  int i = 67;
   printf("%d %c %f\n", i, i, i);
}
```

Resposta: 67 C 0.000000 .

---

2. Variáveis e a sua capacidade de armazenamento

a) sim = 32701? , não = 32823? Imprima o que resulta declarando uma variavelde tipo inteiro cada valor:
b) Qué resultado obteve? **3270** e, **-32713** .Inteiros só podem armazenar até o número: **32767**
c) Como devería modificar o tipo da variável inteira para poder armazenar este último número?: long int

2.2)Que resultado obtem se imprimir? **1.42**
Que acontecia se n3 fose declarada como de tipo inteiro? Resultado = **1.**

---

3. Cálculos

Qual será o output do programa a seguir?

```
#include <stdio.h>

main()
{
   unsigned int a, b;
   a = 2500;
   b = a * a;
   printf("%d*%d=%d\n", a, a, b);
}
```

Resposta: 24080 .

Agora verifica. Elimina o erro. : redeclarar as variáveis como long int; Resultado correcto a imprimir: 6.250.000

---

4. Utilizando a instrução `if` : nota final

Imagina que a uma dada cadeira, a nota de frequência é obtida da seguinte forma: um trabalho com peso 30% e um teste com peso 70%. Fazer um programa que calcula a vossa nota de frequência dessa cadeira. Se a nota de frequência for inferior a 9.5, o computador deve imprimir a mensagem: "tem de ir a exame". Caso contrário, imprime a mensagem: "Passaste com x valores".

```
#include <stdio.h>

void main()
{
 float tra, trab = 0.3;
 float final, tes, teste = 0.7;

 printf("\nnota teste?: "); scanf("%f", &tes);
 printf("\nnota trabalho?: "); scanf("%f", &tra);
 final = (trab * tra) + (teste * tes);
 if  (final  <  9.5)
   printf("\ndeve ir a exame");
 else
   printf("\nPassaste com %.1f valores", final);
}
```

---

5. `if` 2: estruturas mais complicadas.

5a) Faz um programa que pede ao utilizador 3 números e escrevo-os no ecrã na ordem **decrescente:**

```
#include <stdio.h>

void main()
{
  int n1, n2, n3, temp;


  clrscr();
  printf("\nEste programa recebe tres inteiros DISTINTOS e vai escrever-os na\ ordem
decrescente. \n");
  printf("\nprimeiro n£mero: "); scanf("%d", &n1);
  printf("\nsegundo n£mero:  "); scanf("%d", &n2);
  printf("\nterceiro n£mero: ");   scanf("%d", &n3);

  /*estrategia:  sempre manter os numeros na ordem correcta  onde n1 ´= maior */
  if (n2 > n1)
    {temp = n1;  n1= n2;  n2 = temp;  }

  if (n3 > n1)
    {temp = n1; n1= n3; n3 = temp;  }

  if (n3 > n2)
    {temp = n2; n2 = n3; n3 = temp; }
```

```
  printf("\n\ordem decrescente: %d  %d  %d", n1,n2,n3);
}
```

Alternativamente você podia ter feito este programa utilizando 6 instruções if. sem mudar os valores das variáveis onde foram inicialmente atribuídas.

---

6. Mais cálculos

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

void main()
{
  float x1,x2,y1,y2;
  float m;

  printf("\ninserir coordenadas primer punto(x1):");
  scanf("%f", &x1);
  printf("\ninserir coordenadas primer punto(y1):");
  scanf("%f", &y1);
  printf("\ninserir coordenadas segundo punto(x2):");
  scanf("%f", &x2);
  printf("\ninserir coordenadas segundo punto(y2):");
  scanf("%f", &y2);

  m = ((y2 - y1) / (x2 -x1));
  printf("\n Pendente: %.2f", m);
  getch();
}
```

Programação I

Aula Prática 3

Sumário

Programas com utilização de estruturas de controlo de fluxo, selecção ou decisão (if, e if ... else).

Programa 1

Faz um programa que calcule o máximo de 2 números reias. Os 2 números devem ser introduzidos pelo utilizador.

Ao ser executado, o programa vai fazer o seguinte (o que aparece em **bold** é escrito pelo utilizador):

```
Introduza um número:
3
Introduza outro número:
8
O máximo é 8.
```

Programa 2

Imagina que a uma dada cadeira, a nota de frequência é obtida da seguinte forma: o trabalho tem peso 25% e o teste tem peso 75%. Faz um programa que calcula a vossa nota de frequência a essa cadeira. Se a nota de frequência for inferior a 9.5, o computador deve imprimir a mensagem: "tem de ir a exame". Caso contrário, imprime a mensagem: "Passaste com x valores".

Ao ser executado, o programa vai fazer o seguinte:

```
Nota do trabalho:
18
Nota do teste:
13
Passaste com 14.25 valores.
```

Programa 3

Modifica o programa que calcula as raízes de uma equação do segundo grau (dado na aula teórica) de modo a que apresente raízes complexas se for caso disso.

Ao ser executado, o programa vai fazer o seguinte:

```
Introduza os coeficientes de um polinómio de grau 2
a = 1
b = 2
c = 2
As raízes são -1+i e -1-i
```

Programa 4

Faz um programa que pede ao utilizador 3 números e escrevo-os no ecrã por ordem crescente:

Ao ser executado, o programa vai fazer o seguinte:

```
Diz um número:
18
Diz outro número:
-4
Diz outro número:
5
Ordem crescente: -4 5 18
```

Programa 5

Faz um programa que calcula o valor absoluto (módulo) de um número.

Ao ser executado, o programa vai fazer o seguinte:

```
Diz um número:
-1
O valor absoluto de -1 é 1.
```

Programa 6

Faz um programa que, ao receber o valor da largura e do comprimento de uma figura geométrica, detecta se esta é um quadrado ou um rectângulo.

Ao ser executado, o programa vai fazer o seguinte:

```
Diz a largura:
7
Diz o comprimento:
5
A figura é um rectângulo.
```

Programa 7

Faz um programa em C que receba uma data de nascimento e que calcule a idade que a pessoa terá em 01/01/2010.

Programa 8

Faz um programa em C que receba 10 valores inteiros e que indique: Quantos são pares, quantos são ímpares, quantos são positivos e quantos são negativos.

Programa 9

Faz um programa em C que calcule o salário de um empregado baseado no n.º de horas que trabalhou, e no seu salário por hora (as horas extraordinarias (> 40) são pagas a dobrar).

Programação I

Soluções dos exercícios da aula prática 3

Programa 1

Faz um programa que calcule o máximo de 2 números reias. Os 2 números devem ser introduzidos pelo utilizador.

```
#include <stdio.h>

main()
{
  float a, b;

  printf("Introduza um número:\n");
  scanf("%f", &a );
  printf("Introduza outro número:\n");
  scanf("%f", &b);
  if( a > b )
    printf("O máximo é %f.\n", a );
  else
    printf("O máximo é %f.\n", b );
}
```

Programa 2

Imgina que a uma dada cadeira, a nota de frequência é obtida da seguinte forma: o trabalho tem peso 25% e o teste tem peso 75%. Faz um programa que calcula a vossa nota de frequência a essa cadeira. Se a nota de frequência for inferior a 9.5, o computador deve imprimir a mensagem: "tem de ir a exame". Caso contrário, imprime a mensagem: "Passaste com x valores".

```
#include <stdio.h>

main()
{
  float trabalho;  /* nota do trabalho */
  float teste;     /* nota do teste */
  float freq;      /* nota da frequência */

  printf("Nota do trabalho:\n");
  scanf("%f", &trabalho );
  printf("Nota do teste:\n");
  scanf("%f", &teste );
  freq = trabalho*0.25 + teste*0.75;
  if( freq >= 9.5 )
    printf("Passaste com %f valores.\n", freq );
  else
    printf("Tens de ir a exame.\n");
}
```

Programa 3

Modifica o programa que calcula as raízes de uma equação do segundo grau (dado na aula teórica) de modo a que apresente raízes complexas se for caso disso.

A chave deste problema é testar se o valor que aparece dentro da raíz quadrada é negativo ou não. Se for

negativo vai dar raízes complexas, senão vai dar raízes reais.

```
delta = b*b - 4ac
se delta < 0
  Raízes Complexas
senão
  Raízes Reais
```

O caso das raízes reais já tinha sido feito na aula teórica. No caso de dar raízes complexas, as soluções vão ser do tipo **a+bi** e **a-bi**, em que **a** e **b** são números reais. Moral da história: precisamos de variáveis para os valores de **a** e **b**. Para não se confundir com o **a** e **b** dos coeficientes do polinómio ax^2 + bx + c, é melhor chamar outros nomes. Por exemplo, **p** e **q**. Assim, se delta < 0 as raízes complexas serão **p+qi** e **p-qi**. O esqueleto do programa é:

```
delta = b*b - 4ac
se delta < 0
  /* Raízes Complexas */
  p = -b / (2a)
  q = sqrt( -delta ) / (2a)
  ...
senão
  /* Raízes Reais */
  ...
```

O programa completo em C é:

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

main()
{
  float a,b,c,    /* coeficientes do polinómio de grau 2 */
        delta,    /* b^2-4ac */
        raiz1,
        raiz2,    /* raízes no caso de serem soluções reais */
        p,        /* parte real da raiz no caso de ser complexa */
        q;        /* valor absoluto da parte imaginária */

  printf("Introduza os coeficientes de um polinómio de grau 2\n");
  printf("a = "); scanf("%f", &a );
  printf("b = "); scanf("%f", &b );
  printf("c = "); scanf("%f", &c );
  delta = b*b - 4*a*c;
  if( delta < 0 )
   {
     /* Raízes complexas */
    p = -b / (2*a);
    q = sqrt(-delta) / (2*a);
    printf("As raízes são %f+%fi e %f-%fi\n", p, q, p, q );
   }
  else
   {
     /* Raízes reais */
    raiz1 = (-b + sqrt(delta)) / (2*a);
    raiz2 = (-b - sqrt(delta)) / (2*a);
    printf("As raízes são %f e %f \n", raiz1, raiz2 );
   }
}
```

Programa 4

Faz um programa que pede ao utilizador 3 números e escrevo-os no ecrã por ordem crescente:

Dados 3 números a, b, c, há 6 (3 factorial) ordens possíveis:

a b c
a c b
b a c
b c a
c a b
c b a

Há muitas maneiras de fazer este programa. Aqui vai uma delas. (NOTA: é preciso ter cuidado com os números repetidos. Por exemplo, se a pessoa introduzir os números 5, 3, 5, o computador deverá escrever 3 5 5).

```
#include <stdio.h>

main()
{
  int a,b,c;

  printf("a = "); scanf("%d", &a);
  printf("b = "); scanf("%d", &b);
  printf("c = "); scanf("%d", &c);

  if( (a<=b) && (b<=c) )
    printf("Ordem crescente: %d %d %d\n", a, b, c);
  else
    if( (a<=c) && (c<=b) )
      printf("Ordem crescente: %d %d %d\n", a, c, b);
    else
      if( (b<=a) && (a<=c) )
        printf("Ordem crescente: %d %d %d\n", b, a, c);
      else
        if( (b<=c) && (c<=a) )
          printf("Ordem crescente: %d %d %d\n", b, c, a);
        else
          if( (c<=a) && (a<=b) )
            printf("Ordem crescente: %d %d %d\n", c, a, b);
          else
            printf("Ordem crescente: %d %d %d\n", c, b, a);
}
```

Aqui vai outra maneira de resolver o problema:

```
#include <stdio.h>

main()
{
  int a, b, c;

  printf("a = "); scanf("%d", &a);
  printf("b = "); scanf("%d", &b);
  printf("c = "); scanf("%d", &c);

  if (a<=b)
   {
    if (c<=a)
      printf("Ordem crescente: %d %d %d\n", c, a, b);
    else
      if (c>=b)
        printf("Ordem crescente: %d %d %d\n", a, b, c);
      else
        printf("Ordem crescente: %d %d %d\n", a, c, b);
   }
    else /*  b < a  */
     {
      if (c<=b)
        printf("Ordem crescente: %d %d %d\n", c, b, a);
      else if (c>=a)
        printf("Ordem crescente: %d %d %d\n", b, a, c);
      else
```

```
        printf("Ordem crescente: %d %d %d\n", b, c, a);
    }
  }
```

Ainda uma terceira solução para o problema:

Aqui, a ideia é primeiro ordenar os números e fazer um único **printf** no final. Ao ordenar garante-se que a variável **a** fica com o menor dos valores, **b** fica com o valor do meio e **c** fica com o maior valor. Para ordenar compara-se o valor de **a** com o de **b** e com o de **c**, trocando-se os valores se **a** for maior. Faz-se o mesmo para o **b** em relação a **c**. Para fazer a troca de valores entre duas variáveis é **sempre** necessário usar uma terceira variável que guarde temporariamente um dos valores.

Por exemplo: Se **a=5** e **b=3**, vamos querer trocar os seus valores. Se se fizer directamente **a=b**, perde-se o valor que **a** tinha anteriormente (5). Bem como se perderia o valor de **b** (3) se fizessemos **b=a**; Por isso temos que ter a variável de troca. Assim, fazendo:

troca = b; /* troca fica com o valor 3 */
b = a; /* b passa a valer 5 e a vale também 5 */
a = troca; /* a passa a valer 3 */

Está a troca de valores feita.

O programa completo em C é:

```
#include <stdio.h>

main()
{
  int a, b, c, troca;

  printf("a = "); scanf("%d", &a);
  printf("b = "); scanf("%d", &b);
  printf("c = "); scanf("%d", &c);

  if (a>b)
   {
    troca = b;
    b = a;
    a = troca;
   }
  if (a>c)
   {
    troca = c;
    c = a;
    a = troca;
   }
  if (b>c)
   {
    troca = c;
    c = b;
    b = troca;
   }

  printf("Ordem crescente: %d %d %d\n", a, b, c );
}
```

Programa 5

Faz um programa em C que calcula o valor absoluto (módulo) de um número.

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

```
main()
{
  int x, absx;

  printf("Introduz um número: ");
  scanf("%d", &x);
  if (x<0)
    absx = -x;
  else
    absx = x;
  printf("O valor absoluto é %d\n", absx);
}
```

Programa 6

Faz um programa que, ao receber o valor da largura e do comprimento de uma figura geométrica, detecta se esta é um quadrado ou um rectângulo.

```
#include <stdio.h>

main()
{
  float largura, comprimento;


  printf("Diz a largura: ");
  scanf("%f", &largura);
  printf("Diz o comprimento: ");
  scanf("%f", &comprimento);

  if (largura == comprimento)
    printf("A figura é um quadrado ");
  else
    printf("A figura é um rectângulo ");
}
```

Programa 7

Faz um programa em C que receba uma data de nascimento e que calcule que idade a pessoa terá em 01/01/2010.

```
#include <stdio.h>

#define ANO 2010
#define MES 1
#define DIA 1

main()
{
  int dia, mes, ano;

  printf("Diz o dia de nascimento: ");
  scanf("%d", &dia);
  printf("Diz o mes de nascimento: ");
  scanf("%d", &mes);
  printf("Diz o ano de nascimento: ");
  scanf("%d", &ano);

  if ( (ano>ANO) || ( (ano==ANO) && ( (dia>=DIA) || (mes>=MES) ) ) )
    printf("Data invalida");
  else if ( (ano<ANO) && (dia==DIA) && (mes==MES) )
    printf("Terás %d anos no dia 01/01/2010, e estarás de Parabéns!",ANO-ano);
  else
    printf("Terás %d anos no dia 01/01/2010",ANO-ano-1);
}
```

Programa 8

Faz um programa em C que receba 10 valores inteiros e que indique: Quantos são pares, quantos são ímpares, quantos são positivos e quantos são negativos.

```
#include <stdio.h>

main()
{
   int n1, n2, n3, n4, n5, n6, n7, n8, n9, n10;
   int n_pares=0, n_impares=0, n_pos=0, n_neg=0;

   printf("Diz o 1..º número inteiro");
   scanf("%d", &n1);
   if ( (n1%2)==0 )
     n_pares++;
   else
     n_impares++;
   if (n1>0)
     n_pos++;
   else
     n_neg++;

   printf("Diz o 2.º número inteiro");
   scanf("%d", &n2);
   if (n2 % 2 == 0) n_pares++;
   else n_impares++;
   if (n2 > 0) n_pos++;
   else n_neg++;

   printf("Diz o 3.º número inteiro");
   scanf("%d", &n3);
   if ( (n3%2)==0 )
     n_pares++;
   else
     n_impares++;
   if (n3>0)
     n_pos++;
   else
     n_neg++;

   /* ... repetindo sempre o mesmo para os restantes números */

   printf("Resultados\n");
   printf(" %d números pares\n",n_pares);
   printf(" %d números impares\n",n_impares);
   printf(" %d números positivos\n",n_pos);
   printf(" %d números negativos\n",n_neg);
}
```

Repara que desta forma a escrita do programa torna-se repetitiva. Mais tarde, irás aprender uma outra maneira muito mais elegante e eficaz de resolver este exercício.

Programa 9

Faz um programa em C que calcule o salário de um empregado baseado no n.º de horas que trabalhou, e no seu salário por hora. Notas: As horas extrordinárias (> 40) são pagas a dobrar.

```
#include <stdio.h>

#define extras 160      /* contam como horas extras as superiores a 160 */
```

```
                        /* 40 horas por semana * 4 semanas por mes = 160 */

main()
{
 float horas, salario_hora, salario;

 printf("Indique o n.º de horas que trabalhou este mes: ");
 scanf("%f", &horas);
 printf("Indique quanto ganha por hora (em escudos): ");
 scanf("%f", &salario_hora);
 if (horas>160.0)
   salario = 2.0 * salario_hora * (horas - 160.0) + salario_hora * 160.0;
 else
   salario = salario_hora * horas;
 printf("O seu salário vai ser: %.2f", salario);
}
```

Programação 1

Aula prática 4

---

1. Cálculo Booleano

expressão	Resposta: .
`23 & 11`	
`23 && 11`	
`75 | 15`	

Agora verifique. Faz um programa que pede dois operandos e mostra o resultado dos operações (&, && e |).

---

2. `switch` (bifurcação múltipla)

Faz um programa que pede para escolheres uma operação aritmética, e, seguidamente, pede os dois operandos sobre os quais queres realizar essa operação. O computador deve fazer a operação apropriada e escrever o resultado no ecrã.
Exemplo do programa ao ser executado:

```
Escolhe uma operação ( + - * / ):
  /
Introduz o primeiro operando
  7
Introduz o segundo operando:
  2

  7 / 2 = 3.5
```

---

3. bifurcações

Faz um programa que dê um parecer qualitativo a uma classificação numérica de 0..20 nos seguinte termos:
(0..4 -> Mau , 5..9 -> Medíocre , 10..13 -> Suficiente , 14..17 -> Bom , 18..20 -> Muito Bom

Exemplo do programa ao ser executado:

```
 Introduz a classificação:
18
 Muito Bom
```

4a. Ciclos 1

Faz um programa que escreva no ecrã "Universidade do Algarve" 10 vezes.

Ao ser executado, o programa vai fazer o seguinte:

```
Universidade do Algarve
Universidade do Algarve
Universidade do Algarve
Universidade do Algarve
Universidade do Algarve
Universidade do Algarve
Universidade do Algarve
Universidade do Algarve
Universidade do Algarve
Universidade do Algarve
```

4b. Ciclos 2

Faz um programa que escreva no ecrã os números de 1 a 10000.

4c. Ciclos 3

Faz um programa que escreva a tabuada de um determinado número.

Exemplo do programa ao ser executado:

```
Introduz um número:
   9
 9 x 1 = 9
 9 x 2 = 18
 9 x 3 = 27
 9 x 4 = 36
 9 x 5 = 45
 9 x 6 = 54
 9 x 7 = 63
 9 x 8 = 72
 9 x 9 = 81
 9 x 10 = 90
```

---

5. (difícil)

Escreva um programa que pede ao utilizador um número (entre 0 e 255) e o programa mostrará o número em formato binário. (Sugestão: usa uma variável do tipo `unsigned char` e usa as operações `x & 128`, etc. Uma solução elegante usa '`for`') Exemplo

```
numero decimal: 33
binario: 00100001
```

Programação Imperativa
Soluções da aula prática 4

1.

```
#include <stdio.h>
main()
{
  unsigned char a, b, c;

  printf("Give first operand: ");
  scanf("%d", &a);
  printf("Give second operand: ");
  scanf("%d", &b);
  c = a & b
  printf("%d & %d = %d\n", a, b, c);
  c = a && b
  printf("%d && %d = %d\n", a, b, c);
  c = a | b
  printf("%d | %d = %d\n", a, b, c);
}
```

```
Give first operand: 23
Give second operand: 11
23 & 11 = 1
23 && 11 = 3
71 | 15 = 79
```

2.

```
#include <stdio.h>
main()
{
  char opcao;
  double a, b;

  do
   {
    printf("Escolhe uma operação ( + - * / ):\n");
    scanf("%c", &opcao);
   }
  while ( (c!='+') && (c!='-') && (c!='*') && (c!='/') );
  printf("Introduz o primeiro operando\n");
  scanf("%f", &a);
  printf("Introduz o segundo operando\n");
  scanf("%f", &b);
  switch (opcao)
   {
     case '+': printf("%f + %f = %f\n", a, b, a+b);
```

```
                break;
      case '-': printf("%f - %f = %f\n", a, b, a-b);
                break;
      case '*': printf("%f * %f = %f\n", a, b, a*b);
                break;
      case '/': printf("%f + %f = %f\n", a, b, a/b);
    }
}
```

```
Escolhe uma operação ( + - * / ):
  /
Introduz o primeiro operando
  7
Introduz o segundo operando:
  2
7.0000 / 2.0000 = 3.5000
```

3.

```
#include <stdio.h>
main()
{
  float a;

  printf("Introduz a classificacao\n");
  scanf("%d", &a);
  if (a<=4.0)
    printf("Mau\n");
  else
    if (a<=9.0)
      printf("Mediocre\n");
    else
      if (a<=13.0)
        printf("Suficiente\n");
      else
        if (a<=17.0)
          printf("Bom\n");
        else
          printf("Muito bom\n");
}
```

```
Introduz a classificação:
  18
Muito Bom
```

4a.

```
#include <stdio.h>
main()
{
  int i;
```

```
  for (i=1; i<=10; i++)
    printf("Universidade do Algarve\n");
}
```

4b.

```
#include <stdio.h>
main()
{
  int i;

  for (i=1; i<=10000; i++)
   printf("%d ", i);
}
```

4c.

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

  printf("Introduz um numero:\n");
  scanf("%d", &n);
  for (i=1; i<=10; i++)
    printf("%d x %d = %d", n, i, n*i);
}
```

5.

```
#include <stdio.h>
main()
{
  unsigned char i, n;

  printf("Numero decimal (0..255): ");
  scanf("%d", &n);
  printf("binario: ");
  i=128;
  while (i>1)
    {
      printf("%d", (n&i)/i);
      i=i/2;
    }
}
```

```
Numero decimal (0..255): 33
binario: 00100001
```

Note:

```
128 = 10000000, 33 = 00100001, 33&128 = 00000000=0, (33&128)/128 = 0
 64 = 01000000, 33 = 00100001,  33&64 = 00000000=0,   (33&64)/64 = 0
 32 = 00100000, 33 = 00100001,  33&32 = 00100000=32,  (33&32)/32 = 1
 16 = 00010000, 33 = 00100001,  33&16 = 00000000=0,   (33&16)/16 = 0
  8 = 00001000, 33 = 00100001,   33&8 = 00000000=0,     (33&8)/8 = 0
  4 = 00000100, 33 = 00100001,   33&4 = 00000000=0,     (33&4)/4 = 0
  2 = 00000010, 33 = 00100001,   33&2 = 00000000=0,     (33&2)/2 = 0
  1 = 00000001, 33 = 00100001,   33&1 = 00000001=1,     (33&1)/1 = 1
```

Programação 1

Aula prática 4

---

1. Cálculo Booleano

expressão	Resposta: .
`33 & 21`	
`23 && 14`	
`64 | 18`	

Agora verifique. Faz um programa que pede dois operandos e mostra o resultado das operações (&, && e |).

---

2. `switch` - case (escolha múltipla)

Faz um programa que pede o número dum mês do ano [1..12], e converte o número na palavra que expressa o dito mês.
Input:
Introduzir o mês (numérico) do ano: 3

Output:
Trata-se do mês de Março.

Refinamento: se o mês introduzido pelo utilizador esta fora do intervalo aceite, rejeitar e no mesmo ciclo voltar a pedir um número.

---

3. Ampliar o programa da pergunta (2) de maneira que dada uma data expressada em números, o seu programa devolva

a mesma data escrita em texto.
Input do programa:

Dia da semana?: 3
Dia do mês? : 15
Mês?: 9
Ano?: 1993

Output requerido: "Quarta-feira 15 de Setembro de 1993"

4a. Ciclos 1

Escreva um programa que vai lêr uma variavel numérica chamada "tempo". A variável começa com o valor 0.

escreva um ciclo que enquanto esta variável não atingir o valor 11, escreve no ecrã a frase:

"Estamos na iteração 1, 2, ...10".

Dica: utilizar o valor da variavel numérica para construir a frase cada vez.

4b. Ciclos 2

Faz um programa que lê inteiros e imprime seus cubos até que o usuário insira o valor de sentinela -1.

4c. Ciclos 3

Faz um programa que calcula a soma 1 + 2 + 3 + ... + n, para um inteiro n fornecido como entrada..

Input:

```
   Introduzir um inteiro positivo: 5
```

```
Output: A soma dos primeiros 5 inteiros é: 15
```

---

5. Aprenda a lêr um algoritmo e converter para instruções de computador em C

Escreva um programa que:

Vai implementar directamente o operador de quociente "/" e o operador de resto "%" para a divisão de interior positivos.
Numa fração **n** /**d**, subtrair repetidamente o **d** do **n** até que **n** seja menor que **d**.
Nesse ponto, o valor de **n** será o resto e o número **q** de iterações exigidas para alcançá-lo será o quociente.

Input:
Insira o numerador: 30
Insira o denominador: 7

Output:
30 / 7 = 4
30 % 7 = 2
4 * 7 + 2 = 30

Dica: utilizar um ciclo for com valores iniciais apropiados e o programa está prácticamente feito!

Programação 1

Aula prática 4

Sumário

- Programas com utilização de estruturas de controlo de fluxo, selecção ou decisão (if, if ... else, switch ... case).
- Programas com utilização de instruções de iteração (ciclos while).

Programa 1

Faz um programa que pede para escolheres uma operação aritmética, e, seguidamente, pede os dois operandos sobre os quais queres realizar essa operação. O computador deve fazer a operação apropriada e escrever o resultado no ecrã.

Exemplo do programa ao ser executado:

```
Escolhe uma operação ( + - * / ):
/
Introduz o primeiro operando:
7
Introduz o segundo operando:
2
7 / 2 = 3.5
```

Programa 2

Modifica o programa anterior de modo a não permitir a divisão por zero.

Exemplo do programa ao ser executado:

```
Escolhe uma operação ( + - * / ):
/
Introduz o primeiro operando:
7
Introduz o segundo operando:
0
Operação inválida
```

Programa 3

Faz um programa que dê um parecer qualitativo a uma classificação numérica de 0..20 nos seguinte termos:
(0..4 -> Mau , 5..9 -> Medíocre , 10..13 -> Suficiente , 14..17 -> Bom , 18..20 -> Muito Bom

Exemplo do programa ao ser executado:

```
Introduz a classificação:
18
Muito Bom
```

Programa 4

Faz um programa que escreva no ecrã "Universidade do Algarve" 10 vezes.

Ao ser executado, o programa vai fazer o seguinte:

```
Universidade do Algarve
Universidade do Algarve
Universidade do Algarve
Universidade do Algarve
Universidade do Algarve
Universidade do Algarve
Universidade do Algarve
Universidade do Algarve
Universidade do Algarve
Universidade do Algarve
```

Programa 5

Faz um programa que escreva no ecrã os números de 1 a 10000.

Programa 6

Faz um programa que escreva a tabuada de um determinado número.

Exemplo do programa ao ser executado:

```
Introduz um número:
9
9 x 1 = 9
9 x 2 = 18
9 x 3 = 27
9 x 4 = 36
9 x 5 = 45
9 x 6 = 54
9 x 7 = 63
9 x 8 = 72
9 x 9 = 81
9 x 10 = 90
```

Programa 7

Faz um programa que leia um valor numérico inteiro e conte número de dígitos que esse valor numérico tem.

Exemplo do programa ao ser executado:

Introduz um número inteiro:
123
123 tem 3 dígitos.

Programa 8

Faz um programa que leia um *byte* em binário, isto é, um valor numérico de oito bits constituído apenas por **zeros** e/ou **uns**, lembras-te? O programa deve determinar quantos zeros e quantos uns esse *byte*

possui.

Exemplo do programa ao ser executado:

Introduz um numero binário de oito dígitos:
00101100
Tem 5 zeros e 3 uns.

Programação I

Soluções dos exercícios da aula prática 4

Programa 1

```
/*
 *  a4_p1: Faz pequenas operacoes aritmeticas
 *
 *  Input: Uma operacao e dois operandos
 * Output: Resultado da operacao
 */

 #include<stdio.h>

 main()
 {
        char operacao;
        double op1, op2;

        printf("Qual a operacao (+ - * /)? ");
        scanf("%c", &operacao);
        printf("Quais os dois operandos? ");
        scanf("%lf%lf", &op1, &op2);

        switch (operacao)
        {
                case '+': /* adicao */
                    printf("%lf + %lf = %lf\n", op1, op2, op1+op2);
                    break;
                case '-': /* subtraccao */
                    printf("%lf - %lf = %lf\n", op1, op2, op1-op2);
                    break;
                case '*': /* multiplicacao */
                    printf("%lf * %lf = %lf\n", op1, op2, op1*op2);
                    break;
                case '/': /* divisao */
                    printf("%lf / %lf = %lf\n", op1, op2, op1/op2);
                    break;
        }
 }
```

Programa 2

```
/*
 *  a4_p2: Faz pequenas operacoes aritmeticas
 *         (com base em a4_p1, mas, evita divisao por zero)
 *  Input: Uma operacao e dois operandos
 * Output: Resultado da operacao
 */

 #include<stdio.h>

 main()
 {
        char operacao;
        double op1, op2;

        printf("Qual a operacao (+ - * /)? ");
        scanf("%c", &operacao);
```

```
        printf("Quais os dois operandos? ");
        scanf("%lf%lf", &op1, &op2);

        switch (operacao)
        {
                case '+': /* adicao */
                    printf("%lf + %lf = %lf\n", op1, op2, op1+op2);
                    break;
                case '-': /* subtraccao */
                    printf("%lf - %lf = %lf\n", op1, op2, op1-op2);
                    break;
                case '*': /* multiplicacao */
                    printf("%lf * %lf = %lf\n", op1, op2, op1*op2);
                    break;
                case '/': /* divisao */
                    if (op2 == 0.0)
                        printf("Operacao invalida\n");
                    else
                        printf("%lf / %lf = %lf\n", op1, op2, op1/op2);
                    break;
        }
}
```

Programa 3

```
/*
 *  a4_p3: Da um parecer qualitativo
 *         a uma classificacao numerica
 *  Input: Nota em valores (numero inteiro de 0 a 20)
 * Output: Parecer qualitativo
 *         (Mau, Medíocre, Suficiente, Bom, Muito Bom)
 */

#include <stdio.h>

main()
{
        int nota;

        printf("Introduz a classificacao: ");
        scanf("%d", &nota);

        switch(nota)
        {
                case  0:
                case  1:
                case  2:
                case  3:
                case  4:
                        printf("Mau\n");
                        break;
                case  5:
                case  6:
                case  7:
                case  8:
                case  9:
                        printf("Mediocre\n");
                        break;
                case 10:
                case 11:
                case 12:
                case 13:
                        printf("Suficiente\n");
                        break;
                case 14:
                case 15:
                case 16:
                case 17:
                        printf("Bom\n");
                        break;
```

```
                case 18:
                case 19:
                case 20:
                        printf("Muito Bom\n");
                        break;
        }
}
```

Obs: Claro que este programa poderia ter sido feito com instruções **if-else.** Mas foi feito com a instrução **switch-case** para perceberes melhor o seu modo de funcionamento. O que aconteceria se não tivesses usado os **break**s e introduzisses uma classificação, p/ exemplo, de apenas 3 valores? Pensa no assunto.

Programa 4

```
/*
 *  a4_p4: UAlg 10x
 *
 *  Input:
 * Output: Escreve a mesma frase 10 vezes
 */

#include <stdio.h>

main()
{
        int n;  /* contador para o numero de vezes */

        for (n= 1; n<=10; i++)
          printf("Universidade do Algarve\n");
}
```

Programa 5

```
/*
 *  a4_p5:  Grande contador
 *
 *  Input:
 * Output: Escreve no ecra os numeros de 1 a 10000
 */

#include <stdio.h>

main()
{
        int n;  /* contador */

        for (n=1; n<=10000; i++)
          printf("%05d\n", n);
}
```

Obs: Repara no modo como foi formatado o output. Na instrução **printf**, em vez do habitual **%d** aparece **%05d**. Desta forma os números que mandamos escrever no ecrã surgem alinhados à direita num campo de 5 dígitos preenchido com zeros à esquerda. Se puséssemos apenas %5d acontecia o mesmo mas sem o preenchimento com zeros à esquerda.

Experimenta redireccionar o output para um ficheiro de texto para que possas ver bem as diferenças na forma de apresentar os primeiros números. Para isso, ao executares o programa, deverás fazer o que se chama **redireccionamento de saída**. Se o programa executável se chamar "a4_p5" e quiseres criar um ficheiro chamado "numeros.txt" faz assim:

a4_p5 > numeros.txt

Agora vê o conteúdo de numeros.txt

Programa 6

```
/*
 *  a4_p6:  Tabuada
 *
 *  Input: Numero inteiro (o multiplicando)
 * Output: Tabuada desse numero
 */

 #include <stdio.h>

 main()
 {
        int a;          /* multiplicando */
        int b;          /* multiplicador */

        printf("Introduza um numero: ");
        scanf("%d", &a);

        for (b=1; b<=10; b++)
          printf("%d x %d = %d\n", a, b, a*b);
 }
```

Obs: Como poderias formatar a apresentação dos números no ecrã de modo a surgirem todos alinhadinhos à direita? (Lê as observações do programa 5)

Programa 7

```
/*
 *  a4_p7: Conta quantos digitos tem um numero inteiro
 *
 *  Input: Um valor numerico inteiro
 * Output: Numero de digitos que o constitui
 */



    #include <stdio.h>

    main()
    {
      int n_inicial;        /* numero a avaliar */
      int n_digitos;        /* contador de digitos */
      int n;                /* variavel de manobra p/ nao perder n_inicial */

      printf("Introduza um numero inteiro: ");
      scanf("%d", &n_inicial);
      if( n_inicial == 0 )
        n_digitos = 1;
      else
       {
         if( n_inicial < 0 )
           n = -n_inicial;
         else
           n = n_inicial;
         n_digitos = 0;
         while (n != 0)
           {
             n = n / 10;
             n_digitos = n_digitos + 1;
           }
       }
      printf("%d tem %d digitos", n_inicial, n_digitos);
    }
```

Programa 8

```
/*
 *  a4_p8: Conta zeros e uns de um valor binario
 *
 *  Input: Um valor numerico binario
 * Output: Numero de zeros e de uns que ele possui
 */


#include <stdio.h>

main()
{
  unsigned long n_binario;      /* valor a avaliar */
  int uns;                      /* contador de 1s */
  int zeros;                    /* contador de 0s */
  int resto;                    /* resto da divisao */
  int i;                        /* contador p/ o ciclo */

  printf("Introduza um valor binario de oito digitos (so zeros e uns): ");
  scanf("%ld", &n_binario);

  zeros = 0;
  uns = 0;
  i = 8;
  while (i>0)
  {
    resto = n_binario % 10;

    if (resto == 0)
        zeros = zeros + 1;
    if (resto == 1)
        uns = uns + 1;

    n_binario = n_binario / 10;
    i = i-1;
  }
  printf("Tem %d zeros e %d uns.\n", zeros, uns);
}
```

Obs: Podes melhorar este programa para, no caso de o utilizador introduzir um valor não binário, apresentar uma mensagem a indicar "Numero inválido".

Programação Imperativa
Aula Prática 5

Sumário

Programas com utilização de instruções de iteração (while, for e do...while).

Programa 1

Faz um programa que escreve no ecrã uma tabela de conversão de graus Celcius para Fahrenheit. A tabela deve apresentar os graus Celcius de 0 a 40 com intervalos de 2 em 2.

Exemplo do programa ao ser executado:

```
Celcius  Fahrenheit
   0.0     32.0
   2.0     35.6
   4.0     39.2
   6.0     42.8
   ...     ...
  40.0    104.0
```

Programa 2

Modifica o programa que calcula a área do círculo de modo a que o programa funcione ininterruptamente até o utilizador introduzir o valor zero para o raio. Nessa altura, o programa deve dizer que a área é zero, e terminar com um "Até logo".

Exemplo do programa ao ser executado:

```
Introduz o raio: 3
A área é 28.27
Introduz o raio: 2.5
A área é 19.63
Introduz o raio: -2
Esse raio é inválido
Introduz o raio: 1
A área é 3.14
Introduz o raio: 0
A área é 0.00
Até logo.
```

Programa 3

Faz um programa que vai pedindo números ao utilizador até que este introduza o número -1. O computador deve dizer a média dos números introduzidos (excluindo o -1).

Exemplo do programa ao ser executado:

```
Introduz uma série de números e termina com -1.
5
11
6
8
-1
A média é 7.5
```

Programa 4

Modifica o programa anterior, de modo a dar o mínimo, máximo e média.

Exemplo do programa ao ser executado:

```
Introduz uma série de números e termina com -1.
5
11
6
8
-1
A mínimo é 5
O máximo é 11
A média é 7.5
```

Programa 5

Faz um programa para ver se um número é primo ou não (um número só é primo se apenas for divisível por 1 e por si próprio).

Exemplo do programa ao ser executado:

```
Introduza um número:
4
4 não é primo
```

Programa 6

Faz um programa para calcular o factorial de um número.

Tenta fazer três versões deste programa, uma usando a instrução **for**, outra utilizando a instrução **while** e uma terceira com a instrução **do...while**.

Exemplo do programa ao ser executado:

```
Introduza um número:
5
O factorial de 5 é 120.
```

O factorial de um número é definido da seguinte forma:

```
factorial(n) = n * n-1 * n-2 * ... * 1
```

Programa 7

Faz um programa que calcula todos os divisores de um número. Exemplo do programa ao ser executado:

```
Introduz um número:
24
Os divisores de 24 são 1 2 3 4 6 8 12 24
```

Programa 8

Os números de Fibonacci são definidos da seguinte forma:

```
f(1) = 1
f(2) = 1
f(n) = f(n-1) + f(n-2),   para n > 2  (n é um número Natural)
```

Faz um programa que escreve no ecrã os primeiros 20 números de Fibonacci.

Programa 9

Faz um programa que pede um valor decimal inteiro e o converte para binário. Exemplo do programa ao ser executado:

```
Introduz um número:
9
O número em binário: 1001
```

Programa 10

Altera o programa do exercício 9 de forma a que converta o número de decimal para qualquer outra base. Exemplo do programa ao ser executado:

```
Introduz um número:
9
Introduz a base de conversão:
8
O número convertido para a base 8: 11
```

Programação Imperativa
Soluções dos exercícios da aula prática 5

Programa 1

Faz um programa que escreve no ecrã uma tabela de conversão de graus Celcius para Fahrenheit. A tabela deve apresentar os graus Celcius de 0 a 40 com intervalos de 2 em 2.

```
#include <stdio.h>

main()
{
  double c, f;

  printf("Celcius  Fahrenheit\n");
  c = 0.0;
  while( c <= 40.0 )
   {
     f = 9.0/5.0 * c + 32.0;
     printf(" %5.1lf   %5.1lf\n", c, f );
     c = c + 2.0;
   }
}
```

Programa 2

Modifica o programa que calcula a área do círculo de modo a que o programa funcione ininterruptamente até o utilizador introduzir o valor zero para o raio. Nessa altura, o programa deve dizer que a área é zero, e terminar com um "Até logo".

```
#include <stdio.h>

#define PI  3.14159

main()
{
  double area, raio;

  do
   {
     printf("Introduz o raio: ");
     scanf("%lf", &raio);
     if( raio < 0.0 )
       printf("Esse raio é inválido\n");
     else
      {
        area = PI * raio * raio;
        printf("A área é %lf\n", area );
      }
   }
  while( raio != 0 );
  printf("Até logo.\n");
}
```

Programa 3

Faz um programa que vai pedindo números ao utilizador até que este introduza o número -1. O computador deve dizer a média dos números introduzidos (excluindo o -1).

```
#include <stdio.h>

main()
{
```

```
  double x, soma;
  int n;

  n = 0;
  soma = 0.0;
  printf("Introduz uma série de números e termina com -1\n");
  do
    {
      scanf("%lf", &x );
      if (x!=-1.0)
       {
         soma = soma + x;
         n++;
       }
    }
  while ( x != -1 );
  if (n!=0)
    printf("A média é %lf\n", soma / (double) n);
  else
    printf("Média indefinida\n");
}
```

Programa 4

Modifica o programa anterior, de modo a dar o mínimo, máximo e média.

```
#include <stdio.h>

main()
{
  double x, soma, min, max;
  int n;

  n = 0;
  soma = 0.0;
  printf("Introduz uma série de números e termina com -1\n");
  do
   {
     scanf("%lf", &x);
     if (x!=-1.0)
       {
         if (n==0)
           {
             min = x;
             max = x;
           }
         soma = soma + x;
         if (x<min) min = x;
         if (x>max) max = x;
         n++;
       }
   }
  while (x!=-1.0);
  if (n!=0)
    {
      printf("O mínimo é %lf\n", min);
      printf("O máximo é %lf\n", max);
      printf("A média é %lf\n", soma / (double) n);
    }
  else
    printf("Média, máximo e mínimo indefinidos\n");
}
```

Programa 5

Faz um programa para ver se um número é primo ou não (um número só é primo se apenas for divisível por 1 e por si próprio).

Vamos resolver o programa de 4 maneiras diferentes (nota: a definição de número primo é aquela que foi dada no enunciado, mas só é válida para os números Naturais maiores ou iguais a 2. Por outras palavras, o número 1 não é primo. Peço desculpa por não ter referido isto no enunciado).

Solução 1:

```
#include <stdio.h>

main()
{
  int i, n;
  int primo;  /* a variável primo só vai assumir os valores 0 e 1 */
              /* 0 significa que n não é primo */
              /* 1 significa que n é primo */

  printf("Introduz um número: \n");
  scanf("%d", &n);
  if( n <= 1 )
    printf("%d não é primo\n", n );
  else
   {
     i = 2;
     primo = 1;  /* vamos assumir que n é primo */
     for (i=2; i<n; i++)
      {
        if( n % i == 0 )
          /* afinal não é primo */
          primo = 0;
      }
     if( primo == 1 )
       printf("%d é primo\n", n );
     else
       printf("%d não é primo\n", n );
   }
}
```

Solução 2: Tem a vantagem de abandonar o ciclo assim que houver uma divisão que dê resto 0. Apenas faz sentido com o ciclo while. Pois já não é bem contável:

```
#include <stdio.h>

main()
{
  int i, n;
  int primo;  /* a variável primo só vai assumir os valores 0 e 1 */
              /* 0 significa que n não é primo */
              /* 1 significa que n é primo */

  printf("Introduz um número: \n");
  scanf("%d", &n);
  if( n <= 1 )
    printf("%d não é primo\n", n);
  else
   {
     i = 2;
     primo = 1;  /* vamos assumir que n é primo */
     while( (i<n) && (primo==1) )
      {
        if( n % i == 0 )
          /* afinal não é primo */
          primo = 0;
        i++;
      }
     if( primo == 1 )
       printf("%d é primo\n", n );
     else
       printf("%d não é primo\n", n );
   }
```

```
}
```

Solução 3: a mesma coisa que a solução anterior, mas utiliza um `break` para abandonar o ciclo.

```
#include <stdio.h>

main()
{
  int i, n;

  printf("Introduz um número: \n");
  scanf("%d", &n );
  if( n <= 1 )
    printf("%d não é primo\n", n );
  else
   {
     i = 2;
     while (i<n)
      {
        if( (n%i)==0 )
          /* não é primo */
          break;
        i++;
      }
     if( i == n )
       printf("%d é primo\n", n );
     else
       printf("%d não é primo\n", n );
   }
}
```

Solução 4: para ver se um número *n* é primo, não é necessário fazer as divisões todas desde *2* até *n-1*. Basta fazer as divisões desde *2* até *sqrt(n)*. Porquê?

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

main()
{
  int i, n, raiz;

  printf("Introduz um número: \n");
  scanf("%d", &n );
  if( n <= 1 )
    printf("%d não é primo\n", n );
  else
   {
     raiz = (int) sqrt( (double) n);
     i = 2;
     while (i<=raiz)
      {
        if( (n%i)==0 )
          /* não é primo */
          break;
        i++;
      }
     if(i>raiz)
       printf("%d é primo\n", n );
     else
       printf("%d não é primo\n", n );
   }
}
```

Programa 6

Faz um programa para calcular o factorial de um número.

Com a instrução **for**:

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

```
main()
{
  int factorial,n,i;

  printf("Diz um numero: ");
  scanf("%d",&n);
  factorial=1;
  for (i=n; i>1; i--)
     factorial=factorial*i;
  printf("O factorial de %d e %d", n, factorial);
}
```

Programa 7

Faz um programa que calcula todos os divisores de um número.

Resolução

```
#include <stdio.h>

main()
{
  int n, i;

  printf("Introduz um número:\n");
  scanf("%d", &n);
  printf("Os divisores de %d são ", n );
  for (i=1; i<=n; i++)
   {
     if( (n%i)==0 )
       printf("%d ", i);
   }
  printf("\n");
}
```

Programa 8

Faz um programa que escreve no ecrã os primeiros 20 números de Fibonacci.

Resolução

```
#include <stdio.h>


main()
{
  int i,
      fi,      /* F(i)   */
      fi_1,    /* F(i-1) */
      fi_2;    /* F(i-2) */

  fi_1 = 1;
  fi_2 = 1;
  printf("1\n");
  printf("1\n");
  for( i=3; i<=20; i++)
   {
     fi = fi_1 + fi_2;    /* F(i) = F(i-1) + F(i-2) */
     printf("%d\n", fi );
     fi_2 = fi_1;
     fi_1 = fi;
   }
}
```

Programa 9

Faz um programa que pede um valor decimal inteiro positivo e o converte para binário.

Resolução

```
#include <stdio.h>

main()
{
   int dec, pot10;
   unsigned long nbin;

   printf("Introduza um valor decimal: ");
   scanf("%d",&dec);
   pot10=1;
   nbin=0;
   while (dec > 0)
      {
        nbin = nbin + pot10 * (dec % 2);
        dec = dec / 2;
        pot10 = pot10 * 10;
      }
   printf("O valor em binario: %ld\n",nbin);
}
```

Programa 10

Altera o programa do exercício 9 de forma a que converta o número de decimal para qualquer outra base.

Resolução

```
#include <stdio.h>

main()
{
   int dec, pot10, base;
   unsigned long nconv;

   printf("Introduza um valor decimal: ");
   scanf("%d",&dec);
   printf("Introduza a base de conversao: ");
   scanf("%d",&base);
   pot10=1;
   nconv=0;
   while (dec > 0)
      {
        nconv = nconv + pot10 * (dec % base);
        dec = dec / base;
        pot10 = pot10 * 10;
      }
   printf("O valor na base %d: %ld\n", base, nconv);
}
```

A resolução apresentada funciona correctamente apenas para bases de conversão menores ou iguais a 10.

Porque não funciona correctamente para bases superiores? Pensa sobre o assunto.

Programação Imperativa
Aula Prática 5

Sumário

Programas com utilização de instruções de iteração (while, for, do...while, break).

Programa 5.1
Utilizar os ciclos adequados que sejam necessários para:
Contar e imprimir para um dia, todas as horas, todos os minutos e todos os segundos quando se cumprir a seguinte restrição: todos os seis dígitos sejam idênticos. A formatação de saída deve ser: hh.mm.ss
Ex: 22.22.22

Programa 5.2
Escreva um programa capaz de localizar o maior valor duma sequência de números inteiros positivos de entrada. O programa termina quando o utilizador entrar –1.

Ex: Insira inteiros positivos (-1 para sair): 13, 21, 66, 3, 44, 99, 86, 98, 0, -1
O maior número da sequência é o número 99.

Programa 5.3
Escreva um programa que imprime as 12 primeiras tabelas de multiplicação desde o 1 até o 12 inclusivamente. As colunas impressas devem ficar rigorosamente alinhadas
Ex:
1 2 3 4 5 6 7 8 9 10 11 12
2 4 6 8 10 12 14 16 18 20 22 24
3 6 9 ...
etc.

Programa 5.4
Execute um programa que multiplica e imprime no ecrã uma série de dois inteiros i1 e i2 fornecidos pelo utilizador e todas as sequências que seguem quando ambos números são incrementados de 1 unidade cada vez. Isto é: i1, i1+1, i1+2 e simultaneamente i2, i2+1, i2+2,...etc. O incremento de cada variável é de responsabilidade do programa. Existem duas restrições na execução deste algoritmo:
A primeira é que os números iniciais devem ser inteiros positivos inferiores a 100.
A condição para terminar este ciclo de multiplicações acontece quando o produto da última multiplicação efectuada exceder o maior número inteiro capaz de ser armazenado numa variável do tipo inteiro. Este último valor não deve ser impresso no ecrã.

Ex: Insira dois números para iniciar o ciclo de multiplicações: 2 3
2 x 3 = 6, 3 x 4 = 12, 4 x 5 = 20, 5 x 6 = 30, 6 x 7 = 42,

Para o formato de saída imprima 5 multiplicações na mesma alinha do ecrã conforme o formato acima indicado.

Programa 5.5
Modificar o programa anterior, de modo de contabilizar quantos dos resultados produzidos são números impar. Imprima este resultado, assim como o número total de multiplicações efectuadas.

Programa 5.6

Dado um número inteiro i, imprimir todos os números sequenciais seguintes (i, i+1, i+2,...) até que o próximo i seja maior que um número sentinela N também fornecido pelo utilizador. Utilizar a instrução break para sair do ciclo ANTES de imprimir o seguinte número maior que N.

Programa 7

Faz um programa que calcula todos os divisores de um número. Exemplo do programa ao ser executado:

```
Introduz um número:
24
Os divisores de 24 são 1 2 3 4 6 8 12 24
```

Programa 8

Baseado na tabela de multiplicação do exercicio 5.3 utilizar ciclos aninhados para imprimir uma tabela triangular de multiplicação

Ex:

```
1
2  4
3    6  9
4    6  12  16
5    10  15  20  25
6    12  18  24  30  36
...
...
12  24  36  48  60  72 84  96  108  120  132  144
```

Programa 9

Faz um programa que pede um valor decimal inteiro e o converte para binário. Exemplo do programa ao ser executado:

```
Introduz um número:
9
O número em binário: 1001
```

Programa 10

Altera o programa do exercício 9 de forma a que converta o número de decimal para qualquer outra base. Exemplo do programa ao ser executado:

```
Introduz um número:
9
Introduz a base de conversão:
8
O número convertido para a base 8: 11
```

Programação I

Aula Prática 5

Sumário

Programas com utilização de instruções de iteração (while, for e do...while).

Programa 1

Faz um programa que escreve no ecrã uma tabela de conversão de graus Celcius para Fahrenheit. A tabela deve apresentar os graus Celcius de 0 a 40 com intervalos de 2 em 2.

Exemplo do programa ao ser executado:

```
Celcius  Fahrenheit
   0.0     32.0
   2.0     35.6
   4.0     39.2
   6.0     42.8
   ...     ...
  40.0    104.0
```

Programa 2

Modifica o programa que calcula a área do círculo de modo a que o programa funcione ininterruptamente até o utilizador introduzir o valor zero para o raio. Nessa altura, o programa deve dizer que a área é zero, e terminar com um "Até logo".

Exemplo do programa ao ser executado:

```
Introduz o raio: 3
A área é 28.27
Introduz o raio: 2.5
A área é 19.63
Introduz o raio: -2
Esse raio é inválido
Introduz o raio: 1
A área é 3.14
Introduz o raio: 0
A área é 0.00
Até logo.
```

Programa 3

Faz um programa que vai pedindo números ao utilizador até que este introduza o número -1. O computador deve dizer a média dos números introduzidos (excluindo o -1).

Exemplo do programa ao ser executado:

```
Introduz uma série de números e termina com -1.
5
11
```

```
6
8
-1
A média é 7.5
```

Programa 4

Modifica o programa anterior, de modo a dar o mínimo, máximo e média.

Exemplo do programa ao ser executado:

```
Introduz uma série de números e termina com -1.
5
11
6
8
-1
A mínimo é 5
O máximo é 11
A média é 7.5
```

Programa 5

Faz um programa para ver se um número é primo ou não (um número só é primo se apenas for divisível por 1 e por si próprio).

Exemplo do programa ao ser executado:

```
Introduza um número:
4
4 não é primo
```

Programa 6

Faz um programa para calcular o factorial de um número.

Tenta fazer três versões deste programa, uma usando a instrução **for**, outra utilizando a instrução **while** e uma terceira com a instrução **do...while**.

Exemplo do programa ao ser executado:

```
Introduza um número:
5
O factorial de 5 é 120.
```

O factorial de um número é definido da seguinte forma:

```
factorial(n) = n * n-1 * n-2 * ... * 1
```

Programa 7

Faz um programa que calcula todos os divisores de um número. Exemplo do programa ao ser executado:

```
Introduz um número:
24
Os divisores de 24 são 1 2 3 4 6 8 12 24
```

Programa 8

Os números de Fibonacci são definidos da seguinte forma:

```
f(1) = 1
f(2) = 1
f(n) = f(n-1) + f(n-2),   para n > 2  (n é um número Natural)
```

Faz um programa que escreve no ecrã os primeiros 20 números de Fibonacci.

Programa 9

Faz um programa que pede um valor decimal inteiro e o converte para binário. Exemplo do programa ao ser executado:

```
Introduz um número:
9
O número em binário: 1001
```

Programa 10

Altera o programa do exercício 9 de forma a que converta o número de decimal para qualquer outra base. Exemplo do programa ao ser executado:

```
Introduz um número:
9
Introduz a base de conversão:
8
O número convertido para a base 8: 11
```

Programação I

Soluções dos exercícios da aula prática 5

Programa 1

Faz um programa que escreve no ecrã uma tabela de conversão de graus Celcius para Fahrenheit. A tabela deve apresentar os graus Celcius de 0 a 40 com intervalos de 2 em 2.

```
#include <stdio.h>

main()
{
  double c, f;

  printf("Celcius  Fahrenheit\n");
  c = 0;
  while( c <= 40 )
   {
     f = 9.0/5.0 * c + 32;
     printf(" %5.1lf   %5.1lf\n", c, f );
     c = c + 2;
   }
}
```

Programa 2

Modifica o programa que calcula a área do círculo de modo a que o programa funcione ininterruptamente até o utilizador introduzir o valor zero para o raio. Nessa altura, o programa deve dizer que a área é zero, e terminar com um "Até logo".

```
#include <stdio.h>

#define PI  3.14159

main()
{
  double area, raio;

  do
   {
     printf("Introduz o raio: ");
     scanf("%lf", &raio);
     if( raio < 0 )
       printf("Esse raio é inválido\n");
     else
      {
        area = PI * raio * raio;
        printf("A área é %lf\n", area );
      }
   }
  while( raio != 0 );
  printf("Até logo.\n");
}
```

Programa 3

Faz um programa que vai pedindo números ao utilizador até que este introduza o número -1. O computador deve dizer a média dos números introduzidos (excluindo o -1).

```
#include <stdio.h>

main()
{
  double x, soma;
  int n;

  n = 0;
  soma = 0;
  printf("Introduz uma série de números e termina com -1\n");
  scanf("%lf", &x );
  if( x == -1 )
    printf("Média indefinida\n");
  else
    {
      do
       {
         soma = soma + x;
         n++;
         scanf("%lf", &x );
       }
      while ( x != -1 );
      printf("A média é %lf\n", soma/n);
    }
}
```

Programa 4

Modifica o programa anterior, de modo a dar o mínimo, máximo e média.

```
#include <stdio.h>

main()
{
  double x, soma, min, max;
  int n;

  n = 0;
  soma = 0;
  printf("Introduz uma série de números e termina com -1\n");
  scanf("%lf", &x );
  if( x == -1 )
    printf("Média, máximo e mínimo indefinidos\n");
  else
    {
      max = x;
      min = x;
      do
       {
         soma = soma + x;
         if( x < min ) min = x;
         if( x > max ) max = x;
         n++;
         scanf("%lf", &x );
       }
      while ( x != -1 );
      printf("O mínimo é %lf\n", min );
      printf("O máximo é %lf\n", max );
      printf("A média é %lf\n", soma/n);
    }
}
```

Programa 5

Faz um programa para ver se um número é primo ou não (um número só é primo se apenas for divisível por 1 e por si próprio).

Vamos resolver o programa de 4 maneiras diferentes (nota: a definição de número primo é aquela que foi dada no enunciado, mas só é válida para os números Naturais maiores ou iguais a 2. Por outras palavras, o número 1 não é primo. Peço desculpa por não ter referido isto no enunciado).

Solução 1:

```
#include <stdio.h>

main()
{
  int i, n;
  int primo;  /* a variável primo só vai assumir os valores 0 e 1 */
              /* 0 significa que n não é primo */
              /* 1 significa que n é primo */

  printf("Introduz um número: \n");
  scanf("%d", &n );
  if( n <= 1 )
    printf("%d não é primo\n", n );
  else
   {
     i = 2;
     primo = 1;  /* vamos assumir que n é primo */
     while( i < n )
      {
        if( n % i == 0 )
          /* afinal não é primo */
          primo = 0;
        i++;
      }
     if( primo == 1 )
       printf("%d é primo\n", n );
     else
       printf("%d não é primo\n", n );
   }
}
```

Solução 2: tem a vantagem de abandonar o ciclo while assim que houver uma divisão que dê resto 0.

```
#include <stdio.h>

main()
{
  int i, n;
  int primo;  /* a variável primo só vai assumir os valores 0 e 1 */
              /* 0 significa que n não é primo */
              /* 1 significa que n é primo */

  printf("Introduz um número: \n");
  scanf("%d", &n );
  if( n <= 1 )
    printf("%d não é primo\n", n );
  else
   {
     i = 2;
     primo = 1;  /* vamos assumir que n é primo */
     while( (i < n) && (primo == 1) )
      {
        if( n % i == 0 )
          /* afinal não é primo */
          primo = 0;
        i++;
      }
```

```
      if( primo == 1 )
        printf("%d é primo\n", n );
      else
        printf("%d não é primo\n", n );
    }
}
```

Solução 3: a mesma coisa que a solução anterior, mas utiliza um `break` para abandonar o ciclo.

```
#include <stdio.h>

main()
{
  int i, n;

  printf("Introduz um número: \n");
  scanf("%d", &n );
  if( n <= 1 )
    printf("%d não é primo\n", n );
  else
   {
     i = 2;
     while( i < n )
      {
        if( n % i == 0 )
          /* não é primo */
          break;
        i++;
      }
     if( i == n )
       printf("%d é primo\n", n );
     else
       printf("%d não é primo\n", n );
   }
}
```

Solução 4: para ver se um número *n* é primo, não é necessário fazer as divisões todas desde *2* até *n-1*. Basta fazer as divisões desde *2* até *sqrt(n)*. Porquê?

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

main()
{
  int i, n, raiz;

  printf("Introduz um número: \n");
  scanf("%d", &n );
  if( n <= 1 )
    printf("%d não é primo\n", n );
  else
   {
     raiz = (int) sqrt(n);
     i = 2;
     while( i <= raiz  )
      {
        if( n % i == 0 )
          /* não é primo */
          break;
        i++;
      }
     if( i > raiz )
       printf("%d é primo\n", n );
     else
       printf("%d não é primo\n", n );
   }
}
```

Programa 6

Faz um programa para calcular o factorial de um número.

Tenta fazer três versões deste programa, uma usando a instrução FOR, outra utilizando a instrução WHILE e uma terceira com a instrução DO... WHILE.

Versão 1. Com a instrução **for**:

```
#include <stdio.h>

main()
{
  int factorial,n,i;

  printf("Diz um numero: ");
  scanf("%d",&n);
  factorial=1;
  for (i=n; i>1; i--)
     factorial=factorial*i;
  printf("O factorial de %d e %d",n,factorial);
}
```

Versão 2. Com a instrução **while**:

```
#include <stdio.h>

main()
{
  int factorial,n,i;

  printf("Diz um numero: ");
  scanf("%d",&n);
  factorial=1;
  i=n;
  while (i>1)
  {
     factorial=factorial*i;
     i--;
  }
  printf("O factorial de %d e %d",n,factorial);
}
```

Versão 3. Com a instrução **do ... while**:

```
#include <stdio.h>

main()
{
  int factorial,n,i;

  printf("Diz um numero: ");
  scanf("%d",&n);
  factorial=1;
  i=n;
  do
  {
     factorial=factorial*i;
     i--;
  } while (i>1);
  printf("O factorial de %d e %d",n,factorial);
}
```

Programa 7

Faz um programa que calcula todos os divisores de um número.

Resolução

```
#include <stdio.h>

main()
{
  int n, i;

  printf("Introduz um número:\n");
  scanf("%d", &n );
  printf("Os divisores de %d são ", n );
  i = 1;
  while( i <= n )
   {
     if( n % i == 0 )
       printf("%d ", i );
     i++;
   }
  printf("\n");
}
```

Também podiam ter feito assim:

```
#include <stdio.h>

main()
{
  int n, i;

  printf("Introduz um número:\n");
  scanf("%d", &n );
  printf("Os divisores de %d são ", n );
  for( i = 1; i <= n; i++ )
    if( n % i == 0 )
      printf("%d ", i );
  printf("\n");
}
```

NOTA: Em ambos os casos, o ciclo pode ir só até `n/2`.

Programa 8

Faz um programa que escreve no ecrã os primeiros 20 números de Fibonacci.

Resolução

```
#include <stdio.h>

#define N  20

main()
{
  int i,
      fi,      /* F(i)   */
      fi_1,    /* F(i-1) */
      fi_2;    /* F(i-2) */

  fi_1 = 1;
  fi_2 = 1;
  printf("1\n");
  printf("1\n");
  for( i = 3; i <= N; i++ )
   {
     fi = fi_1 + fi_2;    /* F(i) = F(i-1) + F(i-2) */
     printf("%d\n", fi );
     fi_2 = fi_1;
     fi_1 = fi;
   }
```

```
    }
```

Programa 9

Faz um programa que pede um valor decimal inteiro positivo e o converte para binário.

Resolução

```
#include <stdio.h>

main()
{
   int dec, pot10;
   unsigned long nbin;

   printf("Introduza um valor decimal: ");
   scanf("%d",&dec);
   pot10=1;
   nbin=0;
   while (dec > 0)
      {
        nbin = nbin + pot10 * (dec % 2);
        dec = dec / 2;
        pot10 = pot10 * 10;
      }
   printf("O valor em binario: %ld\n",nbin);
}
```

Programa 10

Altera o programa do exercício 9 de forma a que converta o número de decimal para qualquer outra base.

Resolução

```
#include <stdio.h>

main()
{
   int dec, pot10, base;
   unsigned long nconv;

   printf("Introduza um valor decimal: ");
   scanf("%d",&dec);
   printf("Introduza a base de conversao: ");
   scanf("%d",&base);
   pot10=1;
   nconv=0;
   while (dec > 0)
      {
        nconv = nconv + pot10 * (dec % base);
        dec = dec / base;
        pot10 = pot10 * 10;
      }
   printf("O valor na base %d: %ld\n",base,nconv);
}
```

A resolução apresentada funciona correctamente apenas para bases de conversão menores ou iguais a 10.

Porque não funciona correctamente para bases superiores? Pensa sobre o assunto.

Programação Imperativa
Aula prática 6

Sumário

- Introdução à utilização de funções.
- Exercícios com funções.

---

1. Função ímpar

Faz uma função que verifica se um número é ímpar

---

2. - Função primo

Faz uma função que verifica se um número é primo (é só alterar o programa feito numa das aulas anteriores).

---

3. - Testa as funções dos exercícios 1 e 2

Utilizando as funções criadas nos exercícios anteriores faz um programa que vai pedindo números inteiros ao utilizador até que este introduza um número negativo. Para cada valor introduzido, o programa deve dizer se o número é par ou ímpar, e se é primo ou não.

---

4. - Os primeiros 100 primos

Faz um programa para calcular os 100 primeiros números primos. Utiliza a função criada no exercício 2.

---

5. - Desenha um quadrado

Faz um programa para desenhar um quadrado no ecrã. Esse quadrado deverá ser desenhado por uma função para a qual passas três parâmetros: caracter a utilizar, número de linhas e número de colunas.

Este exemplo do programa a ser executado ilustra aquilo que se pretende:

```
Introduz um caracter: z
Introduz o número de linhas: 4
Introduz o número de colunas: 6

zzzzzz
z    z
z    z
zzzzzz
```

---

6. - Factorial

Copia a função factorial que foi dada na aula teórica. Testa essa função fazendo um programa que calcule o factorial de 5.

---

7. - Função aleatório

Faz uma função que devolva números aleatórios inteiros. A função deverá receber um parâmetro **n** que indique a gama de valores pretendidos e, de cada vez que for chamada, devolver um numero aleatório inteiro no intervalo **[0 ... n-1]**.

Por exemplo, se a função for chamada com o parâmetro 6, deverá devolver um número aleatório compreendido entre 0 e 5 (inclusive).

---

8. Moeda ao ar

Utiliza a função que acabaste de criar no exercício anterior para fazer um programa que simule atirar uma moeda ao ar 1000 vezes. O programa deve contar quantas vezes sai "cara" e quantas vezes sai "coroa".

---

9. Mais números aleatórios

Faz um programa que gera 1000 números reais aleatórios no intervalo [0,1] O programa deve contar quantos números caem no intervalo [0.0, 0.5[, quantos caem no intervalo [0.5, 0.8[, e quantos caem no intervalo [0.8, 1.0]. (ver dicas no final da página).

Dicas para gerar números aleatórios:

A linguagem C tem as funções **rand**( ) e **random**( ) para gerar números inteiros pseudo-aleatórios. Estas funções geram os ditos números a partir de algoritmos mais ou menos elaborados e cuja implementação varia ligeiramente de compilador para compilador. A funcão **random**( ), mais recente e também considerada melhor que a **rand**( ), gera números aleatórios do tipo long int pertencentes ao intervalo [0 ... RAND_MAX].

O valor de RAND_MAX também pode ser diferente consoante a implementação (compilador/máquina). Aqui na sala de aulas práticas é igual a 2147483647 mas, para quem usa o Turbo C da Borland em casa, é igual a 32767.

Para gerar números aleatórios a pedido (por exemplo, apenas dentro de um determinado intervalo, ou números reais em vez de inteiros) temos que trabalhar com estes elementos realizando sobre eles algumas operações aritméticas simples.

rand( ), **random**() e RAND_MAX estão definidos em **stdlib.h**.

Programação Imperativa

Soluções aula prática 6

---

1. Função ímpar

```
int impar(int n)
{
  if ((n%2)==0)
    return(0);
  else
    return(1);
}
```

---

2. - Função primo

```
int primo(int n)
{
  int i;
  if (n<2)
    return(0);
  for (i=2; i<n; i++)
  {
    if ((n%i)==)
      return(0);
  }
  return(1);
}
```

3, 4. - Testa as funções dos exercícios 1 e 2

```
void main()
{
  int num, cnt;

  /* pedir o numero: */
  printf("Da um numero: ");
  scanf("%d", &num);

  /* vai mostrar se o numero e impar: */
  if (impar(num)==1)
    printf("%d e impar\n");
  else
    printf("%d e par\n");

  /* vai mostrar se o numero e primo: */
  if (primo(num)==1)
    printf("%d e primo\n");
  else
```

```
    printf("%d nao e primo\n");

  /* vai mostrar os primeiros 100 numeros primos */
  cnt = 0; /* contador de numeros primos ja encontrados */
  num = 1;
  while (cnt<100)
   {
     if (primo(num)==1)
       {
         cnt++;
         printf("%d ", num);
       }
     num++;
   }
}
```

---

5. - Desenha um quadrado

```
void rectangulo(char c, int lin, int col)
{
  int i, j;
  for (j=1; j<=col; j++)
    printf("%c", c);
  for (i=2; i<=lin-1; i++)
    {
      printf("%c", c);
      for (j=2; j<=col-1; j++)
        printf(" ");
      printf("%c", c);
    }
  for (j=1; j<=col; j++)
    printf("%c", c);
}
void main()
{
  char x;
  int nlin, ncol: int;
  printf("Introduz um caracter: ");
  scanf("%c", &x);
  printf("Introduz o numero de linhas: ");
  scanf("%c", &nlin);
  printf("Introduz o numero de colunas: ");
  scanf("%c", &ncol);
}
```

```
Introduz um caracter: z
Introduz o número de linhas: 4
Introduz o número de colunas: 6
zzzzzz
z    z
z    z
zzzzzz
```

---

6. - Factorial

```
int factorial(int n)
 /* returns n! Metodo com ciclo: */
{
  int i, result;

  result = 1;        /* initialize the variable */
  for (i=1; i<=n; i++)
    result = i*result;
  return(result);
}

int factorial (int n)
/* returns n! Metodo recursivo */
{
  if (n==1)
    return(1);
  else
    return(n*factorial(n-1));
}
```

---

7. - Função aleatório

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

int aleatorio(int m)
{
  /* a funcao rand() e uma funcao da biblioteca stdlib
     que retorna um int entre 0 e RAND_MAX; */
  return(rand()%m);
}
```

---

8. Moeda ao ar

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

int aleatorio(int m)
{
  /* a funcao rand() e uma funcao da biblioteca stdlib
     que retorna um int entre 0 e RAND_MAX; */
  return(rand()%m);
}

void main()
{
  int i, ncor;

  srand();  /* faz uma inicializacoa do gerador dos numeros aleatorios */
  ncor = 0;
  for (i=1; i<=1000; i++);
    if (aleatorio(2)==0)
      ncor++;
```

```
  printf("numero de vezes coroa: %d\", ncor);
  printf("numero de vezes cara: %d\", 1000-ncor);
}
```

---

9. Mais números aleatórios

```
float aleat(float xlo, float xhi)
/*************************************
 * gera numeros no intervalo [xlo, xhi] *
 *************************************/
{
   /* numeros entre 0 e 1: */
  x = (float) rand() / (float) RAND_MAX;
   /* converter para numeros entre xlo e xhi: */
  x = xlo + (xhi-xlo) * x;
  return(x);
}

void main()
{
  int i, n05=0, n58=0, n810=0;
  float r, rlo, rhi;


   for (i=0; i<1000; i++)
    {
      r = aleat(0.0, 1.0);
      if (r<0.5)
        n05++;
      else
        if (r<0.8)
          n58++;
        else
          n810++;
    }
  printf("No intervalo [0 - 0.5] cairam %d numeros\n", n05);
  printf("No intervalo [0.5 - 0.8] cairam %d numeros\n", n58);
  printf("No intervalo [0.8 - 1] cairam %d numeros\n", n810);
}
```

Programação Imperativa
Aula prática 6 (2004/05)

Sumário

- Introdução à utilização de funções.
- Exercícios com funções.

---

1. Função raiz quadrada

Faz uma função que devolve a raiz quadrada dum inteiro positivo.

---

2. - Função trigonometrica geral

Faz uma função que devolve o seno, coseno o tangente dum angulo x. A operação requerida é indicada por um segundo argumento passado a função.
(Utilizar as funções existentes nas bibliotecas de C para implementar a sua versão destas funções.)
Exemplo:
Indicar operação: seno
indicar ángulo: 2
Resposta: 0.909297

---

3. - Testa as funções dos exercícios 1 e 2

Utilizando as funções criadas nos exercícios (1) e (2) faz um programa que vai pedindo números e operações ao utilizador e respondendo conforme.
Portanto, para cada número e operação indicada, o programa deve imprimir a raiz quadrada do número e a seguir a operação trigonométrica solicitada.
O programa deve terminar quando o utilizador introduze o número -1.

---

4. - Calcular o seno duma sequência.

Fazer um programa que calcula e imprime todos os senos para os angulos comprendidos entre -90º e 90º graus utilizando a função criada por você em (2). (permitir até cinco decimais nos resultados)

---

5. - Desenhar um rectângulo

Faz um programa para desenhar um rectangulo no ecrã. O desenho deve ser encomendado a uma função que recebe três parâmetros: caracter a utilizar para o desenho, comprimento e altura do rectângulo medido em números de caracteres impressos..

Exemplo:

```
Caracter a utilizar no desenho: *
largura:     4
comprimento: 6

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

O programa deve sempre verificar que o input seja o desejado: a altura deve ser menor que o comprimento. Portanto, deve pedir os números até que estes estejam certos.

---

6. - Função factorial

Criar uma função que calcula o factorial de um número. Utilizar a dita função para dividir o factorial de dois números, nos quais o primeiro será o numerador e o segundo o denominador.

---

7. Função permutação

A permutação é um arranjo de elementos obtidos dum conjunto finito. a função de permutação P(n,k) fornece o número de permutações diferentes de quaisquer k itens retirados dum conjunto de n itens. Uma maneira de calcular essa função é pela fórmula:

P(n,k) = n! / (n - k)! Por exemplo: P(5,2) = 5! / ((5 - 2)! = 5!/3! = 120/6 = 20.

Portanto, há 20 permutações diferentes de dois itens retirados dum conjunto de 5.
Para implementar esta função, que chamaremos perm() utilize a função factorial desenvolvida em (6).
Também não esqueça de verificar que k não pode ser maior que n e que nenhum destes pode ser menor que 0.

Programação 1

Aula prática 6

Sumário

- Introdução à utilização de funções. Noção de função recursiva.
- Exercícios com funções.

Exercício 1 - Função ímpar

Faz uma função que verifica se um número é ímpar

Exercício 2 - Função primo

Faz uma função que verifica se um número é primo (é só alterar o programa feito numa das aulas anteriores).

Programa 1 - Testa as funções dos exercícios 1 e 2

Utilizando as funções criadas nos exercícios anteriores faz um programa que vai pedindo números inteiros ao utilizador até que este introduza um número negativo. Para cada valor introduzido, o programa deve dizer se o número é par ou ímpar, e se é primo ou não.

Programa 2 - Os primeiros 100 primos

Faz um programa para calcular os 100 primeiros números primos. Utiliza a função criada no exercício 2.

Programa 3 - Desenha um quadrado

Faz um programa para desenhar um quadrado no ecrã. Esse quadrado deverá ser desenhado por uma função para a qual passas três parâmetros: caracter a utilizar, número de linhas e número de colunas.

Este exemplo do programa a ser executado ilustra aquilo que se pretende:

```
Introduz um caracter: z
Introduz o número de linhas: 4
Introduz o número de colunas: 6

zzzzzz
z    z
z    z
zzzzzz
```

Programa 4 - Factorial

Copia a função factorial que foi dada na aula teórica. Testa essa função fazendo um programa que calcule o factorial de 5.

Programa 5 - Factorial (versão recursiva)

Substitui a função factorial do programa anterior pela sua definição recursiva e volta a testar o programa.

Exercício 3 - Função aleatório

Faz uma função que devolva números aleatórios inteiros. A função deverá receber um parâmetro **n** que indique a gama de valores pretendidos e, de cada vez que for chamada, devolver um numero aleatório inteiro no intervalo **[0 ... n-1]**.

Por exemplo, se a função for chamada com o parâmetro 6, deverá devolver um número aleatório compreendido entre 0 e 5 (inclusivé).

Programa 6 - Moeda ao ar

Utiliza a função que acabaste de criar no exercício anterior para fazer um programa que simule atirar uma moeda ao ar 1000 vezes. O programa deve contar quantas vezes sai "cara" e quantas vezes sai "coroa".

Programa 7 - Mais números aleatórios

Faz um programa que gera 1000 números reais aleatórios no intervalo [0,1] O programa deve contar quantos números caem no intervalo [0.0, 0.5[, quantos caem no intervalo [0.5, 0.8[, e quantos caem no intervalo [0.8, 1.0]. (ver dicas no final da página).

Dicas para gerar números aleatórios:

A linguagem C tem as funções **rand**( ) e **random**( ) para gerar números inteiros pseudo-aleatórios. Estas funções geram os ditos números a partir de algoritmos mais ou menos elaborados e cuja implementação varia ligeiramente de compilador para compilador. A funcão **random**( ), mais recente e também considerada melhor que a **rand**( ), gera números aleatórios do tipo long int pertencentes ao intervalo [0 ... RAND_MAX].

O valor de RAND_MAX também pode ser diferente consoante a implementação (compilador/máquina). Aqui na sala de aulas práticas é igual a 2147483647 mas, para quem usa o Turbo C em casa, é igual a 32767.

Para gerar números aleatórios a pedido (por exemplo, apenas dentro de um determinado intervalo, ou números reais em vez de inteiros) temos que trabalhar com estes elementos realizando sobre eles algumas operações aritméticas simples.

rand( ), **random**() e RAND_MAX estão definidos em **stdlib.h**.

Programação I

Soluções dos exercícios da aula prática 6

Exercício 1

```
/************************************
 * impar -- Diz se um número é ímpar
 *
 * Parâmetros
 * numero -- um número inteiro
 *
 * Devolve
 * 1 -- se numero é ímpar
 * 0 -- se numero é par
 ************************************/
int impar (int numero)
{
   if (numero % 2 == 0)
     return (0);
   else
     return (1);
}
```

Exercício 2

```
/************************************
 * primo -- Diz se um número é primo
 *
 * Parâmetros
 * numero -- um número inteiro
 *
 * Devolve
 * 1 -- se numero é primo
 * 0 -- se numero não é primo
 ************************************/
int primo (int numero)
{
   int i;  /* divisores sucessivos */

   if ( numero < 2 )
     return (0);
   for (i = 2; i <= numero - 1; i++)
   {
     if ( numero % i  ==  0)
       return (0);
   }
   return (1);
 }
```

Programa 1

```
/************************************************************
 * a6_p1 -- Este programa diz se um determinado numero e' primo
 *      ou não, e se é par ou ímpar.
 *
 * Input : Um número inteiro
 * Output: Mensagem - E' ou NAO E' primo; E' PAR ou IMPAR
 ************************************************************/
#include <stdio.h>

#define TRUE   1
```

```
#define FALSE  0

/************************************
 * impar -- Diz se um número é ímpar
 *
 * Parâmetros
 * n -- um número inteiro
 *
 * Devolve
 * TRUE -- se numero é ímpar
 * FALSE -- se numero é par
 ************************************/
 int impar (int n)
 {
   if( n % 2 == 0)
     return (FALSE);
   else
     return (TRUE);
 }


/************************************
 * primo -- Diz se um número é primo
 *
 * Parâmetros
 * n -- um número inteiro
 *
 * Devolve
 * TRUE -- se numero é primo
 * FLASE -- se numero não é primo
 ************************************/
 int primo (int n)
 {
   int i;  /* divisores intermedios sucessivos */

   if ( numero < 2 )
     return (FALSE);
   for (i = 2; i <= n - 1; i++)
   {
     if ( n % i  ==  0)
       return (FALSE);
   }
   return (TRUE);
 }

void main ()
{
  int numero;

  do
  {
    printf("Introduz um numero inteiro (termina com -1): ");
    scanf( "%d", &numero );

    if (numero != -1)
    {

      if ( impar(numero) == TRUE )
        printf("O numero %d e' impar ", numero);
      else
        printf("O numero %d e' par ", numero);

      if ( primo(numero) == TRUE )
        printf("e e' primo\n\n");
      else
        printf("e nao e' primo\n\n");

    }
  }
  while (numero != -1);
}
```

Programa 2

```
/*************************************************************
 * a6_p2 -- Este programa mostra os primeiros 100 numeros primos
 *
 * Input :
 * Output: Os 100 primeiros primos
 *************************************************************/
#include <stdio.h>

#define TRUE   1
#define FALSE  0

/***********************************
 * primo -- Diz se um número é primo
 *
 * Parâmetros
 * n -- um número inteiro
 *
 * Devolve
 * TRUE -- se numero é primo
 * FALSE -- se numero não é primo
 ***********************************/
 int primo (int n)
 {
   int i;  /* divisores intermedios sucessivos */

   if ( numero < 2 )
     return (FALSE);
   for (i = 2; i <= n - 1; i++)
   {
     if ( n % i  ==  0)
       return (FALSE);
   }
   return (TRUE);
 }



void main ()
{
  int numero, conta_primos;

  numero = 1;
  conta_primos = 1;
  while (conta_primos <= 100)
  {
    if ( primo(numero) == TRUE )
    {
      printf("%03d\t", numero);
      conta_primos++;
    }
    numero++;
  }
}
```

Programa 3

```
/*
 * a6_p3: Desenha um rectangulo
 *
 * Input : Um caracter, num de linhas e num de colunas
 * Output: Rectangulo pedido
 *
 */

#include <stdio.h>

/* Desenha rectangulo a pedido */
void rectangulo(char c, int n_lin, int n_col)
```

```
{
  int i,j;

  for( i=1; i<=n_lin; i++ )
    for( j=1; j<=n_col; j++ )
    {
      if( i==1 || i==n_lin || j==1 || j==n_col )
        printf("%c", c);
      else
        printf(" ");
      if( j==n_col )
        printf("\n");
    }
}

main()
{
  char caracter;
  int n_linhas, n_colunas;

  printf("Introduz um caracter:\n");
  scanf("%c", &caracter );
  printf("Introduz o numero de linhas:\n");
  scanf("%d", &n_linhas );
  printf("Introduz o numero de colunas:\n");
  scanf("%d", &n_colunas );
  rectangulo(caracter, n_linhas, n_colunas);
}
```

Programa 4

```
/*
 * a6_p4: Calcula o factorial de 5
 *
 * Input :
 * Output: factorial de 5
 *
 */

#include <stdio.h>

 /* função iterativa para calcular o factorial */
 int factorial( int n )
 {
     int i,p;

     p = 1;
     for( i=2; i<=n; i++ )
         p = p * i;
     return p;
 }

void main()
{
    printf("O factorial de 5 e' %d\n", factorial(5));
}
```

Programa 5

```
/*
 * a6_p5: Calcula o factorial de 5
 *
 * Input :
 * Output: factorial de 5
 *
 */

#include <stdio.h>
```

```
 /* função recursiva para calcular o factorial */
 int factorial( int n )
 {
     if (n == 0)
         return(1);
     /* else */
     return ( n * factorial(n-1) );
 }

void main()
{
    printf("O factorial de 5 e' %d\n", factorial(5));
}
```

Programa 6

```
/*
 * Lanca moeda ao ar 1000 vezes
 *
 * Input:
 * Output: Quantas vezes saiu 'cara' e quantas saiu 'coroa'
 */

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

#define CARA    0
#define COROA   1

/* Prepara a geracao de numeros aleatorios */
void randomize()
{
  time_t agora;

  agora = time(0); /* tb podia ser  time(&agora); */
  srandom(agora);
}

/* devolve um inteiro aleatorio no intervalo 0..n-1 (inclusive) */
int aleatorio( int n )
{
  int numero;

  numero = (random() % n);
  return( numero );
}

/* Lanca moeda ao ar 1 vez */
int LancaMoeda (void)
{
  if (aleatorio(2) == 0)
    return CARA;
  else
    return COROA;
}

/*  Lanca moedas ao ar varias vezes */
int main()
{
  int resultado, n_caras, n_coroas, i;

  n_caras = 0; n_coroas = 0;
  randomize();
  for ( i = 0; i < 1000; i++ )
    {
      resultado = LancaMoeda();
      if ( resultado == CARA )
        n_caras++;
      else
        n_coroas++;
```

```
    }
  printf("Sairam %d caras e %d coroas\n", n_caras, n_coroas);
  return 0;
}
```

Programa 7

```
/*
 *  a7_p7: Gera 1000 números reais aleatórios entre 0 e 1
 *
 *  Input:
 *  Output: Diz quantos numeros foram gerados por intervalo
 *
 */

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

/* Prepara a geracao de numeros aleatorios */
void randomize()
{
  time_t agora;

  agora = time(0);
  srandom(agora);
}

/* Gera um numero aleatório no intervalo entre 0 e 1 */
 double aleatorio01 (void)
 {
     return (double) rand() / (double) RAND_MAX;
 }

void main()
{
    int i, cont1, cont2, cont3;
    double numero;


    cont1 = 0;
    cont2 = 0;
    cont3 = 0;

    randomize();
    for (i = 0; i < 1000; i++)
    {
        numero = aleatorio01();
        if (numero >= 0.0 && numero < 0.5)
            cont1++;
        else if (numero >= 0.5 && numero < 0.8)
            cont2++;
        else if (numero >= 0.8 && numero < 1.0)
            cont3++;
    }
    printf("No intervalo [0.0, 0.5[ cairam %d numeros\n", cont1);
    printf("No intervalo [0.5, 0.8[ cairam %d numeros\n", cont2);
    printf("No intervalo [0.8, 1.0[ cairam %d numeros\n", cont3);
}
```

Programação Imperativa
Aula Prática 7

Sumário

- Programas com utilização de arrays.
- Programas com ponteiros.

1.

Escreve um programa que atribuí números aleatórios a um array de duas dimensões de 5 por 4 elementos e que apresenta esses números no ecrã por colunas.

2.

Altera o programa anterior de forma a que apresente também o maior e o menor dos números aleatórios.

3.

Escreve um programa que ordena uma lista de N números. N pode ir no máximo até 100.

4.

Escreve um programa que registe o sexo e a idade de um grupo de dez pessoas. Em seguida o programa deve apresentar a média das idades de cada sexo.

5.

O programa a seguir usa a técnica de **passagem por valor**. Muda o programa de forma a usar a técnica de **passagem por referência**.
Qual será o output do programa antes e após esta mudançã?

código original: código modificado:

```
#include <stdio.h>
int twotimesx(int x)
{
  x = 2*x;
  return(x);
}

void main()
{
  int a, b;
```

```
  a = 10.0;
  b = twotimesx(a);
  printf("%a", a);
}
```

(aula relevante: 15)

6.

Escreve uma função que recebe como parâmetros a largura e comprimento de um rectângulo. A função deve calcular e devolver os seguintes pontos

1: a área do rectângulo (lado vezes lado).
2: o perímetro (somar os quatro lados).

A função não pode usar variáveis globais!
(Nota: porque a informação a retornar são **dois** valores, temos de usar ponteiros (passagem por referência). Veja aulas 15 e 14.)

7.

```
void double10(double *dp)
{
  *dp = 10.0;
}

void main()
{
  float y=10.0;
  float x=10.0;

  double10((double *) &x);
  printf("%f %f\n", x, y);
}
```

O programa acima não corre bem (o output não será `10.0 10.0`). Experimente e elimine o erro.
(aula relevante: 15)

8.

Na aula prática 3, o exercício 8 consistia no seguinte: Faz um programa que receba 10 valores inteiros e que indique: Quantos são pares, quantos são ímpares, quantos são positivos e quantos são negativos. Volta a escrever o programa, utilizando **arrays**.

9.

Volta a fazer o exercício de contar 0s e 1s de um byte binário, com arrays.

10.

Volta a fazer o exercício de converter um número decimal para binário, com arrays.

11.

Faz um programa que apresente uma tabela de conversão de decimal para binário:

```
0       0
1       1
2      01
3      11
4     100
...   ...
```

12.

Now safe your work and crash the computer with the following program:

```
#include <stdlib.h>
void main()
{
  int i;
  int *p;

  for (i=0; i<=10000; i++)
   {
     p = (int *) rand();
     *p = 0;
   }
}
```

(Running this program is at your own risk. It will crash the computer and in some cases it can destroy the operating system and in extreme cases even the hardware)

Programação Imperativa
Aula Prática 7- Soluções
(as seguintes, são UMA solução possível a estes exercicios)

---

1 e 2. Escreve um programa que atribuí números aleatórios a um array de duas dimensões de 5 por 4 elementos e que apresenta esses números no ecrã por colunas. No fim apresenta também o maior e o menor dos números aleatórios geridos.

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define COLS 4
#define LINHAS 5

long int a1[LINHAS][COLS];
long int min, max;

void carrega_array_um_indice(int linhas, int cols)
{
  for (int j=0; j < linhas; j++)
    for (int i=0; i < cols; i++)
    {
      a1[j][i] = random(7011) + (random(3023) / 3);
      if (a1[j][i] < min)
        min = a1[j][i];
      if (a1[j][i] > max)
        max = a1[j][i];
    }
}

void imprime_array(int lin, int col)
{
  printf("\nconteudo do array: \n");
  for (int j=0; j < lin; j++)
  {
    printf("\n");
    for (int i=0; i < col; i++)
      printf("%10li  ", a1[j][i]);
  }
  printf("\n\nValores min = %li, e max = %li", min, max);
}

/*---------------------------------------------*/
void main()
{
  int min = 999999L; max = -1;
  randomize();
  carrega_array_um_indice(LINHAS, COLS);
  imprime_array(LINHAS, COLS);
}
```

---

3. Escreve um programa que ordena uma lista de N números. N pode ir no máximo até 100.

```
/* TP7 exercicio 3
 ordenar numeros de menor a maior num array estatico de 1 dimensao
 utiliza metodo de ordenacao da burbulha ("bubble") 'esperto' (porque?) */
#include <stdio.h>
#include <stdlib.h>

#define TAM   100
int a[TAM];    /*definimos array de 1 dimensão */
   /* variavel global */

void inicializa_array(const int tam)
{
  for (int j=0; j < tam; j++)
    a[j] = 0;
}

void carrega_array(const int t)
{
  srand(1);
  for (int j=0; j < t; j++)
  if (j % 2)
    a[j] = random(20001/3) + (random(1103) / (random(57) +1));
  else
    a[j] = random(1001/2) + (random(1703) / (random(23) + 1) );
}

void ordena_array(const int N)  /*método de burbulha */
{
  int k, i;

  for (k = 0; k < N; k++)         /*controla ciclo de tudas as passadas */
  {
    for (i = 0; i < (N-k-1); i++) /*controla uma passada de comparações */
    {
      if (a[i] > a[i+1])          /*termina nos elementos ja ordenados*/
      trocar(&a[i], &a[i+1]);
    }
  }
}

void trocar(int *primeiro, int *segundo)
   /*troca primeiro pelo segundo*/
   /*passagem de parâmetros por referência */
{
  int temp;
  temp      = *segundo;
  *segundo  = *primeiro;
  *primeiro = temp;
}

void imprime_array(const int t, char vez)
{
  if (vez == '1')
    printf("\n\nconteudo do array ANTES de ser ordenado: \n");
  else
    printf("\n\nconteudo do array DEPOIS de ser ordenado: \n");
  for (int j=0; j < t; j++)
```

```
  {
    if (!(j % 8)) printf("\n");
      printf("%7i  ", a[j]);
  }
}

void main()
{
  inicializa_array(TAM);
  carrega_array(TAM);
  imprime_array(TAM, '1');
  ordena_array(TAM);
  imprime_array(TAM, '2');
}
```

---

4. Escreve um programa que registe o sexo e a idade de um grupo de dez pessoas. Em seguida o programa deve apresentar a média das idades de cada sexo.

```
/* TP7, N§ 4.
 array estatico para sexo e idade de 10 pessoas, apresentando a media
 este pgm demostra como podemos armazenar um tipo de dado na B de D e
 amostrar ao utilizador os dados numa outra forma.*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define LINHAS  10
#define COLS    2   /* col 1: idade, col 2: sexo


int pessoas[LINHAS][COLS];  /*definimos 1 so array para idades e sexo    */
int NUM, total;


void limpar_arrays(int linhas, int cols)
{
  for (int j=0; j < linhas; j++)
    for (int i=0; i < cols; i++)
      pessoas[j][i] = 0;   /*exemplo de limpar um char de vez */
  NUM = 0; total = 0;   /*inicializar valores e sempre muito importante!! */
}

void carrega_arrays(int linhas)
{
  char sexc;
  printf("\nVamos carregar a idade e sexo de 10 pessoas:");
  for (int j = 0; j < linhas; ++j)
  {
    printf("\nSexo('F'/'M'): ");
    sexc = toupper(getche());
    pessoas[j][1] = armazena_sexo(sexc);
    printf(" Idade: ");
    scanf(" %i", &pessoas[j][0]);
    total = (float) total + pessoas[j][0];
    NUM++;
```

```
  }
}

int armazena_sexo(char letra_sexo)
  /*converte de letra para numero */
{
  return (letra_sexo == 'F');   /*devolve 1 para F e 0 para M */
}

void imprime_arrays(int lin)
{
    /* declaracao de uma funcao local: */
  char cvt(int tipo)    /*desconverte de numeros para letras */
  {
    if (tipo == 1)
      return 'F';
    else
      return 'M';
  }

  printf("\n\n  Listagem de pessoas indicando sexo e idade: ");
  for (int j=0; j < lin; j++)
    printf("\n%d)Idade:%i, sexo: %c",j+1, pessoas[j][0], cvt(pessoas[j][1]));
    /*agora imprimimos a media das idades: */
  printf("\nMedia das idades= %.2f", (float)(total / NUM));
}

void main()
{
  limpar_arrays(LINHAS,COLS);
  carrega_arrays(LINHAS);
  imprime_arrays(LINHAS);
}
```

---

5. Técnica de **passagem por referência**.

Qual será o output do programa antes e após esta mudançã?
versão com parâmetros por referência:

```
#include <stdio.h>

int quadrado(int *x)
{
  *x = 2 * *x;
  return(*x);
}

void main()
{
  int a, b;

  a = 10.0;
  b = quadrado(&a);
  printf("%a", a);
}
```

---

6. Escreve uma função que recebe como parâmetros a largura e comprimento de um rectângulo. A função deve calcular e devolver os seguintes pontos

1: a área do rectângulo (lado vezes lado).
2: o perímetro (somar os quatro lados).

A função não pode usar variáveis globais!
(Nota: porque a informação a retornar são **dois** valores, temos de usar ponteiros (passagem por referência). Veja aulas 15 e 14.)

```
#include ....

void area_e_peri(float x, float y, float *area, float *peri)
{
  *area = x * y;
  *peri = 2*(x + y);
}

void main()
{
  float a, b;
  float ar, pe;

  printf("Largo:  ");
  scanf("%f", &a);
  printf("Altura:  ");
  scanf("%f", &b);
  area_e_peri(a, b, &ar, &pe);
  printf("\nArea = %.2f, Perimetro = %.2f", ar, pe);
}
```

7.

```
void double10(double *dp)
{
  *dp = 10.0;
}

void main()
{
  float y=10.0;
  float x=10.0;

  double10((double *) &x);
  printf("%f %f\n", x, y);
}
```

O programa acima não corre bem (o output não será 10.0 10.0). Experimente e elimine o erro.
(aula relevante: 15)
SOLUÇÃO: Este pgm. está a misturar os tipos. Na função o parâmetro está definido como 'double', mais a chamada a função está a ser feita com um 'float'. Portanto, utilizar o mesmo tipo nas duas.

---

8. Na aula prática 3, o exercício 8 consistia no seguinte: Faz um programa que recebe 10 valores inteiros e que indica: Quantos são pares, quantos são ímpares, quantos são positivos e quantos são

negativos. Volta a escrever o programa, utilizando **arrays**.

```
/*programa para calcular numeros pares, impares, pos e neg num array.
  a rotina mostra tb. como passamos por referência um array de inteiros. */
#include <stdio.h>
#include <math.h>
#define MAX 5

int Pares(const int N)
  /*devolve 1 se N e par */
{
  if (N == 0)
    return 0;
  else
    return ((N % 2) == 0);
}

int Positivos(const int n)
  /*devolve 1 se n e positivo */
{
  return (n >= 0);
}

void imprime_resultados(int par, int pos, int max, int *nums)
{
  printf("\nOs números armazenados são:\n");
  for (int i=0; i < max; i++)
    printf("%d  ", nums[i]);

  printf("\nEm %d numeros, %d sao par(es), %d impar(e)s, %d positivo(s)\
    %d negativo(s)", max, par, (max-par), pos, (max-pos));
}

void main()
{
  int par = 0, positivo = 0;
  int numeros[MAX];         /*array para guardar os 10 numeros */

  printf("\nPrograma para calcular numeros pares, impares\
     positivos e negativos en 10 números\n");
  for (int i = 0; i < MAX; i++)
  {
    printf("%d) = :", i+1); scanf("%i", &numeros[i]);
    par += Pares(numeros[i]);
    positivo += Positivos(numeros[i]);
  }
  imprime_resultados(par, positivo, MAX, &numeros[0]);
}
```

---

9. Volta a fazer o exercício de contar 0s e 1s de um byte binário, com arrays.

```
/*este programa converte um decimal inteiro num array
  para qualquer outra base */
#include <stdio.h>

void conversao(const int max, int N, char *out)
    /*funcao para convertir o inteiro em binário num array.
```

```
      A passagem do array unidimensional e com um ponteiro */
{
  char modulo(int N)   /*funcao local: se nao há resto, devolve 0; senão 1 */
  {
    if ((N % 2) == 0)
      return '0';
    else
      return '1';
  }
  int n = N, indice = max-1;

  while (n >= 2)             /*enquanto o numero não estiver convertido */
   {out[indice] = modulo(n);  /*obtem o modulo (n % 2) e guarda no array */
    n = (n / 2);          /*tira do num o último dígito pela direita */
    indice--;            /*muda o indice para apontar pxmo esq. no array */
   }
 out[indice] = modulo(n);  /*o último n pode ser um 1!! */
}

void print_numero(const int max, char *e)
{
  for (int x = 0; x < max; x++)
    printf("%c", e[x]);
}

void limpar_numeros(int maximo, int *N, char *array)
{
  for (int x = 0; x < maximo; x++)
    array[x] = '0';
  *N = 0;
}

void main()
{
  int numero;                /* aqui vamos guardar o input */
  char binario[8];           /* output estara num array */
  char r = 's';
  char *ptr = &binario[0];   /*ptr aponta ao inicio do array de output */

  while (r != 'n')
  {
    limpar_numeros(8, &numero, ptr);
    printf("\nNúmero decimal a convertir: ");   scanf("%d",&numero);
    conversao(max, numero, ptr);
    printf("\nnumero: %u na base %d :",  numero, 2);
    print_numero(max, ptr);
    printf("\n\nDeseja continuar ((s/n): "); r = getche();
  }
}
```

---

11. Fazer um programa que apresente uma tabela de conversão de decimal para binário:

```
/*este programa apresenta uma tabela de decimais e o correspondente
  numero binario */
#include <stdio.h>
```

```
char modulo(int N)   /*se nao ha resto, devolve 0; senao 1 */
{   if ((N % 2) == 0)
    return '0';
    else return '1';
}

void print_numero(const int max, char *e)
{
  for (int x = 0; x < max; x++)
    printf("%c", e[x]);
}

void limpar_numeros(int maximo, int *N, char *array)
{
  for (int x = 0; x < maximo; x++)  array[x] = '0';
  *N = 0;
}

void control_de_linha(int n) /* controla salto de linhas no ecrÆ */
{
  if ((((n+1) % 5) == 0) && (n > 0))
    printf("\n");
  else printf(" ");
}

void conversao(const int max, int N, char *out)
    /*funcao para convertir o inteiro em binario num array.
      A passagem do array unidimensional e com um ponteiro */
{
  char modulo(int N);          /*funcao local a esta funcao. */
  int n = N, indice = max-1;

  while (n >= 2)               /*enquanto o numero nao esta convertido */
  {
    out[indice] = modulo(n);   /*obtem o modulo (n % 2) e guarda no array */
    n = (n / 2);               /*tira do num o ultimo digito pela direita */
    indice--;                  /*muda o indice para apontar pxmo esq no array */
  }
 out[indice] = modulo(n);      /* o ultimo n pode nao ser maior que 2 */
}

void main()
{
  int numero;                  /* aqui vamos guardar o input */
  char binario[max];           /* output estara num array */
  char *ptr = &binario[0];     /*ptr aponta ao inicio do array de output */

  for (int num=0; num <= 119; num++)
  {
    limpar_numeros(max, &numero, ptr);
    conversao(max, num, ptr);
    printf("%3d= ",num);
    print_numero(max, ptr);
    control_de_linha(num); /*cada 5 cols saltamos uma linha */
  }
}
```

12. Agora guardar o vosso trabalho. O seguinte programa destroi o Operating System.

```
#include <stdlib.h>
void main()
{
  int i;
  int *p;

  for (i=0; i<=10000; i++)
   {
     p = (int *) rand();
     *p = 0;
   }
}
```

(**Você vai executar este programa da sua propria responsabilidade.** Porque o computador vai congelarse. Pode destruir o sistema operativo, e por acaso, o próprio hardware.
MELHOR NÃO FAZER MOÇOS!!

Programação I

Aula Prática 7

Sumário

Programas com utilização de arrays.

Programa 1

Escreve um programa que atribuí números aleatórios a um array de duas dimensões de 5 por 4 elementos e que apresenta esses números no ecrã por colunas.

Programa 2

Altera o programa anterior de forma a que apresente também o maior e o menor dos números aleatórios.

Programa 3

Escreve um programa que ordena uma lista de N números. N pode ir no máximo até 100.

Programa 4

Escreve um programa que registe o sexo e a idade de um grupo de dez pessoas. Em seguida o programa deve apresentar a média das idades de cada sexo.

Programa 5

Na aula prática 3, o exercício 8 consistia no seguinte: Faz um programa que receba 10 valores inteiros e que indique: Quantos são pares, quantos são ímpares, quantos são positivos e quantos são negativos. Volta a escrever o programa, utilizando arrays.

Programa 6

Volta a fazer o exercício de contar 0s e 1s de um byte binário, com arrays.

Programa 7

Volta a fazer o exercício de converter um número decimal para binário, com arrays.

Programa 8

Faz um programa que apresente uma tabela de conversão de decimal para binário:

```
0        0
1        1
2       01
3       11
4      100
...    ...
```

Programação I

Soluções dos exercícios da aula prática 7

Programa 1

Escreve um programa que atribuí números aleatórios a um array de duas dimensões de 5 por 4 elementos e que apresente esses números no ecrã por colunas.

Resolução:

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

void randomize()
{
  time_t agora;

  agora = time(0);
  srandom(agora);
}

void main()
{
    /* Declara o array */
    int matriz[5][4];
    int a, b;

    /* Preenche o array */
    randomize();
    for( a = 0; a < 5; a++ )
      for ( b = 0; b < 4; b++ )
        matriz[a][b] = random();

    /* Imprime os elementos do array */
    for ( a = 0; a < 5; a++ )
     {
       for ( b = 0; b < 4; b++ )
         printf("%d\t", matriz[a][b]);
       printf("\n");    /* muda de linha */
     }
}
```

Também se pode mostrar os elementos do array à medida que vão sendo gerados. Assim:

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

void randomize()
{
  time_t agora;

  agora = time(0);
  srandom(agora);
}

main()
{
  int a[5][4];
  int i,j;

  randomize();
```

```
  for (i=0; i<5; i++)
  {
    for (j=0; j<4; j++)
    {
       a[i][j] = random();
       printf("%d\t",a[i][j]);
    }
    printf("\n");
  }
}
```

Programa 2

Altera o programa anterior de forma a que apresente também o maior e o menor dos números aleatórios.

Resolução:

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

void randomize()
{
  time_t agora;

  agora = time(0);
  srandom(agora);
}

main()
{
  int a[5][4];
  int i,j,max=0,min=RAND_MAX;

  randomize();
  for (i=0; i<5; i++)
  {
    for (j=0; j<4; j++)
    {
       a[i][j] = random();
       printf("%d\t",a[i][j]);
       if (max < a[i][j])
         max = a[i][j];
       if (min > a[i][j])
         min = a[i][j];
    }
    printf("\n");
  }
  printf("Maximo: %d\n",max);
  printf("Minimo: %d\n",min);
}
```

Programa 3

Escreve um programa que ordena uma lista de N números. N pode ir no máximo até 100.

Resolução:

```
#include <stdio.h>

#define N 100

main()
{
  int a[N];
  int i, k, m, min, temp, tam;

  /* Receber os valores a ordenar */
```

```
  printf("Indique o tamanho do array (maximo 100)\n");
  scanf("%d",&tam);
  for (i=0; i<tam; i++)
   {
     printf("%d.° numero -> ",i+1);
     scanf("%d",&a[i]);
   }

  /* Ordenar o array */
  for( k=0; k<=tam-1; k++ )
   {
     /* descobre o indice do minimo em a[k], a[k+1], ..., a[N-1] */
     min = a[k];
     m = k;
     for( i=k; i<=tam-1; i++ )
       if( a[i] < min )
        {
          min = a[i];
          m = i;
        }

     /* troca a[k] com a[m] */
     temp = a[k];
     a[k] = a[m];
     a[m] = temp;
   }

  /* Escrever os elementos do array ordenados */
  for( i=0; i<tam; i++ )
    printf("%d ", a[i]);
  printf("\n");
}
```

Também se poderia obter os valores a ordenar de outras formas. Por exemplo, através da geração de números aleatórios, ou por atribuição directa.

Programa 4

Escreve um programa que registe o sexo e a idade de um grupo de dez pessoas. Em seguida o programa deve apresentar a média das idades de cada sexo.

Resolução:

```
#include <stdio.h>

#define N 10

main()
{
  int dados[N][2];
  int i, ms=0, fs=0, idade_m=0, idade_f=0;

  /* Registo dos valores */
  for (i=0; i<N; i++)
  {
    printf("Indique os dados da %d.ª pessoa\n",i+1);
    printf("Sexo (0 - M, 1 - F): ");
    do
    {
      scanf("%d",&dados[i][0]);
    } while (dados[i][0] > 1 || dados[i][0] < 0);
    printf("Idade: ");scanf("%d",&dados[i][1]);
  }

  /* Calculo da media das idades por sexo */
  for (i=0; i<N; i++)
  {
    if (dados[i][0] == 0)
```

```
    {
      idade_m = idade_m + dados[i][1];
      ms++;
    }
    else
    {
      idade_f = idade_f + dados[i][1];
      fs++;
    }
  }

  /* Mostra dos resultados */
  printf("Media das idades\n");
  printf("Homens: %.2f\n", (float)idade_m/ms);
  printf("Mulheres: %.2f\n", (float)idade_f/fs);
}
```

Programa 5

Na aula prática 3, o exercício 8 consistia no seguinte: Faz um programa que receba 10 valores inteiros e que indique: Quantos são pares, quantos são ímpares, quantos são positivos e quantos são negativos. Volta a escrever o programa, utilizando arrays.

Resolução:

```
#include <stdio.h>

#define N 10

main()
{
  int n[N];
  int i, pares=0, impares=0, positivos=0, negativos=0;

  printf("Introduza %d numeros inteiros:\n",N);
  for (i=0; i<N; i++)
    scanf("%d",&n[i]);

  for (i=0; i<N; i++)
  {
    if (n[i] % 2 == 0)
      pares++;
    else
      impares++;
    if (n[i] >= 0)
      positivos++;
    else
      negativos++;
  }

  printf("Resultado final\n");
  printf("Pares -> %d\n",pares);
  printf("Impares -> %d\n",impares);
  printf("Positivos -> %d\n",positivos);
  printf("Negativos -> %d\n",negativos);
}
```

Programa 6

Volta a fazer o exercício de contar 0s e 1s de um byte binário, com arrays.

Resolução:

```
#include <stdio.h>

#define N 8
```

```
main()
{
  int byte[N];
  int i, zeros=0, uns=0;

  printf("Introduza um BYTE binario\n");
  for (i=0; i<N; i++)
    scanf("%d",&byte[i]);

  for (i=0; i<N; i++)
  {
    if (byte[i]==0)
      zeros++;
    else if (byte[i]==1)
      uns++;
    else
      {
        printf("O numero nao e binario\n");
        exit();
      }
  }

  printf("Resultado:\n");
  printf("Zeros -> %d\n",zeros);
  printf("Uns -> %d\n",uns);
}
```

Programa 7

Volta a fazer o exercício de converter um número decimal para binário, com arrays.

Resolução:

```
#include <stdio.h>

#define N 16

main()
{
  int decimal,i,j;
  int bin[N];

  printf("Programa para converter de decimal para binario\n");
  printf("Indique o valor decimal: ");
  do
    scanf("%d",&decimal);
  while (decimal < 0);

  i = 0;
  do
  {
    bin[i] = decimal % 2;
    decimal = decimal / 2;
    i++;
  }
  while (decimal != 0);

  for (j=i-1; j>=0; j--)    /* Para mostrar o array invertido */
    printf("%d",bin[j]);
  printf("\n");
}
```

Programa 8

Faz um programa que apresente uma tabela de conversão de decimal para binário:

Resolução:

```
#include <stdio.h>

#define N 16

main()
{
  int decimal,i,j,k,tam;
  int bin[N];

  printf("Tabela de conversao de decimal/binario\n");
  printf("Indique ate que numero deseja a tabela: ");
  do
    scanf("%d",&tam);
  while (tam < 0);

  for (k=0; k<=tam; k++)
  {
    i = 0;
    decimal = k;
    do
    {
      bin[i] = decimal % 2;
      decimal = decimal / 2;
      i++;
    }
    while (decimal != 0);

    printf("%d\t",k);
    for (j=i-1; j>=0; j--)    /* Para mostrar o array invertido */
      printf("%d",bin[j]);
    printf("\n");
  }
}
```

Nota: também podiam ter feito uma função para converter um número para binário. Depois essa função seria chamada dentro de um ciclo.

Programação Imperativa

Aula prática 8

Sumário

---

0. Faça uma função que retorna x^n

(x do tipo float, n do tipo integer)

0a: com um ciclo.
0b: com uma função recursiva.

1.

Faz um programa que receba uma frase escrita pelo utilizador e que nos diga quantas palavras foram escritas.
(contar o número de palavras é igual a contar o número de espaços, `' '`, `'\032'` or `32` mais um)

2

Faz um programa que receba uma frase escrita pelo utilizador e que nos diga quantas vogais foram escritas.

3

Faz um programa que pergunte ao utilizador qual o seu nome próprio e qual o seu apelido. O programa deverá apresentar numa só linha o nome e apelido do utilizador.

Exemplo:

```
Qual o primeiro nome (nome próprio) ?  Pedro
Qual o último nome (apelido) ? Abrunhosa
Pedro Abrunhosa
```

4

Faz um programa que peça ao utilizador para introduzir um nome completo. O programa deverá formatar esse nome eliminando todos os "e", "da", "de" e "do", apresentar o nome formatado escrito apenas em maiúsculas, e dizer qual o tamanho do nome formatado (em caracteres).

Exemplo:

```
Escreva um nome completo: Passos Dias de Aguiar e Mota
PASSOS DIAS AGUIAR MOTA
(23 caracteres)
```

5

Faz um programa que receba um nome completo escrito de forma descuidada (com espaços a mais entre os nomes e/ou maiúsculas mal colocadas) e que apresente esse nome cuidadosamente escrito.

Exemplo:

```
Escreva um nome completo: olinda   barba De jesus
Olinda Barba de Jesus
```

6

Faz um programa que receba uma lista de 10 palavras (p/ex., nomes de amigos, de côres, de frutas...) Após o preenchimento dessa lista o programa deverá ordená-la e apresentá-la no ecrã por ordem alfabética.

Exemplo:

```
Escreva 10 nomes (fazendo Enter apos cada nome)
laranja
uva
pessego
ameixa
pera
cereja
banana
limao
tanjerina
meloa

nomes[0] = laranja
nomes[1] = uva
nomes[2] = pessego
nomes[3] = ameixa
nomes[4] = pera
nomes[5] = cereja
nomes[6] = banana
nomes[7] = limao
nomes[8] = tanjerina
nomes[9] = meloa

Em ordenacao...

nomes[0] = ameixa
nomes[1] = banana
nomes[2] = cereja
nomes[3] = laranja
nomes[4] = limao
nomes[5] = meloa
```

```
nomes[6] = pera
nomes[7] = pessego
nomes[8] = tanjerina
nomes[9] = uva
```

Sugestão: este programa pode ser feito com três funções (além da função main( ), claro): uma para preencher a lista de nomes, outra para ordenar essa lista e outra ainda para mostrar o seu conteúdo. Como ainda não aprendemos a trabalhar com apontadores, sugiro que neste exercício as vossas funções trabalhem sobre um array de caracteres (p/ex.: char nomes[10][80]) definido globalmente.

Programação Imperativa

Soluções da aula prática 8

---

0. Faça uma função que retorna x^n

```
float power(float x, int n)
/* solucao com ciclo */
{
  int i;
  float result;
  result = 1;
  for (i=1; i<=n; i++)
   result = result * x;
  return(result);
}

float power(float x, int n)
/* solucao recursiva */
{
  if (n==1)
    return(x)
  else
    return(x * power(x, n-1));
}
```

1.

```
#include <stdio.h>

void main()
{
  char frase[128];
  int p;
  int numpalavras;

  printf("Escreve uma frase:\n");
  gets(frase);
  p = 0;
  numpalavras = 1;
  while (frase[p] != '\0')
    {
      if (frase[p]==' ')
        numpalavras++;
      p++;
    }
  printf("Foram escritos %d palavras\n", numpalavras);
}
```

ou (para adeptos de C com apontadores):

```
void main()
```

```
{
  char frase[128];
  char *p;
  int numpalavras;

  printf("Escreve uma frase:\n");
  gets(frase);
  p = frase;
  numpalavras = 1;
  while (*p != '\0')
    {
      if (*p==' ')
        numpalavras++;
      p++;
    }
  printf("Foram escritos %d palavras\n", numpalavras);
}
```

2.

```
#include <stdio.h>

void main()
{
  char frase[128];
  char vogais[20] = "aAeEiIoOuU";
  int p;
  char c[2];
  int numvogais;

  printf("Escreve uma frase:\n");
  gets(frase);
  p = 0;
  numvogais = 1;
  while (frase[p] != '\0')
    {
      c[0]=frase[p];
       /* o caracter c esta dentro do string vogais?: */
      if (strstr(vogais, c))
        numvogais++;
      p++;
    }
  printf("Foram escritos %d vogais\n", numvogais);
}
```

Existe na bilbioteca tambem a função strchr que procura a posição de uma caracter dentro do string. Usanda esta função a solução seria

```
void main()
{
  char frase[128];
  char vogais[20] = "aAeEiIoOuU";
  int p;
  char c;
  int numvogais;
```

```
  printf("Escreve uma frase:\n");
  gets(frase);
  p = 0;
  numvogais = 1;
  while (frase[p] != '\0')
    {
      c=frase[p];
       /* o caracter c esta dentro do string vogais?: */
      if (strchr(vogais, c))
        numvogais++;
      p++;
    }
  printf("Foram escritos %d vogais\n", numvogais);
}
```

3.

Faz um programa que pergunte ao utilizador qual o seu nome próprio e qual o seu apelido. O programa deverá apresentar numa só linha o nome e apelido do utilizador.

Exemplo:

```
Qual o primeiro nome (nome próprio) ?  Pedro
Qual o último nome (apelido) ? Abrunhosa
Pedro Abrunhosa
```

```
#include <strdio.h>
#include <string.h>

void main()
{
  char prop[128], nome[128];

  printf("Qual o primeiro nome (nome proprio) ? ");
  gets(prop);
  printf("Qual o ultimo nome (apelido) ? ");
  gets(nome);
  strcat(prop, " ");
  strcat(prop, nome);
  printf("%s", prop);
}
```

4.

Faz um programa que peça ao utilizador para introduzir um nome completo. O programa deverá formatar esse nome eliminando todos os "e", "da", "de" e "do", apresentar o nome formatado escrito apenas em maiúsculas, e dizer qual o tamanho do nome formatado (em caracteres).

Exemplo:

```
Escreva um nome completo: Passos Dias de Aguiar e Mota
PASSOS DIAS AGUIAR MOTA
(23 caracteres)
```

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

void eliminate_characters(char *p, int n)
/**********************************\
 *  eliminates n chars do string p  *
\**********************************/
{
  do
    {
      p[0] = p[n];
      p++;
    }
  while (*p != '\0');
}

void main()
{
  char nome[128];
  int i, mudanca;
  char *p;

  printf("Escreva um nome completo :");
  gets(nome);

    /* convert to upperscript: */
  for (i=1; i<strlen(nome); i++)
    if ((nome[i]<='z') && (nome[i]>='a'))
      nome[i] = nome[i]-32;

   /* eliminate de, etc.: */
  mudanca=1;  /* houve mundancas na ultima passagem? */
  while (mudanca!=0)
    {
      mudanca=0;
      p = strstr(nome, " DE ");
      if (p!=0)
        {
          p++;  /* agora aponta "DE " dentro nome */
          eliminate_characters(p, 3);
          mudanca=1;
        }
      p = strstr(nome, " DA ");
      if (p!=0)
        {
          p++;  /* agora aponta "DA " dentro nome */
          eliminate_characters(p, 3);
          mudanca=1;
        }
    }
  printf("%s\n", nome);
  printf("(%d caracteres)\n", strlen(nome));
}
```

5

Exemplo:

```
    Escreva um nome completo: olinda   barba De jesus
    Olinda Barba de Jesus
```

```
#include <stdio.h>

void convert_to_uppercase(char *p)
/***********************************\
 *  converts char *p to uppercase   *
\***********************************/
{
   /* exemplo, A=65, a=97, B=66, b=98 .... */
  *p = *p-32;
}

void convert_to_lowercase(char *p)
/***********************************\
 *  converts char *p to lowercase   *
\***********************************/
{
  *p = *p+32;
}

void eliminate_character(char *p)
/***********************************\
 *  eliminates 1 char do string p   *
\***********************************/
{
  do
    {
      p[0] = p[1];
      p++;
    }
  while (*p != '\0');
}

void main()
{
  char nome[128];
  char *n;
  int inicio; /* estamos no inicio de uma palavra? 1=verdade */

  printf("Escreva um nome completo: ");
  gets(nome);
  n = nome;
  printf("nome: %s\n", n);
  inicio = 1;
  while (*n != '\0')
    {
      if (inicio==1)
       {
         inicio=0;
         if ((*n>='a') && (*n<='z'))
           convert_to_uppercase(n);
```

```
          else
           if (*n==' ')
             {
               eliminate_character(n);
               n--;
               inicio=1;
             }
       }
     else /* inicio==0 */
       {
         if (*n == ' ')
           inicio=1;
         if ((*n>='A') && (*n<='Z'))
           convert_to_lowercase(n);
       }
     n++;
    }
  printf("corrected: %s", nome);
}
```

6

```
/* TP8 ex.6
   ordenar nomes num array estático de 1 dimensão
   utiliza método de ordenação da burbulha ("bubble") esperto.
   solucao: Patricio Serendero */

#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#include <string.h>
/*----------------------var globais ----------------------*/
const int COL = 10;
const int LIN = 10;
char a[LIN][COL] = {"laranja", "uva", "pessego", "ameixa", "pera", "cereja",
   "banana", "limao", "tanjerina", "meloa"};

void ordena_array(void)
{
  int k, i; int resultado;
  char *s1, *s2;

  for (k=0; k<LIN; k++)       /*controla ciclo de todas as passadas */
  {
    for (i=0; i<(LIN-k-1); i++)
    {
      s1 = a[i]; s2 = a[i+1];
      resultado = strcmp(s1, s2);
      if (resultado > 0)        /*termina nos elementos já ordenados*/
        trocar(a[i], a[i+1]);
    }
  }
}

void trocar(char *primeiro, char *segundo)
```

```
  /*troca primeiro pelo segundo*/
{
  char temp[COL];

  strcpy(temp, segundo);
  strcpy(segundo, primeiro);
  strcpy(primeiro, temp);
}

void imprime_array(char vez)
{
  if (vez == '1')
    printf("\n\n conteúdo do array ANTES de ser ordenado: \n");
  else
    printf("\n\n conteúdo do array DEPOIS de ser ordenado: \n");

  for (int j=0; j < LIN; j++)
  {
    if (!(j % 4)) printf("\n");
      printf("%s  ", a[j]);
  }
}

void main()
{
  clrscr();
  imprime_array('1');
  ordena_array();
  imprime_array('2');
  getch();
}
```

Programação 1

Aula prática 8

Sumário

- Exercícios com arrays de caracteres.
- Manipulação de strings.

Programa 1

Faz um programa que receba uma frase escrita pelo utilizador e que nos diga quantas palavras foram escritas.

Programa 2

Faz um programa que receba uma frase escrita pelo utilizador e que nos diga quantas vogais foram escritas.

Programa 3

Faz um programa que pergunte ao utilizador qual o seu nome próprio e qual o seu apelido. O programa deverá apresentar numa só linha o nome e apelido do utilizador.

Exemplo:

```
Qual o primeiro nome (nome próprio) ?  Pedro
Qual o último nome (apelido) ? Abrunhosa
Pedro Abrunhosa
```

Programa 4

Faz um programa que peça ao utilizador para introduzir um nome completo. O programa deverá formatar esse nome eliminando todos os "e", "da", "de" e "do", apresentar o nome formatado escrito apenas em maiúsculas, e dizer qual o tamanho do nome formatado (em caracteres).

Exemplo:

```
Escreva um nome completo: Passos Dias de Aguiar e Mota
PASSOS DIAS AGUIAR MOTA
(23 caracteres)
```

Programa 5

Faz um programa que receba um nome completo escrito de forma descuidada (com espaços a mais entre os nomes e/ou maiúsculas mal colocadas) e que apresente esse nome cuidadosamente escrito.

Exemplo:

```
    Escreva um nome completo: olinda   barba De jesus
    Olinda Barba de Jesus
```

Programa 6

Faz um programa que receba uma lista de 10 palavras (p/ex., nomes de amigos, de côres, de frutas...) Após o preenchimento dessa lista o programa deverá ordená-la e apresentá-la no ecrã por ordem alfabética.

Exemplo:

```
    Escreva 10 nomes (fazendo Enter apos cada nome)
laranja
uva
pessego
ameixa
pera
cereja
banana
limao
tanjerina
meloa

    nomes[0] = laranja
    nomes[1] = uva
    nomes[2] = pessego
    nomes[3] = ameixa
    nomes[4] = pera
    nomes[5] = cereja
    nomes[6] = banana
    nomes[7] = limao
    nomes[8] = tanjerina
    nomes[9] = meloa

    Em ordenacao...

    nomes[0] = ameixa
    nomes[1] = banana
    nomes[2] = cereja
    nomes[3] = laranja
    nomes[4] = limao
    nomes[5] = meloa
    nomes[6] = pera
    nomes[7] = pessego
    nomes[8] = tanjerina
    nomes[9] = uva
```

Sugestão: este programa pode ser feito com três funções (além da função main( ), claro): uma para preencher a lista de nomes, outra para ordenar essa lista e outra ainda para mostrar o seu conteúdo. Como

ainda não aprendemos a trabalhar com apontadores, sugiro que neste exercício as vossas funções trabalhem sobre um array de caracteres (p/ex.: char nomes[10][80]) definido globalmente.

Programação I

Soluções dos exercícios da aula prática 8

Programa 1

```
/*
 *  a8_p1: Contador de palavras
 *
 *  Input: Uma frase escrita pelo utilizador
 * Output: Numero de palavras da frase
 */

#include <stdio.h>
#include <string.h>

main()
{
    char frase[128];
    char caracter, prox_caracter;
    int n_palavras, i;


    do
    {
        printf("Escreva uma frase:\n");
        gets(frase);
    }
    while ( strlen(frase) == 0 );

    n_palavras = 0;
    i = 0;

    caracter = frase[i];
    prox_caracter = frase[i+1];

    while (caracter != '\0')
    {
        if ( caracter != ' ' && (prox_caracter == ' ' || prox_caracter == '\0') )
            n_palavras++;

        i++;
        caracter = frase[i];
        prox_caracter = frase[i+1];
    }

    if (n_palavras == 0)
        printf("Nao foi escrita qualquer palavra!\n");
    else if (n_palavras == 1)
        printf("Foi escrita apenas %d palavra\n", n_palavras);
    else
        printf("Foram escritas %d palavras\n", n_palavras);
}
```

Programa 2

```
/*
 *  a8_p2: Contador de vogais
 *
 *  Input: Uma frase escrita pelo utilizador
```

```
 * Output: Numero de vogais usadas na frase
 */

#include <stdio.h>
#include <string.h>

main()
{
    char frase[128];
    char caracter;
    int n_vogais, i;


    do
    {
        printf("Escreva uma frase:\n");
        gets(frase);
    }
    while ( strlen(frase) == 0 );

    n_vogais = 0;
    i = 0;
    caracter = frase[i];
    while (caracter != '\0')
    {
        if ( strchr("AEIOUaeiou", caracter) != NULL )
            n_vogais++;
        i++;
        caracter = frase[i];
    }

    if (n_vogais == 0)
        printf("Nao foi escrita qualquer vogal!\n");
    else if (n_vogais == 1)
        printf("Foi escrita apenas %d vogal\n", n_vogais);
    else
        printf("Foram escritas %d vogais\n", n_vogais);
}
```

Programa 3

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

char primeiro[40];
char ultimo[40];
char nome[80];

main()
{
  printf("Escreva o primeiro nome (nome proprio): ");
  fgets(primeiro, sizeof(primeiro), stdin);
  primeiro[strlen(primeiro)-1] = '\0';

  printf("Escreva o ultimo nome (apelido): ");
  fgets(ultimo, sizeof(ultimo), stdin);
  ultimo[strlen(ultimo)-1] = '\0';

  strcpy(nome, primeiro);
  strcat(nome, " ");
  strcat(nome, ultimo);

  printf("O nome e' %s\n", nome);
}
```

Programa 4

```
/*
```

```
 *  a8_p4a: Formata nome completo
 *
 *  Input: Nome completo
 * Output: Nome completo formatado e num de caracteres
 */

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define SIZE 100
#define TRUE   1
#define FALSE  0

main()
{
    char nome[SIZE], buffer[SIZE];
    char caracter;
    int i, i_substr, encontrou;
    long int e_substr;


    /* Recebe um nome completo do teclado */
    do
    {
        printf("Escreva um nome completo:\n");
        fgets(nome, sizeof(nome), stdin );
    }
    while ( strlen(nome) == 0 );
    nome[strlen(nome)-1] = '\0';

    /* Converte nome para letras maiusculas */
    for( i = 0; i < SIZE; i++ )
    {
        buffer[i] = toupper(nome[i]);
    }


    /* Retira substring "E" e "&" */
    do
    {
        encontrou = TRUE;
        if ( strstr(buffer, " E "))
            e_substr = (long int)strstr(buffer, " E ");
        else if( strstr(buffer, " & "))
            e_substr = (long int)strstr(buffer, " & ");
        else
            encontrou = FALSE;

        if (encontrou == TRUE)
        {
            i_substr = e_substr - (long int)buffer;
            for (i = i_substr; i < SIZE-2; i++)
                buffer[i] = buffer[i+2];
        }
    }
    while(encontrou == TRUE);

    /* Retira substrings "DA", "DE", "DO" */
    do
    {
        encontrou = TRUE;
        if ( strstr(buffer, " DA "))
            e_substr = (long int)strstr(buffer, " DA ");
        else if( strstr(buffer, " DE "))
            e_substr = (long int)strstr(buffer, " DE ");
        else if ( strstr(buffer, " DO "))
            e_substr = (long int)strstr(buffer, " DO ");
        else
            encontrou = FALSE;

        if (encontrou == TRUE)
        {
```

```
            i_substr = e_substr - (long int)buffer;
            for (i = i_substr; i < SIZE-3; i++)
                buffer[i] = buffer[i+3];
        }
    }
    while(encontrou == TRUE);

    /* Retira substrings "DAS", "DOS" */
    do
    {
        encontrou = TRUE;
        if ( strstr(buffer, " DAS "))
            e_substr = (long int)strstr(buffer, " DAS ");
        else if ( strstr(buffer, " DOS "))
            e_substr = (long int)strstr(buffer, " DOS ");
        else
            encontrou = FALSE;

        if (encontrou == TRUE)
        {
            i_substr = e_substr - (long int)buffer;
            for (i = i_substr; i < SIZE-4; i++)
                buffer[i] = buffer[i+4];
        }
    }
    while(encontrou == TRUE);

    printf("%s\n(%d caracteres)\n", buffer, strlen(buffer));
}
```

Obs: Repara que esta solução, embora de leitura simples, adopta uma análise por casos que exige várias passagens pela string para a depurar completamente das substrings "de", "do", "das", etc. A solução que se segue, faz a mesma depuração com uma só passagem pela string original e já tira partido de uma técnica que é apanágio da programação estruturada: a definição de funções e passagens de parâmetros para funções.

```
/*
 *  a8_p4b: Formata nome completo
 *
 *  Input: Nome completo
 * Output: Nome completo formatado e num de caracteres
 */

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define SIZE 100
#define TRUE   1
#define FALSE  0

/*
 *  converte a string 's' para maiusculas
 */
void maiusculas( char s[] )
{
  int i;

  for( i = 0; i < strlen(s); i++ )
    s[i] = toupper(s[i]);
}

/*
 *  converte a string 's' para minusculas
 */
void minusculas( char s[] )
{
  int i;

  for( i = 0; i < strlen(s); i++ )
    s[i] = tolower(s[i]);
```

```
}

/*
 *  verifica se um caracter e branco
 *
 *  Input: Um caracter 'c'
 * Output: TRUE se 'c' for um espaco ou um 'tab', FALSE caso contrario.
 */
int eh_branco( char c )
{
  return ( c == ' ' || c == '\t' );
}

/*
 *  salta caracteres brancos num array
 *
 *  Input: Uma string 's' e uma posicao 'k'
 * Output: Devolve a primeira posicao i (i>=k) tal que s[i] nao eh branco
 */
int salta_brancos( char s[], int k )
{
  int i = k;

  while( i<strlen(s) && eh_branco(s[i]) )
    i++;

  return (i);
}

/*
 *  Input: Uma string 's' e uma posicao 'k'.
 *         A funcao assume que s[k] nao eh um caracter branco.
 * Output: 'palavra' vai ser string que contem uma sequencia maxima
 *         de caracteres s[k..i], todos eles nao brancos.
 *         Devolve a posicao i+1
 */
int apanha_palavra( char s[], int k, char palavra[] )
{
  int i, j;

  j = 0;
  i = k;
  while( i<strlen(s) && !eh_branco(s[i]) )
    {
      palavra[j] = s[i];
      j++;
      i++;
    }
  palavra[j] = '\0';

  return (i);
}

/*
 *  Input: Uma string 's'
 * Output: TRUE se 's' eh "e", "&", "de", "da", "do", "das", "dos".
 *         FALSE caso contrario.
 *
 *   Nota: a funcao nao distingue maiusculas e minusculas.
 */
int eh_e_da_de_do_das_dos( char s[] )
{
  char temp[SIZE];

  strcpy( temp, s );
  minusculas( temp );

  return( strcmp( temp, "e")   == 0 ||
          strcmp( temp, "&")   == 0 ||
          strcmp( temp, "da")  == 0 ||
          strcmp( temp, "de")  == 0 ||
          strcmp( temp, "do")  == 0 ||
          strcmp( temp, "das") == 0 ||
```

```
          strcmp( temp, "dos") == 0 );
}

/*
 *  A ideia do programa eh ir engolindo as palavras uma a uma
 *  e filtrar os "de", "do", "da", etc.
 *
 *  Uma frase eh uma sequencia de:
 *         0 ou mais caracteres brancos seguida de
 *         1 ou mais caracteres nao brancos seguida de
 *         1 ou mais caracteres brancos seguida de
 *         1 ou mais caracteres nao brancos seguida de
 *         ...
 *         ...
 *         seguida de '\0'
 */
main()
{
    char nome[SIZE];          /* nome completo */
    char nomef[SIZE] = "";    /* nome formatado */
    char palavra[SIZE];
    int  i;

    /* Recebe um nome completo do teclado */
    do
    {
      printf("Escreva um nome completo:\n");
      fgets(nome, sizeof(nome), stdin );
    }
    while ( strlen(nome) == 0 );
    nome[strlen(nome)-1] = '\0';   /* -1 porque o fgets mete um '\n' */

    maiusculas( nome );
    i = 0;
    i = salta_brancos( nome, i );
    while( i < strlen(nome) )
    {
      i = apanha_palavra( nome, i, palavra );
      if( !eh_e_da_de_do_das_dos( palavra ) )
        {
          strcat( nomef, palavra );
          strcat( nomef, " " );
        }
      i = salta_brancos( nome, i );
    }

    /* elimina o ultimo espaco */
    if( strlen(nomef) > 0 )
      nomef[ strlen(nomef) - 1 ] = '\0';

    printf("%s\n(%d caracteres)\n", nomef, strlen(nomef) );
}
```

Programa 5

```
/*
 *  a8_p5a: Formata nome
 *
 *  Input: Nome escrito descuidadamente
 * Output: Nome escrito adequadamente
 */

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define TRUE  1
#define FALSE 0

main()
```

```
{
    char nome[80], buffer[80];
    int i, j;
    int primeira_letra, encontrou;
    long int e_substr;

    /* Recebe uma string */
    do
    {
        printf("Escreva um nome: ");
        fgets(nome, sizeof(nome), stdin);
    }
    while ( strlen(nome) == 0 );
    nome[strlen(nome)-1] = '\0';

    /* Limpa espacos e atribui maiusculas/minusculas */
    i = 0; j = 0; primeira_letra = TRUE;
    do
    {
            while(nome[i] != ' ' && nome[i] != '\0')
            {
                if(primeira_letra == TRUE)
                {
                    buffer[j] = toupper(nome[i]);
                    primeira_letra = FALSE;
                }
                else
                    buffer[j] = tolower(nome[i]);
                i++; j++;
            }

            if (nome[i] == ' ')
            {
                  buffer[j] = ' ';
                  primeira_letra = TRUE;
                  j++;
            }

            while(nome[i] == ' ')
                    i++;
    }
    while(nome[i] != '\0');
    buffer[j] = '\0';

    /* Transforma "E/Da/De/Do/Das/Dos" em "e/da/de/do/das/dos" */
    do
    {
        encontrou = TRUE;
        if ( strstr(buffer, " E "))
            e_substr = (long int)strstr(buffer, " E ");
        else if ( strstr(buffer, " Da "))
            e_substr = (long int)strstr(buffer, " Da ");
        else if( strstr(buffer, " De "))
            e_substr = (long int)strstr(buffer, " De ");
        else if ( strstr(buffer, " Do "))
            e_substr = (long int)strstr(buffer, " Do ");
        else if( strstr(buffer, " Das "))
            e_substr = (long int)strstr(buffer, " Das ");
        else if ( strstr(buffer, " Dos "))
            e_substr = (long int)strstr(buffer, " Dos ");
        else
            encontrou = FALSE;

        if (encontrou == TRUE)
        {
            i = e_substr - (long int)buffer + 1;
            buffer[i] = tolower(buffer[i]);
        }
    }
    while(encontrou == TRUE);

    strcpy(nome, buffer);
    puts(nome);
```

```
}
```

Obs: Repara que também esta solução adopta uma análise que exige várias passagens pela string para a formatar conforme foi pedido. A solução que se segue, faz o mesmo com uma só passagem pela string original e voltando a tirar partido das funções desenvolvidas para o programa anterior.

```
/*
 *  a8_p5b: Formata nome
 *
 *  Input: Nome escrito descuidadamente
 * Output: Nome escrito adequadamente
 */

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define SIZE 100
#define TRUE   1
#define FALSE  0


/*
 *  incluir as funcoes feitas para o problema 4.
 */



/*
 *  O programa eh quase igual ao problema 4
 */
main()
{
    char nome[SIZE];          /* nome completo */
    char nomef[SIZE] = "";    /* nome formatado */
    char palavra[SIZE];
    int  i;

    /* Recebe um nome completo do teclado */
    do
    {
      printf("Escreva um nome completo:\n");
      fgets(nome, sizeof(nome), stdin );
    }
    while ( strlen(nome) == 0 );
    nome[strlen(nome)-1] = '\0';   /* -1 por causa do '\n' */

    minusculas( nome );
    i = 0;
    i = salta_brancos( nome, i );
    while( i< strlen(nome) )
    {
      i = apanha_palavra( nome, i, palavra );
      if( !eh_e_da_de_do_das_dos( palavra ) )
      {
        /* a primeira letra da palavra passa a maiuscula */
        palavra[0] = toupper( palavra[0] );
      }
      strcat( nomef, palavra );
      strcat( nomef, " " );
      i = salta_brancos( nome, i );
    }

    /* elimina o ultimo espaco */
    if( strlen(nomef) > 0 )
      nomef[ strlen(nomef) - 1 ] = '\0';

    printf("%s\n(%d caracteres)\n", nomef, strlen(nomef) );
}
```

Programa 6

```
/*
 *  a8_p6: Ordena lista de N palavras
 *
 *  Input: N palavras (do teclado)
 * Output: Essas N palavras escritas por ordem crescente
 */

#include <stdio.h>
#include <string.h>

#define N 10

char nomes[N][80];

void PreencheArray(void)
{
    int i;

    printf("Escreva %d palavras (fazendo Enter apos cada palavra)\n", N );
    for (i = 0; i < N; i++)
        gets(nomes[i]);

}

void MostraArray(void)
{
    int i;

    puts("");
    for (i = 0; i < N; i++)
        printf("nomes[%d] = %s\n", i, nomes[i]);

}

void OrdenaArray(void)
{
    char min_palavra[80];
    char temp[80];
    int i, k, m;

    /* Ordenar o array */
    for( k = 0; k <= N-1; k++ )
    {
        /* descobre o indice do minimo em a[k], a[k+1], ..., a[N-1] */

        strcpy(min_palavra, nomes[k]);
        m = k;
        for( i = k; i <= N-1; i++ )
            if ( strcmp(nomes[i], min_palavra) < 0 )
            {
                strcpy(min_palavra, nomes[i]);
                m = i;
            }

         /* troca a[k] com a[m] */
         strcpy(temp, nomes[k]);
         strcpy(nomes[k], nomes[m]);
         strcpy(nomes[m], temp);
     }
}

main()
{
    /* array nomes com 'global scope' */

    PreencheArray();
    MostraArray();
    printf("\nArray em ordenacao...\n");
```

```
    OrdenaArray();
    MostraArray();
}
```

Programação Imperativa

Aula Prática 9

Sumário

Programas com utilização de estruturas.

Programa 1 - Agenda telefónica.

Faz um programa para pesquisar os telefones dos teus amigos. O programa deve funcionar mais ou menos como a agenda de telefones dos telemóveis. Escreves um nome de um amigo e aparece-te o telefone respectivo.

Para fazeres este programa, define um tipo de dados chamado **pessoa** com 2 campos: **nome** e **telefone**. O nome pode ser um array de caracteres com um máximo de 30 posições e o telefone também é um array de caracteres mas com um máximo de 15 posições. Depois podes declarar uma variável chamada **amigo** que vai ser um array de pessoas e que podes inicializar logo com os nomes e telefones dos teus amigos.

O programa deve pedir um nome ao utilizador e como resposta dá o respectivo telefone. Se o nome não existir, o programa deve dizer que esse nome não consta na agenda telefónica. Exemplo do programa a funcionar:

```
Diz um nome:
alex
O número do alex é 937822899
```

Para vos ajudar, aqui está uma parte do programa:

```
#include ...

#define N_AMIGOS 5

typedef struct
{
   char nome[30];
   char telefone[15];
} pessoa;

pessoa amigo[N_AMIGOS] = {
                           { "alice",           "962189289" },
                           { "pedro teixeira",  "289782933" },
                           { "toze",            "913229289" },
                           { "alex",            "937822899" },
                           { "susana",          "218492238" }
                         };

main()
{
  ...
}
```

Programa 2 - Datas

Faz um programa para calcular o número de dias entre duas datas. O programa deve tomar em consideração os anos bissextos. Os anos bissextos são aqueles que são múltiplos de 4, exceptuando os que são múltiplos de 100 mas incluindo os múltiplos de 400 (Por exemplo, 1900 não é bissexto mas 2000 já

é).

Para facilitar a construção do programa, deves definir um tipo de dados **data** com 3 campos: **dia**, **mes**, **ano** Deves também definir uma função chamada **bissexto(ano)** que retorna 1 se o **ano** for bissexto e retorna 0 se não for bissexto. Deves também definir uma função chamada **dias_entre(d1, d2)** que devolve o número de dias entre as datas **d1** e **d2**.

Para testares o programa, pede ao utilizador para introduzir duas datas e apresenta no ecrã o resultado. Podes usar o programa para calcular quantos dias faltam para o fim das aulas, quantos dias faltam para o fim do ano, quantos dias é que já viveste, etc...

Exemplo do programa a funcionar:

```
Introduz uma data (dia mes ano): 6 5 1967
Introduz outra data (dia mes ano): 31 5 2000
Entre 6/5/1967 e 31/5/2000 passaram 12079 dias.
```

DICA NUMERO 1: uma maneira de fazer o programa é arranjar uma data de referência, por exemplo 1/1/1900. Depois podes calcular o número de dias que passaram desde 1/1/1900 até à data d1 e quantos dias passaram desde 1/1/1900 até à data d2. O resultado final será a diferença dos números obtidos.

DICA NUMERO 2: pode também dar jeito declarares um array para guardar o número de dias que cada mês tem.

Programa 3 - Pautas de alunos

Escreve um programa que mostre a pauta final de uma determinada disciplina. Cada disciplina tem um nome, um ano lectivo, um semestre e um conjunto de alunos inscritos (vamos supor que no máximo serão 200). Cada aluno tem um número, um curso, uma nota de frequência, um resultado da frequência (Dispensado, Admitido, Não Admitido), uma nota de exame, e uma classificação final. A classificação final é composta por um valor numérico, e por um resultado final (Aprovado, Reprovado). Nota: O resultado da frequência e o resultado final, devem ser determinados pelo programa.

Exemplo do programa a funcionar:

```
Disciplina: Programação 1
Ano: 2000/2001
Semestre: 1.º

                                                  Classificação final
N.º aluno - Curso | Frequência - Resultado | Exame | Numérico - Resultado
------------------------------------------------------------------------ |
2399        ESC     13.0         Dispensado  -       13.0       Aprovado
12900       IG       8.5         Admitido    12.3    12.0       Aprovado
15693       EI       8.0         Admitido     7.0     7.0       Reprovado
...

Nota: Os dados apresentados são fictícios.
```

Programação Imperativa

Aula Prática 9

Sumário

Programas com utilização de estruturas.

Programa 1 - Agenda telefónica.

Faz um programa para pesquisar os telefones dos teus amigos. O programa deve funcionar mais ou menos como a agenda de telefones dos telemóveis. Escreves um nome de um amigo e aparece-te o telefone respectivo.

Para fazeres este programa, define um tipo de dados chamado **pessoa** com 2 campos: **nome** e **telefone**. O nome pode ser um array de caracteres com um máximo de 30 posições e o telefone também é um array de caracteres mas com um máximo de 15 posições. Depois podes declarar uma variável chamada **amigo** que vai ser um array de pessoas e que podes inicializar logo com os nomes e telefones dos teus amigos.

O programa deve pedir um nome ao utilizador e como resposta dá o respectivo telefone. Se o nome não existir, o programa deve dizer que esse nome não consta na agenda telefónica. Exemplo do programa a funcionar:

```
Diz um nome:
alex
O número do alex é 937822899
```

Para vos ajudar, aqui está uma parte do programa:

```
#include ...

#define N_AMIGOS 5

typedef struct
{
   char nome[30];
   char telefone[15];
} pessoa;

pessoa amigo[N_AMIGOS] = {
                           { "alice",           "962189289" },
                           { "pedro teixeira",  "289782933" },
                           { "toze",            "913229289" },
                           { "alex",            "937822899" },
                           { "susana",          "218492238" }
                         };

main()
{
  ...
}
```

Programa 2 - Datas

Faz um programa para calcular o número de dias entre duas datas. O programa deve tomar em consideração os anos bissextos. Os anos bissextos são aqueles que são múltiplos de 4, exceptuando os que são múltiplos de 100 mas incluindo os múltiplos de 400 (Por exemplo, 1900 não é bissexto mas 2000 já

é).

Para facilitar a construção do programa, deves definir um tipo de dados **data** com 3 campos: **dia**, **mes**, **ano** Deves também definir uma função chamada **bissexto(ano)** que retorna 1 se o **ano** for bissexto e retorna 0 se não for bissexto. Deves também definir uma função chamada **dias_entre(d1, d2)** que devolve o número de dias entre as datas **d1** e **d2**.

Para testares o programa, pede ao utilizador para introduzir duas datas e apresenta no ecrã o resultado. Podes usar o programa para calcular quantos dias faltam para o fim das aulas, quantos dias faltam para o fim do ano, quantos dias é que já viveste, etc...

Exemplo do programa a funcionar:

```
Introduz uma data (dia mes ano): 6 5 1967
Introduz outra data (dia mes ano): 31 5 2000
Entre 6/5/1967 e 31/5/2000 passaram 12079 dias.
```

DICA NUMERO 1: uma maneira de fazer o programa é arranjar uma data de referência, por exemplo 1/1/1900. Depois podes calcular o número de dias que passaram desde 1/1/1900 até à data d1 e quantos dias passaram desde 1/1/1900 até à data d2. O resultado final será a diferença dos números obtidos.

DICA NUMERO 2: pode também dar jeito declarares um array para guardar o número de dias que cada mês tem.

Programa 3 - Pautas de alunos

Escreve um programa que mostre a pauta final de uma determinada disciplina. Cada disciplina tem um nome, um ano lectivo, um semestre e um conjunto de alunos inscritos (vamos supor que no máximo serão 200). Cada aluno tem um número, um curso, uma nota de frequência, um resultado da frequência (Dispensado, Admitido, Não Admitido), uma nota de exame, e uma classificação final. A classificação final é composta por um valor numérico, e por um resultado final (Aprovado, Reprovado). Nota: O resultado da frequência e o resultado final, devem ser determinados pelo programa.

Exemplo do programa a funcionar:

```
Disciplina: Programação 1
Ano: 2000/2001
Semestre: 1.º

                                                  Classificação final
N.º aluno - Curso | Frequência - Resultado | Exame | Numérico - Resultado
------------------------------------------------------------------------ |
2399        ESC     13.0         Dispensado  -       13.0       Aprovado
12900       IG       8.5         Admitido    12.3    12.0       Aprovado
15693       EI       8.0         Admitido     7.0     7.0       Reprovado
...

Nota: Os dados apresentados são fictícios.
```

Programação I

Aula Prática 9

Sumário

Programas com utilização de estruturas.

Programa 1 - Agenda telefónica.

Faz um programa para pesquisar os telefones dos teus amigos. O programa deve funcionar mais ou menos como a agenda de telefones dos telemóveis. Escreves um nome de um amigo e aparece-te o telefone respectivo.

Para fazeres este programa, define um tipo de dados chamado **pessoa** com 2 campos: **nome** e **telefone**. O nome pode ser um array de caracteres com um máximo de 30 posições e o telefone também é um array de caracteres mas com um máximo de 15 posições. Depois podes declarar uma variável chamada **amigo** que vai ser um array de pessoas e que podes inicializar logo com os nomes e telefones dos teus amigos.

O programa deve pedir um nome ao utilizador e como resposta dá o respectivo telefone. Se o nome não existir, o programa deve dizer que esse nome não consta na agenda telefónica. Exemplo do programa a funcionar:

```
Diz um nome:
alex
O número do alex é 937822899
```

Para vos ajudar, aqui está uma parte do programa:

```
#include ...

#define N_AMIGOS 5

typedef struct
{
   char nome[30];
   char telefone[15];
} pessoa;

pessoa amigo[N_AMIGOS] = {
                           { "alice",           "962189289" },
                           { "pedro teixeira",  "289782933" },
                           { "toze",            "913229289" },
                           { "alex",            "937822899" },
                           { "susana",          "218492238" }
                         };

main()
{
  ...
}
```

Programa 2 - Datas

Faz um programa para calcular o número de dias entre duas datas. O programa deve tomar em

consideração os anos bissextos. Os anos bissextos são aqueles que são múltiplos de 4, exceptuando os que são múltiplos de 100 mas incluindo os múltiplos de 400 (Por exemplo, 1900 não é bissexto mas 2000 já é).

Para facilitar a construção do programa, deves definir um tipo de dados **data** com 3 campos: **dia**, **mes**, **ano** Deves também definir uma função chamada **bissexto(ano)** que retorna 1 se o **ano** for bissexto e retorna 0 se não for bissexto. Deves também definir uma função chamada **dias_entre(d1, d2)** que devolve o número de dias entre as datas **d1** e **d2**.

Para testares o programa, pede ao utilizador para introduzir duas datas e apresenta no ecrã o resultado. Podes usar o programa para calcular quantos dias faltam para o fim das aulas, quantos dias faltam para o fim do ano, quantos dias é que já viveste, etc...

Exemplo do programa a funcionar:

```
Introduz uma data (dia mes ano): 6 5 1967
Introduz outra data (dia mes ano): 31 5 2000
Entre 6/5/1967 e 31/5/2000 passaram 12079 dias.
```

DICA NUMERO 1: uma maneira de fazer o programa é arranjar uma data de referência, por exemplo 1/1/1900. Depois podes calcular o número de dias que passaram desde 1/1/1900 até à data d1 e quantos dias passaram desde 1/1/1900 até à data d2. O resultado final será a diferença dos números obtidos.

DICA NUMERO 2: pode também dar jeito declarares um array para guardar o número de dias que cada mês tem.

Programa 3 - Pautas de alunos

Escreve um programa que mostre a pauta final de uma determinada disciplina. Cada disciplina tem um nome, um ano lectivo, um semestre e um conjunto de alunos inscritos (vamos supor que no máximo serão 200). Cada aluno tem um número, um curso, uma nota de frequência, um resultado da frequência (Dispensado, Admitido, Não Admitido), uma nota de exame, e uma classificação final. A classificação final é composta por um valor numérico, e por um resultado final (Aprovado, Reprovado). Nota: O resultado da frequência e o resultado final, devem ser determinados pelo programa.

Exemplo do programa a funcionar:

```
Disciplina: Programação 1
Ano: 2000/2001
Semestre: 1.º

                                                  Classificação final
N.º aluno - Curso | Frequência - Resultado | Exame | Numérico - Resultado
-------------------------------------------------------------------------- |
2399        ESC     13.0         Dispensado  -       13.0       Aprovado
12900       IG       8.5         Admitido    12.3    12.0       Aprovado
15693       EI       8.0         Admitido     7.0     7.0       Reprovado
...

Nota: Os dados apresentados são fictícios.
```

Programação I

Soluções dos exercícios da aula prática 9

Programa 1 - Agenda telefónica.

Resolução:

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

#define N_AMIGOS 5

typedef struct
{
  char nome[30];
  char telefone[15];
} pessoa;

pessoa amigo[N_AMIGOS] = {
                           { "alice",           "96-2189289" },
                           { "pedro teixeira", "289-7829933" },
                           { "toze",            "91-3229289" },
                           { "alex",            "93-7822899" },
                           { "susana",          "21-8492238" }
                         };

main()
{
  char nome[200];
  int i;

  printf("Diz um nome: ");
  gets( nome );
  for( i=0; i< N_AMIGOS; i++ )
    if( strcmp( nome, amigo[i].nome ) == 0 )
      break;
  if( i < N_AMIGOS )
    printf("O numero e %s\n", amigo[i].telefone );
  else
    printf("%s nao esta na minha agenda\n", nome );
}
```

Programa 2 - Datas

Resolução:

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

typedef struct
{
  int dia;
  int mes;
  int ano;
} data;

/* dias_mes é uma tabela que tem os dias de cada mês */
int dias_mes[2][13] = {
  { 0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }, /* não bissexto */
  { 0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 }  /* bissexto */
};

/* retorna 1 se o ano é bissexto, retorna 0 se não é bissexto */
int bissexto( int ano )
{
   return (ano % 4 == 0) && (ano % 100 != 0 || ano % 400 == 0);
}

/* calcula o numero de dias entre a 1/1/1900 e a data d.
   assume que a data d é posterior a 1/1/1900
*/
int dias( data d )
{
   int i, soma, b;

   soma = 0;
   for( i=1900; i< d.ano; i++ )       /* anos completos */
```

```
     if( bissexto( i ) )
       soma = soma + 366;
     else
       soma = soma + 365;
   b = bissexto( d.ano );
   for( i=1; i< d.mes; i++ )          /* meses completos */
     soma = soma + dias_mes[b][i];
   soma = soma + d.dia;               /* dias */
   return soma;
}

/* calcula o numero de dias entre as datas d1 e d2 */
int dias_entre( data d1, data d2 )
{
   int dias1, dias2;
   dias1 = dias( d1 );
   dias2 = dias( d2 );
   return abs( dias1-dias2 );
}

main()
{
  data d1,d2;

  printf("Introduz uma data (dia mês ano): ");
  scanf("%d%d%d", &d1.dia, &d1.mes, &d1.ano );
  printf("Introduz outra data (dia mês ano): ");
  scanf("%d%d%d", &d2.dia, &d2.mes, &d2.ano );
  printf("Entre %d/%d/%d e %d/%d/%d passaram %d dias.\n",
          d1.dia, d1.mes, d1.ano,
          d2.dia, d2.mes, d2.ano,
          dias_entre( d1,d2 ) );
}
```

Programa 3 - Pautas de alunos

Resolução:

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

typedef struct
{
  int nota_final;
  char resultado[15];
} classificacao;

typedef struct
{
  int numero;
  char curso[5];
  float nota_freq;
  char resultado_freq[15];
  float nota_exame;
  classificacao classifica;
} discente;

typedef struct
{
  char nome[32];
  char ano_lectivo[12];
  int semestre;
  discente aluno[200];
} disciplina;

int arredonda (float nota)
{
  if (nota-0.5 >= (int)nota)
    return (int) ceil ((double)nota);
  else
    return (int) floor ((double)nota);
}

float recebe_nota (void)
{
  float nota;

  do
    {
      scanf("%f",&nota);
    }
  while ((nota < 0) || (nota > 20));
  return nota;
}

main()
{
```

```
  disciplina pauta;
  int i,j;
  char c='';

  /* Receber os dados */
  printf("INTRODUZA OS DADOS\n");
  printf("\nDisciplina\n");
  printf("Nome: ");gets(pauta.nome);
  printf("Ano lectivo: ");gets(pauta.ano_lectivo);
  printf("Semestre: ");scanf("%d",&pauta.semestre);
  printf("\n\nAlunos\n");
  for (i=0;c!='0' && i<200;i++)
  {
    printf("-- %d.§ Aluno --\n",i+1);
    printf("Numero:");scanf("%d",&pauta.aluno[i].numero);
    scanf("%c",&c);         /* por causa do enter */
    printf("Curso: ");gets(pauta.aluno[i].curso);
    printf("Nota de Frequencia: ");pauta.aluno[i].nota_freq=recebe_nota();

    if (pauta.aluno[i].nota_freq >= 9.5)
      strcpy(pauta.aluno[i].resultado_freq,"Dispensado");
    else if (pauta.aluno[i].nota_freq >= 6.0)
      {
        strcpy(pauta.aluno[i].resultado_freq,"Admitido");
        printf("Nota de exame: ");pauta.aluno[i].nota_exame=recebe_nota();
      }
    else
      strcpy(pauta.aluno[i].resultado_freq,"Nao Admitido");

    if (strcmp(pauta.aluno[i].resultado_freq,"Dispensado") == 0)
      {
        pauta.aluno[i].nota_exame = 0;
        pauta.aluno[i].classifica.nota_final = arredonda(pauta.aluno[i].nota_freq);
        strcpy(pauta.aluno[i].classifica.resultado, "Aprovado");
      }
    else if ((strcmp(pauta.aluno[i].resultado_freq,"Admitido") == 0) && (pauta.aluno[i].nota_exame >= 9.5))
      {
        pauta.aluno[i].classifica.nota_final = arredonda(pauta.aluno[i].nota_exame);
        strcpy(pauta.aluno[i].classifica.resultado, "Aprovado");
      }
    else if ((strcmp(pauta.aluno[i].resultado_freq, "Admitido") == 0) && (pauta.aluno[i].nota_exame < 9.5))
      {
        pauta.aluno[i].classifica.nota_final = arredonda(pauta.aluno[i].nota_exame);
        strcpy(pauta.aluno[i].classifica.resultado, "Reprovado");
      }
    else
      {
        pauta.aluno[i].nota_exame = 0;
        pauta.aluno[i].classifica.nota_final = 0;
        strcpy(pauta.aluno[i].classifica.resultado, "Reprovado");
      }

    printf("\nPrima 0 para terminar\n");
    scanf("%c",&c);         /* por causa do enter */
    c=getchar();
  }

  /* Escrever a pauta */
  printf("Disciplina: %s \n",pauta.nome);
  printf("Ano: %s \n",pauta.ano_lectivo);
  printf("Semestre: %d.§ \n",pauta.semestre);
  printf("                                              Classificacao final\n");
  printf("N.§ aluno - Curso | Frequencia - Resultado | Exame | Numerico - Resultado |\n");
  printf("-------------------------------------------------------------------------------\n");
  for (j=0;j<i;j++)
    {
      printf("%d\t",pauta.aluno[j].numero);
      printf("    %s\t",pauta.aluno[j].curso);
      printf("    %4.1f\t",pauta.aluno[j].nota_freq);
      printf("%s\t",pauta.aluno[j].resultado_freq);
      printf("%4.1f",pauta.aluno[j].nota_exame);
      printf("   %3d.0\t",pauta.aluno[j].classifica.nota_final);
      printf("%s\n",pauta.aluno[j].classifica.resultado);
    }
}
```

Aula prática 10

Once upon a midnight dreary, while I pondered, weak and weary
Over many a quaint and curious volume of forgotten lore,
While I nodded, nearly napping, suddenly there came a tapping,
As of some one gently rapping, rapping at my chamber door.
" 'Tis some visitor," I muttered, "tapping at my chamber door -
Only this, and nothing more."

The Raven, Edgar Allan Poe, 1845.

1a. Neste aula vamos escrever um programa que vai ler um ficheiro. Como exemplo vamos ler um ficheiro que contem o poema *The Raven* de Edgar Allan Poe (um excerto, a primeira parte, fica no início desta página) e vamos contar quantas vezes cada letra aparece no texto.

- Só conta as letras normais ('A' .. 'Z')
- Não distingue entre 'A' e 'a', etc.
- Põe de lado todos os outros caracteres.

Aqui encontra-se o ficheiro `theraven.txt` com o texto inteiro do *The Raven* de Edgar Allan Poe. Guarda este ficheiro no seu disco (Y:). Carrega o botão direito do rato em cima da palavra azul 'aqui' e depois: Em Microsoft Internet Explorer "Save Target As...". Em Netscape Navigator "Save Link As...".

1b. Muda o programa da 1a de forma que vai guardar a informação no ficheiro 'contas.txt'.

1c. Qual letra aparece o mais frequente? (escreve um programa). Isto segue a regra 'ETAOIN SHRDLU' (em inglês)

ASCII table and description

ASCII stands for American Standard Code for Information Interchange. Computers can only understand numbers, so an ASCII code is the numerical representation of a character such as 'a' or '@' or an action of some sort. ASCII was developed a long time ago and now the non-printing characters are rarely used for their original purpose. Below is the ASCII character table and this includes descriptions of the first 32 non-printing characters. ASCII was actually designed for use with teletypes and so the descriptions are somewhat obscure. If someone says they want your CV however in ASCII format, all this means is they want 'plain' text with no formatting such as tabs, bold or underscoring - the raw format that any computer can understand. This is usually so they can easily import the file into their own applications without issues. Notepad.exe creates ASCII text, or in MS Word you can save a file as 'text only'

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k

12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.asciitable.com

[soluções]

Soluções da aula prática 10

1a.

```
/* Edgar Alan Poe */
#include <stdio.h>

void main()
{
  FILE *f;
  char c;
  int ci;
  int cont[91];
  int i;

  for (i=65; i<=90; i++)
    cont[i] = 0;
  f = fopen("theraven.txt", "r");
  while (!feof(f))
  {
    c=fgetc(f);              /* get a single character from file */
    printf("%c", c);         /* show it on screen */
    if ((c>=97) && (c<=122)) /* convert ot uppercase */
      c=c-32;
    if ((c>=65) && (c<=90))  /* count it in our histogram 'cont' */
      cont[c] = cont[c] + 1;
  }
  fclose(f);
  for (i=65; i<=90; i++)
    printf("%c : %d\n", i, cont[i]);
}
```

output (ecrã)

```
A  339
B  94
C  71
D  194
E  618
F  94
G  122
H  290
I  318
J  2
K  32
L  225
M  158
N  374
O  370
```

```
P  95
Q  9
R  336
S  278
T  437
U  121
V  66
W  79
X  3
Y  100
Z  0
```

1b.

```
/* Edgar Alan Poe output to file */
#include <stdio.h>

void main()
{
  FILE *f;
  char c;
  int ci;
  int cont[91];
  int i;

  for (i=65; i<=90; i++)
    cont[i] = 0;
  f = fopen("theraven.txt", "r");
  while (!feof(f))
  {
    c=fgetc(f);
    printf("%c", c);
    if ((c>=97) && (c<=122))
      c=c-32;
    if ((c>=65) && (c<=90))
      cont[c] = cont[c] + 1;
  }
  fclose(f);
  f = fopen("contas.txt", "w");
  for (i=65; i<=90; i++)
    fprintf(f, "%c : %d\n", i, cont[i]);
  flcose(f);
}
```

1c.

```
/* Edgar Alan Poe */
#include <stdio.h>

void main()
{
```

```
  FILE *f;
  char c;
  int ci;
  int cont[91];
  int i;
  int maxn;

  for (i=65; i<=90; i++)
    cont[i] = 0;
  f = fopen("theraven.txt", "r");
  while (!feof(f))
  {
    c=fgetc(f);
    printf("%c", c);
    if ((c>=97) && (c<=122))
      c=c-32;
    if ((c>=65) && (c<=90))
      cont[c] = cont[c] + 1;
  }
  fclose(f);
  maxn=0;
  maxc=0;
  for (i=65; i<=90; i++)
    if (cont[i]>maxn)
      {
        maxc = i;
        maxn = cont[i];
      }
  printf("Max: %c  %d\n", maxc, maxn);
}
```

output (ecrã)

```
Max: E  618
```

Aula prática 11

---

1. Faz um programa que determine quanto tempo leva dobrar o capital numa conta do banco. O utilizador deve dar os dados relevantes (por exemplo a taxa de juros).

---

2. Declare um array de 1000 elementos. Enche o array com números aleatórios e escreve o código para ordenar o array..

---

3. O programa que se segue deveria calcular o factorial e o somatório de um número introduzido pelo utilizador, no entanto tem alguns erros. Assinale e corrija os erros do programa para que realize o que é pretendido.

```
/* Factorial */

int num, somatorio, factorial;

void main() {
  printf("Indique um numero inteiro ");
  scanf("%d", num);
  while (num>0)
    {
      factorial = factorial * num;
      somatorio = somatorio + num;
    }
  printf("Factorial %d, Somatorio %f",
          factorial, factorial);
}
```

---

4. Define um novo tipo de variável para guardar um coordenado ou vector (x, y, z). Depois escreve uma função que recebe um coordenado e devolve o comprimento do vector (a distância até o origem).

---

soluções

Soluções da aula prática 11

1.

```
/* Dobrar Capital */

void main()
{
  float cap, juros;
  int ano;

  printf("Taxa de juros (%): ");
  scanf("%d", &juros);
  ano = 0;
  cap = 1;
  do
   {
    cap = cap + cap*(juros/100.0);
    ano++;
   }
  while(cap<2.0);
  printf("numero de anos: %d\n", ano);
end.
```

2.

```
/* Ordenar1000 */

#define N 1000

void main()
{
   float ra[N];
   int i, j;
   float min;
   int jmin;
   float temp;

  for (i=0; i<N; i++)
    ra[i] = rand();
  for (i=0; i<N-1; i++)
    {
      min = 2.0; /* com certeza maior que numero maximo do array */
      jmin = i;
        /* procura minimo no resto do array: */
      for (j=i; j<N; j++)
        if (ra[j]<min)
```

```
          {
            min = ra[j];
            jmin = j;
          }
        /* troca o numero minimo com ra[i]: */
      temp = ra[i];
      ra[i] = ra[jmin];
      ra[jmin] = temp;
    }
   /* mostrar resultado: */
  for (i=0; i<N; i++)
    printf("%d  %0.6f", i, ra[i]);
}
```

3.

```
/* Factorial */

int num, somatorio, factorial;

{
  printf("Indique um numero inteiro");
  scanf("%d", &num);
  factorial = 1;
  somatorio = 0;
  while (num>0)
    {
      factorial = factorial * num;
      somatorio = somatorio + num;
      num = num - 1;
    }
  printf("Factorial %d, Somatorio %d', factorial,
         somatorio);
}
```

4.

```
/* Distance */
#include <math.h>

typedef struct {
  float x, y, z;
} coordenate;

float VectorLength(coordenate co)
{
  return (sqrt(sqr(co.x) + sqr(co.y) + sqr(co.z));
}

void main()
{
  /* outras instrucoes */
```

```
}
```
