ÍNDICE

1. INTRODUÇÃO À LINGUAGEM C

1.1. Introdução

A linguagem C foi desenvolvida nos laboratórios Bell na década de 70, tendo surgido a partir da necessidade de escrever programas, que utilizassem as potencialidades da linguagem máquina, mas de uma forma mais simples e portátil que esta. Usada inicialmente para a programação de sistemas, viria pela sua flexibilidade e poder, a tornar-se numa linguagem de uso geral nos mais diversos contextos, sendo actualmente muito utilizada pelos programadores profissionais.
A linguagem C pode ser considerada uma linguagem de médio nível, porque para além de possuir instruções de alto nível e ser tão estruturada como por exemplo o PASCAL, também possui instruções de baixo nível.

1.2. Características

As suas principais características são:

- Potencialidades e conveniência das linguagens quer de alto nível quer de baixo nível.
- Pequeno conjunto de palavras reservadas.
- Permitir a criação de novos tipos de dados pelo utilizador.
- "Portabilidade", isto é, os programas escritos em linguagem C podem ser transportados entre máquinas de diferentes arquitecturas com um número muito reduzido de alterações.
- Eficiência, através de uma boa ligação aos recursos da máquina.

1.3. Estrutura de um Programa

Um programa em linguagem C é constituído por uma ou mais funções e variáveis. Uma função contém instruções que especificam as operações de computação a serem executadas, enquanto as variáveis armazenam os valores. A função permite

"encapsular"[a] alguma computação, podendo depois ser utilizada sabendo-se apenas o que faz e sem preocupações quanto à forma como o faz. As funções em C assemelham-se aos procedimentos e funções do PASCAL. A execução de um programa começa tipicamente numa função de nome *main*, que deverá existir em qualquer parte do programa. O aspecto geral de uma função é o seguinte:

```
tipo_retornado nome_função(declarações de parâmetros)
  {
        declarações  locais
        instruções
  }
```

Como se vê uma função é composta por um cabeçalho seguido do respectivo corpo. O cabeçalho é constituído pelo nome da função a que se segue entre parêntesis, uma lista de declarações de parâmetros. Caso estes não existam os parêntesis surgem vazios. Quanto ao corpo da função, este é delimitado por chavetas { }, e contém declarações de variáveis, ou mesmo de outras funções, a serem utilizadas, seguidas de instruções.
Sempre que se pretenderem incluir comentários num programa, pode-se fazê-lo colocando esses comentários entre /* e */, como se verá num exemplo mais à frente.
Uma função pode retornar um valor para a função que a chama, o que é conseguido com a instrução *return*, sendo o seu aspecto:

```
return expressão;
```

em que, *expressão* pode ser qualquer expressão. Por exemplo:

```
return  5;              /* Retorna a constante inteira 5 */
return val * val;       /* Retorna o resultado do produto */
```

As funções são chamadas (invocadas para execução) dando-se o nome da função juntamente com os argumentos, entre parêntesis:

```
nome_função(prim_arg, seg_arg,... )
```

Os valores dos argumentos na chamada da função, são atribuídos aos parâmetros da função pela ordem em que se encontram.

Exemplo1: Programa que imprime no ecrã: O meu primeiro programa!

```
#include <stdio.h>                          /* inclui dados sobre funções de entrada
                                            /* e saída standards, como a printf */
main()                                      /* definição da função main */
{                                           /* Chaveta de abertura do corpo */
      printf ("O meu primeiro programa!");  /* Chama função printf */
}                                           /* Chaveta de fecho do corpo da função */
```

O resultado da execução do programa será a impressão no ecrã do texto contido entre aspas na função *printf*. A instrução *#include <stdio.h>*, tem por finalidade incluir no programa informação

---

[a] Esconder certa complexidade

sobre funções de entrada e saída de dados, tal como a função *printf* que será explicada no capítulo 4.

Exemplo2: O seguinte programa chama uma função *mul()*, que multiplica dois números inteiros e retorna o resultado. O valor retornado passa a ser o argumento da função *printf* que o imprime. Os dois parâmetros *a* e *b* da função *mul(),* recebem respectivamente os dois argumentos *5* e *4*, com que a função é chamada.

```
#include <stdio.h>

main()
{
        printf("A multiplicação de 5 com 4 é igual a: %d", mul(5,4));
}

int mul(int a, int b)
{
        return (a * b);
}
```

A impressão será: *A multiplicação de 5 com 4 é igual a: 20,* o que significa que *%d* vai ser substituído pelo valor retornado pela função *mul()*. Notar ainda, que as variáveis *a* e *b*, definidas na função só são reconhecidas no interior desta.

Uma variável é o nome que é dado a uma zona de memória, cujo tamanho varia conforme o tipo dessa variável. No exemplo anterior, *a* e *b* são variáveis capazes de armazenar inteiros. Este assunto será abordado detalhadamente na secção 2.3.
A produção e execução de programas, depende do sistema operativo em que se esteja a trabalhar, bem como do compilador utilizado. Em qualquer caso o programa fonte deve ser criado com a extensão ".c" (por exemplo, exemplo1.c).
A função *main*, como função que é, também pode retornar um valor. Neste caso, o valor será retornado ao ambiente exterior, responsável pela execução do programa. Habitualmente o valor 0 significa um final sem erro, e um valor diferente de 0 um final com erro. No entanto, e para simplificação não será utilizada a instrução *return 0* nas várias funções *main* aqui apresentadas.

1.4. Instruções

Uma expressão seguida de ';' é uma instrução. Expressões podem ser formadas por constantes, variáveis e operadores combinados de uma forma válida. As instruções dividem-se em simples e compostas. Simples quando se trata de apenas uma instrução, e compostas quando se trata de um conjunto (bloco) de instruções. Estes conceitos, de instrução simples e composta, farão mais sentido quando, no capítulo 3, forem discutidas as instruções de controle de sequência.

- Instrução simples:

expressão;

Exemplo:

```
printf("O meu primeiro programa!");
```

- Instruções compostas: Instruções simples dentro de chavetas

```
{
        instrução
        instrução
        ...
}
```


Exemplo:

```
{
        x = 10;
        y = x + x;
}
```

O sinal = significa atribuição, isto é, para o exemplo acima à variável x foi-lhe atribuído o valor 10.

2. OPERANDOS E OPERADORES

2.1. Introdução

Na base de qualquer programa estão os operandos e operadores, pois é através destes que se constroem as expressões que acabarão por formar um programa. As variáveis e as constantes constituem os operandos que representam os dados, enquanto os operadores determinam o que lhes fazer. Os operandos classificam-se através de três atributos:

nome
tipo
classe de armazenamento

As próximas secções tratam exactamente destes atributos, fundamentais para um correcto entendimento da linguagem C.

2.2. Nomes

Um nome, ou identificador, é uma sequência de letras ou dígitos que começa obrigatoriamente por uma letra. O sinal _ ("underscore") conta como uma letra, e é normalmente usado para simular um espaço num nome.
Nomes com letras maiúsculas são diferentes de nomes com letras minúsculas. Assim sendo, *A* e *a* representarão dois nomes distintos. Normalmente em C, usam-se letras minúsculas para representar as variáveis, e letras maiúsculas para representar as constantes. Esta utilização de maiúsculas e minúsculas não é uma regra da linguagem, mas ao ser seguida pelos programadores facilita a leitura dos programas.
As palavras reservadas *int, if, else, etc.* da linguagem C, não são permitidas como nomes. Em seguida apresenta-se a lista das palavras reservadas do C.

Palavras Reservadas

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

2.3. Tipos de Dados

Todas as variáveis são de um determinado tipo de dado que é especificado na definição da variável. Ao definir-se uma variável, seja qual for o tipo, o compilador reserva o espaço na memória necessário a essa variável. Em C existem quatro tipos de dados básicos, que são:

char - carácter; 1 byte.
int - inteiro; normalmente uma palavra de computador: 2 / 4 bytes.
float - número de vírgula flutuante; normalmente 4 bytes.
double - não inferior à precisão do float; normalmente o dobro da precisão do float: 8 bytes.

Para além dos tipos de dados existem qualificadores que lhes podem ser aplicados, e que são:

short, long, signed, unsigned

O qualificador *short* aplica-se a inteiros e tem como objectivo reduzi-los para metade do seu tamanho (ex: numa máquina em que os inteiros sejam de 4 bytes, o short int será de 2 bytes). O *long* aplicado a inteiros tem a função inversa do *short*. O qualificador *long* também se aplica ao tipo *double*, mas com efeitos que dependem da sua implementação no compilador.
Os qualificadores *signed* e *unsigned* são aplicados ao tipo *char* ou a qualquer inteiro, e indicam se o número tem ou não sinal. No caso dos *signed* o seu bit mais significativo está reservado ao sinal. Números *unsigned* são sempre superiores ou iguais a zero, enquanto os *signed* poderão ser negativos. Exemplificando, para o caso do *char*, uma variável *signed char* tomaria valores entre -128 e 127, enquanto que se fosse *unsigned char*, tomaria valores entre 0 e 255. Isto significa que uma variável *char* pode conter qualquer carácter ASCII, mais exactamente os respectivos códigos, pois estes variam de 0 a 127[a].
Todas as variáveis devem ser declaradas antes de serem utilizadas. Uma declaração especifica um tipo, e contém uma ou mais variáveis desse tipo. Exemplos disso são:

---

[a] Existem extensões a este código onde se incluem caracteres de desenho.

int ano,mes,dia;

declara três variáveis do tipo inteiro cujos nomes são ano, mes e dia.

unsigned char tamanho;

declara uma variável do tipo char sem sinal, e cujo nome é tamanho. Para este caso o compilador "alocaria" espaço para um carácter (1 byte). As variáveis podem ser inicializadas no momento da sua declaração:

```
int x = -691;
double media = 0;
```

Existe em C um operador para a determinação do espaço em memória ocupado por um determinado tipo de dado, e cujo nome é *sizeof*. O operador sizeof devolve um valor inteiro que é o número de bytes correspondente ao tamanho desse objecto. Por exemplo para determinar o espaço ocupado por um inteiro, usa-se: *sizeof (int)*.

Exercício: Escreva um programa que diga qual o espaço reservado em memória para os seguintes tipos de dados: char, short, int, long int, float e double.

2.4. Classes de Armazenamento

As classes de armazenamento de um identificador definem o modo como o compilador lhes reserva espaço. A classe de armazenamento determina para uma variável o seu tempo de vida, e o seu âmbito (escopo), isto é:

tempo de vida: Tempo de vida da variável no programa
escopo: Porção de programa onde a variável é reconhecida.

Os objectos em C caem em duas classes gerais dependendo aonde são definidas.

locais - objectos definidos dentro de um bloco, normalmente numa função.
globais - objectos definidos no nível mais exterior, fora da função.

As declarações/definições das variáveis incluindo a sua classe de armazenamento têm a forma geral:

classe_armazenamento tipo_dado nome_variável

2.4.1. Tipos de Classes de Armazenamento

São quatro as classes de armazenamento de objectos em C:

auto Classe de armazenamento interna para variáveis "locais" que existem e são reconhecidas somente ao longo da duração da função. Todas as variáveis locais são implicitamente auto.

Exemplo: No programa seguinte é chamada uma função *soma* que adiciona os dois argumentos que lhe são fornecidos em cada chamada. O resultado é retornado à função principal, que o imprime.

```
/* Programa soma.c */

#include <stdio.h>

main()
{
        int x = 0;
        x=soma(x,10);
        printf("x = %d",x);
        x=5;
        x=soma(x,20);
        printf("x = %d",x);
}

soma(int a, int incr)
{
        int r=a;
        r = r + incr;
        return r;
}
```

O resultado do exemplo seria a impressão no monitor de, 10 para o primeiro *printf*, e de 25 para o segundo *printf.*
Na primeira chamada *r* é inicializado com o valor de *a*, igual a 0, que é incrementado com o valor de *incr*, igual a 10, sendo o valor devolvido 10. Na segunda chamada da função *r* é inicializado novamente com o valor de *a*, que é 5, sendo depois incrementado com o valor de *incr*, que é 20, o valor devolvido desta vez é 25.

static O seu significado varia consoante o objecto é definido dentro ou fora de uma função. Uma variável *static* quando é inicializada na sua definição só é inicializada na primeira vez que a função é chamada. Nas chamadas seguintes vai apresentar o valor da sua última utilização. O interesse desta classe surge quando por exemplo se pretende que o valor de uma variável de uma função não seja destruído por uma nova chamada a essa função. O tempo de vida destas variáveis é o da duração do programa.

Exemplo: No programa seguinte, é invocada uma função chamada *adicao.* A variável (*static*) da função é inicializada com um valor passado no primeiro argumento. Esta função adiciona o segundo argumento à variável static. O resultado da adição é devolvido à *main*.

```
/* Programa static.c */

#include <stdio.h>

main()
{
        int x = 0;
        x=adicao(x,10);
        printf("x = %d",x);
        x=5;
        x=adicao(x,20);
        printf("x = %d",x);
}

adicao(int a, int incr)
{
        static int r=a;
        r = r + incr;
        return r;
}
```

O resultado do exemplo seria a impressão no monitor de, 10 para o primeiro *printf*, e de 30 para o segundo *printf*, porque a variável *r*, em virtude de estar definida como static é inicializada apenas na primeira chamada. Na segunda chamada, a variável *r* já não é inicializada, e ao seu valor 10, é adicionado o novo incremento de 20, que resultará no valor 30 que é retornado. Portanto o parâmetro *a* só é utilizado na primeira chamada da função, não tendo qualquer efeito a atribuição a *x* do valor 5.

extern Para variáveis globais a todo o programa. Nesta classe caem por exemplo as variáveis definidas externamente num ficheiro e que se pretende utilizar noutro ficheiro do mesmo programa.

Exemplo: Neste exemplo são apresentados extractos de dois ficheiros pertencentes ao mesmo programa. No ficheiro f1.c é definida a variável *x_ext*. No segundo ficheiro, f2.c, é declarada a variável *x_ext* como externa, o que significa que se encontra definida em outro local.

```
/* ficheiro f1.c */

...
int x_ext;
```

```
/* ficheiro f2.c */

...
extern int x_ext;
```

register O mesmo que auto, excepto que, o compilador, se possível, escolhe locais de memória de acesso mais rápido (registos). É normalmente utilizada para variáveis de ciclo. Só aplicável a variáveis do tipo *int* ou *char.*

Exemplo:

```
register int contador;
```

declara a variável inteira *contador* como sendo para armazenar num registo.

2.5. Constantes

As constantes são, como o seu nome diz, valores fixos que não podem ser alterados durante o programa e, têm associado a cada uma delas um determinado tipo de dado. Os tipos de constantes inteiras são:

decimal	7	11
octal	07	013
hexadecimal	0x7	0xb

Para identificar o tipo de dado como *unsigned* usa-se o sufixo *u* ou *U*, enquanto para significar *long* usa-se o sufixo *l* ou *L*, e o sufixo *ul* ou *UL* para *unsigned long*. O tipo *int* é assumido por defeito desde que a constante seja inteira e portanto não necessita de sufixo.

Exemplos:

1066	int, decimal
03301	int, octal
0x7bF	int, hexadecimal
86243L	long, decimal
0xe4e1c00L	long, hexadecimal
2.718281828459045	double, decimal
6.624e-27	double, decimal

Um ponto decimal numa "string" (cadeia) de dígitos indica que o objecto é do tipo *double*. O expoente pode ser denotado usando *e* ou *E*. Uma constante de vírgula flutuante com o sufixo *l* ou *L* é considerada do tipo *long double.* Para significar *float* usa-se o sufixo *f* ou *F*.
Os caracteres são denotados através de plicas, por exemplo *'q'*, representa o carácter *q*. Os caracteres têm uma representação interna numérica. No código ASCII o carácter *'q'* é representado pelo inteiro (em octal) *0161*. Então,

c = 0161;	é equivalente a	c = 'q' ;
c = 49;	é equivalente a	c = '1';

Vários caracteres especiais, podem ser representados como se segue:

\n	newline
\t	tab
\b	backspace
\\	backslash
\r	carriage return

Qualquer carácter pode ser especificado pelo seu equivalente numérico usando a notação '*ddd*', onde *ddd* é um inteiro octal, ou por *'\xddd'*, onde *ddd* é um inteiro em hexadecimal.

Exemplo:

```
char c=´\0161´;
char c=´\033´;
char c=´\1b´;
```

As sequências de escape para caracteres de não impressão são também válidas dentro de cadeias de caracteres (“strings”). Por exemplo:

```
main()
{
        printf ("%s", "OLA\n");
}
```

imprime no ecrã a palavra *OLA* e em seguida há uma mudança de linha provocada pelo carácter '*n*'.

Uma constante também pode ser uma cadeia de caracteres, ou melhor, um "array" (vector) de caracteres em que o último elemento é o carácter nulo (i.e., carácter cujo código ASCII é 0). Por exemplo para a constante:

"ARRAY" string de caracteres

a sua representação interna seria o conjunto de caracteres, 'A' 'R' 'R' 'A' 'Y' '\0'.

Exemplo:

```
#include <stdio.h>

main()
{
        int c;

        c = 0xa + 010 + 21;          /* 0xa é 10 em decimal e 010 é 8 em decimal */
        printf("0xa + 010 + 21 = %x em hexadecimal\n",c);
        printf("0xa + 010 + 21 = %o em octal\n",c);
        printf("0xa + 010 + 21 = %d em decimal\n",c);
        printf("Impressao da string %s\n","ARRAY");
}
```

O resultado do programa anterior consistiria em:

```
0xa + 010 + 21 = 27 em hexadecimal
0xa + 010 + 21 = 47 em octal
0xa + 010 + 21 = 39 em decimal
Impressao da string ARRAY
```

Note-se a utilização no *printf* de *%x %o %d* para impressão de inteiros respectivamente em hexadecimal, octal e decimal, e de *%s* para impressão da “string”.

Exercício: Escreva um programa que permita visualizar a representação hexadecimal dos seguintes valores em decimal: 0, 10, 15, 16, 29, 30.

2.6. Operadores

Os operadores em C podem dividir-se nas três classes:

aritméticos,
relacionais e lógicos
bit a bit (binários)

Consoante os operadores envolvam um, dois ou três operandos, classificam-se em unários, binários e ternários.

2.6.1. Operadores Aritméticos

Os operadores aritméticos são:

+	adição
-	subtracção
*	multiplicação
/	divisão
%	módulo (resto da divisão de inteiros)

O operador % não pode ser aplicado a números de vírgula flutuante. Na divisão de inteiros a parte fraccionária é sempre truncada. Por exemplo de 5 / 2 resultaria 2.
Em C qualquer operador aritmético pode ser combinado com a atribuição, o que significa que:

```
x += 2      <=>     x = x + 2
a /= b + c  <=>     a = a / (b + c)
```

2.6.2. Operadores Incremento e Decremento

Os operadores incremento e decremento, ++ e -- respectivamente, podem ser usados tanto na forma prefixa como na forma sufixa. O operador ++ soma uma unidade à variável, enquanto o operador -- subtrai, como se pode ver nos casos que se seguem:

```
x++ ;          <=>   x = x + 1;
y = x++;       <=>   y = x;
                     x = x + 1;
y = ++x;       <=>   x = x + 1;
                     y = x ;
```

Como se constata, na forma prefixa acontece primeiro o incremento e só depois a atribuição, enquanto na sufixa sucede o contrário.

2.6.3. Operadores relacionais e Operadores lógicos

Estes operadores permitem a comparação de expressões. No caso de a relação ser verdadeira resulta o valor 1 (verdade), no caso de a relação ser falsa resulta o valor 0.

Relacionais:

x < y	menor que
x <= y	menor ou igual
x > y	maior que
x >= y	maior ou igual
x == y	comparar igualdade
x != y	comparar diferença

Lógicos:

&&	e (and)
||	ou (or)
!	negação (not)

Perigo! = = não é o mesmo que =

O operador unário de negação '!' converte o verdadeiro, de valor numérico 1, em falso, de valor numérico 0. No caso de ser aplicado a falso, converte o falso , de valor 0, em verdadeiro, de valor 1.

Exemplo:

```
if ((x != y) && (a > b ))
        - - n;
```

Implicitamente é avaliada primeiro a condição *x!=y* , e depois se, e só se, a esta tiver resultado lógico verdadeiro, será avaliada a segunda condição *a>b*. Se ambas forem verdadeiras, então será executada a instrução *--n*.
Neste caso, para o compilador ter ou não ter parêntesis nas condições é exactamente igual. O interesse é tornar a expressão mais clara. Noutras situações poderá ser o de alterar as precedências.

2.6.4. Operador Condicional

O operador condicional é um operador ternário usado em C. O valor da expressão:

expr0 ? expr1 : expr2

é o valor de *expr1* se *expr0* for verdadeira ou o valor de *expr2* se *expr0* for falsa.

Exemplo: A instrução

```
x = a > b ? a : b;
```

é exactamente equivalente à seguinte instrução :

```
if ( a > b )
        x = a;
else
        x = b;
```

2.6.5. Operadores Bit a Bit

Os operadores bit a bit (binários) permitem, como o nome o diz, operações ao nível dos bits. São seis os operadores bit a bit, os quais só poderão ser aplicados a expressões do tipo inteiro :

~	complementa bits (inverte o estado dos bits)
<<	desloca para a esquerda os bits
>>	desloca para a direita os bits
&	conjunção
^	ou exclusivo
|	disjunção

Exemplos: Supor duas variáveis *x* e *y* do tipo char, cuja representação binária é:

x = 10111011
y = 10101010

Os resultados de expressões envolvendo as várias operações são (as operações são executadas sobre os valores originais de x e y):

~x	01000100
x << 2	11101100
y >> 2	00101010
x & y	10101010
x | y	10111011
x ^ y	00010001

Outros exemplos de carácter mais prático :

a = b & 037	atribuir a *a* os 5 bits de menor ordem de *b*
a |= 0200	o 8º bit de *a* é posto a 1
a &= 017	limpa todos os bits *a* excepto os 4 finais
a ^= 077	reversa os últimos 6 bits
a <<= 2	*a* é multiplicado por 4
a =~ b	*a* fica complementar a *b*
a = b & 1	*a* será 1 se o primeiro bit de *b* estiver ligado

O operador << equivale a multiplicar por uma potência de 2 :

x *= 8 <=> x <<= 3 <=> x = x << 3

O operador >> equivale a dividir por uma potência de 2.

x /= 8 <=> x >>= 3 <=> x = x >> 3

Atenção ! Ter sempre cuidado em não confundir os operadores lógicos *&&* e **||,** com os operadores binários *&* e |.

Exercício1: Qual seria a saída do seguinte *printf* :

```
printf("6&1=%d     6&&1=%d\n",6&1,6&&1);
```

Exercício2: Considerando que duas variáveis inteiras *a* e *b*, possuem os valores 1 e 2 respectivamente, diga qual o resultado das seguintes operações:

```
a  &  b
a  |  b
a  ^  b
a  <<= b
b  >>= a
```

2.7. Precedências

O conhecimento da ordem de precedência dos vários operadores é fundamental para que se possam construir expressões correctamente. A tabela seguinte lista as precedências por ordem decrescente bem como o sentido de avaliação (associatividade). Ter em atenção que alguns dos operadores indicados na tabela ainda não foram discutidos.

Operadores	Associatividade
( ) [ ] -> .	esquerda para a direita
! ~ ++ -- + - * & (tipo) sizeof	direita para a esquerda
* / %	esquerda para a direita
+ -	esquerda para a direita
<< >>	esquerda para a direita
< <= > >=	esquerda para a direita
== !=	esquerda para a direita
&	esquerda para a direita
^	esquerda para a direita
|	esquerda para a direita
&&	esquerda para a direita
||	esquerda para a direita
?:	direita para a esquerda
= += -= *= /= %= &= ^= |= <<= >>=	direita para a esquerda
,	esquerda para a direita

Tabela 2.1 Tabela das precedências dos operadores e sentido de associatividade.

2.8. Conversão de Tipos

Por vezes surge a necessidade de trabalhar operandos cujos tipos de dados diferem uns dos outros. Nestas situações há que proceder à conversão de tipos, isto é conversão desses diferentes tipos num tipo comum, de forma a possibilitar a resolução das expressões. As conversões caem em três classes: *automática*, *requerida pela expressão* e *forçada*.

i) *Automática*: Feita automaticamente pelo compilador sempre que a variável seja do tipo *char*, *short* ou *float*, não existindo possibilidade de perda de informação. As conversões automáticas são feitas do seguinte modo:

char —> int
short —> int
float —> double

ii) *Requerida pela expressão*: Quando há mistura de tipos, sendo necessária para a resolução da expressão.

Exemplo:

```
int x;
double y;
x = x + y ;          /* conversão num double seguida de */
                     /* uma conversão num inteiro */
```

a conversão para inteiro faz com que o resultado seja truncado.

iii) *Forçada* ("cast"): Este tipo de conversão pode ser requerida pelo utilizador sempre que este a considere necessária, e faz-se precedendo a variável pelo novo tipo

entre parêntesis. Ter em atenção que a conversão para tipos menores pode truncar a variável. Forma geral:

(tipo de dado) expressão;

Exemplo: A seguinte função devolve o resto da divisão de duas variáveis de vírgula flutuante.

```
double mod(double a, double b)
{
        double temp;

        temp = (int) (a/b);
        temp *= b;
        return (a - temp);
}
```

Neste exemplo o "*cast"* tornou-se necessário para que o resultado da divisão fosse um inteiro, perdendo-se a parte fraccionária.

2.9. Typedef

Se um tipo de dado é precedido pela palavra *typedef* então o identificador na declaração é entendido pelo compilador como um novo nome para esse tipo de dado.

Exemplo: Programa que calcula a área de um rectângulo. O resultado (área) é guardado numa variável do tipo *AREA*, definido usando-se *typedef*. Assim em todas as declarações de variáveis que representem áreas, poder-se-ia usar ÁREA em lugar de *float*.

```
/* Programa area.c */

#include <stdio.h>

typedef float AREA;

main()
{
        AREA x;
        float lado1, lado2;

        printf("\nIntroduza os valores dos lados: ");
        scanf("%f%f", &lado1, &lado2); /* scanf  lê dados do teclado com o tipo indicado
                                          entre aspas. Neste caso 2 floats %f%f */
        x = lado1*lado2;
        printf("A área é igual a  %f",x);    /* %f indica um float para escrita */
}
```

Ter em atenção que em última instância a variável *x* é um *float*, como se prova pela formatação (com %f) utilizada no *printf*.

A utilização de *typedef* é mais frequente com tipos de dados compostos, como se poderá ver no capítulo dedicado a esses tipos de dados.

No exemplo anterior foi utilizada a função *scanf()* com dois ou mais argumentos. O primeiro argumento, entre aspas (do tipo "string"), indica os tipos dos dados a ler para as variáveis que ocupam os argumentos seguintes.

2.10. Exercícios

1. Escrever um programa que leia três inteiros correspondentes a uma hora em horas minutos e segundos, a converta em segundos, e imprima o resultado. Supondo as seguintes variáveis para as horas minutos e segundos, a leitura poderia ser:

```
int horas, minutos, segundos;
long int result_em_segundos;
. . .
scanf("%d%d%d", &horas, &min, &seg);          /* notar mais uma vez o operador & */
```

2. Escreva um programa que efectue a operação inversa da pedida no exercício anterior.

3. INSTRUÇÕES DE CONTROLE DE SEQUÊNCIA

3.1. Introdução

A linguagem C, como já foi dito, apresenta características de uma linguagem estruturada. Assim sendo, possui instruções de controle de sequência, que permitem especificar a ordem pela qual são executadas as instruções de um programa. As instruções de controle de sequência são de dois tipos: condicionais (*if* e *switch*) e de ciclo (*while*, *do/while* e *for*). Ter em atenção que quando se tem mais do que uma instrução (instrução composta) estas deverão ser colocadas entre chavetas, como já foi afirmado anteriormente.
Existem duas instruções que permitem alterar a sequência normal de execução dos ciclos: a instrução *break* e a instrução *continue. A* instrução *break* interrompe definitivamente a execução do ciclo, enquanto que, a instrução *continue* interrompe a execução de uma iteração saltando para o início da iteração seguinte. A instrução *break* é também usada na instrução *switch*.

3.2. Instrução IF

Esta é a instrução mais simples para controle de sequência. O seu funcionamento é o seguinte: faz-se a avaliação de uma *expressão*, e caso ela seja verdadeira é executado um primeiro bloco contendo uma ou mais instruções, em contrário, será executado se existir, um segundo bloco iniciado pela palavra *else*. Há duas formas de utilização:

```
        if (expressão)
        instrução;
ou
        if (expressão)
        instrução;
        else
        instrução;
```

Exemplo1: Neste exemplo, a função *idade*, verifica em que intervalo se situa uma idade *i* dada como argumento. No caso de *i* ser maior do que 70 imprime "Idoso", senão se for maior do que 30 imprime "Adulto", senão imprime "Jovem".

```
void idade ( int i)          /* void antes do nome da função para significar */
{                            /* que esta não retorna qualquer valor. */
        if ( i > 70)
                printf("Idoso");
        else    if ( i > 30)
                        printf("Adulto");
                else    printf("Jovem");
}
```

Exemplo2: Escreva um programa que verifique se um ano, lido do teclado, é ou não bissexto.

Resolução: Um ano é bissexto se for múltiplo de 400, ou então se for múltiplo de 4 mas não de 100. Relembra-se que, ser múltiplo de, é o mesmo que ser divisível por. Assim o programa será:

```
/* Programa bissexto.c */

#include <stdio.h>

main()
{
        int ano;

        printf("\n Qual o ano que pretende testar: ");
        scanf("%d", &ano);
        /* ano % valor == 0 -  Testa se ano é divisível por valor (resto 0) */
        if ( ano % 400 == 0 || (ano % 4 == 0 && ano % 100 != 0) )
                printf("\nO ano de %d É bissexto!\n", ano);
        else    printf("\nO ano de %d NÃO é bissexto\n", ano);
}
```

Exercício: Escreva um programa que dados dois valores e um operando, imprima o resultado da respectiva operação. As operações possíveis são a adição, a subtracção, a multiplicação e a divisão, a que correspondem os caracteres +, -, * e / respectivamente.

3.3. Instrução SWITCH

Esta instrução avalia uma expressão do tipo inteiro ou carácter, e depois compara-a com constantes de tipo inteiro ou carácter, permitindo assim várias decisões (vias). Esta instrução usa-se normalmente em situações em que existam várias possibilidades de decisão.

Sintaxe:

```
switch (expressão)
{
        case constante1 : instrução;
                          break;
        case constante1 : instrução;
                          break;
        ...
                  default : instrução;
                          break ;
}
```

Exemplo: A função seguinte verifica se uma cor, dada como argumento, pertence ao sistema RGB e imprime mensagens de acordo com a situação.

```
void sistema_cor(int cor)
{
        switch cor {
                case 'R': case 'r': printf("\ncor vermelha");
                                    break;
                case 'G': case 'g': printf("\ncor verde");
                                    break;
                case 'B': case 'b': printf("\ncor azul");
                                    break;
                default: printf("\ncor inválida");
                                    break;
        }
}
```

Como se pode ver neste exemplo vários cases podem conduzir à execução da mesma instrução.

A instrução *break* permite sair (quebrar) do *switch*. Caso esta não seja incluída todas as instruções abaixo do *case* seleccionado também seriam executadas até que surgisse um *break* ou surgisse a chaveta que termina o *switch*. No caso do exemplo, supondo que a cor era 'G', se não tivesse a instrução *break*, as instruções correspondentes aos cases seguintes e *default* seriam executadas.
A instrução *default* é opcional e é avaliada quando nenhuma das constantes coincide com o resultado da expressão.

Exercício: Resolver o exercício proposto anteriormente para a instrução *if*, mas desta vez usando um *switch*.

3.4. Ciclo WHILE

Esta instrução é usada quando se pretende que um bloco de instruções seja executado enquanto uma determinada condição for verdadeira. O teste da condição é feito no início de cada ciclo, sendo possível que as instruções não sejam executadas uma única vez.

O ciclo *while* sem nenhuma condição é considerado sempre verdadeiro, dando origem a um ciclo infinito ("do forever").

Sintaxe:

```
while (expressão)
    instrução simples ou composta
```

A instrução é executada enquanto a expressão é verdadeira.

Exemplo1: O programa deste exemplo (*contach.c*), conta os caracteres inseridos através do teclado (ou obtidos pelo redireccionamento de um ficheiro para o programa), até que o carácter lido seja igual a *EOF*. *EOF* é uma constante definida no ficheiro *stdio.h* e que representa o carácter de fim de ficheiro. Se a entrada de dados se fizer pelo teclado o utilizador terá de digitar o carácter de fim de ficheiro válido para o seu sistema operativo. No caso do DOS será Ctrl Z. A função utilizada para a leitura de um carácter é a *scanf()* com *%c*. Caso se pretendesse contar os caracteres de um ficheiro a utilização consistiria em: *contach<fich_inp*, em que obviamente o número de caracteres lidos seria o tamanho do ficheiro.

```
/* Programa  contach.c */

#include <stdio.h>

main()
{
        long caracteres_lidos=0;
        char carácter;

        scanf("%c", &carácter);
        while( caracter!=EOF) {
                caracteres_lidos++;
                scanf("%c", &carácter);
        }
        printf("\nForam lidos %ld caracteres!\n", caracteres_lidos);
}
```

Exemplo2: Programa que testa a função *pot_int_pos* que calcula a potência inteira positiva de um número.

```
/* Programa pot.c */

#include <stdio.h>

main()
{
        int base, exp;
        printf("Introduza uma base e um expoente (inteiros >0): ");
        scanf("%d%d", &base, &exp);
        printf("%d levantado a %d = %d", base, exp, pot_int_pos(base,exp));
}
```

```
pot_int_pos(int base, int exp)
{
        int result,i;

        result=i=1;                 /* Inicializa result e i a 1 */
        while ( i <=exp) /* Enquanto i menor ou igual ao expoente */
         result=result*base;
        i++;
        return result;
}
```

Exercício1: Altere o exercício anterior de modo a que os números introduzidos para a base e expoente sejam apenas aceites se forem ambos positivos.

Exercício2: Proceda às alterações necessárias para que a função anterior também calcule potência inteiras de expoente negativo.

3.5. Ciclo DO - WHILE

Esta construção é idêntica ao ciclo *while*. Enquanto que no ciclo *while* o teste da condição é feito no início de cada iteração, no *do-while* o teste é feito no fim. Assim, mesmo que a condição seja sempre falsa o bloco de instruções é executado pelo menos uma vez (a primeira).

Sintaxe:

```
do
        instrução simples ou composta
while (expressão);
```

Exemplo1: Programa que testa uma função que apenas lê dígitos. A função lê caracteres do teclado até que um seja dígito. O dígito é depois retornado.

```
/* Programa ledigito.c */

#include <stdio.h>

main()
{
        char digito;

        digito=le_digito();
        printf("O digito lido foi: %c\n",digito);
}
```

```
char le_digito()
{
        char c;
        do      /* Como o ciclo contém apenas uma instrução não
                necessita de chavetas. */
                scanf("%c",&c);
        while (c<'0' || c>'9');
        return c;
}
```

Como se pode ver no exemplo, neste caso é preferível que a avaliação da expressão ocorra depois da instrução, de leitura de um carácter. Só quando o carácter lido corresponde a um dígito (0-9), é que o ciclo termina, sendo retornado o respectivo dígito.

Exercício1: Utilizando a resolução do exercício proposto anteriormente para a instrução *switch*, introduzir as alterações necessárias para que o programa permita resolver operações enquanto o utilizador o pretender. Para tal deverá fazer uso do ciclo *do/while*.

3.6. Ciclo FOR

A instrução *for* é aquela que habitualmente mais se utiliza na construção de ciclos em C. Isto deve-se ao seu poder e flexibilidade, como poderá ser observado. Identicamente ao *while*, a condição de teste é feita no início do ciclo.
Sintaxe:

```
for (início; teste; incremento)
        instrução
```

Gramaticalmente, os três componentes são expressões. A *início* correspondem expressões normalmente de inicialização da(s) variável(is) de controle do ciclo (se forem mais do que uma as expressões deverão ser separadas por vírgula). A *teste* correspondem as condições de execução de cada ciclo. Em *incremento* são normalmente efectuados os incrementos das variáveis de ciclo (se forem mais do que uma as expressões deverão ser separadas por vírgula).
Na primeira iteração do ciclo *for* é executado *início*, depois *teste*, em seguida, se *teste* resultar verdadeiro é executado instrução (composto por uma ou mais instruções), e por último *incremento*. Nas restantes iterações do *for*, *início* não é executado.

O ciclo *for* pode ser convertido facilmente num ciclo *while*, com o seguinte aspecto:

```
início;
while (teste)
{
        instrução;
        incremento;
}
```

O ciclo *for* é preferível quando se tem um início e incremento simples. Para um entendimento mais claro observe-se o exemplo seguinte:

Exemplo1: Programa que executa um ciclo *for* duas vezes,

```
#include <stdio.h>

main()
{
        int i;

        printf("i = %d\n", i);
}
```

e cuja saída será:
```
0
1
```

O ciclo *for* foi executado enquanto *i* menor do que *2*.

Exercício: Reescreva o programa do exemplo 2 da secção 3.3 para o ciclo *while* (de cálculo de uma potência), usando desta vez um ciclo *for*.

Exemplo2: O programa seguinte converte em minúsculas e escreve no monitor as letras lidas do teclado, até que seja lido o carácter de fim de ficheiro.

```
#include <stdio.h>

main()
{
        char carácter;
        long int caracteres_lidos=0;

        scanf("%c", &carácter);
        while( caracter!=EOF) {                 /* Enquanto carácter lido diferente de
                caracteres_lidos++;                                     EOF executa ciclo */
                if(caracter>='A' && caracter<='Z')
                        caracter=caracter - 'A' + 'a';
                printf("%c",carácter);
                scanf("%c", &carácter);
        }
        printf("\nForam lidos %ld caracteres!\n", caracteres_lidos);
}
```

3.7. Exercícios

1. Converta o seguinte algoritmo em pseudo-código, que calcula o maior e o menor de 3 números inteiros, para C. O programa determina o maior e o menor entre apenas dois dos valores. Posteriormente determina o máximo e o mínimo entre o terceiro valor e o maior e o menor dos dois anteriores.

```
Prog MaiorMenor
        Ler(v1,v2,v3)
        Se v1>v2
        Então   max <- v1
                min <- v2
        Senão   max <- v2
                min <. v1
        Fse
        Se v3>max
        Então   max <- v3
        Senão   Se v3<min
                Então   min <- v3
                Fse
        Fse
        Escrever('O maior =', max, ' e o menor =', min)
Fprog
```

2. Altere o programa anterior de modo que determine quais os dois maiores e os dois menores de quatro números.

3. Escreva um programa que calcula a média de *n* notas lidas do teclado.

4. Escreva um programa que leia um número e calcule o seu factorial.

Resolução: O cálculo do factorial de um número consiste em multiplicar o número por (número - 1), em seguida por (número - 2) e assim sucessivamente até 1. O que implica um ciclo para esses produtos sucessivos, e uma variável que recebe os valores desde (número -1) até 1.

```
/* Programa fatorial.c */
#include <stdio.h>

long factorial(int num)          /* A função retorna um inteiro long */
{
        long result=1;

        if (num>0)               /* equivalente à instrução:    num=num-1; */
           num--;

        /* Como num já foi decrementado, o primeiro produto vai ser entre num e
        o número imediatamente abaixo deste. Notar ainda a inexistência da
        primeira expressão no ciclo for. */

        for( ; num>1; num--)
                result*=num;
        return (result);
}

main()
{
        int numero;

        printf("\nCALCULO DO FACTORIAL\n");
        printf("\nIntroduza um número: ");
        scanf("%d", &numero);
        printf("%d! == %ld\n", factorial(num));
}
```

5. Escreva um programa que após ler três inteiros referentes ao dia, mês e ano de uma data, bem como a duração de uma tarefa em dias, determine qual a data de conclusão dessa tarefa.

4. OPERAÇÕES DE ENTRADA / SAÍDA

4.1. Introdução

Nos capítulos anteriores utilizaram-se algumas funções de entrada e saída de dados que, só agora serão explicadas com mais pormenor. Estas facilidades, não são intrínsecas à linguagem, tendo sido desenvolvidas à parte e encontram-se em bibliotecas (conjunto de ficheiros fonte compilados e agrupados num ficheiro). As funções básicas de "input" (entrada) e "output" (saída) disponíveis em todas as implementações de C constituem a "biblioteca de rotinas padrão" ("Standard library routines"). Várias declarações e macros necessárias a estas rotinas são incluídas num programa através da linha

include <stdio.h>[a]

4.2. Saída Formatada

As rotinas de "output" formatado que se vão estudar são:

printf (s_control, arg1, arg2, ...): escreve a "string" de controle *s_control* no terminal. Os argumentos serão introduzidos na "string" de controle de acordo com especificações nelas existentes.
sprintf ((buf, s_control, arg1, arg2, ...): o mesmo que *printf* excepto que a saída é colocada num "buffer" (i.e., um vector) especificado por *buf*.

4.2.1. Especificações da Saída Formatada

As especificações do formato para *printf* começam com um carácter % e terminam com um carácter de conversão. Existem várias opções e caracteres de conversão possíveis. Os caracteres de conversão, e as resultantes interpretações dos correspondentes argumentos são os seguintes:

c: um simples carácter.

---

[a]Alguns dos compiladores actuais, incluem automaticamente as funções da standard library, não necessitando desta linha.

d: inteiro notação decimal.
o: inteiro notação octal (não esquecer o zero do número que deve ser incluído explicitamente).
x: inteiro em notação hexadecimal; não esquecer o 0x do número
u: inteiro sem sinal em notação decimal.
s: vector de caracteres: os caracteres são imprimidos em sucessão até o carácter nulo ser alcançado.
f: float ou double, imprimido em notação decimal de acordo com a especificação opcional descrita abaixo.
e: float ou double imprimido em notação científica de acordo com a especificação opcional descrita abaixo.
g: float ou double, imprimindo como um %e ou %f.

Qualquer outro carácter é imprimido literalmente; por exemplo, *printf("%%")* imprime o sinal de percentagem. Os números são escritos justificados à direita. As especificações podem opcionalmente incluir (na ordem seguinte):

-Um sinal menos que indica que o argumento deve ser justificado à esquerda no campo de saída.
-Uma "string" de dígitos que especifica a largura mínima do campo para este argumento. Se o argumento convertido tiver menos caracteres é encostado à esquerda (ou à direita se o argumento é um ente justificado à esquerda).
-Um ponto que separa a largura do campo da precisão.
-Uma "string" que especifica o nº máximo de caracteres a ser imprimidos da "string", ou (se foi imprimido um nº de vírgula flutuante) o nº de dígitos a ser imprimido à direita do ponto decimal. Se isto é omitido para nº de vírgula flutuante, a precisão por defeito é 6.
-O carácter *l* indica que o argumento é um *long*.

Exemplos: Saídas da impressão com *printf* da cadeia de caracteres "Instituto Politécnico":

```
/%16s/          /Instituto Politécnico/
/%-16s/         /Instituto Politécnico/
/%24s/          /  Instituto Politécnico/
/%-24s/         /Instituto Politécnico   /
/%24 16s/       /         Instituto Polité/
/%-24 16s/      /Instituto Polité        /
/%16s/          /Instituto Polité/
```

4.3. Entrada formatada

A rotina *scanf( )* de "input" formatado torna a entrada numa forma que é rigorosamente o inverso do *printf( )*. A sintaxe é:

```
scanf (s_control, arg1, arg2, ...);
```

onde *s_control* é uma cadeia de caracteres. Os dados de entrada são lidos e interpretados de acordo com a "string" de controle. Cada um dos restantes argumentos

deverá ser um endereço, a entrada é armazenada nesse endereço tendo em atenção as especificações de conversão da “string” de controle. Se se pretender ler um inteiro, a tendência será para escrever

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

tal, no entanto não funciona, porque os argumentos da *scanf* devem ser endereços. Assim, dever-se-á escrever,

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

em que *&* é o operador que permite obter o endereço da variável. Deve-se ter em atenção que a situação é distinta quando se pretende ler uma “string”, pois o nome da “string” é um endereço, como se discutirá no capítulo sobre "arrays". Um dos erros mais comuns com *scanf* é esquecer isto; fazer sempre com que todos os argumentos da função *scanf* sejam endereços. A função

```
sscanf (buf, s_control, arg1, arg2, ...)
```

funciona mais ou menos como a *scanf* excepto que lê dados de uma “string” *buf* em vez de os ler do teclado.
A “string” de controle das funções *scanf*, ou *sscanf* pode conter espaços brancos (os quais são ignorados), especificações de conversão, e caracteres vulgares.

4.3.1. Especificações da entrada Formatada

Especificações de conversão consistem no carácter %, no carácter opcional de supressão de atribuição *, num nº opcional especificando a largura máxima do campo, e no carácter de conversão. Os caracteres vulgares são esperados para combinar com o próximo carácter não branco da entrada. Se a supressão de atribuição é indicada com * então o próximo campo é "saltado". Os caracteres de conversão possíveis e as correspondentes interpretações da entrada são:

- d: inteiro decimal
- 0: inteiro octal
- x: inteiro hexadecimal
- c: carácter simples: neste caso o salto normal de espaços brancos é suprimido no vector de caracteres.
- s: neste caso o argumento deve ser o endereço de um vector de caracteres suficientemente grande para armazenar o “input” (incluindo o carácter nulo). Os caracteres são lidos até ser encontrado um espaço.
- f: número de vírgula flutuante, possivelmente incluindo o sinal e expoente..
- e: o mesmo que f.

Se os caracteres de conversão *d*, *o*, ou *x*, são precedidos por um *l*, o "input" correspondente é interpretado como um *long*. Se o carácter de conversão *e* ou *f* é

precedido por um *l*, então a entrada correspondente é interpretado como nº de vírgula flutuante de precisão dupla *(double*). O argumento correspondente deve ser um endereço de um *double* em vez de um *float*. Esquecer isto é outro erro comum quando se utiliza a função *scanf*.
A função *scanf* pára quando atinge o fim da "string" de controle ou quando a entrada falha ao tentar cumprir com a especificação de controle. Retorna como valor, o nº de campos de entrada que cumpriram e aos quais tenham sido atribuídos valores.

4.4. Outras Funções da Biblioteca Standard

Para além das funções de entrada/saída formatada já estudadas, existem outras funções também muito importantes. Estas funções também requerem a instrução de inclusão do ficheiro *stdio.h*.

Ler um carácter do "standard input"[a] (teclado):

```
int c;
c = getchar();
```

Esta função retorna um inteiro correspondente ao carácter entrado, ou o valor da constante EOF se atingiu o fim do ficheiro ou ocorreu um erro.

Escrever um carácter no "standard output" (ecrã):

```
putchar(c);
```

Escrever uma "string" na saída padrão:

```
char string[80];
puts(string);
```

Ler uma "string" da entrada padrão terminada com '\n':

```
gets(string);
```

A "string" lida é colocada em *string*. A função retorna o valor da constante *NULL* quando encontra o fim do ficheiro ou em caso de erro.

Exemplo1: O programa *meu_type.c* lê caracteres do teclado, ou de um ficheiro redireccionado para o programa, e escreve-os no "stantard output"[b], até que seja encontrado o carácter de fim de ficheiro. Caso se pretendesse ler dados de um ficheiro a utilização seria: *meu_type<fich_inp*. Para ler de um ficheiro e escrever noutro viria: *meu_type<fich_inp>fich_out*.

---

[a] O mesmo que entrada padrão, por defeito o teclado.

[b] O mesmo que saída padrão, por defeito o monitor.

```
/* Programa  meu_type.c */

#include <stdio.h>

main()
{
        int carácter;

        caracter=getchar();
        while(caracter!=EOF) {
                putchar(carácter);
                caracter=getchar();
        }
}
```

Exercício1: Altere o exemplo 1 da secção 3.3, substituindo a função *scanf()* pela *getchar()*.

4.5. Outras Funções

Todas as funções apresentadas têm em comum fazer parte da livraria standard, no entanto os compiladores oferecem outras funções. Estas dependem normalmente do sistema, e como tal exploram bastante bem as suas potencialidades. São apresentadas as duas funções seguintes, que existem na generalidade dos compiladores:

```
c = getche();
```

em que, *c* é um inteiro. Esta função permite ler um carácter do teclado, sem necessidade de em seguida se digitar a tecla *Return*. O carácter lido é mostrado (*"ecoado"*) no ecrã.

```
c = getch();
```

Idêntica à anterior, apenas com a diferença de não mostrar o carácter lido. O facto de estas funções não necessitarem que a tecla *Return* seja premida, faz com que seja de grande interesse a sua utilização. Normalmente, é necessário a inclusão da instrução,

```
#include <conio.h>
```

para a sua utilização.

Exemplo: É agora apresentada uma versão da função *le_digito()* utilizando a função getch(), para só seja visualizado o carácter quando se tratar de um digito.

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

le_digito()
{
        char c;
        do
                c=getch();
        while (c<'0' || c>'9');
        putchar(c);
        return c;
}
```

4.6. Exercícios

1. Escreva um programa que conte e imprima o número de dígitos, vogais e consoantes lidas do teclado.

5. FUNÇÕES

5.1. Introdução

Um programa em linguagem C pode ser composto por uma (função main) ou mais funções, distribuídas por um ou mais ficheiros. As funções são a única forma de se agruparem as instruções que se pretendem executar, constituindo assim uma unidade básica na programação em C. A linguagem C está feita de modo a que se construam facilmente funções muito eficientes, de tal forma que podem ser utilizadas em diferentes programas sem alterações.

5.2. Definição

Relembrando do primeiro capítulo, a forma geral da definição de uma função é:

```
classe_armazenamento tipo_dado nome (declarações_lista_de_parâmetros)
{
        declarações das variáveis locais
        instruções
}
```

Sempre que se pretenda que uma função retorne um valor, então deverá ser incluída a instrução *return*.
As variáveis da lista de parâmetros podem ser consideradas como variáveis locais, que são inicializadas com os valores dos argumentos da chamada da função. O tipo de dado que precede o nome da função é o tipo do objecto que a função retorna. Por exemplo, a definição *double raiz(double num)* significa uma função que retorna um *double*. Os tipos podem ser portanto os mesmos que os definidos para as variáveis.
Se o tipo de uma função não é declarado explicitamente, então o tipo retornado por defeito é *int*. E se algum dos parâmetros não tiver a declaração explícita do tipo, este é também assumido como *int*.
Uma função retorna automaticamente quando atinge o fim do corpo da função, a não ser que antes encontre a instrução *return*. A sintaxe da instrução *return*:

```
return (expressão);
```

Os parêntesis em torno da expressão são opcionais. O tipo da expressão deverá estar de acordo com o tipo da função.

Se uma função retorna um não inteiro então qualquer rotina que a chame deve declarar o tipo do objecto que ela retorna. Eis a distinção entre:

- *Definição da função* -> dá o código da função.
- *Declaração da função* -> somente informa o compilador do tipo do objecto retornado.

Exemplo: Neste exemplo é calculada a raiz quadrada de um número, pelo método de Newton. Em *main* é declarada a função *raiz()*, para que o compilador saiba qual o tipo de dado que ela retorna.

```
/* Programa raiz.c */

main( )
{
        float a, r, raiz( );        /* declaração da função raiz */
        printf ("\nIntroduza um número:");
        scanf ("%f",&a);
        r = raiz(a);
        printf ("A raiz quadrada de %f é %f",a,r);
}

/*      Função raiz quadrada através da iteração do método de Newton.*/
float raiz(float x)         /* definição da função raiz que retorna um float*/
{
        float y, z;

        y = x;
        do {
                z = y;
                y = (z + x/z)/2;
        } while (z != y);
        return y;
}
```

No contexto das funções existem dois conceitos importantes que por vezes dão origem a alguma confusão:

argumentos - lista de variáveis, constantes ou expressões colocados entre os parêntesis da função aquando da chamada desta. No exemplo anterior, na chamada *raiz(a)*, *a* é o argumento.

parâmetros - lista de declarações de variáveis aquando da definição da função. No exemplo anterior a lista de parâmetros da função *raiz* é apenas *float x*.

Muitas funções, não retornam qualquer valor. A forma de o declarar explicitamente é usando o tipo de dado *void*. Uma declaração simples deste tipo, pode ser:

```
void inverte(char str[ ])       /* função que não retorna qualquer valor */
{
        ...
```

Quanto à classe de armazenamento de uma função pode ser:

- static, significando que a função só é visível no ficheiro onde está definida.
- extern, significando que a função já foi definida em outro local ou ficheiro.
- não existir classe de armazenamento tratando-se portanto de uma função global.

5.3. Parâmetros de uma Função

Em C, os argumentos para as funções são passados via "chamada por valor", ao passo que em Pascal e Fortran são chamados via "chamada por referência". A passagem por valor significa que a uma função (mais concretamente aos seus parâmetros), são apenas passados os valores dos argumentos, enquanto na chamada por referência é como se fossem passadas as próprias variáveis.
Em Pascal, duas variáveis podem ser trocadas através da chamada ao seguinte procedimento:

```
procedure troca (var x,y: integer)
var         temp : integer
begin
            temp := x;
            x := y;
            y := temp
end
```

O equivalente em Fortran::

```
SUBROUTINE TROCA(X,Y)
INTEGER X,Y,TEMP
TEMP=X
X = Y
Y = TEMP
RETURN
END
```

O equivalente em C ,resultaria ineficaz:

```
troca(x,y)
int x,y;
{
        int temp;

        temp = x;
        x = y;
        y = temp;
}
```

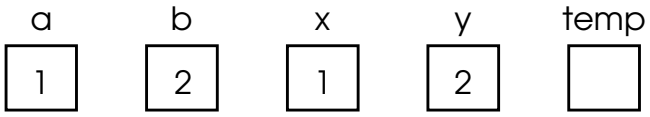
Esta rotina troca unicamente os valores das variáveis da função (na "stack"), e deixa os valores originais da rotina que chama inalteráveis. Supondo que *troca()* era invocada do seguinte modo:

```
int a=1, b=2;
troca(a,b);
```

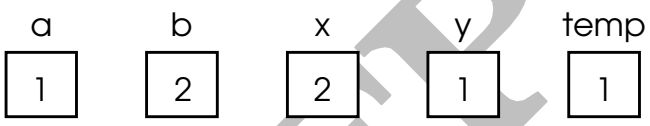
seriam trocados os valores de *x* e *y* da função mas manter-se-iam inalteráveis os de *a* e *b*, como a seguir se demonstra:

Com a execução da primeira instrução são criadas as duas variáveis *a* e *b*, e atribuídos respectivamente os valores 1 e 2.

Na segunda instrução, é chamada a função *troca()* que na sua execução cria as variáveis *x* e *y*, que recebem os valores 1 e 2 respectivamente de *a* e *b*. Em seguida é também criada a variável *t*. Passam-se agora a ter **cinco** variáveis:



Após a execução da função os valores das variáveis são os seguintes:



Como se pode observar *x* e *y* são variáveis distintas das dos argumentos (*a* e *b)*, limitando-se a receber os valores destas, como se queria demonstrar.
Para que na função se pudessem alterar os valores das variáveis da função que chama, haveria que passar os endereços dessas variáveis, e os parâmetros da função teriam de ser apontadores. Então a função *troca()* seria invocada do seguinte modo:

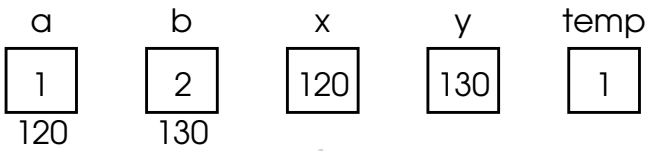
```
int a=1, b=2;
troca( &a, &b);
```

e teria o seguinte aspecto, com as variáveis *x* e *y* definidas como apontadores para inteiros (ou seja, variáveis que guardam endereços de posições de memória onde estão armazenados inteiros):

```
troca( int *x, int *y)
{
        int temp;
        temp = *x;
        *x = *y;
        *y = temp;
}
```

Quer isto dizer que *x* e *y* receberam os endereços de *a* e *b* respectivamente, e que através do operador de conteúdo de um endereço (*), podem aceder aos conteúdos desses endereços. Supondo que os endereços de *a* e *b* eram 120 e 130

respectivamente, então após a execução da função as variáveis apresentariam o seguinte aspecto:



5.4. Conversão de Tipos

O valor de uma expressão numa instrução *return* é convertido para o tipo da função através das regras de conversão de tipos de dados. Com efeito, um "cast" para o tipo apropriado é inserido automaticamente em qualquer instrução *return*.
Por exemplo, a função *atoi* (que converte uma "string" num inteiro) pode ser escrita em termos duma rotina de biblioteca *atof* (a qual converte uma "string" para um número de vírgula flutuante) como se segue:

```
atoi(char s[])
{
        double atof();
        return atof(s);
}
```

O valor produzido pela função *atof* é automaticamente convertido para inteiro na instrução *return*, em virtude da função *atoi* retornar um inteiro.

As únicas conversões de tipo efectuadas durante a avaliação de expressões numa lista de argumentos são as conversões de tipo *automático*. Assim, nas outras situações há que forçar a conversão, como no exemplo seguinte, em que a função *log* espera um argumento do tipo double:

```
int n;
double x, log();
x = log((double)n);
```

e em que portanto foi feito o "cast" da variável *n* para um *double*.

5.5. Argumentos da Linha de Comando

É possível em C passar valores para o programa através da linha de comando. Estes valores são passados obviamente à função *main*. Nesta função usam-se habitualmente 0 ou 2 parâmetros formais. Esses dois parâmetros formais da *main()* estão relacionados com os argumentos de linha de comando que são dados na chamada do programa. A maior parte dos programadores escolhe os nomes *argc* e *argv* para estes dois parâmetros:

```
main (int argc, char argv[][])
{
...
}
```

O primeiro parâmetro *argc* é o número de argumentos da linha de comando (isto é, palavras na linha de comando separadas por espaços brancos).
O segundo parâmetro *argv* é um vector de "strings" (cada "string" é um vector de caracteres) representando os argumentos da linha de comando.
Como o nome do próprio programa que é executado está em *argv[0]* então o *argc* é sempre no mínimo 1. O primeiro argumento 'verdadeiro' está em *argv[1]* e o último argumento 'verdadeiro' está em *argv[argc -1]*. *argv[argc]* possui normalmente um valor para distinção (por exemplo *NULL*). Compreender-se-á melhor após a leitura do capítulo 6 sobre vectores.

Exemplo 1: O seguinte exemplo imprime os argumentos do programa.

```
/* Programa printarg.c */

#include <stdio.h>

main (int argc, char argv[][])
{
        int i;

        for (i = l; i < argc; i++)
                /* argv[i] representa 1 argumento, mais concreta/e o seu endereço */
                printf ("Arg. %d é\t(%s)\n", i,argv[i]);
        printf("E o programa chama-se  %s\n",argv[0]);
}
```

Considerando a seguinte execução do programa: *printarg 1 2 ultimo*
O resultado seria:

```
Arg. 1 é (1)
Arg 2 é  (2)
Arg3 é           (ultimo)
E o programa chama-se  printarg.exe
```

5.6. Conclusão

Como conclusão, recapitulam-se em seguida alguns aspectos mais importantes sobre as funções:

- Permitem dividir o programa em partes funcionais mais pequenas.
- Podem residir em bibliotecas e/ou noutros ficheiros fonte.
- A comunicação do exterior para a função é feita através dos argumentos.
- A comunicação da função para o exterior é feita através de um valor de retorno, sendo este do mesmo tipo da função.
- Todas as funções têm um determinado tipo associado.
- A definição da função pode ser feita em qualquer ponto.

- A função pode invocar-se a si própria (recursividade).
- Uma função cujo tipo é antecedido por *static* só é visível no ficheiro aonde está definida.

5.7. Exercícios

1. Escreva um programa que calcule a seguinte série:

$$-\frac{x^1}{1!}+\frac{x^2}{3!}-\frac{x^3}{5!}+\frac{x^4}{7!} - \ldots$$

Para o cálculo do factorial e da potência poderá ser utilizada a função *factorial()* desenvolvida no capítulo 3. No mesmo capítulo encontra-se a função *pot_int_pos()*, mas que apenas calcula a potência de números inteiros.

2. Escreva um função que retorne o número de dígitos que compõem um número inteiro positivo passado como argumento da função, bem como um programa que a teste.

3. Escreva um função que retorne o número de dígitos que compõem a parte inteira de um número real positivo, passado como argumento da função, Escreva também um programa que teste a função.

6. VECTORES

6.1. Introdução

Supor, que se pretendia um programa para ler a média das notas de cada um dos cerca de 3000 alunos do ISEP, calcular a média das médias, e depois para cada aluno determinar o desvio da sua média relativamente à média das médias. Uma solução para guardar cada uma das médias, consistiria em definir 3000 variáveis, por exemplo:

float med_1, med_2, med_3, med_4, med_5, med_6, med_7, med_8, med_9, med_10, med_11, med_12, med_13, med_14, med_15, med_16, med_17, med_18, med_19, . . .

até *med_3000*, mais as instruções para a leitura das médias:

```
printf("\nIntroduza a média do aluno nº 1: ");
scanf("%f", & med_1 );
. . .
```

3000 vezes, o que se revela completamente impraticável. Seria preferível a possibilidade de definir as 3000 variáveis de uma só vez, por exemplo da seguinte forma:

```
float media[3000];
```

em que *media[0]* guardaria a média do aluno 1, *media[1]* a do aluno 2, e assim sucessivamente fazendo variar o valor do índice até 2999. Para melhorar o processo seria conveniente definir uma variável para índice, por exemplo:

```
int num;
```

Deste modo, para a leitura das 3000 médias, poder-se-ia utilizar um ciclo, como a seguir se ilustra:

```
for ( num=0; num<3000; num++ )  {
        printf("\nIntroduza a média do aluno nº %d: ", num);
        scanf("%f", & media[num] );
}
```

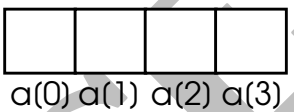
o que representaria uma melhoria extraordinária relativamente à primeira solução. Felizmente a generalidade das linguagens de programação fornecem este tipo de dados, chamados vectoresa, os quais se explicam nas próximas secções.

6.2. Vectores na Linguagem C

A linguagem C permite definir novos tipos de dados a partir dos existentes. Os vectores, são um conjunto de elementos do mesmo tipo, agrupados com o mesmo nome, e diferenciados através de um índice entre parêntesis rectos. Antes de mais, convém salientar que os vectores estão fortemente ligados aos apontadores, os quais não serão aqui abordados. Por este motivo, poderão surgir algumas dificuldades, que se tentarão minimizar.
Em C existem apenas vectores unidimensionais e o tamanho de um vector deve ser fixo e definido por uma constante na altura da compilação. No entanto, um elemento de um vector pode ser um objecto de qualquer tipo, inclusivé outro vector. Isto faz com que seja possível simular vectores multidimensionais.
A melhor forma de entender o modo de funcionamento dos vectores, é perceber a sua declaração. Por exemplo,

```
int a[4];
```

diz que *a* é uma variável do tipo vector de 4 elementos do tipo *int*. A sua representação será:



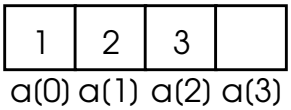
em que cada posição guarda um inteiro.
Como se pode ver, os elementos do vector ocupam posições de memória contíguas e o seu índice varia obrigatoriamente de 0 a 3. Ou seja, tem-se, para essas 4 posições os elementos

```
a[0],  a[1],  a[2],  a[3]
```

que são tratados como se fossem quatro variáveis distintas.
Atribuindo os valores 1,2 e 3 respectivamente, aos três primeiros elementos,

```
a[0] = 1;
a[1] = 2;
a[2] = 3;
```

virá:



---

[a] Neste texto é usado o também o termo array para significar vector.

Convém ainda saber que o nome de um vector é uma constante e representa o endereço da 1ª posição do vector. Isto é:

```
a == &a[0]      /* O operador & dá o endereço da variável */
                /* Neste caso dá o endereço de a[0] */
```

Supondo que o vector se iniciava na posição de memória com o endereço *100*, e que cada inteiro "gasta" 2 bytes, ter-se-ia:

```
a == 100
&a[0] == 100
&a[1] == 102
&a[2] == 104
&a[3] == 106
```

Em virtude de ser uma constante, não é possível atribuir valores ao nome do vector, como se ilustra no seguinte caso:

```
int a[4];
a = 2;
a[0] =2;
a[0]++;
a[1]=2*a[0];
```

A atribuição na segunda instrução é inválida. Também quando se passa um vector como argumento de uma função, na realidade o que é passado para a função é o seu endereço, como se viu com a função *scanf*.
A dimensão de um vector é determinado pelo produto das suas linhas pelas suas colunas. A seguinte declaração:

```
int ecra[25][80];
```

diz que *ecra* é um vector de 25 vectores de 80 elementos inteiros cada. Assim, sizeof(ecra) será 2000 (25*80).
A inicialização de um vector pode ser feita no momento da sua definição:

```
int a[4] = {1, 2, 3};
```

irá definir (criar) um vector de quatro inteiros e inicializar a[0] a 1, a[1] a 2 e a[2] a 3. No entanto a definição:

```
int a[] = {1,2,3};
```

irá definir um vector de apenas três elementos e inicializá-los de forma idêntica ao anterior. O vector é criado com apenas 3 posições em virtude de não ser explicitado entre parêntesis o número de elementos, e portanto este número será determinado pela lista de inicialização.
No caso de um vector ser multidimensional a inicialização obedece às mesmas regras. Por exemplo:

```
int m[][3] = {1, 2, 3, 11, 22, 33};
```

ou

```
int m[][3] = {{1, 2, 3},{11,22,33}};
```

em que se obtém os seguintes valores para cada elemento:

m[0][0] é igual a 1
m[0][1] é igual a 2
m[0][2] é igual a 3
m[1][0] é igual a 11

1	2	3
11	22	33

m[0][0] é igual a 11
m[1][1] é igual a 22
m[1][2] é igual a 33

Exemplo 1: O programa *media.c* lê 100 notas para um vector, e calcula a média das notas.

```
/* Programa media.c */

#include <stdio.h>

main()
{
        int i;
        float notas[100], soma;

        /* Ciclo que lê 100 notas e armazena-as no vector notas.*/
        for( i = 0; i < 100; i++) {
                printf("Nota %d=?",i+1);
                scanf("%f,&notas[i]);
        }
        /* A variável acumuladora soma inicializada a zero. */
        soma=0;
        /* Ciclo que soma as 100 notas. */
        for( i = 0; i < 100; i++)
                soma = soma + notas[i];
        printf("A média é =%f",soma/100);
}
```

Como cada elemento do vector *notas* é um inteiro, há necessidade na função *scanf* de lhe aplicar o operador *&* como acontece com qualquer outro inteiro.

Exercício 1: Altere o exemplo anterior de modo a que calcule também o desvio de cada valor relativamente à média, e os guarde num segundo vector. Determine também a maior e menor notas.

6.3. Vectores como Argumentos de Funções

Para melhor se compreender este assunto observe o seguinte exemplo:

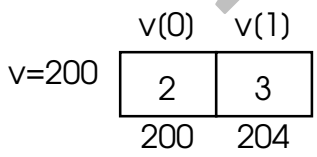
O seguinte programa testa uma função que troca os dois primeiros elementos de um vector dado como argumento.

```
#include <stdio.h>

void troca( );     /* Declara a função como não retornando qualquer valor */

main( )
{
        int v[2];

        v[0]=2;
        v[1]=3;
        troca( v );
        printf ("v[0] = %d,v[1] = %d", v[0],v[1]);
}

void troca(int x[])
{
        int t;

        t = x[0];
        x[0] = x[1];
        x[1] = t;
}
```

O resultado será v[0] = 3 e v[1] = 2. Neste exemplo é passado como argumento da função o nome do vector, que é o endereço da 1ª posição do vector. Assim *x* vai representar também um endereço o de *v[0]* que é o mesmo de *x[0]*, e portanto as alterações feitas na função terão repercussões nas variáveis da função que chama.

Na definição da função também seria possível e com os mesmos resultados, ter *int x[]* em lugar de *int x[2]*, sendo nesse caso *x* dimensionado automaticamente a *2*.

Supor que o endereço de v[0] é 200, então é porque v = 200 como mostra a figura:

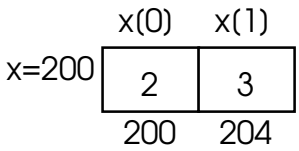


(Supondo uma máquina de inteiros com 4 bytes, v[1] virá 4 bytes à frente (204))

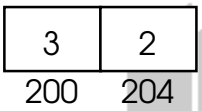
então a chamada da função

troca(v); <=> troca(200);

em que ao nome do vector *x* corresponderá o valor 200, ou seja, *x* será igual a 200.



o que implica que após as trocas ficará:



Estas posições de memória correspondem também a v[0] e a v[1],respectivamente.

Exercício 1: Supondo as seguintes alterações à função *main()*, em que o vector *v* passa a ser de quatro elementos:

```
main( )
{
        int v[4];

        v[2]=2;
        v[3]=3;
        troca( ? );
        printf ("v[0] = %d,v[1] = %d", v[2],v[3]);
}
```

pense como faria a chamada da função *troca()*, se pretendesse agora trocar os dois elementos *v[2]* e *v[3]* do vector.

Exercício 2: Escrever um função que some duas matrizes e coloque o resultado numa terceira.

Resolução:

```
soma_mat( int m1[][M_COL],int m2[][M_COL],int res[][M_COL], int lin,int col)
{
        int i, j;

        for (i=0; i < lin; i++)
                for (j=0; j < col; j++)
                        res[i][j] = m1[i][j] + m2[i][j];
}
```

M_COL é uma constante (as constantes serão estudadas no capítulo 8). *lin* e *col* são respectivamente o número de linhas e colunas das matrizes. No ciclo externo são percorridas todas as linhas, e para cada uma são percorridas todas as colunas.

6.4. "Strings" (Cadeias de caracteres)

Em C, não existe o tipo de dado “string”. Pode no entanto ser simulado através de um vector de caracteres. Por convenção, delimita-se a “string” com o carácter nulo (\0).

Isto é, coloca-se o carácter nulo na primeira posição não preenchida do vector. Assim, quando se define o tamanho do vector de caracteres para ser usado como “string” é necessário contar com mais uma posição.

Exemplo 1: Supor que se pretende armazenar uma palavra de até 10 caracteres numa “string”. Para tal seria necessário criar um vector de 11 posições (10 para a palavra, mais uma para o carácter nulo):

```
char pal[11];
```

A standard library fornece um conjunto de funções para a manipulação de "strings". Estas funções pressupõem que as “strings”s que lhes são passadas são terminadas com o carácter nulo.

- int strlen (char s[]);

devolve o comprimento da “string” *s*. Exemplo:

```
char s[] = "OLA";
printf ("Comprimento Da String == %d",strlen(s));
```

A definição anterior além de inicializar o vector com a “string”, também o dimensiona como um vector de quatro caracteres, como mostra a figura seguinte:

'O'	'L'	'Á'	'\0'

Quanto à saída do *printf*, ela seria: Comprimento Da String == 3.

- strcpy (char destino[], char origem[]);

copia *origem* para *destino*. Exemplo:

```
char destino[] = "ADEUS", origem[] = "OLA";
strcpy(destino,origem);
printf ("%s",destino);
```

a saída é OLA.

- strcat (char primeira[],char segunda[]);

concatena (acrescenta) a “string” *segunda* à “string” *primeira*.

- int strcmp (char s1[], char s2[]);

compara *s1* com *s2*, e devolve 0 se *s1* igual a *s2*, > 0 se *s1*> *s2* e < 0 se *s1*<*s2*.

Uma lista completa destas funções pode ser encontrada nos manuais de qualquer compilador de C. Para a utilização destas funções, é requerida a linha

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

Estas funções podem também ser facilmente construídas pelo programador.

Exemplo 2: Neste exemplo é apresentada uma possível versão da função *strlen.*

```
strlen (char str[])
{
        int conta = 0;

        while (str[conta] != '\0')
                conta++;
        return conta;
}
```

O vector é percorrido até ser encontrado o carácter nulo. Por cada posição percorrida *conta* é incrementado.

Exemplo 3: Neste exemplo é apresentada uma versão simplificada da função *itoa*, em que os parâmetros representam o número a converter e a "'string"' que conterá o número convertido.
A conversão de um número numa "string", passa pela repetição de dois passos, da forma que se segue:

Passo 1 - Leitura de um digito do número.
Passo 2 - Acrescentar o dígito à "string".
Passo 3 - Repetir os passos anteriores até que todos os dígitos que constituem o número tenham sido atribuídos à "string".

O primeiro passo é conseguido obtendo-se o resto da divisão do número por 10, enquanto o segundo consiste apenas em atribuir o dígito a uma nova posição do vector ("string"). Cada vez que se lê um dígito, esse dígito é retirado ao número. Assim o ciclo termina quando o número for igual a zero.

```
i_to_a(int num, char str[])
{
        int i=0;

        do {
                str[i]=num%10;
                i++;
                num/=10;
        } while (num>0);
}
```

Exercício 1: Reescreva o exemplo 1 da secção 4.4, usando desta vez a função *gets()*. Esta versão poderá não funcionar quando o input vier de um ficheiro executável, pelo facto de estes não estarem organizados em linhas, e portanto não possuírem os respectivos caracteres '\n'.

6.5. Ordenação e Pesquisa

A ordenação de vectores e a pesquisa de um dado elemento num vector, são operações muito comuns em programação. Vão ser aqui abordados dois métodos para

cada uma das operações. Por uma questão de simplificação serão utilizados vectores de números inteiros. No entanto estes métodos poder-se-iam adaptar facilmente a vectores de outro tipo de dados.

6.5.1. Ordenação por Selecção

O algoritmo do método de ordenação por selecção consiste em seleccionar repetidamente o menor elemento dos que ainda não foram tratados, daí o nome do método. Pretendendo-se uma ordenação por ordem crescente, primeiro selecciona-se o menor elemento do vector e faz-se a sua troca com o elemento na primeira posição do vector, em seguida selecciona-se o segundo menor elemento e faz-se a sua troca com o elemento na segunda posição do vector, repetindo-se o processo até que todo o vector fique ordenado.

Em seguida apresenta-se a versão em C de uma função que implementa este método e onde *vec* é o vector a ordenar e *num_elem* o número de elementos do vector.

```
void ord_selec(int vec[], int num_elem)
{
        int i,j,min,temp;

        for(i=0; i<num_elem-1;i++) {
                min=i;
                for(j=i+1; j<num_elem; j++)
                        if (vec[j]<vec[min])
                                min=j;
                temp=vec[min];
                vec[min]=vec[i];
                vec[i]=temp;
        }
}
```

Este método é bastante eficiente para pequenos e médios vectores.

6.5.2. Ordenação por Inserção

Na ordenação por inserção considera-se um novo elemento de cada vez, e este é inserido no seu devido lugar no vector, entre os elementos já considerados, tendo-se o cuidado de os manter ordenados. Não se trata de um método que ordena um vector existente, mas antes que insere ordenadamente elementos num vector. Em seguida é apresentado o algoritmo da respectiva função. Para maior facilidade na conversão para a linguagem C considere-se que o vector se inicia na posição zero.

0. Considerar como parâmetros: o vector, o novo elemento e uma variável que indique o número de elementos do vector.
1. Testar se vector existe. Isto ,é se o seu número de elementos é maior ou igual que zero.
   1.1 Testar se vector vazio.

1.1.1 Se sim então colocar o novo elemento na primeira posição.
1.1.2 Se não, puxar todos os elementos maiores que novo uma posição acima e colocar o novo elemento na sua posição. Isto é:
1.1.2.1 Atribuir à posição índice i o elemento da posição índice i-1.
1.1.2.2 Repetir 1.1.2.1 Enquanto não percorrer todo o vector (i>0) e o novo elemento for menor que o elemento na posição índice i.
1.1.2.3 Colocar o novo elemento na posição índice i.
1.2 Actualizar o número de elementos do vector.
2. Fim

Exercício 1: Escreva o programa em pseudo-código correspondente ao algoritmo anterior.

Exercício 2: Considerando o seguinte programa em pseudo-código, como uma possível solução do exercício anterior, altere-o de modo a que deixe de existir explicitamente a condição que testa se o vector está vazio.

```
Função ord_inserção( vec[], novo, num_elem)

[ Testa validade se vector existe ]
Se num_elem>=0
        [ Testa se vector vazio ]
        Se num_elem==0
        Então   [ Coloca o  novo elemento na primeira posição ]
                vec[num_elem]<-novo
        Senão   [ Puxa todos os elementos maiores q. novo uma posição acima ]
                i<-num_elem
                Enquanto i>0 e novo<vec[i] Fazer
                        vec[i]<-vec[i-1]
                        i<-i-1
                Fenq
                [ Coloca o novo elemento na sua posição ]
                vec[i]<-novo
        Fse
        [ actualiza o número de elementos do vector ]
        num_elem<-num_elem+1
Fse
Ffunc
```

6.5.3. Pesquisa Sequencial

Trata-se do método mais simples de pesquisa, e que consiste em pesquisar sequencial e exaustivamente um vector na procura de um dado valor. Uma função em C, que executasse este método poderia ter o seguinte aspecto:

```
int pesq_seq(int vec[], int valor, int num_elem)
{
        int i=0;

        while (i<num_elem && valor!=vec[i])
                i++;
        return (i);
}
```

Esta função, no caso de o valor pesquisado existir no vector retorna a posição onde foi encontrado. No caso de não existir retorna um *i* igual ao número de elementos do vector.

6.5.4. Pesquisa Binária

A pesquisa binária trabalha com vectores ordenados, e consiste em "dividir" a meio o vector a ser pesquisado. Isto é, determina o meio do vector e em seguida verifica se o valor pesquisado é menor que o elemento no meio do vector. Se sim, faz o mesmo na primeira metade do vector, se não, fá-lo na metade superior, e assim sucessivamente até que a pesquisa tenha sucesso ou que o vector não possa ser mais fraccionado.
A função *pesq_bin()* implementa este método. Se a pesquisa tiver sucesso a função retorna a posição onde foi encontrado o valor em questão. Caso a pesquise falhe é retornado um valor correspondente à da posição acima da última.

```
pesq_bin( int vec[], int valor, int num_elem)
{
        int i, ini, fim;

        ini=0; fim=num_elem-1;
        do {    /* "Divide" o vector a meio, isto é calcula o meio do vector. */
                i=(ini+fim)/2;
                /* Testa se valor se encontra na primeira metade */
                if( valor<vec[i] )
                        fim=i-1;
                else    ini=i+1;
        } while( valor!=vec[i] && ini<fim );
        if(valor==vec[i])
                return (i);
        else    return (num_elem);
}
```

6.6. Exercícios

1. Escreva uma função que multiplique duas matrizes dadas como argumentos. Faça um programa que teste a função. Para a leitura das matrizes desenvolva uma função.

2. Escreva uma função que dado como argumento um vector de inteiros o ordene por ordem crescente ou decrescente consoante o valor de uma opção dada também como argumento.

3. Desenvolva a sua versão das funções *strcpy*, *strcmp* e *strcat*. Desenvolva também um programa que as teste. As declarações e objectivos destas funções são os seguintes:

- strcpy(char destino[],char origem[]), copia origem para destino;

- strcmp(char str1[], char str2[]), compara str1 com str2. Retorna >0 se str1>str2, <0 se o inverso e zero se iguais;
- strcat(char str1[], char str2[]), concatena (acrescenta), str2 a str1.

4. Escreva uma função que dada uma "string" (como argumento) a converta num inteiro.

5. Desenvolva uma função que devolve o índice da primeira ocorrência de um dado carácter numa "string". O cabeçalho da função deverá ser:

encontra_char (char str[], char c);

6. Escreva uma função que dado um vector de "strings", o ordene. Cada "string" do vector é uma palavra. Escreva também o programa que lê o vector e testa a função.

7. [Exame] Escreva uma função que dadas duas "strings" pesquise a existência da segunda na primeira, e caso a encontre retorne o índice correspondente à sua posição. Se não existir a função deverá retornar -1. A posição a partir da qual se inicia a pesquisa é passada como argumento. Não deve usar a função já existente do C *strstr()*. A primeira linha da função deverá ser:

pesquisa_str (char primeira[], char segunda[], int pos_inicio);

Resolução: A resolução consiste em comparar o primeiro carácter da segunda "string" com o carácter da posição *inicio* na primeira "string". Se não forem iguais compara com o carácter que se encontrar na posição seguinte a *inicio*, e assim sucessivamente. Quando forem iguais compara o segundo da segunda com o seguinte da primeira e assim sucessivamente. Se quando deixarem de ser iguais ou terminar a segunda "string" o número de caracteres percorridos for igual ao tamanho da segunda "string" então é porque esta existe na primeira.
Note-se que para tudo isto são necessários dois ciclos. Um ciclo percorrerá a primeira "string" à procura nesta, de um carácter igual ao primeiro carácter da segunda. Quando acontecer essa igualdade, iniciar-se-á o outro ciclo, imbricado no anterior, e que irá comparando os caracteres que compõem as duas "strings".

```
strpstr(char str[],char pesq[],int inicio)
{
        int i,compstr,comppesq,j;
```

```
        /* Determina comprimento das duas strings */
        compstr=strlen(str);
        comppesq=strlen(pesq);
        /* Só inicia pesquisa se forem ambas de comprimento > 0 */
        if(comppesq!=0 && compstr!=0) {
                /* A pesquisa inicia-se a partir da posição inicio , e decorre enquanto não
chegar ao fim da primeira "string" e o comprimento da segunda for inferior ou igual ao que falta
percorrer.*/
                for(i=inicio; str[i] != '\0' && comppesq<= compstr - i ; i++) {
                        /* Neste ciclo percorre as duas ""strings" enquanto os caracteres
de ambas forem iguais e pesq não chega ao fim*/
                        for(j=0; pesq[j]!='\0' && str[i+j]==pesq[j]; j++);
                        /* Se o número de caracteres percorridos (iguais) for igual ao
tamanho da segunda string, então retorna a posição onde se iniciou na primeira a igualdade. */
                        if(j==comppesq)
                                return i;
                }
        }
        /* Só chega a este ponto se a pesq não existir em str. */
        return -1;
}
```

8. Escreva uma função que dada uma "string", como argumento, a inverta. Escreva também um programa que a teste.

7. O PRÉ-PROCESSADOR

7.1. Introdução

O préprocessador é o responsável pela primeira fase da compilação de um programa em C. Nesta fase são processadas todas as linhas começadas pelo caracter #. São várias as directivas de préprocessamento.

7.2. Include

Se um programa em C contém a linha

```
# include FILE
```

então o préprocessador irá substituir essa linha pelo conteúdo do ficheiro referido por FILE. Isto é feito antes de qualquer compilação. Isto é por vezes usado por ficheiros "header" contendo definições dos dados usados em ficheiros diferentes. 'FILE' pode ter uma das duas formas:

"nome-ficheiro" ficheiro do utilizador

<*nome-ficheiro*> normalmente ficheiro do compilador.

Na primeira forma o préprocessador primeiro procura o ficheiro no directório corrente, e só depois , caso não o encontre, é que o procura no directório i*nclude* (directório onde se encontram todas as headers do sistema). Na segunda forma o préprocessador apenas procura o ficheiro no directório *include*. Como exemplo, um programa poderá começar com as seguintes linhas:

```
# include "pilha.h"
# include < stdio.h >
# include < ctype.h >
# include < math.h >
```

que inclui um ficheiro do utilizador e três ficheiros do compilador. O ficheiro do utilizador contém instruções em C, como qualquer outro programa escrito nesta linguagem.

7.3. Define

Se um ficheiro contém uma linha da forma

```
#define nome string
```

então cada ocorrência do identificador *nome* no resto do ficheiro irá ser substituida por *string***.** Isto é feito pelo préprocessador antes da compilação.
É um bom estilo de programação usar *#define* para todas as constantes. É muito mais simples alterar uma linha da forma.

```
#define     SIZE    1024
```

que modifica correctamente todas as ocorrências de *SIZE* no ficheiro.
Alguns pontos técnicos sobre o *define*:

- A *string* inclui todos os caracteres (incluindo o espaço branco) até encontrar o caracter newline.
- O *nome* segue exactamente as regras dos nomes das variáveis.
- Uma convenção standard é usar *nome* em maiúsculas para se distinguir as macros das variáveis vulgares.
- Não são possíveis substituições nas “strings”, isto é, vendo no fragmento,

```
#define     SIM     1

printf ("SIM = %d\n", SIM);
```

o préprocessador só faz uma substituição.

- A “string” no define pode ser superior a uma linha; para isso o caracter newline terá de ser precedido pelo caracter '\' barra invertida (“backslash”).

7.4. Define Com Argumentos

Uma das facilidade da linguagem C consiste em invocar *#define* com argumentos. Por exemplo,

```
#define     isdigit(c)      ('0' < = (c) && (c) < = 9')
```

pode ser usado exactamente como se existisse uma função que determine quando é que o caracter *c* é um digito. Outro exemplo é a definição,

```
#define     ABC(x)      ((x < 0)?   (-x) : (x))
```

a qual faria com que uma linha da forma

```
y = ABC (a+ b)
```

fosse substituida (pelo préprocessador antes da compilação) por

```
y = (a + b < 0      ?   -a + b      :    a +b);
```

O que iria imprimir o seguinte fragmento?

```
#define        NEG(x)    -x

int a=1;

printf ("%", - NEG(a) );
```

Moral ! Usar parêntesis nas “strings” de substituição!

Como neste caso,

```
#define     MAX (a, b)       ((a) > (b)  ?  (a) : (b))
```

Alguns pontos sobre o uso de *#define* com argumentos:

- *#define* pode ter vários argumentos

- Em

```
#define nome(arg1, arg2, _)
```

não deverá ser deixado qualquer espaço entre o identificador *nome* e o parêntesis esquerdo.

- Os argumentos *arg1*,. . . obedecem às mesmas condições dos nomes das variáveis.

O uso abusivo do *#define* com argumentos pode levar a alguns erros aborrecidos. Notar que a macro anterior ABS tem a vantagem de que trabalhará para variáveis de diferentes tipos. Se uma macro tem uma “string” de substituição muito longa ou complicada, talvez seja preferível implementá-la como uma função.

Existem outras instruções de préprocessamento que não são aqui referidas, mas que poderão ser encontradas em qualquer manual.