ISCTE-IUL – Instituto Universitário de Lisboa
DCTI
Sistemas Operativos

Introdução à Linguagem C

Jrg, versão 1.3 – 26-02-2010

Conteúdo

1 - "Hello World"

Considere o seguinte programa, muito simples, que apenas escreve a mensagem “Hello World!”.

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

A primeira linha do programa é um "`include`". Esta directiva inclui o ficheiro indicado, neste caso o ficheiro `<stdio.h>`. Este `<stdio.h>` contém a definição de uma série de funções standard de input/output (I/O) normalmente usadas nos programas em C.

A execução de um programa em linguagem C começa na função `main`., que neste caso chama a função `printf` para escrever a mensagem "`Hello World!\n`" no ecrã.

Para executar este programa tem que:

1 - escrever o programa num ficheiro de texto com extensão .c; por exemplo hello.c. Pode fazer isso em qualquer editor de texto como o vi, o kwrite ou o nano.

2 - compilar o programa com o comando

```
gcc hello.c -o hello
```

3 - executar o programa invocando o ficheiro executável criado.

Tipicamente na consola (janela de comandos) seria feita a seguinte sequência:

```
> vi hello.c                    (ou outro editor...); escrever o programa
> gcc hello.c -o hello
> ./hello                       (hello pode bastar...); executar o programa
```

2 - Variáveis

2.1 - Tipos básicos

Os tipos básicos mais comuns são ilustrados nas seguintes declarações de variáveis:

```
int a = 10;
float f = 12.5;
char c = 'x';
```

A estes acrescem algumas variantes de tamanho e de aritmética. Por exemplo:

```
short int si = 10;
long int l = 10;
double d = 12.5;
unsigned int i = 10;
```

O `char` é um tipo inteiro com tamanho de 1 byte. Os outros tipos têm tamanho dependente da máquina. O operador `sizeof()` dá o tamanho de um tipo. Por exemplo:

```
sizeof(int)          dá o tamanho do tipo int; por exemplo 4 (32 bits)
int i;
sizeof(i)            dá, identicamente, o tamanho do tipo a que i pertence;
```

A sintaxe dos caracteres é consistente com o código ASCII. Assim, por exemplo:

`c='A'` é o mesmo que `c=65`;
`'A'+1` é `'B'` ou seja 66;
se `c == 'B'` então `c - 'A'` dá 1.

As conversões entre tipos básicos são automáticas, sem necessidade de explicitar o cast. Assim, por exemplo:

```
int i =0;
float f = 12.5;
i = f;              // é o mesmo que  i = (int) f;
```

O tipo de uma expressão deriva implicitamente do tipo dos operandos. Assim, por exemplo:

```
int i = 11;
i = 2*(i / 2);      // atribui a i o valor 2*5 (resultado da divisão inteira de 10 por 2);
i = (float) i / 2;  // atribui a i o valor (int) 2*2.5;
```

2.2 - Escrever com *printf*

O `printf` permite escrever o valor de variáveis (em geral expressões) na forma ilustrada nos seguinte exemplo

```
#include <stdio.h>
int main() {
    int i=10;
    printf ("Para passar é preciso ter %d ou mais\n", i);
}
```

O `printf` escreve no ecrã sempre e só a mensagem indicada no primeiro argumento. Mas, nessa mensagem, o símbolo `%d` não é para ser escrito literalmente: é para ser substituído por um número inteiro. O número a usar vem do argumento seguinte do `printf`, neste caso o valor de `i`.

Por cada símbolo `%d` inserido na mensagem é necessário colocar um argumento adicional no `printf`. Por exemplo:

```
printf("Um int ocupa %d bytes e um float %d bytes\n",
  sizeof(int), sizeof(float) );
```

O símbolo %d está associado ao tipo inteiro. Da mesma forma os símbolos `%c` e `%f` estão associados ao tipo `char` e `float` respectivamente.

```
int i = 10;
char c = 'Y';
float f = 12.7;
printf("Exemplo de \num int: %d \num char: %c \num float: %f\n", i, c, f );
```

Como é óbvio, os símbolos de formatação presentes têm que concordar em número e em tipo com o número e de argumentos adicionais que lhes vão fornecer os valores. Exemplos:

```
int i = 10;
char c = "Y";
float f = 12.7;
printf ("O código ASCII de %c é %d\n", c, c );
printf ("%f arredonda para %d\n", f, (int)(f+0.5) );
```

Maus exemplos:

```
printf ("O que é isto ? %d\");                  // falta um argumento
printf ("Valor=%d\n", i, j);                    // um argumento a mais
printf ("Escrever a parte inteira %d\n", f );   // discordância de tipo
```

2.3 - Leitura com `scanf`

A função homóloga do `printf` para leitura é o `scanf`. Um exemplo:

```
#include <stdio.h>
main() {
    int n;
    printf ("Diga um número: ");
    scanf ("%d", &n);
    printf ("O número seguinte é: %d\n", n + 1);
}
```

Neste exemplo é lido um número, através do `scanf`, para a variável n.

A especificação da leitura é dada pelo símbolo `%d` no primeiro argumento e indica, neste caso, que se pretende ler um inteiro. No segundo argumento indica-se a variável que vai receber o valor lido: `&n`. Note especialmente o símbolo `&` antes do `n`.

Outros exemplos:

```
scanf("%d%d", &i, &j);      // lê dois inteiros, o primeiro para i o segundo para j;
scanf ("%c", &c );          // lê um caractere para a variável c;
```

3 - Estruturas de controlo.

3.1 - if

O `if` escreve-se com a condição entre parêntesis e com um `else` opcional, controlando uma instrução ou um bloco de instruções (entre chavetas). Pode-se encadear uma sequência de "`else if`".

Exemplo 1:

```
#include <stdio.h>
main() {
     int a, b, m;
     printf ("Diga dois números: ");
      scanf ("%d%d", &a, &b);
      m = a;
  if ( a > b )
    m = b;
     printf ("O maior é: %d\n", b);
}
```

Exemplo 2:

```
#include <stdio.h>
main() {
     int a, b, m;
   printf ("Diga dois números: ");
      scanf ("%d%d", &a, &b);
  if ( a > b )
       printf ("O maior é: %d\n", a);
  else if ( b > a )
       printf ("O maior é: %d\n", b);
  else
    printf ("São iguais\n);
}
```

Exemplo 3:

```
#include <stdio.h>
main() {
     int a, b;
   printf ("Diga dois números: ");
      scanf ("%d%d", &a, &b);
  if ( b > a ) {
    int t = a;
    b = a;
    a = t;
  }
  printf ("O maior é: %d\n", a);
}
```

3.2 - Ciclos: while, for

O ciclo `while` tem a condição à cabeça (pode executar 0 ou mais vezes). Lê-se: "enquanto a condição for verdadeira... executar".

```
// ler 10 números e escreve o maior
#include <stdio.h>
main() {
     int i, n, maior=0;
      i = 0;
  while ( i < 10 ) {
         printf ("Diga um número: ");
          scanf ("%d", &n);
          if ( n > maior ) maior = n;
    i++;
  }
     printf ("O maior é %d\n", maior);
}
```

O `for` é uma forma compacta do `while` em que escreve o controlo do ciclo (inicialização; condição; incremento) na mesma linha de cabeçalho do ciclo.

Exemplo 1: ler 10 números e escreve o maior

```
#include <stdio.h>
main() {
    int i, n, maior=0;
    for ( i = 0; i <10; i++ ) {
        printf ("Diga um número: ");
        scanf ("%d", &n);
        if ( n > maior ) maior = n;
    }
    printf ("O maior deles é %d\n", maior);
}
```

Exemplo 2: ler uma sequência de números terminada pelo número 0 e calcular a soma

```
#include <stdio.h>
main() {
    int n=1, soma=0;                          // é preciso inicializar n
    while ( n != 0 ) {
        printf ("Diga um número: ");
        scanf ("%d", &n);
        soma += n;
    }
    printf ("A soma é %d\n", soma);
}
```

3.3 - Ciclo `do-while`

Ocasionalmente pode ter interesse o ciclo `do while` (condição no fim), que executa 1 ou mais vezes.

Exemplo: ler uma sequência de números terminada pelo número 0 e calcular a soma

```
#include <stdio.h>
main() {
    int n, soma=0;
    do {
        printf ("Diga um número: ");
        scanf ("%d", &n);
        soma += n;
    } while ( n != 0 );
    printf ("A soma é %d\n", soma);
}
```

3.4 - break

A instrução `break` é usada para sair de um ciclo, ou da instrução `switch` (que abordaremos de seguida). Ao encontrar a instrução `break`, a execução do programa continua na primeira instrução que segue o ciclo ou o `switch`.

No exemplo que se segue a variável `i` assume os valores 0 a 5. Quando `i` é 5 executa-se a instrução `break` que passa o controlo para o `printf`, imediatamente após o ciclo.

```
#include <stdio.h>
int main() {
  int i = 0;
  while (i < 10000) {
    if (i==5) break;
    printf("i: %d\n", i);
    i++;
  }
  printf("O Valor de i é %d\n", i);
}
```

3.5 - switch

A instrução `switch` permite decidir entre os vários valores de uma variável de um tipo enumerável (`int`, `long`, `short` ou `char`). A ideia consiste em descrever o que fazer para cada um dos valores que a variável pode assumir. No final, pode existir um caminho de execução por omissão, que apanha todos os outros valores.

O Exemplo seguinte devolve uma descrição para cada número inteiro (Nenhum, Um, Dois, Vários ou Muitos). Todos os casos importantes são listados e correspondem a uma acção, todos os outros casos são apanhados por `default`. A execução do programa prossegue sempre dentro do `switch` até ser encontrado um `break`.

```
switch(numero) {
    case 0 : printf("Nenhum\n"); break;
    case 1 : printf("Um\n"); break;
    case 2 : printf("Dois\n"); break;
    case 3 :
    case 4 :
    case 5 :
        printf("Vários\n");
        break;
    default :
        printf("Muitos\n");
        break;
  }
```

4 - Expressões condicionais e lógicas

4.1 - Valores lógicos

O valor inteiro 0 é tomado como *false*. Qualquer valor não 0 é tomado como *true*.

Exemplo: detectar se numa sequência de 10 números aparece o 7.

```
#include <stdio.h>
main() {
    int aparece = 0;    // inicializa a false
    int i, n;
    for ( i = 0; i <10; i++ ) {
        printf ("Diga um número: ");
        scanf ("%d", &n);
        if ( n == 7)
      aparece = 1;
    }
    if ( aparece )
    printf ("O 7 aparece\n");
  else
    printf ("O 7 não aparece\n");
}
```

Os operadores relacionais são:

`==`	igual	`>`	maior	`>=`	maior ou igual
`!=`	diferente	`<`	menor	`<=`	menor ou igual

Os operadores lógicos são

`&&`	and	`||`	or	`!`	not

O resultado de uma expressão condicional ou lógica é 0 (false) ou 1 (true).

Exemplo:

`1==2`	dá 0 (falso)
`1==2 || 1 == 1`	dá 1 (true)
`!0`	dá 1
`!1`	dá 0
`!5`	dá 0

4.2 - Cuidados e abreviações

Construções do género

`if (a != 0)` ou `while ( a != 0 )`

podem-se abreviar para

`if ( a )` ou `while ( a )`

A primeira forma dá 1 (`true`) para valores de a diferentes de 0. A segunda forma usa a circunstância de todos os valores diferentes de 0 serem tomados como `true`.

A construção seguinte configura um ciclo infinito.

```
while ( 1 ) {
    ...
}
```

Um dos erros mais chatos é pôr um = em vez de ==.

Por exemplo `if( i = 7)` em vez de `if ( i == 7 )`. Neste exemplo: o efeito do `if ( n = 7 )` seria atribuir a `n` o valor 7 (o sinal igual sozinho é uma atribuição!). Desta forma o resultado da "condição" do `if` seria também o valor 7, ou seja, `true`. Por consequência programa daria sempre "O 7 aparece".

5 - Funções

A sintaxe de definição de uma função inclui o tipo, nome da função e lista de argumentos. Por exemplo:

```
#include <stdio.h>
main() {
   func2 ( 1, 2 );
}

int func2 ( int a, int b ) {
    printf ("Recebi argumentos %d e %d\n", a, b );
    return a + b;
}
```

A instrução `return` tem uma função dupla de controlo (termina a execução da função) e formação do valor de retorno. Se a função não tiver valor de retorno (se for um "procedimento") pode ser definida com o tipo `void`.

Exemplo: escrever n vezes um caractere dado

```
void espaco ( int a, char c ) {
    while ( a-- ) printf ("%c", c);
}
```

Dependendo da variedade de compilador de C que estiver a usar pode ter que ser obrigado a declarar as funções antes de usar. A sintaxe de declaração das dois exemplos anteriores seria:

```
int func(int, int);
void espaco(int, char);
```

6 - Arrays

A sintaxe de declaração e inicialização de arrays é ilustrada nos seguintes exemplos:

```
int a[5];                 // dimensão 5 (índices de 0 a 4);
int a[] = { 1, 2, 3};     // dimensão 3
int c[5] = { 1, 2 };      // dimensão 5; inicializa só as 3 primeiras posições;
```

A inicialização é a única operação onde se pode manipular o array como um conjunto; em todas as outras só se pode aceder a uma posição através de um índice. Os índices começam em 0.

Exemplo: ler 10 números e escrevê-los por ordem inversa da de leitura

```
#include <stdio.h>
#include <stdio.h>
main() {
    int i, a[10];
    printf ("Escreva 10 números: ");
    for ( i=0; i<=9; i++ )
        scanf ( "%d", &a[i] );
    printf ("%d\n", somaa ( a, 10 ) );

}
int somaa ( int a[], int n ) {
    int soma = 0;
    for ( n--; n>=0; n-- )
        soma += a[n];
    return soma;
}
```

A referência `&a[i]` significa `&(a[i])`, mas os parêntesis podem ser omitidos uma vez que `[]` tem precedência sobre `&`.

Para receber um array como argumento pode-se usar a forma a[]. A dimensão, sendo necessária, terá que ser passada num argumento adicional.

Exemplo: ler 10 números e somar

```
#include <stdio.h>
main() {
    int i, a[10];
    printf ("Escreva 10 números: ");
    for ( i=0; i<=9; i++ )
        scanf ( "%d", &a[i] );
    printf ("%d\n", somaa ( a, 10 ) );

}
int somaa ( int a[], int n ) {
    int soma = 0;
    for ( n--; n>=0; n-- )
        soma += a[n];
    return soma;
}
```

7 - Arrays de caracteres ("strings")

7.1 - Terminador

Os arrays de caracteres podem ser usados da mesma forma que qualquer outro array, mas efectivamente têm algumas características particulares.

A inicialização de um array de caracteres pode ser feita da forma compacta ilustrada no seguinte exemplo:

```
char s[10] = { 'H', 'e', 'l', 'l', 'o'};
char s[10] = "Hello";
```

A segunda forma inicializa da mesma forma as 5 primeiras posições do array mas com uma diferença: inicializa também a posição seguinte com um caractere terminador da string; deste modo, com a segunda inicialização, o array fica com o seguinte conteúdo:

H	e	l	l	o	\0	?	?	?	?

O caractere terminador representa-se pelo símbolo '\0' e é o caractere com código ASCII 0.

Esta forma de terminação, embora convencional, é praticamente uma regra de programação em linguagem C. Todas as funções e mecanismos da linguagem a usam e pressupõem.

Os arrays de caracteres têm, como qualquer outro array, uma dimensão "física" relacionada com o espaço de memória que lhe está atribuído; neste exemplo são 10 posições (10 bytes). Além disso, têm a dimensão "efectiva" correspondente ao conteúdo com significado, que vai do início até ao terminador; neste caso a dimensão efectiva seria 5.

Com base nesta convenção pode-se usar um array de caracteres sem saber, à partida, a sua dimensão física; basta saber a dimensão real que está marcada, no próprio array, pelo terminador.

Por exemplo, para imprimir um dado array de caracteres s, independentemente do número de posições do array, pode-se aplicar o procedimento

```
for ( i=0; s[i] != '\0'; i++ ) {
    printf ("%c", s[i];
}
```

ou seja imprimir todos os caracteres até ao '\0'

Este tipo de convenção é usado por todas as funções que mexem em arrays. É o caso do `printf` que permite imprimir um array de caracteres usando o formatador %s. Por exemplo:

```
char s[] = "Hello World";
printf ( "%s\n", s );
```

Note que, face a esta convenção, é preciso salvaguardar sempre uma posição do array para conter o '\0'. Assim, por exemplo, para guardar a palavra "Hello" é preciso um array de, pelo menos, 6 posições.

No exemplo anterior a dimensão física do array s é dada implicitamente pela inicialização. Já não deve espantar que a dimensão resultante neste exemplo seja 12: os 11 caracteres indicados mais uma posição para o '\'0'.

7.2 - Funções de manipulação de strings

Suportadas na mesma convenção de terminação há uma série de funções de biblioteca que facilitam a manipulação de arrays de caracteres.

A função `int strlen( char[] )` devolve a dimensão efectiva do array indicado como argumento (ou seja, o número de caracteres deste o início até ao '\0'). Isto corresponde basicamente a um raciocínio do género:

```
int foo_strlen( char s[] ) {
    int n=0;
    while ( s[n] )
        n++;
    return n;
}
```

Outras funções são

`strcpy ( char s[], char a[] )`	copia `a` para `s`;
`strcat ( char s[] , char a[] )`	junta ("concatena") `a` a `s`;
`n = strcmp ( char a[], char b[] )`	dá o resultado da comparação entre `a` e `b`;

O `strcpy` corresponde à noção de atribuição aplicada a strings; por exemplo dadas as declarações:

```
char a[] = "Hello";
char s[10];
```

a chamada

```
strcpy ( s, a );
```

copia o conteúdo de `a` para `s`, de forma que fica também s com "Hello". Evidentemente s fica, como a, terminado com '\0'. O `strcpy` corresponde ao raciocínio:

```
foo_strcpy (char s[], char a[]) {
    int i;
    for ( i = 0; a[i]; i++ ) {
        s[i] = a[i];
    }
    s[i] = 0;
}
```

O `strcat` permite juntar duas strings. Por exemplo dados:

```
char s[20] = "Hello";
char s[10] = "World";
```

a sequência de instruções

```
strcat ( s, " ");
strcat ( s, a );
```

forma em `s` a expressão "Hello World".

A função `strcmp` compara duas strings e devolve 0 caso sejam iguais ou outro valor caso sejam diferentes. Mais concretamente devolve a diferença entre os dois primeiros caracteres em que as duas strings difiram. Desta forma pode-se o usar o resultado do `strcmp` quer para comparar quer para ordenar as strings.

Por exemplo, dados:

```
char a[10] = "Hello";
char b[10] = "World";
char c[10] = "Worx";
```

`n = strcmp(a, "Hello")` devolve 0 indicando que as duas strings são iguais;

`n = strcmp ( a, b )` devolve um número negativo indicando que é `a` menor que `b`; a diferença resulta da comparação entre `H` e `W`;

`n = strcmp ( c, b )` devolve um número positivo indicando que `c` é maior que `b`; a diferença resulta da comparação de `x` e `l`, os primeiros caracteres diferentes.

8 - Leitura e escrita de caracteres

8.1 - Ler e escrever (com strings)

Para ler e escrever strings há algumas funções adicionais.

O `printf` com %s escreve uma string no ecrã.

A função `gets` lê uma linha do ecrã para um array.

```
#include <stdio.h>
main() {
  char s[100];
  printf ("Nome :");
  gets ( s );
  printf ("O seu nome é %s\n", s );
}
```

Note-se o papel do terminador: o `gets`() lê a linha e transfere-a para o array, pondo o terminador no fim; o `printf` escreve os caracteres desde o início do array até ao terminador.

Nenhuma das funções conhece a dimensão física do array. No caso do `gets` isso pode ser perigoso porque, sendo assim, não tem forma de impedir a entrada de mais caracteres do que aqueles que cabem no array (19 neste caso, salvaguardando uma posição para o terminador).

A alternativa é a função `fgets` que permite fazer este limite.

```
#include <stdio.h>
main() {
  char s[100];
  printf ("Nome :");
  fgets ( s, 100, stdin );
  printf ("O seu nome é %s\n", s );
}
```

Ambas as funções lêem uma linha, ou seja todos os caracteres que apareçam até ser encontrado um fim de linha ('\n'). A função `gets` lê mas descarta o caractere de fim de linha; a `fgets`, ao contrário, deixa-o no array.

Considerando os exemplos anteriores, depois do `gets` o array poderia ficar

J	o	a	o	\0	?	?	?	...	?	?

e depois do `fgets` ficaria

J	o	a	o	\n	\0	?	?	...	?	?

ou seja, no caso da leitura com `fgets` o próprio '\n' fica no array.

Querendo podemos eliminá-lo com a seguinte instrução:

```
s[strlen(s)-1] = '\0';
```

8.2 - Ler caracteres

A função `fgetc(stdin)` lê um caractere.

Exemplo:

```
#include <stdio.h>
main() {
    char s[100], c;
    int i = 0;
    printf ("Nome :");
    while ( ( c = fgetc(stdin) ) != '\n' && i < 100 ) {
        s[i++] = c;
    }
    s[i] = 0;

    printf ( "O seu nome tem %d caracteres", strlen(s) );
    if ( strcmp( s, "João" ) != 0 )
        printf (" e não é João");
    printf (".\n", s );
}
```

8.3 - Disciplina de leitura

Há questões e cuidados a ter quando se mistura a leitura de números e strings

Em primeiro lugar é preciso perceber que a leitura é um mecanismo de consumo sequencial (por ordem) dos caracteres que vão sendo escritos.

O `scanf` com `%d` consome todos os separadores que se apresentem (espaços, fins de linha) até encontrar o primeiro dígito, depois consome todos os dígitos (parando, isto é, já não consumindo o primeiro não dígito).

O `gets` (ou `fgets`) consome todos os caracteres até encontrar um enter (inclusivé).

Exemplo: são dados os seguintes caracteres

```
 123 abc
345
Hello
```

Aplicando a seguinte sequência de funções:

`scanf ("%d", &x)`	consome o espaço e os dígitos 123; deixa o espaço a seguir ao 3; x fica com 123
`gets (s)`	consome o espaço a seguir ao 3, os caracteres abc e o fim da linha; s fica com " abc" (ou " abc\n" no caso do fgets);
`gets (s)`	consome 345 e o fim da linha; s fica com "345" (ou " 345\n" no caso do fgets);
`scanf ("%d", &x)`	termina no H (não consome nada) x fica como estava (não é alterado)

Exemplo: são dados os seguintes caracteres

```
123
456
Hello
```

Aplicando a seguinte sequência de funções:

`scanf ("%d", &x)`	consome os dígitos 123; deixa o enter a seguir ao 3; x fica com 123
`scanf ("%d", &x)`	consome o enter e os dígitos 456; deixa o enter a seguir ao 6; x fica com 456
`gets (s)`	consome o enter; s fica com "" (ou " \n" no caso do fgets);
`gets (s)`	consome o Hello e o enter; s fica com "Hello " (ou "Hello \n" no caso do fgets);

9 - Ler e escrever em ficheiros

As funções e leitura e escrita em ficheiros são muito semelhantes às de leitura e escrita no ecrã. Antes de ler ou escrever é preciso abrir o ficheiro.

9.1 - Escrever

O seguinte exemplo ilustra um exemplo tipo "Hello World" para ficheiro:

```
#include <stdio.h>
main() {
  FILE *f;
  f = fopen ("teste.txt", "w");
  fprintf (f, "Hello World\n" );
  fclose ( f );
}
```

A variável f (de um tipo que por enquanto não discutimos) representa um "canal" para acesso a um ficheiro. A função f abre o ficheiro para escrita (é dado o nome, neste caso "teste.txt" e a operação, neste caso "w" de write).

A função `fprintf()` permite escrever num ficheiro. É parecida com o `printf`: tem um primeiro argumento indicando o ficheiro; depois é igual.

Nem sempre a abertura de um ficheiro resulta (permissões, ...).

Se a abertura falhar o `fopen` devolve NULL:

```
main() {
  FILE *f;
  f = fopen ("teste.txt", "w");
  if ( f == NULL ) {
    printf ( "não correu bem....\n");
    exit ( 1 );
  }
  fprintf (f, "Hello World\n" );
  fclose ( f );
}
```

Exemplo: ler 10 números e escrevê-los num ficheiro:

```
#include <stdio.h>
main() {
  FILE *f;
  f = fopen ("numeros_1.txt", "w");
  int i, n;
for ( i = 0; i < 10; i ++ ) {
    printf ("Diga: ");
    scanf ("%d", &n);
    fprintf ( f, "%d", n);
  }
  fclose ( f );
}
```

Versão 2

```
#include <stdio.h>
main() {
  FILE *f;
  f = fopen ("numeros_2.txt", "w");
  int i, n;
  for ( i = 0; i < 10; i ++ ) {
    printf ("Diga: ");
    scanf ("%d", &n);
    fprintf ( f, "%d ", n);
  }
  fclose ( f );
}
```

Na primeira versão os números ficam todos colados na primeira linha (uma desgraça, quando se tentarem ler). Na segunda versão ficam separados por espaços na primeira linha. Uma terceira versão ficando um em cada linha seria conseguida acrescentando **\n** no `fprintf`.

```
…
   f = fopen ("numeros_3.txt", "w");
…
     fprintf ( f, "%d\n",  n);
…
```

Exemplo: ler várias linhas do ecrã e escrevê-las no ficheiro (termina quando se der "fim").

```
#include <stdio.h>
main() {
  FILE *f;
  char linha[80], i = 1;
  f = fopen ("texto.txt", "w");
  while ( 1 ) {
    printf ("%d: ", i++);
    fgets ( linha, 80, stdin);
    if ( ! strcmp ( linha, "fim\n" ) ) break;
    fprintf ( f, "%s", linha );
  }
  fclose( f);
}
```

9.2 - Ler

A leitura usa também funções semelhantes às já conhecidas:

`fscanf ( f, "%d", &n )` ler um inteiro do ficheiro f para a variável n

`fgets ( s, 100, f )` ler uma linha do ficheiro f para o array f

O primeiro passo é ainda abrir o ficheiro, no caso com a operação "r" (de "read").

O seguinte exemplo tenta ler os números do ficheiro `numeros_1.txt`

```
#include <stdio.h>
main() {
        FILE *f;
        f = fopen ("numeros_1.txt", "r");
        if ( f == NULL ) {
                perror ("file");
                exit(1);
        }
        int i, n;
        printf ("Numeros:\n");
        for ( i = 0; i < 10; i ++ ) {
                fscanf (f, "%d", &n);
                printf ( "%d: %d\n", i+1, n);
        }
        printf ("\n");
        fclose ( f );
}
```

O resultado da leitura poderá não ser grande coisa: o ficheiro só tem um número que pode ser gigante e, nesse caso, aparece mal (mas isso é mais uma questão das cadeiras de AC).

A leitura dos ficheiros das outras versões, `numeros_2.txt` e `numeros_3.txt`, deverá correr melhorzinho.

9.3 - Ler: fim de ficheiro

Em qualquer dos casos a leitura de um ficheiro nunca se faz como no exemplo anterior presumindo que se encontram lá 10 números. Lê-se de um ficheiro o que lá estiver, terminando "no fim".

As funções de leitura assinalam esse fim. Por exemplo, o `scanf` assinala o fim de ficheiro devolvendo EOF ("end of file"):

```
#include <stdio.h>
main() {
        FILE *f;
        f = fopen ("numeros_1.txt", "r");
        if ( f == NULL ) {
                perror ("file");
                exit(1);
        }
        int n, i = 0;
        printf ("Numeros\n");
        while ( fscanf (f, "%d", &n) != EOF ) {
                printf ( "%d : %d", ++i, n);
        }
        printf ("\n");
        fclose ( f );
}
```

O `fgets` assinala o fim de ficheiro devolvendo NULL:

```
main() {
  FILE *f;
  char s[80];
  int i = 0;
  f = fopen ("texto.txt", "r");
  while ( fgets (s, 100, f ) != NULL ) {
    printf ( "%d: %s", ++i, s );
  }
  printf ("\n");
  fclose ( f );
}
```

10 -Estruturas

10.1 -Declaração

Uma estrutura é uma variável composta de vários campos (ou "membros"), podendo cada um deles ser uma variável comum ou outra estrutura.

```
struct Aluno {
  int num;
  char nome[100];
  int nota;
} a, b, c;
```

Este exemplo representa uma estrutura correspondente a uma pauta de alunos; o aluno é representado por um número, nome e nota.

As variáveis a, b e c representam, cada uma, um aluno. Cada um delas tem, portanto, 3 campos, designados num (um inteiro), nome (um array de 100 caractares) e nota (outro inteiro).

Para aceder a um campo da variável usa-se o . (ponto).

Assim, por exemplo:

`a.num`	campo num do aluno a
`b.nome`	campo nome do aluno b

Face àquelas declarações fazem sentido as seguintes construções:

```
a.num = 1001;
strcpy ( b.nome, "Joao" );
printf  ("%d - %s - %d\n", a.num, a.nome. a.nota );
```

10.2 -Typedef

O `typedef` é uma instrução permite dar um novo nome a um tipo de dados. Considere o seguinte:

```
typedef int Inteiro;
typedef float Peso;
```

Desta forma poderíamos definir variáveis do tipo Inteiro e Peso.

```
Inteiro a = 10;
Peso meupeso = 120.5;
```

O `typedef` é especialmente útil para ajudar a definir tipos compostos, designadamente estruturas. Por exemplo:

```
typedef struct {
  int num;
  char nome[100];
  int nota;
} TipoAluno;
```

define um `TipoAluno` que permite, depois, fazer declarações equivalentes às anteriores:

```
TipoAluno a, b, c;
```

10.3 -Arrays de estruturas

A seguinte declaração

```
TipoAluno a, pauta[20];
```

declara duas variáveis: a uma estrutura do tipo `TipoAluno` e um array `pauta` com 20 elementos do mesmo tipo. Face a estas declarações fazem sentido as seguintes construções:

```
pauta[1].num = 1070;
strcpy ( pauta[1].nome , "João" );
printf ("O nome do 3º aluno começa pela letra %c\n", pauta[2].nome[0]);
```

11 -Exemplo: Gestão de uma Pauta

11.1 -Versão 1: só o menu

O seguinte exemplo implementa uma Pauta com as opções constantes do seguinte menu:

```
#include <stdio.h>
main() {
        int opcao;
        do {
                printf ("1. Inserir aluno\n" );
                printf ("2. Apagar aluno\n" );
                printf ("3. Inserir nota\n" );
                printf ("4. Lista de alunos\n" );
                printf ("5. Pauta por nome\n" );
                printf ("0. Sair \n" );
                scanf ("%d", &opcao );
                if ( opcao == 1 ) nop ( );
                if ( opcao == 2 ) nop ( );
                if ( opcao == 3 ) nop ( );
                if ( opcao == 4 ) nop ( );
        } while  ( opcao != 0 );
}

int nop ( ) {
  printf ("Opção ainda em construção\n" );
  printf ("Carregue enter para continuar...");
  fgetc (stdin);
}
```

Há um pequeno problema: a função `nop()` tinha como intenção obrigar a uma paragem para o utilizador carregar em enter mas não resulta ("não espera"). A razão é simples: o `scanf` lê a opção mas deixa o enter por lêr; o `fgetc()` encontra esse enter e, por isso, não espera.

Podemos ter o cuidado de limpar o resto da linha depois de ler a opção.

Por exemplo com:

```
char dummy[100];
...
scanf ("%d", &opcao );
gets (dummy);
...
```

ou

```
scanf ("%d", &opcao );
while ( fgetc(stdin) != '\n' );
...
```

11.2 -Versão 2: estrutura de dados

Vamos guardar os dados da pauta num array de estruturas

```
typedef struct {
  int num;
  char nome[100];
  int nota;
} TipoAluno;

TipoAluno pauta[20];
int n_alunos = 0;
```

complementado com uma variável que indica o número de alunos (inicialmente 0). Estas variáveis vão ser partilhadas (ou seja vai ser globais) para conjunto de funções. Para que várias funções possam usar as mesmas variáveis vamos declará-las como estáticas (fora de qualquer função).

Nesta versão 2 vamos implementar duas operações: a 1 para inserir um aluno e a 3 para listar (de forma a poder, de imediato, verificar se a inserção está a funcionar).

```
#include <stdio.h>
typedef struct {
        int num;
        char nome[100];
        int nota;
} TipoAluno;

TipoAluno pauta[20];
int n_alunos = 0;

main() {
        int opcao;
        do {
                printf ("1. Inserir aluno\n" );
                printf ("2. Apagar aluno\n" );
                printf ("3. Inserir nota\n" );
                printf ("4. Lista de alunos\n" );
                printf ("5. Pauta por nome\n" );
                printf ("0. Sair \n" );
                scanf ("%d", &opcao );
                limpar_linha();
                if ( opcao == 1 ) inserir_aluno ();
                if ( opcao == 2 ) nop ();
                if ( opcao == 3 ) nop ();
                if ( opcao == 4 ) listar ();
                if ( opcao == 5 ) nop ();
        } while  ( opcao != 0 );

}
```

```
int limpar_linha() {
    while ( fgetc(stdin) != '\n');
}

int nop () {
        printf ("Opção em construção\n" );
        printf ("Carregue enter para continuar...");
        limpar_linha();
}

int inserir_aluno () {
        int num;
        char nome[100];

        printf ("N° aluno: " );
        scanf ("%d", &num );
        limpar_linha();         //senão o nome vai apanhar o enter e ficar vazio

        printf ("Nome: " );
        gets( nome );
        pauta[n_alunos].num = num;
        strcpy ( pauta[n_alunos].nome, nome );
        n_alunos++;
}

int listar () {
        int i;
        for ( i = 0; i < n_alunos; i++ ) {
           printf ("%5d %s\n", pauta[i].num, pauta[i].nome );
        }
}
```

11.3 -Versão 3

Juntam-se nesta versão mais duas operações: a 3 lançar nota e a 5 para mostrar a pauta.

```
int lancar_nota ( ) {

  int i, num;
  printf ("N° aluno: " );
  scanf ("%d", &num );
  limpar_linha();

  for ( i = 0; i < n_alunos; i++ ) {
    if (pauta[i].num == num) break;
  }
  if ( i == n_alunos ) {
    printf ("Aluno %d não consta\n", i );
    return;
  }

  printf ("Nota: " );
  scanf ("%d", &num );
  pauta[i].nota = num;
  limpar_linha();
}
```

```
int pauta() {
  int i;

  espacos( '=', 62 ); printf ("\n");
  printf ( "N°   %-50s  Nota\n", "Nome");
  espacos ( '=', 62 ); printf ("\n");

  for ( i = 0; i < n_alunos; i++ ) {
     printf ( "%-5d %-50s %4d\n", pauta[i].num,
                                  pauta[i].nome, pauta[i].nota );
  }
  espacos ( '=', 62 ); printf ("\n");
}

int espacos ( char c, int n) {
  int i;
  for ( i = 1; i < n; i++ ) putchar (c);
}
```

Acrescentamos ainda uma função para ordenar a pauta, a chamar após a inserção de um novo aluno.

```
int ordenar_nome ( ) {

   int i, j;

     for ( i = 0; i < n_alunos; i++ ) {
        for ( j = i + 1; j < n_alunos; j++ ) {
             if ( strcmp( pauta[i].nome, pauta[j].nome ) > 0 ) {
                 TipoAluno a;
                  a = pauta[i];
                  pauta[i] =  pauta[j];
                  pauta[j] = a;
          }
    }
}
```

11.4 -Versão 4 : guardar em ficheiro

Nesta versão vamos guardar os dados em ficheiro. O plano é: no início do programa lê-se do ficheiro para o array, onde depois se faz toda a manipulação dos dados; no fim do programa volta a gravar-se o array para ficheiro.

Para gravar usa-se a seguinte função

```
int gravar() {

  FILE *f;
  f = fopen ("pauta.txt", "w" );
  int i;
   for ( i = 0; i < n_alunos; i++ ) {
    fprintf ( f, "%d\n%s\n%d\n" , pauta[i].num, pauta[i].nome,
                      pauta[i].nota );
  }
  fclose(f);

}
```

Fica, portanto, o número numa linha, o nome noutra e a nota ainda noutra (houve uma primeira ideia de pôr tudo na mesma linha mas isso não iria facilitar a leitura !).

Para ler usa-se a seguinte função:

```
int ler() {

        FILE *f;
        f = fopen ("pauta.txt", "r" );
        if ( f == NULL ) return;
        printf ("Carregar ficheiro...");
        int num, nota;
        char nome[100];

        while ( fscanf ( f, "%d", &num ) != EOF )  {
                fgets ( nome, 100, f );
                nome[strlen(nome)-1] = 0;
                fscanf ( f, "%d", &nota );

                pauta[n_alunos].num = num;
                strcpy ( pauta[n_alunos].nome , nome );
                pauta[n_alunos].nota = nota;
                n_alunos++;
        }
        fclose(f);
        printf ("\n");

}
```

A função ler é chamada no início do main e a função escrever é chamada no fim do main.

12 -Ponteiros

12.1 -Memória dinâmica

Considere o seguinte exemplo:

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int a, *p;
    p = malloc ( sizeof(int) );
    a = 10;
    (*p) = 10;
    printf ( "%d;%d\n", a, (*p) );
}
```

A função `malloc()` permite ao programa arranjar mais memória. Neste caso arranjar `sizeof(int)` bytes, ou seja, o suficiente para guardar um número inteiro.

Ficamos assim no programa com duas variáveis `int`. A variável `a` que é declarada e fica disponível para utilização através do nome, como é normal. E essa outra que é criada com o `malloc()` e, a partir daí, fica também disponível para utilização, através de `(*p)`.

A primeira variável é criada pelos mecanismos normais previstos na linguagem, ou seja, a declaração. A segunda é criada pela chamada à função `malloc()` dizemos que é criada dinamicamente (ou que é "memória dinâmica").

O `p` tem um papel importante neste mecanismo de utilização de memória dinâmica. É através do p que se chega ao espaço de memória criado pelo `malloc()`. Ao fazer

```
malloc ( sizeof(int) );
```

estamos a invocar o `malloc()` que vai criar e dar ao nosso programa um novo espaço de memória.

Ao fazer

```
p = malloc ( sizeof(int) );
```

estamos a dizer que `p` fica a apontar para esse espaço. Ou seja será através de p que se vai poder aceder a esse espaço para usá-lo como variável. Daí dizer-se que p é um ponteiro (ou a apontador, se quiser).

E como é que se acede a esse espaço através de p. Obviamente, como a sintaxe do programa sugere, através de

```
(*p)
```

que dizemos ser "o que é apontado por p".

Em suma, a sequência

```
p = malloc ( sizeof(int) );
(*p) = 10;
```

- cria um espaço de memória com tamanho para um inteiro (ou seja uma variável int), ficando p a apontar para essa variável;
- nessa variável (a tal para que aponta) colocamos o valor 10.

12.2 -Inicialização

Podemos experimentar este programa

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int *p;
    *p = 10;
    printf ( "%d\n", *p );
}
```

O resultado será muito possivelmente um erro. Ou este, que não falha - dá mesmo erro:

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int *p = NULL;
    *p = 10;
    printf ( "%d\n", *p );
}
```

No primeiro caso estamos a usar o que é "apontado por p" antes de ter posto p a apontar para algum lado; só por muita sorte p estaria a apontar para um sítio válido. No segundo caso estamos a inicializar p indicando que aponta para "lado nenhum"; é esse o sentido lógico da inicialização com p=NULL,

Ambos estão mal, claro. Só podemos usar o espaço para que p aponta depois de pôr p a apontar para um espaço que se possa usar (pois, é de lapalisse).

12.3 -Memória dinâmica : arrays

Considere o seguinte exemplo:

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int a, *p=NULL;
    p = malloc ( 10 * sizeof(int) );
    *p =  10;
    *(p+1) =  11;
    *(p+2) =  *p + 2;
    printf ("%d %d %d\n", *p, *(p+1), *(p+2) );
}
```

Neste caso estamos a usar o `malloc` parra arranjar espaço correspondente a 10 inteiros.

Em alternativa podemos usar a função `calloc()` que cria um conjunto de posições consecutivas que são, também, inicializadas com 0 (contrariamente ao `malloc()` que não inicializa).

```
p = calloc (sizeof(int), 10 );
```

A semelhança entre este array e o "array dinâmico" do ponto 13.4 é a origem do espaço de memória. Trata-se em qualquer dos casos de um espaço de memória semelhante: num caso e noutro um conjunto de posições consecutivas onde se podem guardar 10 inteiros.

12.4 -Aritmética de ponteiros

Para usar estas 10 posições temos que ampliar a exploração do *p da forma ilustrada no exemplo. Assim, tal como

`*p`	representa a posição de memória apontada por p,
`*(p+1)`	representa a posição de memória seguinte.

Dizemos, "apontada por p+1". Desta forma podemos aceder a cada um das 10 posições de criadas pela chamada ao `malloc()`, desde a primeira, *p até à última *(p+9).

Na realidade, uma vez que p é uma variável, e portanto modificável, podemos fazer variar o próprio p para aceder às várias posições. Assim, por exemplo:

```
*p = 10;
p = p + 1;
*p = 11;
```

Coloca 10 numa posição de memória e 11 na posição seguinte.

Temos assim uma aritmética específica dos ponteiros, com um significado lógico. Dado um ponteiro p

`p + i`	aponta para uma posição de memória i posições à frente, e
`p - i`	aponta para uma posição de memória i posições para trás.

Na mesma lógica funcionam as operações relacionais. Por exemplo, dados dois ponteiros p1 e p2,

`p1 == p2`	se apontam para a mesma posição;
`p1 > p2`	se p1 aponta para uma posição à frente da apontada por p2.

O seguinte exemplo inicializa um conjunto de 10 posições de memória com números de 10 a 10 que, depois, depois escreve por ordem inversa no ecrã.

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int i, *p=NULL;
    p = malloc ( 10* sizeof(int) );
    for ( i=0; i<10;i++) {
        *(p+i) = 10+i;
    }
    int *p1 = p + 9;
    for ( ; p1 >= p; p1-- ){
        printf ("%d\n", *p1 );
    }
}
```

12.5 -Sintaxe de arrays

Na realidade os ponteiros podem ser usados com a mesma sintaxe dos arrays. O conceito

```
*(p+i)
```

a posição de memória apontada por p+i, ou seja, a posição que se encontra i casas à frente da posição inicial p é o mesmo expresso pela notação

```
p[i]
```

As duas sintaxes são intermutáveis.

O seguinte exemplo cria um conjunto de 10 posições de memória e inicializa-as com números de 10 a 19.

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int i, *p=NULL;
    p = malloc ( sizeof(int), 10 );
    for ( i=0; i<10;i++) {
        p[i] = i + 10;
    }
}
```

12.6 -Ponteiros e arrays

A semelhança entre ponteiros e arrays funciona também no sentido contrário: um array pode também ser manipulado com a sintaxe dos arrays.

Na realidade um array é um ponteiro, que aponta para a primeira posição do array). Assim, querendo, podemos aceder às suas diversas posições usando a sintaxe dos arrays, como acontece neste exemplo:

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int a[10] = { 10, 11, 12, 13, 14, 15, 16, 17, 18, 19};
    int i;
    for ( i=0; i<10;i++) {
        printf ("%d\n", *(a+i) );
    }
}
```

Em suma, o a deste exemplo é da mesma natureza que o p dos exemplos anteriores. A diferença é que o a é um ponteiro constante; aponta sempre para a mesma posição, que é o início do array declarado. Já uma variável p declarada como nos exemplos anteriores

```
int *p;
```

sendo uma variável pode ser modificada - tendo neste caso por consequência apontar para posições diferentes. Assim, por exemplo, pode-se fazer:

```
int a[10];
p = a;
```

desta forma p com o mesmo valor que a, ou seja fica a apontar para a mesma posição que a. Desta forma p[i] parra a ser o mesmo que a[i]. Fazendo, por exemplo:

```
int a[10];
p = a+1;
```

`p` fica a apontar para a posição a seguir a `a`. Desta forma `p[i]` passa a ser o mesmo que `a[i+1]`. E, por exemplo, `p[-1]` será, na circunstância, o mesmo que `a[0]`.

Não é possível obviamente fazer:

```
a=p;
```

porque a é um ponteiro constante: aponta sempre para a posição onde começa o array `a`.

12.7 -Ponteiros para estruturas

O seguinte exemplo cria uma variável dinâmica do tipo Aluno.

```
typedef struct {
        int num;
        char nome[100];
        int nota;
} TipoAluno;

main() {
   TipoAluno *p;
  p = malloc( sizeof(TipoAluno) );
  (*p).num = 100;
  strcpy ( (*p).nome, "Zé Carlos" );
  (*p).nota = 17;
}
```

Para o efeito é preciso dispor de um ponteiro para uma variável `TipoAluno`. È isso que é definido na declaração:

```
TipoAluno *p;
```

A função malloc é chamada neste caso para obter um espaço com o tamanho da estrutura; o ponteiro p fica a apontar para esse espaço. O *p denota, assim, uma variável do tipo `TipoAluno`. Esta variável é depois usada como uma estrutura normal, acedendo com o **.** a cada um dos seus campos.

Entretanto há nestas circunstâncias uma sintaxe alternativa que se pode utilizar,

`p->nome` em vez de `(*p).num`

Assim, o programa ficaria

```
...
p = malloc( sizeof(TipoAluno) );
p->num = 100;
strcpy ( p->nome, "Zé Carlos" );
p->nota = 17;
```

12.8 -Tipo dos ponteiros

Considere as seguintes declarações:

```
int *p_inteiro;
TipoAluno *p_aluno;
```

Trata-se de dois ponteiros, com tipos diferentes. O primeiro é um ponteiro para inteiro, ou seja é um ponteiro do tipo

```
int *
```

O segundo é um ponteiro para TipoAluno. É do tipo

```
TipoAluno *
```

As declarações têm aliás estas duas formas interessantes de serem lidas. Na perspectiva do tipo apontado podemos ler

```
int *p
```

salientando que o conjunto (*p) denota um inteiro. Na perspectiva do próprio apontador podemos ler

```
int *p;
```

que indica que p é um "apontador para inteiro".

O ponto importante é que um ponteiro aponta para um tipo de dados.

As funções malloc() devolvem um tipo

```
void *
```

que deve ser convertido, através de um cast, para um tipo específico. Por exemplo:

```
int *p = (int *) malloc ( 10 * sizeof(int) );
```

O cast de void * para outro tipo de ponteiro é automático; pode ser o omitido, como aconteceu nos exemplos anteriores.

13 -Exemplo: pauta com memória dinâmica

```
typedef struct _tipo_aluno {
    int num;
    char nome[100];
    int nota;
    struct _tipo_aluno *prox;
} TipoAluno;

TipoAluno *pauta = NULL

int inserir_aluno () {
    int num;
    char nome[100];

    printf ("N° aluno: " );
    scanf ("%d", &num );
    limpar_linha();
    printf ("Nome: " );
    gets( nome );

    TipoAluno *p;
    p = malloc ( sizeof(TipoAluno) );

    p->num = num;
    strcpy ( p->nome, nome );
    p->nota = -1;

    if ( pauta == NULL ) {
        pauta = p;
        p->prox = NULL;
    }
    else {
        TipoAluno *pa =pauta;
        pauta = p;
        p->prox = pa;
    }
}

int listar() {
    TipoAluno *p = pauta;
    while ( p ) {
        printf ("%d %s %d\n", p->num, p->nome, p->nota );
        p = p->prox;
    }
}

int inserir_aluno () {
    TipoAluno *p;
    p = malloc ( sizeof(TipoAluno) );

    printf ("N° aluno: " );
    scanf ("%d", &(p->num) );

    limpar_linha();
    printf ("Nome: " );
    gets( p->nome );

    p->nota = -1;
    p->prox=NULL;
```

```
    if ( pauta == NULL ) {
        pauta = p;
    }
    else {
        TipoAluno *pa = pauta;
        while ( pa->prox != NULL )
            pa = pa ->prox;
        pa->prox = p;
    }
}


int ler() {
     FILE *f;
     f = fopen ("pauta.txt", "r" );
     if ( f == NULL ) return;
     printf ("Carregar ficheiro...");

     int num, nota;
     char nome[100];

     TipoAluno *pa = pauta, *p;
     while ( fscanf ( f, "%d", &num ) != EOF )  {
        fgets ( nome, 100, f );
        nome[strlen(nome)-1] = 0;
        fscanf ( f, "%d", &nota );

        p = malloc( sizeof(TipoAluno) );
        p->num = num;
        strcpy ( p->nome, nome );
        p->nota = nota;
        p->prox = NULL;

        if ( pauta == NULL ) {
            pauta = p;
        }
        else
            pa->prox=p;
        pa=p;
    }
    fclose(f);
    printf ("\n");
}
```

14 -Algumas situações com ponteiros

14.1 -Ponteiros e endereços

Fisicamente um ponteiro contém um endereço de memória: o endereço da posição para que aponta. Por exemplo na instrução:

```
int *p = malloc ( sizeof(int) );
```

o p recebe o endereço da posição de memória que o malloc obteve para ser usada pelo programa.

O operador & permite obter o endereço de uma variável comum. Por exemplo, data a declaração:

```
int i;
```

a expressão `&i` denota o endereço de `i`.

Sendo &i um endereço pode ser atribuído a um apontador, que fica então a apontar para `i`:

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

Nestas circunstâncias pode-se aceder a i através da p. Por exemplo:

```
*p = 10
```
ou
```
p[0] = 10
```

fazem o mesmo que `i = 10`.

14.2 -Passagem por referência

Na linguagem C os argumentos são sempre passados por valor. Ou seja, quando se invoca uma função são criadas variáveis locais da função para receber os argumentos.

```
main() {
  int x = 3;
  f ( x, 10 );
  ...
}

void f ( int a, int b ) {
  // a e b são variáveis locais...
  // neste caso a vai ser inicializada com o valor de x,
  //    ou seja com 3; e b com 10;
  ...
}
```

Por consequência, nunca é possível a uma função alterar as variáveis passadas como argumento.

Em certas circunstâncias dá, de facto, jeito que uma função possa alterar variáveis passadas como argumento. Não sendo possível fazê-lo directamente pode-se obter o mesmo efeito passando como argumento não a variável mas o seu endereço.

```
main() {
  int x = 3;
  f ( &x );
  printf ("%d\n", x );
}

int f ( int *p ) {
    *p = 4;
}
```

A função `f()` recebe como argumento um ponteiro `int *`. Neste caso é passado como argumento `&x` ou seja o endereço de `x`, ou seja, o argumento `p` fica a apontar para `x`. Desta forma quando dentro da função se aceder a `*p` está-se a mexer na variável `x`.

Exemplo:

```
main() {
    int x = 3, y = 2;
    troca ( &x,  &y);
    printf ("%d %d\n", x, y );
}

void troca ( int *pa, int *pb ) {
    int x = *pa;
    *pa = *pb;
    *pb = x;
}
```

Tratando-se de um array a situação é diferente. O array é um ponteiro (para a primeira posição do array). A partir dele a função pode aceder aos elementos do array, designadamente para os alterar se for o caso.

Exemplo: a seguinte função inicializa um array com o valor indicado no argumento.

```
main() {
    int a[10];
    init ( a, 10, 20);
}
void init ( int *p, int n, int v) {
    int i;
    for ( i=0; i<n; i++ ) {
        p[i] = v;
    }
}
```

14.3 -Argumentos nas funções de leitura

Na instrução

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

o scanf recebe como argumento o endereço da variável n. Isto é indispensável para que a função possa alterar o valor de n, atribuindo-lhe o valor lido do teclado.

A situação é diferente quando se trata de um array como nos exemplos:

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

A passagem do array com argumento permite às funções alterar o seu conteúdo.

14.4 -Arrays de ponteiros

// TODO

14.5 -Argumentos na linha de comando (argc, argv)

O seguinte exemplo escreve os argumentos recebidos na linha de comando.

```
int main( int argc, char *argv[] ) {
    int i;
    for ( i=0; i<argc; i++ ) {
        printf ("%d-> %s\n", i, argv[i] );
    }
}
```

O `argv[]` é um array de ponteiros para `char` que veicula os argumentos recebidos na linha de comando. O argc indica quantos elementos existem nesse array.

Suponha que este programa existe num ficheiro teste e é executado dando o comando:

```
teste A B "1+2"
```

O primeiro elemento `argv[0]` contém sempre o nome do programa invocado. Os seguintes os argumentos adicionais dados na linha de comando, neste caso `arrgv[1]` com A, etc.