

Introdução à Programação: Elementos Básicos de Programação em C

1ª edição: Dezembro/2001

RICARDO J. MACHADO

Email: rmac@dsi.uminho.pt
URL: <http://www.dsi.uminho.pt/~rmac>



Universidade do Minho

Departamento de Sistemas de Informação

Sumário

- 1. A Linguagem C**
- 2. Estrutura dos Programas**
- 3. Controlo dos Programas**
- 4. Atribuições e Comparações**
- 5. Funções, Variáveis e Protótipos**
- 6. Defines e Macros**
- 7. Strings e Arrays**
- 8. Apontadores**
- 9. Standard Input/Output**
- 10. Ficheiros**
- 11. Bibliotecas de Funções**
- 12. Erros Típicos**

© RMAC XII-2001

2

1. A Linguagem C (1/3)

■ Linguagens de programação

- a computação de um algoritmo exige que ele seja descrito recorrendo a uma linguagem artificial cujas instruções possam ser directamente executadas por uma máquina de computação
- estas linguagens designam-se de *linguagens de programação*
- de facto, as linguagens de programação de alto nível (tipo C, Pascal, Ada) não são directamente executadas por uma máquina de computação, mas, como existem processadores de linguagens (compiladores) capazes de realizar automaticamente a transformação para linguagem máquina, por relaxe de linguagem, é frequente dizer que aquelas são directamente executáveis
- uma linguagem de programação implementa um algoritmo, suportando a introdução de decisões de concepção e de implementação para tornar possível a computação da sequência de instruções elementares especificada no algoritmo original

© RMAC XII-2001

3

1. A Linguagem C (2/3)

■ A linguagem C

- a linguagem C foi criada em 1972 nos *Bell Telephone Laboratories* (E.U.A.) por *Dennis Ritchie*
- o objectivo da criação da linguagem C foi implementação do sistema operativo UNIX a um nível elevado de abstracção, evitando o recurso a linguagens *assembly*
- a linguagem C foi uma evolução da linguagem B, criada anteriormente por *Ken Thompson* nos *Bell Telephone Laboratories* (E.U.A.)
- em 1978, *Brian Kernighan* e *Dennis Ritchie* publicaram o livro *The C Programming Language* que se tornou numa norma *de facto* até surgir a normalização oficial do ANSI C, em 1988

© RMAC XII-2001

4

1. A Linguagem C ^(3/3)

■ A linguagem C (cont.)

- em 1983 a *American National Standards Institute (ANSI)* iniciou a definição das regras para escrever um programa em C *standard*
- um programa escrito em ANSI C (publicado em 1988) deve poder ser compilado por qualquer compilador de C e deve funcionar exactamente da mesma forma, independentemente do compilador de C utilizado
- a divulgação generalizada da linguagem C deveu-se ao facto de
 - ser uma linguagem de utilização genérica
 - seguir o paradigma imperativo
 - ser procedimental e modular
 - ser sintética e poderosa

© RMAC XII-2001

5

2. Estrutura dos Programas ^(1/11)

- função *main* () -

■ trivial.c

```
main ( )  
{  
}
```

- a função *main* () deve aparecer uma única vez num programa em C
- os parêntesis permitem passagem de parâmetros
- as chavetas delimitam blocos de código
- programa "trivial.c" não faz nada (inexistência de instruções)

© RMAC XII-2001

6

2. Estrutura dos Programas (2/11)

- chavetas e ponto e vírgula -

■ wrtsome.c

```
main ()
{
    printf("This is a line of text to output.");
}
```

- as instruções executáveis de qualquer programa em C devem localizar-se entre as 2 chavetas, a de início de programa ("{" e a de fim ("}")
- **printf ()** é uma função que envia informação para o écran (tudo o que estiver entre aspas)
- todas as instruções devem terminar com ponto e vírgula (";")

© RMAC XII-2001

7

2. Estrutura dos Programas (3/11)

- função **printf ()** -

printf

Function Writes formatted output to stdout.

Syntax `#include <stdio.h>`
`int printf(const char *format[, argument, ...]);`

DOS	UNIX	Windows	ANSI C	C++ only
■	■		■	

Remarks **printf** accepts a series of arguments, applies to each a format specifier contained in the format string given by *format*, and outputs the formatted data to *stdout*. There must be the same number of format specifiers as arguments.

Return Value **printf** returns the number of bytes output. In the event of error, **printf** returns EOF.

See also **cprintf**, **ecvt**, **fprintf**, **fread**, **fscanf**, **putc**, **puts**, **putw**, **scanf**, **sprintf**, **vprintf**, **vsprintf**

© RMAC XII-2001

8

2. Estrutura dos Programas (4/11)

- execução sequencial -

■ wrtmore.c

```
main ( )
{
    printf ("This is a line of text to output.\n");
    printf ("And this is another ");
    printf ("line of text.\n\n");
    printf ("This is a third line.\n");
}
```

- as instruções são executadas sequencialmente, tal como na leitura da fita da máquina de Turing
- "\" (*backslash*) indica o surgimento de carácter especial
- "\n" (*newline, carriage return* ou *line feed*) posiciona o cursor no início da próxima linha

© RMAC XII-2001

9

2. Estrutura dos Programas (5/11)

- caracteres especiais -

■ Backslash codes

\7	<i>Bell</i> (sinal sonoro do computador)
\a	<i>Bell</i> (sinal sonoro do computador)
\b	<i>BackSpace</i>
\n	<i>New Line</i> (mudança de linha)
\r	<i>Carriage Return</i>
\t	Tabulação Horizontal
\v	Tabulação Vertical
\\	Carácter \ (forma de representar o próprio carácter especial \)
\'	Carácter ' (plica)
\"	Carácter " (aspas)
\?	Carácter ? (ponto de interrogação)
\ooo	Carácter cujo código <i>ASCII</i> em Octal é ooo
\xnn	Carácter cujo código <i>ASCII</i> em Hexadecimal é nn
%%	carácter %

© RMAC XII-2001

10

2. Estrutura dos Programas (6/11)

- declaração de variáveis -

■ oneint.c

```
main ( )
{
  int index;
  index = 13;
  printf ("This value of the index is %d\n", index);
  index = 27;
  printf ("This value of the index is %d\n", index);
  index = 10;
  printf ("This value of the index is %d\n", index);
}
```

- ***int index*** declara uma variável do tipo inteiro chamada ***index***
- ***index*** não é uma palavra reservada em C e segue a regra de identificadores

© RMAC XII-2001

11

2. Estrutura dos Programas (7/11)

- declaração de variáveis -

■ oneint.c (cont.)

- atribuição de valores distintos à variável durante a execução
- **"%"** indica o surgimento de caracter especial de formatação de saída de dados
- **"%d"** (*decimal base*) indica saída de dados do tipo inteiro
- valor a sair corresponde à variável indicada depois da vírgula

© RMAC XII-2001

12

2. Estrutura dos Programas (8/11)

- comentários -

■ comments.c

```
/* This is a comment ignored by the compiler */

main ( ) /* This is a comment ignored by the compiler */
{
    printf ("We are looking at how comments are "); /* A comment is
                                                    allowed to be
                                                    continue on
                                                    another line */

    printf ("used in C.\n");
}
/* One more comment for effect */
```

- tudo o que estiver entre "/*" e "*/" é um comentário, i.e., é ignorado pelo compilador

© RMAC XII-2001

13

2. Estrutura dos Programas (9/11)

- cabeçalhos -

■ comments.c (cont.)

- os comentários auxiliam na compreensão dos programas
- o programa "comments.c" não está adequadamente comentado
- todos os programas devem ser intensamente comentados e em língua inglesa
- exemplo de comentário de início de programa

```
/* coments.c : Program to elucidate the usage of comments in C */
/* author : Ricardo J. Machado */
/* date : 1995/11/17 */

main ( )
{
    ...
}
```

© RMAC XII-2001

14

2. Estrutura dos Programas (10/11)

- formatação do código -

■ goodform.c

```
main ( ) /* Main program starts here */
{
    printf ("Good form ");
    printf      ("can aid in ");
    printf      ("understanding a program.\n");
    printf ("And bad form ");
    printf      ("can make a program ");
    printf      ("unreadable.\n");
}
```

- o compilador ignora todos os espaços e mudanças de linha a mais, deixando ao programador uma grande liberdade de formatação gráfica do seu código
- o programa "goodform.c" está adequadamente formatado

© RMAC XII-2001

15

2. Estrutura dos Programas (11/11)

- formatação do código -

■ uglyform.c

```
main ( ) /* Main program starts here */ { printf("Good form ");printf
("can aid in ");printf("understanding a program.\n")
; printf("And bad form ");printf("can make a program ");
printf("unreadable.\n");
}
```

- o programa "uglyform.c" não está adequadamente formatado, apesar de não dar erros de compilação
- os comentários têm um impacto directo na manutibilidade do software, bem como no prolongamento do ciclo de vida dos sistemas

© RMAC XII-2001

16

3. Controlo dos Programas (1/23)

- ciclo *while* -

■ while.c

```
/* This is an example of a while loop */

main ( )
{
    int count;
    count = 0;
    while (count < 6) {
        printf ("The value of count is %d\n", count);
        count = count + 1;
    } /* end of while loop */
}
```

- todas as instruções dentro das chavetas do ciclo *while* são executadas ciclicamente (segundo a sequência em que surgem), enquanto a condição entre parêntesis for avaliada como verdadeira

© RMAC XII-2001

17

3. Controlo dos Programas (2/23)

- ciclo *while* -

■ while.c (cont.)

- é possível ter um ciclo *while* que nunca chega a executar as instruções do seu bloco
 - exemplo: se, no programa "while.c", a variável *count* fosse iniciada com um valor superior a 5, então o ciclo do *while* (*count* < 6) nunca chegaria a executar
- é possível ter um ciclo *while* que nunca chega a terminar o ciclo de execução das instruções do seu bloco
 - exemplo: se, no programa "while.c", a variável *count* não fosse incrementada dentro do ciclo, então o ciclo do *while* (*count* < 6) nunca chegaria a terminar
- se, num ciclo *while*, o seu bloco de código for composto por uma única instrução, então as chavetas não necessitam de estar presentes
 - exemplo: *while (condicao) instrucao_unica;*
- o ciclo *while* é conveniente ser utilizado em circunstâncias em que não existe *a priori* nenhuma ideia de quantas vezes irá ser executado o bloco de código do ciclo

© RMAC XII-2001

18

3. Controlo dos Programas (3/23)

- ciclo *do-while* -

■ dowhile.c

```
/* This is an example of a do-while loop */

main ( )
{
    int i;
    i = 0;
    do {
        printf ("The value of i is now %d\n", i);
        i = i + 1;
    } while (i < 5);
}
```

- todas as instruções dentro das chavetas do ciclo *do-while* são executadas ciclicamente (segundo a sequência em que surgem), até a condição entre parêntesis ser avaliada como falsa

© RMAC XII-2001

19

3. Controlo dos Programas (4/23)

- ciclo *do-while* -

■ dowhile.c (cont.)

- uma vez que o teste da condição do ciclo *do-while* é efectuado somente no final do seu bloco de código, as instruções do seu bloco são executadas, pelo menos, uma vez
- é possível ter um ciclo *do-while* que nunca chega a terminar o ciclo de execução das instruções do seu bloco
 - exemplo: se, no programa "dowhile.c", a variável *i* não fosse incrementada dentro do ciclo, então o ciclo *do while* (*i* < 5) nunca chegaria a terminar
- se, num ciclo *do-while*, o seu bloco de código for composto por uma única instrução, então as chavetas não necessitam de estar presentes
 - exemplo: *do instrucao_unica; while (condicao);*
- o ciclo *do-while* é raramente utilizado, mas, tal como no ciclo *while*, também deve ser utilizado em circunstâncias em que não existe *a priori* nenhuma ideia de quantas vezes irá ser executado o bloco de código do ciclo

© RMAC XII-2001

20

3. Controlo dos Programas (5/23)

- ciclo *for* -

■ forloop.c

```
/* This is an example of a for loop */  
  
main ()  
{  
    int index;  
    for (index = 0; index < 6; index = index + 1)  
        printf ("The value of the index is %d\n", index);  
}
```

- todas as instruções dentro das chavetas do ciclo *for* são executadas ciclicamente (segundo a sequência em que surgem), enquanto a condição existente entre os parêntesis *for* avaliada como verdadeira
- o ciclo *for* constitui uma forma diferente de descrever um ciclo *while*

© RMAC XII-2001

21

3. Controlo dos Programas (6/23)

- ciclo *for* -

■ forloop.c (cont.)

- os parêntesis do ciclo *for* delimitam três campos separados entre si por ponto e vírgula
 - o primeiro campo é utilizado para iniciar as variáveis contadoras, pelo que qualquer expressão existente neste campo é executada sempre antes da primeira execução do bloco de código do ciclo (mais do que uma iniciação neste campo devem ser separadas por vírgula)
 - o segundo campo é utilizado para descrever a condição a testar antes de começar a execução do bloco de código do ciclo
 - o terceiro campo é utilizado para colocar as expressão de actualização das variáveis contadoras a executar de cada vez que o ciclo se repete
- se, num ciclo *for*, o seu bloco de código *for* composto por uma única instrução, então as chavetas não necessitam de estar presentes, tal como aparece no programa "forloop.c"
- o ciclo *for* é tipicamente utilizado quando se conhece exactamente quantas vezes irá ser executada o bloco de código do ciclo

© RMAC XII-2001

22

3. Controlo dos Programas (7/23)

- ciclo *for* -

■ forloop.c (cont.)

- o ciclo *for*, comparativamente com os outros dois tipos de ciclos, tem a vantagem de concentrar num só local (dentro dos parêntesis) toda a informação de controlo de ciclo, em vez de a misturar com o bloco de código, tal como sucede com o ciclo *while* e com o ciclo *do-while*
- ciclos encadeados são ciclos (*while*, *do-while* e *for*) que estejam presentes dentro de ciclos
- em C, não existe qualquer limitação ao número de ciclos que pode ocorrer dentro de outros ciclos

3. Controlo dos Programas (8/23)

- ciclos -

■ Resumo dos ciclos

- ciclo *while*
 - *while (condicao) instrucao;*
 - executa zero ou mais vezes
 - testa a condição antes da instrução
 - ciclo infinito → *while (1) instrucao;*
- ciclo *do-while*
 - *do instrucao while (condicao)*
 - executa uma ou mais vezes
 - testa a condição depois da instrução
 - ciclo infinito → *do instrucao; while (1);*
- ciclo *for*
 - *for (iniciacao; condicao; actualizacao) instrucao;*
 - executa zero ou mais vezes
 - testa a condição antes da instrução
 - ciclo infinito → *for (; ;) instrucao;*

3. Controlo dos Programas (9/23)

- instrução *if-else* -

■ ifelse.c

```
/* This is an example of the if and if-else statements */

main ( )
{
    int data;
    for (data = 0; data < 10; data = data + 1) {
        if (data == 2) printf ("Data is now equal to %d\n", data);
        if (data < 5)
            printf ("Data is now %d, which is less than 5\n", data);
        else
            printf ("Data is now %d, which is greater than 4\n", data);
    } /* end of for loop */
}
```

© RMAC XII-2001

25

3. Controlo dos Programas (10/23)

- instrução *if-else* -

■ ifelse.c (cont.)

- cada um dos *if* dentro do ciclo *for* é executado 10 vezes
- em C, a instrução *se-então* da linguagem natural não possui sintacticamente o "então", uma vez que coloca a acção logo depois da condição:
if (condicao) accao;
- a instrução *if-else* é semelhante ao *se-então-senão* da linguagem natural:
if (condicao) accao_1; else accao_2;
- o operador "==" é relacional e não de atribuição, ou seja, permite construir uma proposição booleana

© RMAC XII-2001

26

3. Controlo dos Programas (11/23)

- instruções *break* e *continue* -

■ breakcon.c

```
/* This is an example of the break and continue statements */

main ( )
{
    int xx;
    for (xx = 5; xx < 15; xx = xx + 1) {
        if (xx == 8) break;
        printf ("In the break loop, xx is now %d\n", xx);
    } /* end of break loop */

    for (xx = 5; xx < 15; xx = xx + 1) {
        if (xx == 8) continue;
        printf ("In the continue loop, xx is now %d\n", xx);
    } /* end of continue loop */
}
```

© RMAC XII-2001

27

3. Controlo dos Programas (12/23)

- instruções *break* e *continue* -

■ breakcon.c (cont.)

- a instrução *break* provoca o término do ciclo e o retomar da execução na primeira instrução a seguir ao ciclo
- a instrução *break* é imprescindível para sair de um ciclo, quando determinados cálculos efectuados dentro do ciclo atingem certos valores
- a instrução *continue* provoca um salto da iteração actual do ciclo para a próxima, sem terminar o mesmo

© RMAC XII-2001

28

3. Controlo dos Programas (13/23)

- instrução *switch-case* -

■ switch.c

```
/* This is an example of the switch statement */

main ( )
{
    int truck;
    for (truck = 3; truck < 13; truck = truck + 1) {
        switch (truck) {
            case 3 : printf ("The value is three\n");
                     break;
            case 4 : printf ("The value is four\n");
                     break;
            case 5 :
            case 6 :
            case 7 :
```

(...)

© RMAC XII-2001

29

3. Controlo dos Programas (14/23)

- instrução *switch-case* -

■ switch.c (cont.)

```
(...)

        case 8 : printf ("The value is between 5 and 8\n");
                 break;
        case 11 : printf ("The value is eleven\n");
                 break;
        default : printf ("It is one of the undefined values\n");
                 break;
    } /* end of switch */
} /* end of for loop */
}
```

© RMAC XII-2001

30

3. Controlo dos Programas (15/23)

- instrução *switch-case* -

■ *switch.c* (cont.)

- a instrução *switch* permite executar uma acção distinta para cada valor que a variável em teste possa tomar em tempo de execução
- cada uma das acções (bloco de acções) surge associada a um *case* onde aparece um determinado valor da variável (aquele que a variável tem que apresentar para que as acções sejam executadas)
- no final de cada bloco de acções de uma entrada *case*, deve ser utilizado um *break*, caso contrário serão executadas todas as acções existentes no *switch* a partir do *case* utilizado para entrar no mesmo
- a instrução *switch* é equivalente a uma sequência de instruções *if-else* encadeadas

© RMAC XII-2001

31

3. Controlo dos Programas (16/23)

- instrução *goto* -

■ *gotoex.c*

```
/* This is an example of the goto statement */

main ( )
{
    int dog, cat, pig;
    goto real_start;

    some_where:
    printf ("The is another line of the mess\n");
    goto stop_it;

    /* the following section is the only section with a useable goto */
    real_start:
    for (dog = 1; dog < 6; dog = dog + 1) {
```

(...)

© RMAC XII-2001

32

3. Controlo dos Programas (17/23)

- instrução *goto* -

■ gotoex.c (cont.)

```
(...)  
  
for (cat = 1; cat < 6; cat = cat + 1) {  
    for (pig = 1; pig < 4; pig = pig + 1) {  
        printf ("Dog = %d  Cat = %d  Pig = %d\n", dog, cat, pig);  
        if ( (dog + cat + pig) > 8 ) goto enough;  
    }; /* end of 3th loop */  
}; /* end of 2nd loop */  
}; /* end of 1st loop */  
enough: printf ("Those are enough animals for now.\n");  
/* this is the end of the section with a useable goto statement */  
  
printf ("\nThis is the first line of the spaghetti code.\n");  
goto there;  
  
(...)
```

© RMAC XII-2001

33

3. Controlo dos Programas (18/23)

- instrução *goto* -

■ gotoex.c (cont.)

```
(...)  
  
where:  
    printf ("This is the third line of spaghetti.\n");  
    goto some_where;  
  
there:  
    printf ("This is the second line of the spaghetti code.\n");  
    goto where;  
  
stop_it :  
    printf ("This is the last line of this mess.\n");  
}
```

© RMAC XII-2001

34

3. Controlo dos Programas (19/23)

- instrução *goto* -

■ gotoex.c (cont.)

- a instrução *goto* permite saltar para um local do programa identificado por uma etiqueta (*label*) seguida de ":"
- com uma instrução *goto* é possível sair de um ciclo, mas já não é possível nem entrar num ciclo, nem saltar entre funções
- a instrução *goto* deve ser utilizada com parcimónia para evitar o chamado *código espaguete*, em que o fluxo de execução saltita erráticamente de local em local, gerando uma enorme confusão na lógica de estruturação do programa, dificultando a sua compreensão, promovendo o surgimento de erros e colocando em causa a futura actividade de manutenção do software

3. Controlo dos Programas (20/23)

- programa bem escrito -

■ tempconv.c

```
/* This is a temperature conversion program written in */  
/* the C programming language. This program generates */  
/* and displays a table of fahrenheit and centigrade */  
/* temperatures, and lists the freezing and boiling */  
/* of water. */
```

```
main ()  
{  
    int count; /* a loop control variable */  
    int fahrenheit; /* the temperature in fahrenheit degrees */  
    int centigrade; /* the temperature in centigrade degrees */  
  
    printf ("Centigrade to Fahrenheit temperature table\n\n");
```

(...)

3. Controlo dos Programas (21/23)

- programa bem escrito -

■ tempconv.c (cont.)

```
(...)  
  
for (count = -2; count <= 12; count = count + 1) {  
    centigrade = 10 * count;  
    fahrenheit = 32 + (centigrade * 9)/5;  
    printf (" C = %4d F = %4d ", centigrade, fahrenheit);  
    if (centigrade == 0) printf (" Freezing point of water");  
    if (centigrade == 100) printf (" Boiling point of water");  
    printf("\n");  
} /* end of for loop */  
}
```

- programa bem comentado, bem estruturado e com uma selecção de nomes sugestivos para as variáveis

© RMAC XII-2001

37

3. Controlo dos Programas (22/23)

- programa mal escrito -

■ dumbconv.c

```
main ()  
{  
    int x1, x2, x3;  
    printf ("Centigrade to Farenheit temperature table\n\n");  
    for (x1 = -2; x1 <= 12; x1 = x1 + 1) {  
        x3 = 10 * x1;  
        x2 = 32 + (x3 * 9)/5;  
        printf (" C = %4d F = %4d ", x3, x2);  
        if (x3 == 0) printf (" Freezing point of water");  
        if (x3 == 100) printf (" Boiling point of water");  
        printf ("\n");  
    } /* end of for loop */  
}
```

© RMAC XII-2001

38

3. Controlo dos Programas (23/23)

- programa mal escrito -

■ dumconv.c (cont.)

- programa não comentado, mal estruturado e com uma selecção de nomes nada sugestivos para as variáveis
- esta forma de escrever programas é vivamente desaconselhada por dificultar a sua compreensão, promover o surgimento de erros e colocar em causa a futura actividade de manutenção do software

4. Atribuições e Comparações (1/25)

- atribuições simples e múltiplas -

■ intasign.c

```
/* This program will illustrate the assignment statements */  
  
main ()  
{  
  int a, b, c; /* integer variables for examples */  
  
  a = 12;  
  b = 3;  
  c = a + b; /* simple addition */  
  c = a - b; /* simple subtraction */  
  c = a * b; /* simple multiplication */  
  c = a / b; /* simple division */  
  c = a % b; /* simple modulo (remainder) */  
  
  (...)
```

4. Atribuições e Comparações (2/25)

- atribuições simples e múltiplas -

■ intasign.c (cont.)

```
(...)  
  
c = 12*a + b/2 - a*b*2/(a*c + b*2);  
c = c/4+13*(a + b)/3 - a*b + 2*a*a;  
a = a + 1;      /* incrementing a variable */  
b = b * 5;  
  
a = b = c = 20; /* multiple assignment */  
a = b = c = 12*13/4;  
  
printf ("\n%d %d %d\n", a, b, c);  
}
```

- ilustração de operações aritméticas elementares sobre dados do tipo inteiro

© RMAC XII-2001

41

4. Atribuições e Comparações (3/25)

- atribuições simples e múltiplas -

■ intasign.c (cont.)

- nas atribuições múltiplas, as atribuições simples são efectuadas da direita para a esquerda até consumir todas as atribuições
- nas atribuições múltiplas podem ser utilizadas operações aritméticas
- esta forma de atribuição é muito útil na iniciação de várias variáveis simultaneamente
- em C, qualquer variável tem que ser declarada antes de ser utilizada

© RMAC XII-2001

42

4. Atribuições e Comparações (4/25)

- tipos *int*, *char* e *float* -

■ moretypes.c

```
/* The purpose of this file is to introduce additional data types */

main ( )
{
    int a, b, c;          /* -32768 to +32767 with no decimal point */
    char x, y, z;         /* -128 to +127 with no decimal point */
    float num, toy, thing; /* 3.4E-38 to 3.4E+38 with decimal point */

    a = b = c = -27;
    x = y = z = 'A';
    num = toy = thing = 3.6792;
    a = y;               /* a is now 65 (character A) */
    x = b;               /* x is now -27 */
    num = b;             /* num will now be -27.00 */
    a = toy;             /* a will now be 3 */
}
```

© RMAC XII-2001

43

4. Atribuições e Comparações (5/25)

- tipos *int*, *char* e *float* -

■ moretypes.c (cont.)

- **bit** = *binary digit* e constitui a unidade básica de informação num sistema de computação, uma vez que representa uma informação booleana
- **byte** = 8 bits
- o tipo de uma variável condiciona a sua forma de utilização nos programas, uma vez que a cada tipo corresponde um determinado espaço (número de bits) para guardar informação
- o tipo *char* é semelhante ao tipo *int*, excepto pelo facto de só poder receber valores numéricos entre -127 e +127, uma vez que dispõe de um único byte para armazenar a sua informação, enquanto que o tipo *int* dispõe de 2 bytes
- o tipo *char* é tipicamente utilizado para armazenar caracteres alfanuméricos (ver códigos ASCII mais à frente)

© RMAC XII-2001

44

4. Atribuições e Comparações (6/25)

- tipos *int*, *char* e *float* -

■ moretypes.c (cont.)

- em C, é, quase sempre, possível utilizar indistintamente variáveis do tipo *char* ou *int*, excepto quando os valores em causa necessitarem de 2 bytes
- exemplos do "motypes.c":
 - `a = y`
 - `x = b`
- o tipo *float* (de *floating point*) é utilizado para representar números reais em 4 bytes, ou seja, entre $3.4E-38$ e $3.4E+38$
- pelo facto do tipo *float* possuir uma capacidade de representação superior ao tipo *int*, é possível transformar um valor inteiro na sua representação em vírgula flutuante utilizando uma mera atribuição e vice versa (realizando, neste caso a truncagem do valor real, ou seja, guardando a sua parte inteira)
- exemplos do "motypes.c":
 - `num = b`
 - `a = toy`

© RMAC XII-2001

45

4. Atribuições e Comparações (7/25)

- conversão de tipos -

Target Type	Expression Type	Possible Info Loss
signed char	char	If value > 127, the targets will be negative
char	short int	High-order 8 bits
char	int	High-order 8 bits
char	long int	High-order 24 bits
short int	int	None
short int	long int	High-order 16 bits
int	long int	High-order 16 bits
int	float	Fractional part and possibly more
float	double	Precision, result rounded
double	long double	Precision, result rounded

© RMAC XII-2001

46

4. Atribuições e Comparações (8/25)

- tipos *long*, *short*, *unsigned* e *double* -

■ lottypes.c

```
main ( )
{
    int a;          /* simple integer type */
    long int b;     /* long integer type */
    short int c;    /* short integer type */
    unsigned int d; /* unsigned integer type */
    char e;         /* character type */
    float f;        /* floating point type */
    double g;       /* double precision floating point */

    a = 1023;
    b = 2222;
    c = 123;
    d = 1234;
```

(...)

© RMAC XII-2001

47

4. Atribuições e Comparações (9/25)

- tipos *long*, *short*, *unsigned* e *double* -

■ lottypes.c (cont.)

```
(...)

    e = 'X';
    f = 3.14159;
    g = 3.1415926535898;

    printf ("a = %d\n", a); /* decimal output */
    printf ("a = %o\n", a); /* octal output */
    printf ("a = %x\n", a); /* hexadecimal output */
    printf ("b = %ld\n", b); /* decimal long output */
    printf ("c = %d\n", c); /* decimal short output */
    printf ("d = %u\n", d); /* unsigned output */
    printf ("e = %c\n", e); /* character output */
    printf ("f = %f\n", f); /* floating output */
```

(...)

© RMAC XII-2001

48

4. Atribuições e Comparações (10/25)

- tipos *long*, *short*, *unsigned* e *double* -

■ lottypes.c (cont.)

```
(...)  
  
printf ("g = %f\n", g);    /* double float output */  
printf ("\n");  
printf ("a = %d\n", a);    /* simple int output */  
printf ("a = %7d\n", a);   /* use a field width of 7 */  
printf ("a = %-7d\n", a);  /* left justify in field of 7 */  
  
c = 5;  
d = 8;  
printf ("a = %5d\n", c, a); /* use a field width of 5 */  
printf ("a = %8d\n", d, a); /* use a field width of 8 */  
printf ("\n");  
printf ("f = %f\n", f);    /* simple float output */  
  
(...)
```

© RMAC XII-2001

49

4. Atribuições e Comparações (11/25)

- tipos *long*, *short*, *unsigned* e *double* -

■ lottypes.c (cont.)

```
(...)  
  
printf ("f = %12f\n", f);  /* use field width of 12 */  
printf ("f = %12.3f\n", f); /* use 3 decimal places */  
printf ("f = %12.5f\n", f); /* use 5 decimal places */  
printf ("f = %-12.5f\n", f); /* left justify in field */  
}  
  
(...)
```

- os tipos *int*, *short int* e *long int* são todos semelhantes, uma vez que são capazes de representar valores numéricos positivos e negativos; a única diferença entre eles consiste no número de bytes de que cada um dispõe, condicionando, portanto, a gama de valores representáveis
- as variantes *unsigned* (sem sinal) daqueles 3 tipos (*unsigned int*, *unsigned short int* e *unsigned long int*) utilizam o mesmo número de bytes do que os tipos *signed* (*int*, *short int* e *long int*), mas para representar unicamente valores numéricos positivos

© RMAC XII-2001

50

4. Atribuições e Comparações (12/25)

- tipos *long*, *short*, *unsigned* e *double* -

■ *lottypes.c* (cont.)

- o tipo *double* é utilizado para representar números reais, tal como o tipo *float*, mas, como utiliza 8 bytes, cobre uma gama mais vasta de valores, i.e., entre $1.7E-308$ e $1.7E+308$
- para enviar para o écran cada um daqueles tipos de dados, é necessário utilizar, na função *printf* (), um carácter a seguir ao símbolo "%" que indica o tipo de dado envolvido na operação de saída, tal como no caso do tipo *int* se utiliza o carácter "d"

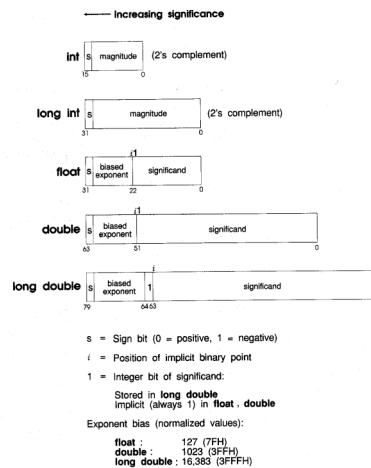
4. Atribuições e Comparações (13/25)

- tipos de dados -

Type	Bit Width	Range
char	8	-128 to 127
unsigned char	8	0 to 255
signed char	8	-128 to 127
int	16	-32768 to 32767
unsigned int	16	0 to 65535
signed int	16	-32768 to 32767
short int	16	-32768 to 32767
unsigned short int	16	0 to 65535
signed short int	16	-32768 to 32767
long int	32	-2147483648 to 2147483647
unsigned long int	32	0 to 4294967295
signed long int	32	-2147483648 to 2147483647
float	32	$3.4E-38$ to $3.4E+38$
double	64	$1.7E-308$ to $1.7E+308$
long double	64	$1/7E-308$ to $1.7E+308$

4. Atribuições e Comparações (14/25)

- tipos de dados -



© RMAC XII-2001

53

4. Atribuições e Comparações (15/25)

- formatação dos dados -

Code	Format
%c	A single character
%d	Decimal
%i	Decimal
%e	Scientific notation
%f	Decimal floating-point
%g	Uses %e or %f, whichever is shorter
%o	Octal
%s	String of characters
%u	Unsigned decimal
%x	Hexadecimal
%%	Prints a % sign
%p	Displays a pointer
%n	The associated argument will be an integer pointer into which is placed the number of characters written so far

© RMAC XII-2001

54

4. Atribuições e Comparações (16/25)

- comparações lógicas -

■ compares.c

```
/* This file will illustrate logical compares */

main ( )
{
    int x = 11, y = 11, z = 11;
    char a = 40, b = 40, c = 40;
    float r = 12.987, s = 12.987, t = 12.987;

    /* first group of compare statements */
    if (x == y) z = -13; /* this will set z = -13 */
    if (x > z) a = 'A'; /* this will set a = 65 */
    if (!(x > z)) a = 'B'; /* this will change nothing */
    if (b <= c) r = 0.0; /* this will set r = 0.0 */
    if (r != s) t = c/2; /* this will set t = 20 */

    (...)
}
```

© RMAC XII-2001

55

4. Atribuições e Comparações (17/25)

- comparações lógicas -

■ compares.c (cont.)

```
(...)

/* second group of compare statements */
if (x = (r != s)) z = 1000; /* this will set x = some positive
                           number and z = 1000 */
if (x = y) z = 222; /* this sets x = y, and z = 222 */
if (x != 0) z = 333; /* this sets z = 333 */
if (x) z = 444; /* this sets z = 444 */

/* third group of compare statements */
x = y = z = 77;
if ((x == y) && (x == 77)) z = 33; /* this sets z = 33 */
if ((x > y) || (z > 12)) z = 22; /* this sets z = 22 */
if (x && y && z) z = 11; /* this sets z = 11 */

(...)
```

© RMAC XII-2001

56

4. Atribuições e Comparações (18/25)

- comparações lógicas -

■ compares.c (cont.)

```
(...)  
  
if ((x = 1) && (y = 2) && (z = 3)) r = 12.00; /* this sets  
                                              x = 1, y = 2, z = 3, r = 12.00 */  
  
if ((x == 2) && (y = 3) && (z = 4)) r = 14.56; /* this doesn't  
                                              change anything */  
  
/* fourth group of compares */  
if (x == x); z = 27.345; /* z always gets changed */  
if (x != x) z = 27.345; /* nothing gets changed */  
if (x = 0) z = 27.345; /* this sets x = 0, z is unchanged */  
}
```

- em C, é possível iniciar as variáveis ao mesmo tempo em que são declaradas

© RMAC XII-2001

57

4. Atribuições e Comparações (19/25)

- operadores relacionais -

■ compares.c (cont.)

- em "compares.c", o primeiro grupo de instruções utiliza comparações simples, uma vez que em cada comparação estão envolvidas unicamente 2 variáveis

Relational Operators

Operator	Action
>	Greater than
>=	Greater than or equal
<	Less than
<=	Less than or equal
==	Equal
!=	Not equal

© RMAC XII-2001

58

4. Atribuições e Comparações (20/25)

- operadores lógicos -

■ compares.c (cont.)

- em “compares.c”, o segundo grupo de instruções utiliza comparações mais elaboradas, em que é necessário avaliar proposições lógicas compostas
- em C, as comparações booleanas utilizam como verdade e falsidade os seguintes conceitos:
 - verdade: qualquer valor numérico diferente de 0 (zero)
 - falsidade: o valor 0 (zero)
- em “compares.c”, o terceiro grupo de instruções utiliza as operações booleanas elementares

Logical Operators

Operator	Action
&&	AND
	OR
!	NOT

© RMAC XII-2001

59

4. Atribuições e Comparações (21/25)

- comparações lógicas -

■ compares.c (cont.)

- em “compares.c”, o quarto grupo de instruções utiliza comparações com “requintes de malvadez”
- exemplo de “compares.c”:
 - em `if (x == x); z = 27.345;` o “;” depois da condição faz com que a atribuição a z seja sempre executada
 - em `if (x != x) z = 27.345;` a condição é sempre falsa, pelo que a atribuição a z nunca é executada
 - em `if (x = 0) z = 27.345;` a atribuição a x garante que a condição é sempre falsa, pelo que a atribuição a z nunca é executada

© RMAC XII-2001

60

4. Atribuições e Comparações (22/25)

- código críptico -

■ cryptic.c

```
main ( )
{
  int x = 0, y = 2, z = 1025;
  float a = 0.0, b = 3.14159, c = -37.234;

  /* incrementing */
  x = x + 1; /* this increments x */
  x++;      /* this increments x */
  ++x;      /* this increments x */
  z = y++;  /* z = 2, y = 3 */
  z = ++y;  /* z = 4, y = 4 */
  (...)
}
```

© RMAC XII-2001

61

4. Atribuições e Comparações (23/25)

- código críptico -

■ cryptic.c (cont.)

```
(...)

/* decrementing */
y = y - 1; /* this decrements y */
y- -;      /* this decrements y */
- -y;      /* this decrements y */
y = 3;
z = y- -;  /* z = 3, y = 2 */
z = - -y;  /* z = 1, y = 1 */

/* arithmetic op */
a = a + 12; /* this adds 12 to a */
a += 12;    /* this adds 12 more to a */
a *= 3.2;   /* this multiplies a by 3.2 */
(...)
```

© RMAC XII-2001

62

4. Atribuições e Comparações (24/25)

- código críptico -

■ cryptic.c (cont.)

```
(...)  
  
a -= b;    /* this subtracts b from a */  
a /= 10.0; /* this divides a by 10.0 */  
  
/* conditional expression */  
a = (b >= 3.0 ? 2.0 : 10.5); /* this expression */  
if (b >= 3.0)                /* and this expression */  
    a = 2.0;                  /* are identical, both */  
else                          /* will cause the same */  
    a = 10.5;                 /* result */  
  
c = (a > b ? a : b);           /* c will have the max of a or b */  
c = (a > b ? b : a);           /* c will have the min of a or b */  
}
```

© RMAC XII-2001

63

4. Atribuições e Comparações (25/25)

- código críptico -

■ cryptic.c (cont.)

- em C, existem operações de incremento que não são nada intuitivas, uma vez que apresentam uma sintaxe pouco convencional

<code>y = x++;</code>	<code>y = ++x;</code>
Acontecem duas coisas por esta ordem: 1. O valor de x é atribuído a y 2. O valor de x é incrementado	Acontecem duas coisas por esta ordem: 1. O valor de x é incrementado 2. O valor de x é atribuído a y

- o mesmo ocorre para as outras operações aritméticas elementares, bem como para as expressões condicionais
- exemplos do "cryptic.c"
 - `a *= 3.2;` é equivalente a `a = a * 3.2;`
 - `c = (a > b ? a : b);` é equivalente a `if (a > b) c = a; else c = b;`

© RMAC XII-2001

64



exercícios

1. Escreva um programa que imprima o seu nome no écran.
2. Modifique o programa anterior para que, para além do seu nome, imprima também o seu endereço e o seu número de telefone, em linhas seguidas no écran.
3. Escreva um programa que imprima o seu nome no écran 10 vezes. Escreva este programa de três formas diferentes, recorrendo a um tipo de ciclo distinto.
4. Escreva um programa que conte de 1 a 10, imprima os valores em linhas separadas para cada um deles e que imprima, adicionalmente, duas mensagens à sua escolha, uma quando a contagem estiver em 3 e uma outra, quando a contagem estiver em 7.



exercícios

5. Escreva um programa que conte de 1 a 12 e imprima, em linhas separadas, os valores e os seus quadrados.
6. Escreva um programa que conte de 1 a 12 e imprima, em linhas separadas, os valores e os seus inversos com uma precisão de 5 casas decimais.
7. Escreva um programa que conte de 1 a 100 e imprima, numa única linha, somente os valores compreendidos entre 32 e 39.

5. Funções, Variáveis e Protótipos (1/26)

- invocação de funções -

■ sumsqres.c

```
int sum; /* this is a global variable */

main ( )
{
    int index;

    header ( ); /* this calls the function named header */

    for (index = 1; index <= 7; index++)
        square (index); /* this calls the square function */

    ending ( ); /* this calls the ending function */
}

(...)
```

© RMAC XII-2001

67

5. Funções, Variáveis e Protótipos (2/26)

- invocação de funções -

■ sumsqres.c (cont.)

```
(...)

header ( ) /* this is the function named header */
{
    sum = 0; /* initialize the variable sum */
    printf ("This is the header for the square program\n\n");
}

square (number) /* this is the square function */
int number;
{
    int numsq;

    (...)
}
```

© RMAC XII-2001

68

5. Funções, Variáveis e Protótipos (3/26)

- invocação de funções -

■ sumsqres.c (cont.)

```
(...)  
  
    numsq = number * number; /* this produces the square */  
    sum += numsq;  
    printf ("The square of %d is %d\n", number, numsq);  
} /* end of square function */  
  
ending () /* this is the ending function */  
{  
    printf ("\nThe sum of the squares is %d\n", sum);  
}
```

- em C, qualquer função é seguida de parêntesis, quer existam, ou não, parâmetros, como forma de não serem sintacticamente confundidas com variáveis

© RMAC XII-2001

69

5. Funções, Variáveis e Protótipos (4/26)

- invocação de funções -

■ sumsqres.c (cont.)

- em "sumsqres.c", para além das, já conhecidas, *main ()* e *printf ()* são definidas três funções:
 - *header ()*, chamada uma vez no início do programa
 - *square ()*, chamada 7 vezes (dentro do ciclo for) no programa
 - *ending ()*, chamada uma vez no final do programa
- quando uma função é chamada, o fluxo de controlo do programa passa para a primeira instrução da função chamada, regressando, posteriormente, à primeira instrução depois da chamada (na função chamadora), logo que a função chamada termine
- é sempre possível dispensar a utilização de funções, inserindo na *main ()* as instruções existentes no interior de cada uma das funções chamadas pela *main ()*
- as variáveis globais declaradas antes da *main ()* ficam visíveis (*scope*) a qualquer função do programa

© RMAC XII-2001

70

5. Funções, Variáveis e Protótipos (5/26)

- passagem de parâmetros por valor -

■ sumsqres.c (cont.)

- em "sumsqres.c", a função *square* () possui um parâmetro
 - exteriormente, a função *square* () é chamada, na *main* (), sendo-lhe passada, como parâmetro, a variável *index*
 - interiormente, a função *square* () denomina o parâmetro recebido do exterior como *number* (uma variável com um *scope* interno à função *square* () e declarada antes da chaveta de início de função - método clássico)
- de facto, quando se passa uma variável (por valor) como parâmetro numa chamada de uma função é realizada a cópia do valor da variável passada, valor esse que é atribuído à variável que internamente a representa
- desta forma, a variável passada preserva o seu valor independentemente do que a função chamada faça no seu interior com o valor copiado
- uma vez que as variáveis passadas como valor não permitem devolver resultados da função chamada à função chamadora, é possível utilizar as variáveis globais (apesar de tal não ser recomendável) para realizar essa tarefa

© RMAC XII-2001

71

5. Funções, Variáveis e Protótipos (6/26)

- devolução de resultados -

■ squares.c

```
main ( ) /* This is the main program */
{
    int x, y;

    for (x = 0; x <= 7; x++) {
        y = squ (x); /* go get the value of x*x */
        printf ("The square of %d is %d\n", x, y);
    }

    for (x = 0; x <= 7; ++x)
        printf ("The value of %d is %d\n", x, squ (x));
} /* end of main function */
```

(...)

© RMAC XII-2001

72

5. Funções, Variáveis e Protótipos (7/26)

- devolução de resultados -

■ squares.c (cont.)

```
(...)  
  
squ (in) /* function to get the value of in squared */  
int in;  
{  
    int square;  
  
    square = in * in;  
    return (square); /* this sets squ ( ) = square */  
}
```

© RMAC XII-2001

- em C, a instrução *return* permite que uma função devolva um (e SÓ um) resultado à função que a chamou, colocando entre os parêntesis a seguir ao *return* o valor (ou uma variável que o contenha) pretendido
- nestas circunstâncias, a própria função chamada pode ser vista como uma variável cujo tipo coincide com o do valor retornado

73

5. Funções, Variáveis e Protótipos (8/26)

- tipos dos resultados -

■ floatsq.c

```
float z; /* this is a global variable */  
  
main ( )  
{  
    int index;  
    float x, y, sqr ( ), glsqr ( );  
  
    for (index = 0; index <= 7; index++) {  
        x = index; /* convert int to float */  
        y = sqr (x); /* square x to a floating point variable */  
        printf ("The square of %d is %10.4f\n", index, y);  
    } /* end of for */  
  
    (...)
```

© RMAC XII-2001

74

5. Funções, Variáveis e Protótipos (9/26)

- tipos dos resultados -

■ floatsq.c (cont.)

```
(...)  
  
for (index = 0; index <= 7; index++) {  
    z = index;  
    y = glsqr ( );  
    printf ("The square of %d is %10.4fn", index, y);  
} /* end of for */  
} /* end of main function */  
  
float sqr (inval) /* square a float, return a float */  
float inval;  
{  
    float square;  
  
    (...)
```

© RMAC XII-2001

75

5. Funções, Variáveis e Protótipos (10/26)

- tipos dos resultados -

■ floatsq.c (cont.)

```
(...)  
  
    square = inval * inval;  
    return (square);  
}  
  
float glsqr ( ) /* square a float, return a float */  
{  
    return (z*z);  
}
```

© RMAC XII-2001

- em C, é possível definir o tipo dos valores retornados por uma função, como se ela fosse uma variável
- em C, por defeito, se nada for declarado em contrário, as funções quando retornam algo é do tipo *int*

76

5. Funções, Variáveis e Protótipos (11/26)

- *scope* das variáveis -

■ scope.c

```
#include "stdio.h" /* prototypes for input/output */
void head1 (void); /* prototype for head1      */
void head2 (void); /* prototype for head2      */
void head3 (void); /* prototype for head3      */

int count; /* this is a global variable */

main ( )
{
    register int index; /* this variable is available only in main */

    head1 ( );
    head2 ( );
    head3 ( );
```

(...)

© RMAC XII-2001

77

5. Funções, Variáveis e Protótipos (12/26)

- *scope* das variáveis -

■ scope.c (cont.)

```
(...)

for (index = 8; index > 0; index- -) {
    int stuff; /* this variable is only available in these braces */
    for (stuff = 0; stuff <= 6; stuff++) printf ("%d ", stuff);
    printf (" index is now %d\n", index);
} /* end of 1st for loop */
} /* end of main function */

int counter; /* this is available from this point on */
void head1 (void)
{
    int index; /* this variable is available only in head1 */
    index = 23;
```

(...)

© RMAC XII-2001

78

5. Funções, Variáveis e Protótipos (13/26)

- *scope* das variáveis -

■ scope.c (cont.)

```
(...)  
  
    printf ("The header1 value is %d\n", index);  
} /* end of head1 */  
  
void head2 (void)  
{  
    int count; /* this variable is available only in head2 */  
    count = 53;  
    printf ("The header2 value is %d\n", count);  
    counter = 77;  
}  
  
void head3 (void) {  
    printf ("The header3 value is %d\n", counter);  
}
```

© RMAC XII-2001

79

5. Funções, Variáveis e Protótipos (14/26)

- *scope* das variáveis -

■ scope.c (cont.)

- uma variável global é uma variável definida fora de qualquer função, no entanto, dependendo do local onde são declaradas podem, ou não, ser visíveis a todas as funções do programa
- uma variável global só é visível às funções que surgem depois da sua declaração
- exemplo do "scope.c"
 - a variável global *count* é visível a todas as funções, porque está declarada antes da *main* ()
 - a variável global *counter* só é visível às funções que surgem depois da sua declaração
- em C, dentro de uma função (variáveis locais), o *scope* de uma variável está limitado ao "espaço" delimitado pelas chavetas dentro das quais ela foi declarada

© RMAC XII-2001

80

5. Funções, Variáveis e Protótipos (15/26)

- variáveis *register* -

■ scope.c (cont.)

- em C, é permitida a utilização de um mesmo identificador para designar variáveis diferentes, não exigindo que se conheça a implementação de todas as funções para evitar a repetição de identificadores
- quando duas variáveis com o mesmo nome possuem *scopes* com coberturas sobrepostas, a variável com o *scope* mais alargado "cede" espaço à outra, reduzindo o seu *scope* para as zonas do programa em que não existam sobreposições
- um processador pode guardar dados em memória ou em registos, sendo que a memória possui uma capacidade de armazenamento de dados superior aos registos, mas apresenta tempos de acesso (para leitura e escrita) consideravelmente superiores
- quando, num programa, determinadas variáveis são intensiva e permanentemente utilizadas (para leitura e/ou escrita de dados) é possível indicar ao compilador a intenção de as implementar directamente num registo (*register int index*), de forma a acelerar a execução do programa

© RMAC XII-2001

81

5. Funções, Variáveis e Protótipos (16/26)

- variáveis *automatic* e *static* -

■ scope.c (cont.)

- por defeito, todas as variáveis (não globais, ou seja, locais) são do tipo *automatic*, i.e., são criadas sempre que o bloco de código (delimitado pelas chavetas dentro das quais são declaradas) inicia a sua execução e são destruídas sempre que o mesmo bloco termina a execução
- desta forma, as variáveis locais do tipo *automatic* não preservam o seu valor ao longo das várias execuções do bloco onde são declaradas
- pelo contrário, as variáveis locais do tipo *static* preservam o seu valor ao longo das várias execuções do bloco onde são declaradas
- as variáveis globais do tipo *static* não podem ser utilizadas fora do ficheiro em que são declaradas

© RMAC XII-2001

82

5. Funções, Variáveis e Protótipos (17/26)

- *prototypes* de funções -

■ scope.c (cont.)

- a utilização de *protótipos* (*prototypes*) das funções permite que o compilador realize uma série de verificações adicionais, nomeadamente no que diz respeito aos parâmetros e aos tipos dos valores retornados
- os *prototypes* das funções devem surgir antes da sua implementação
- *void* antes do nome da função declara a inexistência de valores retornados
- *void* entre os parêntesis a seguir ao nome da função declara a inexistência de parâmetros
- é possível armazenar *prototypes* de funções em ficheiros à parte e inclui-los com a instrução *#include file_name.h*

© RMAC XII-2001

83

5. Funções, Variáveis e Protótipos (18/26)

- *scope* das variáveis -

```
global declarations
main( )
{
    local variables
    statement sequence
}
f1( )
{
    local variables
    statement sequence
}
f2( )
{
    local variables
    statement sequence
}
.
.
fN( )
{
    local variables
    statement sequence
}
```

© RMAC XII-2001

84

5. Funções, Variáveis e Protótipos (19/26)

- funções recursivas -

■ recurson.c

```
main ()
{
    int index;
    index = 8;
    count_dn (index);
}

count_dn (count)
int count;
{
    count- -;
    printf ("The value of the count is %d\n", count);
    if (count > 0) count_dn (count);
    printf ("Now the count is %d\n", count);
}
```

© RMAC XII-2001

85

5. Funções, Variáveis e Protótipos (20/26)

- funções recursivas -

■ recurson.c (cont.)

- recursividade é uma função chamar-se a ela própria
- uma função recursiva chama-se sucessivamente até atingir um estado em que a condição de paragem actua, começando, então, o "caminho de regresso" até à chamada original na qual termina o ciclo de auto-invocações
- o ciclo de auto-invocações não é exactamente igual à situação em que se possuem tantas funções iguais quantos os níveis de recursividade atingido e em que as várias funções se chamam umas às outras
- de facto, a grande diferença consiste no facto de que com a recursividade todos os recursos utilizados numa invocação são exactamente os mesmos que foram utilizados na invocação anterior, pelo que se torna necessário recorrer à *stack* do computador para armazenar os dados anteriores até ao momento em que "caminho de regresso" traga o fluxo de volta àquele nível de recursividade

© RMAC XII-2001

86

5. Funções, Variáveis e Protótipos (21/26)

- definição de funções pelo método moderno -

■ backward.c

```
#include "stdio.h" /* prototypes for standard input/output */
#include "string.h" /* prototypes for string operations */

void forward_and_backwards (char line_of_char [], int index);

void main (void)
{
    char line_of_char [80];
    int index = 0;

    strcpy (line_of_char, "This is a string.\n");

    forward_and_backwards (line_of_char, index);
}
```

(...)

© RMAC XII-2001

87

5. Funções, Variáveis e Protótipos (22/26)

- definição de funções pelo método moderno -

■ backward.c (cont.)

```
(...)

void forward_and_backwards (char line_of_char [], int index)
{
    if (line_of_char [index]) {
        printf ("%c", line_of_char [index]);
        index++;
        forward_and_backwards (line_of_char, index);
    } /* end of if */
    printf ("%c", line_of_char [index]);
}
```

- em C, é possível definir *prototypes* e funções pelo método moderno, movendo para dentro dos parêntesis os tipos dos parâmetros

© RMAC XII-2001

88

5. Funções, Variáveis e Protótipos (23/26)

- função *strcpy* () -

strcpy

Function Copies one string into another.

Syntax `#include <string.h>`
`char *strcpy(char *dest, const char *src);`

DOS	UNIX	Windows	ANSI C	C++ only
■	■	■	■	

Remarks Copies string *src* to *dest*, stopping after the terminating null character has been moved.

Return value **strcpy** returns *dest*.

See also **strcpy**

5. Funções, Variáveis e Protótipos (24/26)

- definição de funções pelo método moderno -

■ floatsq2.c

```
#include "stdio.h" /* prototypes for standard input/outputs */
```

```
float sqr (float inval);
```

```
float glsqr (void);
```

```
float z; /* this is a global variable */
```

```
main ( )
```

```
{
```

```
int index;
```

```
float x, y;
```

```
for (index = 0; index <= 7; index++) {
```

```
    x = index; /* convert int to float */
```

```
    y = sqr (x); /* square x to a floating point variable */
```

```
(...)
```

5. Funções, Variáveis e Protótipos (25/26)

- definição de funções pelo método moderno -

■ floatsq2.c (cont.)

```
(...)  
  
printf ("The square of %d is %10.4f\n", index, y);  
}  
  
for (index = 0; index <= 7; index++) {  
    z = index;  
    y = glsqr ( );  
    printf ("The square of %d is %10.4f\n", index, y);  
}  
}  
  
float sqr (float inval) /* square a float, return a float */  
{  
    float square;
```

(...)

© RMAC XII-2001

91

5. Funções, Variáveis e Protótipos (26/26)

- definição de funções pelo método moderno -

■ floatsq2.c

```
(...)  
  
square = inval * inval;  
return (square);  
}  
  
float glsqr (void) /* square a float, return a float */  
{  
    return (z*z);  
}
```

- "floatsq.c" é algoritmicamente equivalente ao "floatsq2.c" com a única diferença de que "floatsq2.c" declara os *prototypes* e as funções com o método moderno

© RMAC XII-2001

92

6. Defines e Macros (1/4)

■ define.c

```
#define START 0 /* starting point of loop */
#define ENDING 9 /* ending point of loop */
#define MAX (A, B) ((A) > (B) ? (A) : (B)) /* Max macro definition */
#define MIN (A, B) ((A) > (B) ? (B) : (A)) /* Min macro definition */

main ()
{
    int index, mn, mx;
    int count = 5;
    for (index = START; index <= ENDING; index++) {
        mx = MAX (index, count);
        mn = MIN (index, count);
        printf ("Max is %d and min is %d\n", mx, mn);
    } /* end of for */
} /* end of main */
```

© RMAC XII-2001

93

6. Defines e Macros (2/4)

■ define.c (cont.)

- o primeiro passo de compilação consiste num pre-processamento do texto do programa em que todos os *#define* são resolvidos, i.e., TODOS os identificadores e macros são substituídos no programa, onde quer que estejam, pelos valores e expressões com os quais foram definidos
- com é possível definir constantes cuja alteração é realizada por uma mera modificação de UMA única linha, em vez de tantas modificações quantas as vezes que a constante aparece em todos os ficheiros do programa
- uma macro é um tipo especial de *#define* em que é realizada uma operação matemática simples ou em que é tomada uma decisão booleana simples

© RMAC XII-2001

94

6. Defines e Macros (3/4)

■ macro.c

```
#define WRONG (A) A*A*A /* wrong macro for cube */
#define CUBE (A) (A)*(A)*(A) /* right macro for cube */
#define SQUR (A) (A)*(A) /* right macro for square */
#define START 1
#define STOP 9
main ()
{
    int i, offset;
    offset = 5;
    for (i = START; i <= STOP; i++) {
        printf ("The square of %3d is %4d, and its cube is %6d\n",
            i+offset, SQUR (i+offset), CUBE (i+offset));
        printf ("The wrong of %3d is %6d\n", i+offset, WRONG (i+offset));
    }
}
```

© RMAC XII-2001

95

6. Defines e Macros (4/4)

■ macro.c (cont.)

- a macro “wrong” não funciona em todas as situações devidos a problemas nas regras de precedência na execução das operações aritméticas
- exemplo:
 - $WRONG(1+5) \rightarrow 1+5 * 1+5 * 1+5$ $CUBE(1+5) \rightarrow (1+5) * (1+5) * (1+5) = 216$
- comparativamente, as macros são mais rápidas do que as funções (em tempo de execução), mas exigem mais memória

© RMAC XII-2001

96

7. Strings e Arrays (1/18)

- declaração de strings -

■ chrstrg.c

```
#include "stdio.h" /* prototypes for standard input/output */
#include "string.h" /* prototypes for string operations */
main ( )
{
    char name [5]; /* define a string of characters */

    name [0] = 'D';
    name [1] = 'a';
    name [2] = 'v';
    name [3] = 'e';
    name [4] = 0; /* null character - end of text */

    printf ("The name is %s\n", name);
    printf ("One letter is %c\n", name [2]);
    printf ("Part of the name is %s\n", &name [1]);
}
```

© RMAC XII-2001

97

7. Strings e Arrays (2/18)

- declaração de strings -

■ chrstrg.c (cont.)

- um **array** é uma sequência de dados homogêneos (do mesmo tipo) armazenados sequencialmente na memória
- uma **string** é um caso particular de um **array**, em que os seus elementos são do tipo **char** (codificados segundo a tabela de códigos ASCII)
- em C, a declaração de **arrays** exige a identificação, entre parênteses rectos, do número de elementos no **array**
- em C, como o primeiro elemento é indexado a 0, o índice do último elemento do é sempre igual ao valor declarado na definição do **array** subtraído de uma unidade
- em C, como uma **string** tem que acabar **SEMPRE** com o valor nulo 0 (zero), de facto, o número máximo de caracteres de uma **string** é sempre igual ao valor declarado na definição do **array** subtraído de uma unidade
- o parâmetro **"%s"**, permite a utilização de **strings** na função **printf ()**

© RMAC XII-2001

98

7. Strings e Arrays (3/18)

■ Um array e os endereços de memória

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
1000	1001	1002	1003	1004	1005	1006

7. Strings e Arrays (4/18)

- códigos ASCII e EBCDIC -

ASCII Symbol	HEX Code for 7-Bit ASCII	EBCDIC Symbol	HEX Code for EBCDIC	ASCII Symbol	HEX Code for 7-Bit ASCII	EBCDIC Symbol	HEX Code for EBCDIC	ASCII Symbol	HEX Code for 7-Bit ASCII	EBCDIC Symbol	HEX Code for EBCDIC
NUL	00	NUL	00	*	2A	*	5C	T	54	T	E3
SOH	01	SOH	01	+	2B	+	4E	U	55	U	E4
STX	02	STX	02	.	2C	.	6B	V	56	V	E5
ETX	03	ETX	03	-	2D	-	60	W	57	W	E6
EOF	04	EOF	37	-	2E	-	4B	X	58	X	E7
ENQ	05	ENQ	2D	/	2F	/	61	Y	59	Y	E8
ACK	06	ACK	2E	0	30	0	F0	Z	5A	Z	E9
BEL	07	BEL	2F	1	31	1	F1	[5B	[EA
BS	08	BS	16	2	32	2	F2	\	5C	\	EB
HT	09	HT	05	3	33	3	F3]	5D]	EC
LF	0A	LF	25	4	34	4	F4	^	5E	^	ED
VT	0B	VT	08	5	35	5	F5	_	5F	_	EE
FF	0C	FF	0C	6	36	6	F6	RES			
CR	0D	CR	0D	7	37	7	F7	a	61	a	E1
SO	0E	SO	0E	8	38	8	F8	b	62	b	E2
SI	0F	SI	0F	9	39	9	F9	c	63	c	E3
DLE	10	DLE	10	:	3A	:	7A	d	64	d	E4
DC1	11	DC1	11	;	3B	;	5E	e	65	e	E5
DC2	12	DC2	12	—	3C	—	4C	f	66	f	E6
DC3	13	DC3	13	=	3D	=	7E	g	67	g	E7
DC4	14	DC4	35	\	3E	\	6E	h	68	h	E8
NAK	15	NAK	3D	?	3F	?	6F	i	69	i	E9
SYN	16	SYN	32	@	40	@	7C	j	6A	j	EA
ETB	17	ETB	26	A	41	A	C1	k	6B	k	EB
CAN	18	CAN	18	B	42	B	C2	l	6C	l	EC
EM	19	EM	19	C	43	C	C3	m	6D	m	ED
SUB	1A	SUB	3F	D	44	D	C4	n	6E	n	EE
ESC	1B	ESC	24	E	45	E	C5	o	6F	o	EF
FS	1C	FS	1C	F	46	F	C6	p	70	p	F0
GS	1D	GS	1D	G	47	G	C7	q	71	q	F1
RS	1E	RS	1E	H	48	H	C8	r	72	r	F2
US	1F	US	1F	I	49	I	C9	s	73	s	F3
SP	20	SP	40	J	4A	J	DA	t	74	t	F4
!	21	!	5A	K	4B	K	DB	u	75	u	F5
"	22	"	7F	L	4C	L	DC	v	76	v	F6
#	23	#	7B	M	4D	M	DD	w	77	w	F7
\$	24	\$	5B	N	4E	N	DE	x	78	x	F8
%	25	%	6C	O	4F	O	DF	y	79	y	F9
&	26	&	50	P	50	P	DA	z	7A	z	FA
'	27	'	7D	Q	51	Q	DB	[7B	[FB
(28	(4D	R	52	R	DC]	7C]	FC
)	29)	5D	S	53	S	DD	^	7D	^	FD
								_	7E	_	FE
								DEL	7F	DEL	FF

7. Strings e Arrays (5/18)

- códigos ASCII de 7 bits -

Valor ASCII	Caracter	Valor ASCII	Caracter	Valor ASCII	Caracter
032	branco	063	?	093	l
033	!	064	@	094	.
034	"	065	A	095	—
035	#	066	B	096	'
036	\$	067	C	097	a
037	%	068	D	098	b
038	&	069	E	099	c
039	'	070	F	100	d
040	(071	G	101	e
041)	072	H	102	f
042	*	073	I	103	g
043	+	074	J	104	h
044	,	075	K	105	i
045	—	076	L	106	j
046	.	077	M	107	k
047	/	078	N	108	l
048	0	079	O	109	m
049	1	080	P	110	n
050	2	081	Q	111	o
051	3	082	R	112	p
052	4	083	S	113	q
053	5	084	T	114	r
054	6	085	U	115	s
055	7	086	V	116	t
056	8	087	W	117	u
057	9	088	X	118	v
058	:	089	Y	119	w
059	;	090	Z	120	x
060	<	091	[121	y
061	=	092	\	122	z
062	>				

© RMAC XII-2001

101

7. Strings e Arrays (6/18)

- operações sobre strings -

■ strings.c

```
main ( )
{
char name1 [12], name2 [12], mixed [25];
char title [20];

strcpy (name1, "Rosalinda"); /* name1 receives "Rosalinda" */
strcpy (name2, "Zeke"); /* name2 receives "Zeke" */
strcpy (title, "This is the title."); /* title receives "This is the title." */

printf (" %s\n\n", title);
printf ("Name 1 is %s\n", name1);
printf ("Name 2 is %s\n", name2);
}
```

(...)

© RMAC XII-2001

102

7. Strings e Arrays (7/18)

- operações sobre strings -

■ strings.c (cont.)

```
(...)  
  
if (strcmp (name1, name2) > 0) /* returns 1 if name1 > name2 */  
    strcpy (mixed, name1);  
else  
    strcpy (mixed, name2);  
  
printf ("The biggest name alphabetically is %s\n", mixed);  
  
strcpy (mixed, name1);  
strcat (mixed, " ");  
strcat (mixed, name2);  
printf ("Both names are %s\n", mixed);  
}
```

© RMAC XII-2001

103

7. Strings e Arrays (8/18)

- função *strcmp* () -

strcmp

Function Compares one string to another.

Syntax #include <string.h>
int strcmp(const char *s1, const char *s2);

DOS	UNIX	Windows	ANSI C	C++ only
■	■	■	■	

Remarks **strcmp** performs an unsigned comparison of *s1* to *s2*, starting with the first character in each string and continuing with subsequent characters until the corresponding characters differ or until the end of the strings is reached.

Return value **strcmp** returns a value that is

< 0 if *s1* is less than *s2*
== 0 if *s1* is the same as *s2*
> 0 if *s1* is greater than *s2*

See also **strcmpi**, **strcoll**, **stricmp**, **strncmp**, **strncmpi**, **strnicmp**

© RMAC XII-2001

104

7. Strings e Arrays (9/18)

- função *strcat* () -

strcat, *_fstrcat*

Function Appends one string to another.

Syntax #include <string.h>

Near version: char **strcat*(char **dest*, const char **src*);

Far version: char far * far _*fstrcat*(char far **dest*, const char far **src*)

Near version

DOS	UNIX	Windows	ANSI C	C++ only
■	■	■	■	
■		■		

Far version

Remarks *strcat* appends a copy of *src* to the end of *dest*. The length of the resulting string is **strlen(*dest*) + strlen(*src*)**.

Return value *strcat* returns a pointer to the concatenated strings.

© RMAC XII-2001

105

7. Strings e Arrays (10/18)

- declaração de arrays -

■ *intarray.c*

```
main ()
{
    int values [12];
    int index;

    for (index = 0; index < 12; index++)
        values [index] = 2 * (index + 4);

    for (index = 0; index < 12; index++)
        printf ("The value at index = %2d is %3d\n", index, values [index]);
}
```

- em C, na declaração de um *array* não é necessário contabilizar um caracter terminador, pelo que o valor entre parêntesis rectos é igual ao número de elementos a armazenar no *array*

© RMAC XII-2001

106

7. Strings e Arrays (11/18)

- declaração de strings sem dimensionamento -

■ bigarray.c

```
char name1 [ ] = "First Program Title";

main ( )
{
    int index;
    int stuff [12];
    float weird [12];
    static char name2 [ ] = "Second Program Title";

    for (index = 0; index < 12; index++) {
        stuff [index] = index + 10;
        weird [index] = 12.0 * (index + 7);
    }
}
```

(...)

© RMAC XII-2001

107

7. Strings e Arrays (12/18)

- declaração de strings sem dimensionamento -

■ bigarray.c (cont.)

```
(...)

printf ("%s\n", name1);
printf ("%s\n\n", name2);
for (index = 0; index < 12; index++)
    printf ("%5d %5d %10.3f\n", index, stuff [index], weird [index]);
}
```

- é possível declarar e iniciar *strings*, deixando o seu dimensionamento a cargo do compilador

© RMAC XII-2001

108

7. Strings e Arrays (13/18)

- passagem de arrays como parâmetros -

■ passback.c

```
main ( )
{
    int index;
    int matrix [20];
    for (index = 0; index < 20; index++) /* generate data */
        matrix [index] = index + 1;

    for (index = 0; index < 5; index++) /* print original data */
        printf ("Start matrix [%d] = %d\n", index, matrix [index]);

    dosome (matrix); /* go to a function & modify matrix */

    for (index = 0; index < 5; index++) /* print modified matrix */
        printf ("Back matrix [%d] = %d\n", index, matrix [index]);
}
```

(...)

© RMAC XII-2001

109

7. Strings e Arrays (14/18)

- passagem de arrays como parâmetros -

■ passback.c (cont.)

```
(...)

dosome (list) /* this will illustrate returning data */
int list [ ];
{
    int i;

    for (i = 0; i < 5; i++) /* print original matrix */
        printf ("Before matrix [%d] = %d\n", i, list [i]);

    for (i = 0; i < 20; i++) /* add 10 to all values */
        list [i] += 10;

    for (i = 0; i < 5; i++) /* print modified matrix */
        printf ("After matrix [%d] = %d\n", i, list [i]);
}
```

© RMAC XII-2001

110

7. Strings e Arrays (15/18)

- passagem de arrays como parâmetros -

■ passback.c (cont.)

- a passagem do nome de um *array* como parâmetro de uma função, resulta, de facto, na passagem do endereço do primeiro elemento do *array*
- desta forma, como os elementos de um *array* se encontram sequencialmente na memória torna-se possível, dentro da função chamada, ter acesso a TODOS os elementos do *array*
- assim, esta passagem (por referência) permite que, dentro da função chamada, seja possível alterar os valores dos vários elementos do *array*

7. Strings e Arrays (16/18)

- arrays bidimensionais -

■ multiary.c

```
main ( )
{
  int i, j;
  int big [8] [8], large [25] [12];

  for (i = 0; i < 8; i++)
    for (j = 0; j < 8; j++)
      big [i] [j] = i * j; /* this is a multiplication table */

  for (i = 0; i < 25; i++)
    for (j = 0; j < 12; j++)
      large [i] [j] = i + j; /* this is an addition table */

  (...)
```


7. Strings e Arrays (17/18)

- arrays bidimensionais -

■ multiary.c (cont.)

```
(...)  
  
big [2] [6] = large [24] [10] * 22;  
big [2] [2] = 5;  
big [ big [2] [2] ] [ big [2] [2] ] = 177; /* this is big [5] [5] = 177; */  
  
for (i = 0; i < 8; i++) {  
    for (j = 0; j < 8; j++) printf ("%5d ", big [i] [j]);  
    printf ("\n"); /* newline for each increase in i */  
}
```

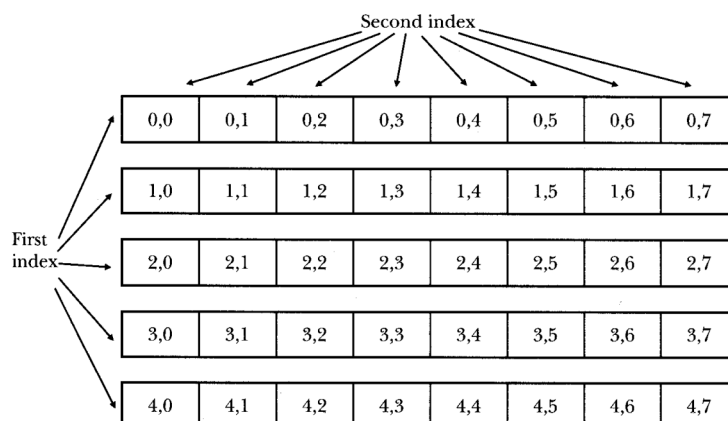
- é possível declarar **arrays** bidimensionais
- dois ciclos encadeados permitem percorrer todos os elementos de um **array** bidimensional

© RMAC XII-2001

113

7. Strings e Arrays (18/18)

- arrays bidimensionais -



© RMAC XII-2001

114

8. Apontadores (1/10)

■ pointer.c

```
main ( ) /* illustration of pointer use */
{
    int index,*pt1, *pt2; /* last two declarations correspond to pointers*/

    index = 39; /* any numerical value */
    pt1 = &index; /* the address of index */
    pt2 = pt1;
    printf ("The value is %d %d %d\n", index, *pt1, *pt2);
    *pt1 = 13; /* this changes the value of index */
    printf ("The value is %d %d %d\n", index, *pt1, *pt2);
}
```

- de uma forma simplista, um apontador é um endereço, i.e, em vez de ser uma variável, é um apontador para uma variável armazenada algures no espaço de endereçamento do programa

© RMAC XII-2001

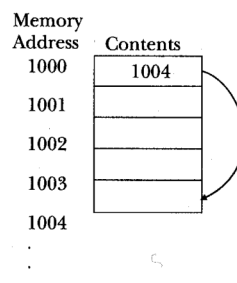
115

8. Apontadores (2/10)

- endereço e conteúdo de uma variável -

■ pointer.c (cont.)

- o identificador de uma variável precedido de um & (*ampersand*) define o endereço da variável, pelo que aponta para a variável
- um identificador precedido de um * (*star*) refere-se ao valor da variável para o qual o apontador aponta



© RMAC XII-2001

116

8. Apontadores (3/10)

- string como um apontador -

■ pointer2.c

```
main ( )
{
    char strg [40], *there, one, two;
    int *pt, list [100], index;

    strcpy (strg, "This is a character string.");

    one = strg [0]; /* one and two are identical */
    two = *strg;
    printf ("The first output is %c %c\n", one, two);

    one = strg [8]; /* one and two are identical */
    two = *(strg+8);
    printf ("the second output is %c %c\n", one, two);
    (...)
}
```

© RMAC XII-2001

117

8. Apontadores (4/10)

- string como um apontador -

■ pointer2.c (cont.)

```
(...)

there = strg+10; /* strg+10 is identical to strg [10] */
printf ("The third output is %c\n", strg [10]);
printf ("The fourth output is %c\n", *there);

for (index = 0; index < 100; index++)
    list [index] = index + 100;
pt = list + 27;
printf ("The fifth output is %d\n", list [27]);
printf ("The sixth output is %d\n", *pt);
}
```

© RMAC XII-2001

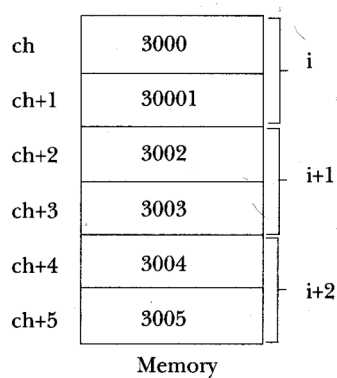
- uma *string* é em tudo igual a um apontador, excepto na impossibilidade de ser alterado

118

8. Apontadores (5/10)

- aritmética de apontadores -

```
char *ch=3000;  
int *i=3000;
```



© RMAC XII-2001

119

8. Apontadores (6/10)

- passagem de parâmetros por referência -

■ twoway.c

```
main ()  
{  
    int pecans, apples;  
  
    pecans = 100;  
    apples = 101;  
    printf ("The starting values are %d %d\n", pecans, apples);  
  
    /* when we call "fixup" */  
    fixup (pecans, &apples); /* we take the value of pecans */  
    /* we take the address of apples */  
  
    printf ("The ending values are %d %d\n", pecans, apples);  
}
```

(...)

© RMAC XII-2001

120

8. Apontadores (7/10)

- passagem de parâmetros por referência -

■ twoway.c (cont.)

```
(...)  
  
fixup (nuts, fruit) /* nuts is an integer value */  
int nuts,*fruit; /* fruit points to an integer */  
{  
    printf ("The values are %d %d\n", nuts, *fruit);  
    nuts = 135;  
    *fruit = 172;  
    printf ("The values are %d %d\n", nuts, *fruit);  
}
```

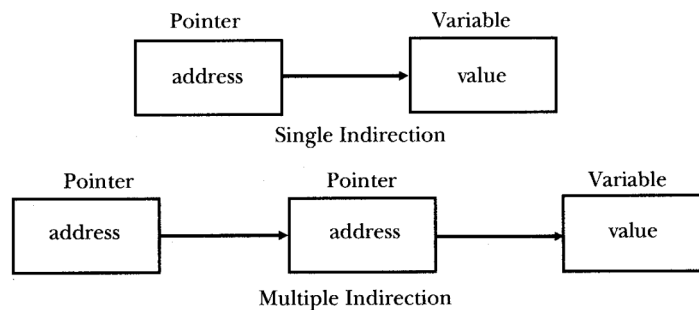
- em passagens de parâmetros por referência, passam-se os endereços das variáveis, de tal forma que, contrariamente às passagens por valor, no interior da função chamada, a variável pode ser alterada por intermédio do apontador, mas este, mesmo que seja alterado, não causa danos externos (o apontador interno é uma cópia do externo)

© RMAC XII-2001

121

8. Apontadores (8/10)

- indirecção simples e múltipla -



© RMAC XII-2001

122

8. Apontadores (9/10)

- tipos de apontadores -

Declarator syntax	Implied type of name	Example
type name;	type	int count;
type name[];	(open) array of type	int count[];
type name[3];	Fixed array of three elements, all of type (name[0], name[1], and name[2])	int count[3];
type *name;	Pointer to type	int *count;
type *name[];	(open) array of pointers to type	int *count[];
type *(name[]);	Same as above	int *(count[]);
type (*name)[];	Pointer to an (open) array of type	int (*count) [];
type &name;	Reference to type (C++ only)	int &count;
type name();	Function returning type	int count();
type *name();	Function returning pointer to type	int *count();
type *(name());	Same as above	int *(count());
type (*name)();	Pointer to function returning type	int (*count)();


Note the need for parentheses in *(*name)[]* and *(*name)()*, since the precedence of both the array declarator *[]* and the function declarator *()* is higher than the pointer declarator ***. The parentheses in **(name[])* are optional.

© RMAC XII-2001

123

8. Apontadores (10/10)

- apontadores para funções -



B8 87 55	6A 64
C8 02 00 00	9A 96 0E 0F 05
56	FF 76 06
57	6A 32
1E	6A 64
8E D8	9A 96 0E 0F 05
FF 76 06	FF 76 06
6A 32	6A 32
6A 32	6A 32
9A AA 0E 0F 05	9A 96 0E 0F 05
FF 76 06	E9 00 00
6A 64	1F
6A 32	5F
9A 96 0E 0F 05	5E
FF 76 06	C9
6A 64	CA 02 00

© RMAC XII-2001

124



exercícios

1. Re-escreva o programa "tempconv.c", de forma a que o cálculo da temperatura seja realizada dentro de uma função.
2. Escreva um programa que imprima o seu nome no écran 10 vezes, chamando uma função para realizar a impressão. Escreva este programa de três formas diferentes, recorrendo a um tipo de ciclo distinto.
3. Introduza os *prototypes* necessários ao programa "recurson.c", para eliminar os *warnings* do compilador.
4. Escreva um programa que conte de 7 a -5 (contagem decrescente). Recorra a *#define* para definir os limites da contagem.
5. Escreva um programa que declare e inicie três pequenas *strings*, cada uma com 6 caracteres de comprimento. Utilize a função *strcpy ()* para copiar "one", "two" e "three" para cada uma das três *strings*. Concatene as três *strings* e imprima o resultado 10 vezes.

© RMAC XII-2001

125



exercícios

6. Declare dois *arrays* de inteiros, cada um com 10 elementos, chamados "array1" e "array2". Com o auxílio de um ciclo, escreva valores sem significado em cada um dos *arrays* e adicione termo a termo os dois *arrays*, colocando o resultado num terceiro *array* chamado "arrays". Finalmente, imprima todos os resultados numa tabela, juntamente com o respectivo índice.

1	2 + 10 = 12	
2	4 + 20 = 24	
3	6 + 30 = 36	etc.
7. Declare uma *string* e recorra à função *strcpy ()* para copiar uma outra *string* para dentro da primeira. Imprima a *string*, recorrendo a um ciclo e a um apontador para imprimir um carácter de cada vez (inicie o apontador para o primeiro elemento e utilize ++ para incrementar o apontador).
8. Modifique o programa anterior, de forma a imprimir a *string* de trás para a frente (inicie o apontador para o último elemento e utilize -- para decrementar o apontador).

© RMAC XII-2001

126

9. Standard Input/Output (1/19)

■ simpleio.c

```
#include "stdio.h" /* standard header for input/output */

main ( )
{
    char c;

    printf ("Enter any characters, X = halt program.\n");

    do {
        c = getchar ( ); /* get a single character from the kb */
        putchar (c); /* display the character on the monitor */
    } while (c != 'X'); /* until an X is hit */

    printf ("\nEnd of program.\n");
}
```

© RMAC XII-2001

127

9. Standard Input/Output (2/19)

- *stdin* e *stdout* -

■ simpleio.c (cont.)

- a expressão *standard I/O* refere-se aos "locais" mais frequentes de onde os dados são lidos (*stdin*) e para onde os dados são enviados (*stdout*)
- desta forma, leituras e escritas de e naqueles dispositivos periféricos não necessitam ser explicitamente referidos nas instruções dos programas
- *#include "stdio.h"* é tratada pelo pré-processador, de forma a incluir no ficheiro do programa todo o conteúdo do ficheiro de *header*
- cada *header file* possibilita o acesso a um conjunto bem definido de funções; o *stdio.h*, por exemplo, possibilita a utilização das funções de acesso ao *stdin* (teclado) e ao *stdout* (monitor)
- quando os ficheiros de *header* são referidos entre aspas (*#include "file_name.h"*), o pré-processador vai iniciar a procura do ficheiro na directoria corrente
- quando os ficheiros de *header* são referidos entre maior e menor (*#include <file_name.h>*), o pré-processador vai iniciar a procura do ficheiro na directoria especificada no ambiente

© RMAC XII-2001

128

9. Standard Input/Output (3/19)

- ficheiros de *header* das bibliotecas de C -

Header File	Purpose or Use	Header File	Purpose or Use
DIRECT.H	Directory management (C++)	ALLOC.H	Dynamic allocation functions
DIRENT.H	Support for POSIX (C++)	ASSERT.H	Defines the assert() macro (ANSI C)
DOS.H	DOS interfacing functions	BCD.H	Defines the bcd class (C++)
ERRNO.H	Defines error codes (ANSI C)	BIOS.H	ROM-BIOS functions
FCNTL.H	Defines constants used by open() function	COMPLEX.H	Defines the complex number class (C++)
FLOAT.H	Defines implementation-dependent floating-point values (ANSI C)	CONIO.H	Screen-handling functions
FSTREAM.H	File I/O class definitions (C++)	CONSTEA.H	Console I/O (C++)
GENERIC.H	Macros for making generic class declarations (C++)	CTYPE.H	Character-handling functions (ANSI C)
GRAPHICS.H	Graphics functions	DIR.H	Directory-handling functions
IO.H	UNIX-like I/O routines		
IOMANIP.H	Defines I/O manipulators (C++)		
IOSTREAM.H	Defines I/O stream class (C++)		
LIMITS.H	Defines various implementation-dependent limits (ANSI C)		
LOCALE.H	Country and language specific functions (ANSI C)		
MALLOC.H	Dynamic allocation		
MATH.H	Various definitions used by the math library (ANSI C)		
MEM.H	Memory manipulation functions		
MEMORY.H	Same as MEM.H		
NEW.H	Allows alternative function to be called when NEW fails		
PROCESS.H	Spawn() and exec() functions		
SEARCH.H	Searching and sorting		
SETJMP.H	Nonlocal jumps (ANSI C)		
SHARE.H	File sharing		
SIGNAL.H	Defines signal values (ANSI C)		
STDARG.H	Variable-length argument lists (ANSI C)		
STDDEF.H	Defines some commonly used constants (ANSI C)		
STDIO.H	Declarations for standard I/O streams (ANSI C)		
STDIOSTR.H	Stream classes that use FILE structures (C++)		
STDLIB.H	Miscellaneous declarations (ANSI C)		
STREAM.H	Defines old stream class (C++)		

© RMAC XII-2001

129

9. Standard Input/Output (4/19)

- função *getchar* () -

getchar

Function Gets character from stdin.

Syntax `#include <stdio.h>`
`int getchar(void);`

DOS	UNIX	Windows	ANSI C	C++ only
■	■		■	

Remarks **getchar** is a macro that returns the next character on the named input stream stdin. It is defined to be **getc(stdin)**.

Return value On success, **getchar** returns the character read, after converting it to an **int** without sign extension. On end-of-file or error, it returns EOF.

See also **fgetc**, **fgetchar**, **getc**, **getch**, **getche**, **gets**, **putc**, **putchar**, **scanf**, **ungetc**

© RMAC XII-2001

130

9. Standard Input/Output (5/19)

- função *getc* () -

getc

Function Gets character from stream.

Syntax #include <stdio.h>
int getc(FILE *stream);

DOS	UNIX	Windows	ANSI C	C++ only
■	■	■	■	

Remarks **getc** is a macro that returns the next character on the given input stream and increments the stream's file pointer to point to the next character.

Return value On success, **getc** returns the character read, after converting it to an **int** without sign extension. On end-of-file or error, it returns EOF.

See also **fgetc**, **getch**, **getchar**, **getc**, **gets**, **putc**, **putchar**, **ungetc**

9. Standard Input/Output (6/19)

- função *putchar* () -

putchar

Function Outputs character on stdout.

Syntax #include <stdio.h>
int putchar(int c);

DOS	UNIX	Windows	ANSI C	C++ only
■	■	■	■	

Remarks **putchar(c)** is a macro defined to be **putc(c, stdout)**.

Return Value On success, **putchar** returns the character *c*. On error, **putchar** returns EOF.

See also **fputchar**, **getc**, **getchar**, **printf**, **putc**, **putch**, **puts**, **putw**, **vprintf**

9. Standard Input/Output (7/19)

- função **putc** () -

putc

Function Outputs a character to a stream.

Syntax `#include <stdio.h>`
`int putc(int c, FILE *stream);`

DOS	UNIX	Windows	ANSI C	C++ only
■	■	■	■	

Remarks **putc** is a macro that outputs the character *c* to the stream given by *stream*.

Return Value On success, **putc** returns the character printed, *c*. On error, **putc** returns EOF.

See also **fprintf**, **fputc**, **fputchar**, **fputs**, **fwrite**, **getc**, **getchar**, **printf**, **putch**, **putchar**, **putw**, **vprintf**

9. Standard Input/Output (8/19)

■ singleio.c

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

main ()
{
    char c;

    printf ("Enter any characters, terminate program with X\n");

    do {
        c = getch (); /* get a character */
        putchar (c); /* display the hit key */
    } while (c != 'X');

    printf ("\nEnd of program.\n");
}
```

9. Standard Input/Output (9/19)

- função *getch* () -

getch

Function Gets character from keyboard, does not echo to screen.

Syntax #include <conio.h>
int getch(void);

DOS	UNIX	Windows	ANSI C	C++ only
■				

Remarks *getch* reads a single character directly from the keyboard, without echoing to the screen.

Return value *getch* returns the character read from the keyboard.

9. Standard Input/Output (10/19)

- caracteres especiais CR (13) e LF (10) -

■ *betterin.c*

```
#include "stdio.h"
#include "conio.h"
#define CR 13 /* this defines CR to be 13 */
#define LF 10 /* this defines LF to be 10 */
main ()
{
    char c;
    printf ("Input any characters, hit X to stop.\n");
    do {
        c = getch (); /* get a character */
        putchar (c); /* display the hit key */
        if (c == CR) putchar (LF); /* if it is a carriage return put out a linefeed too */
    } while (c != 'X');
    printf ("\nEnd of program.\n");
}
```

9. Standard Input/Output (11/19)

- leitura de inteiros -

■ intin.c

```
#include "stdio.h"

main ( )
{
    int valin;

    printf ("Input a number from 0 to 32767, stop with 100.\n");

    do {
        scanf ("%d", &valin); /* read a single integer value in */
        printf ("The value is %d\n", valin);
    } while (valin != 100);

    printf ("End of program\n");
}
```

© RMAC XII-2001

137

9. Standard Input/Output (12/19)

- função *scanf* () -

scanf

Function Scans and formats input from the stdin stream.

Syntax `#include <stdio.h>`
`int scanf(const char *format[, address, ...]);`

DOS	UNIX	Windows	ANSI C	C++ only
■	■		■	

Remarks **scanf** scans a series of input fields, one character at a time, reading from the stdin stream. Then each field is formatted according to a format specifier passed to **scanf** in the format string pointed to by *format*. Finally, **scanf** stores the formatted input at an address passed to it as an argument following *format*. There must be the same number of format specifiers and addresses as there are input fields.

Return value **scanf** returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored.

If **scanf** attempts to read at end-of-file, the return value is EOF.

If no fields were stored, the return value is 0.

See also `atof`, `cscanf`, `fscanf`, `getc`, `printf`, `sscanf`, `vfprintf`, `vscanf`, `vsscanf`

© RMAC XII-2001

138

9. Standard Input/Output (13/19)

- leitura de strings -

■ stringin.c

```
#include "stdio.h"

main ( )
{
    char big [25];

    printf ("Input a character string, up to 24 characters.\n");
    printf ("An X in column 1 causes the program to stop.\n");

    do {
        scanf ("%s", big);
        printf ("The string is -> %s\n", big);
    } while (big [0] != 'X');

    printf ("End of program.\n");
}
```

© RMAC XII-2001

139

9. Standard Input/Output (14/19)

■ inmen.c

```
main ( )
{
    int numbers [5], result [5], index;
    char line [80];

    numbers [0] = 74;
    numbers [1] = 18;
    numbers [2] = 33;
    numbers [3] = 30;
    numbers [4] = 97;

    sprintf (line, "%d %d    %d %d %d\n", numbers [0], numbers [1],
            numbers [2], numbers [3], numbers [4]);
    (...)
}
```

© RMAC XII-2001

140

9. Standard Input/Output (15/19)

■ inmen.c (cont.)

```
(...)  
  
printf ("%s", line);  
  
sscanf (line, "%d %d %d %d %d", &result [4], &result [3],  
        (result+2), (result+1), result);  
  
for (index = 0; index < 5; index++)  
    printf ("The final result is %d\n", result [index]);  
}
```

9. Standard Input/Output (16/19)

- função *sprintf* () -

sprintf

Function Writes formatted output to a string.

Syntax #include <stdio.h>
int sprintf(char *buffer, const char *format[, argument, ...]);

DOS	UNIX	Windows	ANSI C	C++ only
■	■	■	■	

Remarks **sprintf** accepts a series of arguments, applies to each a format specifier contained in the format string pointed to by *format*, and outputs the formatted data to a string.

See **printf** for details on format specifiers.

sprintf applies the first format specifier to the first argument, the second to the second, and so on. There must be the same number of format specifiers as arguments.

Return value **sprintf** returns the number of bytes output. **sprintf** does not include the terminating null byte in the count. In the event of error, **sprintf** returns EOF.

See also fprintf, printf

9. Standard Input/Output (17/19)

- função *sscanf* () -

sscanf

Function Scans and formats input from a string.

Syntax `#include <stdio.h>`
`int sscanf(const char *buffer, const char *format[, address, ...]);`

DOS	UNIX	Windows	ANSI C	C++ only
■	■	■	■	

Remarks *sscanf* scans a series of input fields, one character at a time, reading from a string. Then each field is formatted according to a format specifier passed to *sscanf* in the format string pointed to by *format*. Finally, *sscanf* stores the formatted input at an address passed to it as an argument following *format*. There must be the same number of format specifiers and addresses as there are input fields.

See *scanf* for details on format specifiers.

sscanf might stop scanning a particular field before it reaches the normal end-of-field (whitespace) character, or it might terminate entirely, for a number of reasons. See *scanf* for a discussion of possible causes.

Return value *sscanf* returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored. If no fields were stored, the return value is 0.

If *sscanf* attempts to read at end-of-string, the return value is EOF.

See also *fscanf*, *scanf*

143

© RMAC XII-2001

9. Standard Input/Output (18/19)

■ *special.c*

```
#include "stdio.h"
#include "stdlib.h"

main ( )
{
    int index;

    for (index = 0; index < 6; index++) {
        printf ("This line goes to the standard output.\n");
        fprintf (stderr, "This line goes to the error device.\n");
    }

    exit (4); /* This can be tested with the DOS errorlevel or UNIX-like #? */
}
```

144

© RMAC XII-2001

9. Standard Input/Output (19/19)

- função *exit* () -

exit

Function Terminates program.

Syntax `#include <stdlib.h>`
`void exit(int status);`

DOS	UNIX	Windows	ANSI C	C++ only
■	■	■	■	

Remarks **exit** terminates the calling process. Before termination, all files are closed, buffered output (waiting to be output) is written, and any registered "exit functions" (posted with **atexit**) are called.

status is provided for the calling process as the exit status of the process. Typically a value of 0 is used to indicate a normal exit, and a nonzero value indicates some error. It is set with one of the following

EXIT_SUCCESS Normal program termination.

EXIT_FAILURE Abnormal program termination; signal to operating system that program has terminated with an error.

Return value None.

See also **abort, atexit, exec..., _exit, keep, signal, spawn...**

© RMAC XII-2001

145

10. Ficheiros (1/16)

■ formout.c

```
#include "stdio.h"

main ( )
{
    FILE *fp;
    char stuff [25];
    int index;

    fp = fopen ("TENLINES.TXT", "w"); /* open for writing */
    strcpy (stuff, "This is an example line.");

    for (index = 1; index <= 10; index++)
        fprintf (fp, "%s Line number %d\n", stuff, index);

    fclose (fp); /* close the file before ending program */
}
```

© RMAC XII-2001

146

10. Ficheiros (2/16)

- função *fopen* () -

fopen

Function Opens a stream.

Syntax `#include <stdio.h>`
`FILE *fopen(const char *filename, const char *mode);`

DOS	UNIX	Windows	ANSI C	C++ only
■	■	■	■	

Remarks *fopen* opens the file named by *filename* and associates a stream with it. *fopen* returns a pointer to be used to identify the stream in subsequent operations.

Return value On successful completion, *fopen* returns a pointer to the newly opened stream. In the event of error, it returns null.

See also *creat*, *dup*, *fclose*, *fdopen* *error*, *_fmode* (global variable), *fread*, *freopen*, *fseek*, *fwrite*, *open*, *rewind*, *setbuf*, *setmode*

10. Ficheiros (3/16)

- modos de abertura de ficheiros -

Mode	Meaning
"r"	Open file for reading
"w"	Create a file for writing
"a"	Append to file
"rb"	Open binary file for reading
"wb"	Create binary file for writing
"ab"	Append to a binary file
"r+"	Open file for read/write
"w+"	Create file for read/write
"a+"	Open file for read/write
"rb+"	Open binary file for read/write
"wb+"	Create binary file for read/write
"ab+"	Open binary file for read/write
"rt"	Open a text file for reading
"wt"	Create a text file for writing
"at"	Append to a text file
"r+t"	Open a text file for read/write
"w+t"	Create a text file for read/write
"a+t"	Open or create a text file for read/write

10. Ficheiros (4/16)

- função *fprintf* () -

fprintf

Function Writes formatted output to a stream.

Syntax `#include <stdio.h>`
`int fprintf(FILE *stream, const char *format[, argument, ...]);`

DOS	UNIX	Windows	ANSI C	C++ only
■	■	■	■	

Remarks *fprintf* accepts a series of arguments, applies to each a format specifier contained in the format string pointed to by *format*, and outputs the formatted data to a stream. There must be the same number of format specifiers as arguments.
See *printf* for details on format specifiers.

Return value *fprintf* returns the number of bytes output. In the event of error, it returns EOF.

See also *cprintf*, *fscanf*, *printf*, *putc*, *sprintf*

10. Ficheiros (5/16)

- função *fclose* () -

fclose

Function Closes a stream.

Syntax `#include <stdio.h>`
`int fclose(FILE *stream);`

DOS	UNIX	Windows	ANSI C	C++ only
■	■	■	■	

Remarks *fclose* closes the named stream. All buffers associated with the stream are flushed before closing. System-allocated buffers are freed upon closing. Buffers assigned with *setbuf* or *setvbuf* are not automatically freed. (But if *setvbuf* is passed null for the buffer pointer, it *will* free it upon close.)

Return value *fclose* returns 0 on success. It returns EOF if any errors were detected.

See also *close*, *fcloseall*, *fdopen*, *fflush*, *flushall*, *fopen*, *freopen*

10. Ficheiros (6/16)

- escrita de caracteres -

■ charout.c

```
#include "stdio.h"
main ()
{
    FILE *point;
    char others [35];
    int indexer, count;
    strcpy (others, "Additional lines.");
    point = fopen ("tenlines.txt", "a"); /* open for appending */
    for (count = 1; count <= 10; count++) {
        for (indexer = 0; others [indexer]; indexer++)
            putc (others [indexer], point); /* output a single character */
        putc ('\n', point); /* output a linefeed */
    }
    fclose (point);
}
```

© RMAC XII-2001

151

10. Ficheiros (7/16)

- função *putc* () -

putc

Function Outputs a character to a stream.

Syntax #include <stdio.h>
int putc(int *c*, FILE **stream*);

DOS	UNIX	Windows	ANSI C	C++ only
■	■	■	■	

Remarks **putc** is a macro that outputs the character *c* to the stream given by *stream*.

Return Value On success, **putc** returns the character printed, *c*. On error, **putc** returns EOF.

See also fprintf, fputc, fputch, fputcchar, fputs, fwrite, getc, getchar, printf, putch, putchar, putw, vprintf

© RMAC XII-2001

152

10. Ficheiros (8/16)

- leitura de caracteres -

■ readchar.c

```
#include "stdio.h"
main ()
{
    FILE *funny;
    char c;
    funny = fopen ("TENLINES.TXT", "r");
    if (funny == NULL) printf ("File doesn't exist\n");
    else {
        do {
            c = getc (funny); /* get one character from the file */
            putchar (c); /* display it on the monitor */
        } while (c != EOF); /* repeat until EOF (end of file) */
    }
    fclose (funny);
}
```

© RMAC XII-2001

153

10. Ficheiros (9/16)

- função *getc* () -

getc

Function Gets character from stream.

Syntax #include <stdio.h>
int getc(FILE *stream);

DOS	UNIX	Windows	ANSI C	C++ only
■	■	■	■	

Remarks **getc** is a macro that returns the next character on the given input stream and increments the stream's file pointer to point to the next character.

Return value On success, **getc** returns the character read, after converting it to an **int** without sign extension. On end-of-file or error, it returns EOF.

See also **fgetc**, **getch**, **getchar**, **getcche**, **gets**, **putc**, **putchar**, **ungetc**

© RMAC XII-2001

154

10. Ficheiros (10/16)

- leitura de palavras -

■ readtext.c

```
#include "stdio.h"

main ( )
{
    FILE *fp1;
    char oneword [100];
    char c;
    fp1 = fopen ("TENLINES.TXT", "r");

    do {
        c = fscanf (fp1, "%s", oneword); /* got one word from the file */
        printf ("%s\n", oneword); /* display it on the monitor */
    } while (c != EOF); /* repeat until EOF */

    fclose (fp1);
}
```

© RMAC XII-2001

155

10. Ficheiros (11/16)

- função *fscanf* () -

fscanf

Function Scans and formats input from a stream.

Syntax `#include <stdio.h>`
`int fscanf(FILE *stream, const char *format[, address, ...]);`

DOS	UNIX	Windows	ANSI C	C++ only
*	*	*	*	

Remarks **fscanf** scans a series of input fields, one character at a time, reading from a stream. Then each field is formatted according to a format specifier passed to **fscanf** in the format string pointed to by *format*. Finally, **fscanf** stores the formatted input at an address passed to it as an argument following *format*. The number of format specifiers and addresses must be the same as the number of input fields.

See **scanf** for details on format specifiers.

fscanf can stop scanning a particular field before it reaches the normal end-of-field character (whitespace), or it can terminate entirely for a number of reasons. See **scanf** for a discussion of possible causes.

Return value **fscanf** returns the number of input fields successfully scanned, converted, and stored; the return value does not include scanned fields that were not stored.

If **fscanf** attempts to read at end-of-file, the return value is EOF. If no fields were stored, the return value is 0.

See also **atof**, **cscanf**, **fprintf**, **printf**, **scanf**, **sscanf**, **vfscanf**, **vscanf**, **vsscanf**

© RMAC XII-2001

156

10. Ficheiros (12/16)

- caracter especial EOF -

■ readgood.c

```
#include "stdio.h"

main ( )
{
    FILE *fp1;
    char oneword [100];
    char c;
    fp1 = fopen ("TENLINES.TXT", "r");
    do {
        c = fscanf (fp1, "%s", oneword); /* got one word from the file */
        if (c != EOF) printf ("%s\n", oneword); /* display it on the monitor */
    } while (c != EOF); /* repeat until EOF */
    fclose (fp1);
}
```

© RMAC XII-2001

157

10. Ficheiros (13/16)

- leitura de uma linha -

■ readline.c

```
#include "stdio.h"

main ( )
{
    FILE *fp1;
    char oneword [100];
    char *c;
    fp1 = fopen ("TENLINES.TXT", "r");
    do {
        c = fgets (oneword, 100, fp1); /* get one line from the file */
        if (c != NULL) printf ("%s", oneword); /* display it on the monitor */
    } while (c != NULL); /* repeat until NULL */
    fclose (fp1);
}
```

© RMAC XII-2001

158

10. Ficheiros (14/16)

- função *fgets* () -

fgets

Function Gets a string from a stream.

Syntax `#include <stdio.h>`
`char *fgets(char *s, int n, FILE *stream);`

DOS	UNIX	Windows	ANSI C	C++ only
■	■	■	■	

Remarks *fgets* reads characters from *stream* into the string *s*. The function stops reading when it reads either *n* - 1 characters or a newline character, whichever comes first. *fgets* retains the newline character at the end of *s*. A null byte is appended to *s* to mark the end of the string.

Return value On success, *fgets* returns the string pointed to by *s*; it returns null on end-of-file or error.

See also *cgets*, *fputs*, *gets*

10. Ficheiros (15/16)

- caracter especial NULL -

■ anyfile.c

```
#include "stdio.h"
main ( )
{
    FILE *fp1;
    char oneword [100], filename [25];
    char *c;
    printf ("Enter filename -> ");
    scanf ("%s", filename); /* read the desired filename */
    fp1 = fopen (filename, "r");
    do {
        c = fgets (oneword, 100, fp1); /* get one line from the file */
        if (c != NULL) printf ("%s", oneword); /* display it on the monitor */
    } while (c != NULL); /* repeat until NULL */
    fclose (fp1);
}
```


10. Ficheiros (16/16)

■ printdat.c

```
#include "stdio.h"
main ( )
{
    FILE *funny, *printer;
    char c;
    funny = fopen ("TENLINES.TXT", "r"); /* open input file */
    printer = fopen ("PRN", "w"); /* open printer file */
    do { c = getc (funny); /* get one character from the file */
        if (c != EOF) { putchar (c); /* display it on the monitor */
                        putc (c, printer); /* print the character */
        }
    } while (c != EOF); /* repeat until EOF (end of file) */
    fclose (funny);
    fclose (printer);
}
```

© RMAC XII-2001

161



exercícios

1. Escreva um programa que leia caracteres em ciclo e que imprima no monitor o caracter lido, bem como o seu código decimal. O caracter "\$" deve terminar o ciclo de leitura. Experimente o programa com todas as teclas do teclado do seu computador.
2. Escreva um programa que solicite ao utilizador o nome de um ficheiro para leitura e o nome de um outro ficheiro para escrita. O programa deve, em ciclo, ler os caracteres presentes no primeiro ficheiro e escrevê-los no segundo ficheiro e enviá-los para o monitor e para a impressora. O programa deve terminar quando for lido um EOF.
3. Escreva um programa que solicite ao utilizador o nome de um ficheiro para leitura. O programa deve ler, do ficheiro, uma linha de cada vez e enviá-la para o monitor com o número de linha.

© RMAC XII-2001

162

11. Bibliotecas de Funções (1/14)

- classificação -

Classification

routines

These routines classify ASCII characters as letters, control characters, punctuation, uppercase, etc.

isalnum	(ctype.h)	isdigit	(ctype.h)	ispunct	(ctype.h)
isalpha	(ctype.h)	isgraph	(ctype.h)	isspace	(ctype.h)
isascii	(ctype.h)	islower	(ctype.h)	isupper	(ctype.h)
iscntrl	(ctype.h)	isprint	(ctype.h)	isxdigit	(ctype.h)

11. Bibliotecas de Funções (2/14)

- conversão -

Conversion

routines

These routines convert characters and strings from alpha to different numeric representations (floating-point, integers, longs) and vice versa, and from uppercase to lowercase and vice versa.

atof	(stdlib.h)	ltoa	(stdlib.h)	toascii	(ctype.h)
atoi	(stdlib.h)	_strdate	(time.h)	_tolower	(ctype.h)
atol	(stdlib.h)	_strtime	(time.h)	tolower	(ctype.h)
ecvt	(stdlib.h)	strtod	(stdlib.h)	_toupper	(ctype.h)
fcvt	(stdlib.h)	strtol	(stdlib.h)	toupper	(ctype.h)
gcvt	(stdlib.h)	_strtold	(stdlib.h)	ultoa	(stdlib.h)
ltoa	(stdlib.h)	strtoul	(stdlib.h)		

11. Bibliotecas de Funções (3/14)

- controlo de directorias -

Directory control routines

These routines manipulate directories and path names.

chdir	(dir.h)	_fullpath	(stdlib.h)
_chdrive	(direct.h)	getcurdir	(dir.h)
closedir	(dirent.h)	getcwd	(dir.h)
_dos_findfirst	(dos.h)	_getcwd	(direct.h)
_dos_findnext	(dos.h)	getdisk	(dir.h)
_dos_getdiskfree	(dos.h)	_getdrive	(direct.h)
_dos_getdrive	(dos.h)	_makepath	(stdlib.h)
_dos_setdrive	(dos.h)	mkdir	(dir.h)
findfirst	(dir.h)	mktemp	(dir.h)
findnext	(dir.h)	opendir	(dirent.h)
fnmerge	(dir.h)	readdir	(dirent.h)
fnsplit	(dir.h)	rewinddir	(dirent.h)
rmdir	(dir.h)	setdisk	(dir.h)
_searchenv	(stdlib.h)	_splitpath	(stdlib.h)
searchpath	(dir.h)		

11. Bibliotecas de Funções (4/14)

- diagnóstico -

Diagnostic routines

These routines provide built-in troubleshooting capability.

assert	(assert.h)
matherr	(math.h)
_matherrl	(math.h)
perror	(errno.h)

11. Bibliotecas de Funções (5/14)

- entradas/saídas -

Input/output routines

These routines provide stream-level and DOS-level I/O capability.

access	(io.h)	_dos_creat	(dos.h)
cgets	(conio.h)	_dos_creatnew	(dos.h)
_chmod	(io.h)	_dos_getfileattr	(dos.h)
chmod	(io.h)	_dos_gettime	(dos.h)
chsize	(io.h)	_dos_open	(dos.h)
clearerr	(stdio.h)	_dos_read	(dos.h)
_close	(io.h)	_dos_setfileattr	(dos.h)
close	(io.h)	_dos_settime	(dos.h)
cprintf	(conio.h)	_dos_write	(dos.h)
cputs	(conio.h)	dup	(io.h)
_creat	(io.h)	dup2	(io.h)
creat	(io.h)	eof	(io.h)
creatnew	(io.h)	fclose	(stdio.h)
creattemp	(io.h)	fcloseall	(stdio.h)
cscanf	(conio.h)	fdopen	(stdio.h)
_dos_close	(dos.h)	feof	(stdio.h)

© RMAC XII-2001

167

11. Bibliotecas de Funções (6/14)

- entradas/saídas -

ferror	(stdio.h)	printf	(stdio.h)
fflush	(stdio.h)	putc	(stdio.h)
fgetc	(stdio.h)	putch	(conio.h)
fgetchar	(stdio.h)	putchar	(stdio.h)
fgetpos	(stdio.h)	puts	(stdio.h)
fgets	(stdio.h)	putw	(stdio.h)
filelength	(io.h)	_read	(io.h)
fileno	(stdio.h)	read	(io.h)
flushall	(stdio.h)	remove	(stdio.h)
fopen	(stdio.h)	rename	(stdio.h)
fprintf	(stdio.h)	rewind	(stdio.h)
fputc	(stdio.h)	rmtmp	(stdio.h)
fputchar	(stdio.h)	scanf	(stdio.h)
fputs	(stdio.h)	setbuf	(stdio.h)
fread	(stdio.h)	setcursortype	(conio.h)
freopen	(stdio.h)	settime	(io.h)
fscanf	(stdio.h)	setmode	(io.h)
fseek	(stdio.h)	setvbuf	(stdio.h)
fsetpos	(stdio.h)	sopen	(io.h)
_fsopen	(stdio.h)	sprintf	(stdio.h)
fstat	(sys\stat.h)	sscanf	(stdio.h)

© RMAC XII-2001

168

11. Bibliotecas de Funções (7/14)

- entradas/saídas -

ftell	(stdio.h)	stat	(sys\stat.h)
fwrite	(stdio.h)	_strerror	(string.h, stdio.h)
getc	(stdio.h)	strerror	(stdio.h)
getch	(conio.h)	tell	(io.h)
getchar	(stdio.h)	tempnam	(stdio.h)
getche	(conio.h)	tmpfile	(stdio.h)
getftime	(io.h)	tmpnam	(stdio.h)
getpass	(conio.h)	umask	(io.h)
gets	(stdio.h)	ungetc	(stdio.h)
getw	(stdio.h)	ungetch	(conio.h)
ioctl	(io.h)	unlock	(io.h)
isatty	(io.h)	utime	(utime.h)
kbhit	(conio.h)	vfprintf	(stdio.h)
lock	(io.h)	vfscanf	(stdio.h)
locking	(io.h)	vprintf	(stdio.h)
lseek	(io.h)	vscanf	(stdio.h)
_open	(io.h)	vsprintf	(stdio.h)
open	(io.h)	vsscanf	(io.h)
perror	(stdio.h)	_write	(io.h)

© RMAC XII-2001

169

11. Bibliotecas de Funções (8/14)

- manipulação -

Manipulation routines

These routines handle strings and blocks of memory: copying, comparing, converting, and searching.

mblen	(stdlib.h)	memset	(mem.h, string.h)
mbstowcs	(stdlib.h)	movedata	(mem.h, string.h)
mbtowc	(stdlib.h)	movmem	(mem.h, string.h)
memccpy	(mem.h, string.h)	setmem	(mem.h)
memchr	(mem.h, string.h)	stpcpy	(string.h)
memcmp	(mem.h, string.h)	strcat	(string.h)
memcpy	(mem.h, string.h)	strchr	(string.h)
memcmp	(mem.h, string.h)	strcmp	(string.h)
memmove	(mem.h, string.h)	strcoll	(string.h)
strcpy	(string.h)	strnset	(string.h)
strcspn	(string.h)	strpbrk	(string.h)
strdup	(string.h)	strrchr	(string.h)
strerror	(string.h)	strrev	(string.h)
stricmp	(string.h)	strset	(string.h)
strcmpi	(string.h)	strspn	(string.h)
strlen	(string.h)	strstr	(string.h)
strlwr	(string.h)	strtok	(string.h)
strncat	(string.h)	strupr	(string.h)
strncmp	(string.h)	strxfrm	(string.h)
strncmpi	(string.h)	wcstombs	(stdlib.h)
strncpy	(string.h)	wctomb	(stdlib.h)
strnicmp	(string.h)		

© RMAC XII-2001

170

11. Bibliotecas de Funções (9/14)

- matemática -

Math routines

These routines perform mathematical calculations and conversions.

abs	(complex.h, stdlib.h)	expl	(math.h)
acos	(complex.h, math.h)	fabs	(math.h)
acosl	(math.h)	fabsl	(math.h)
arg	(complex.h)	fcvt	(stdlib.h)
asin	(complex.h, math.h)	floor	(math.h)
asinl	(math.h)	floorl	(math.h)
atan	(complex.h, math.h)	fmod	(math.h)
atanl	(math.h)	fmodl	(math.h)
atan2	(complex.h, math.h)	_fpreset	(float.h)
atan2l	(math.h)	frexp	(math.h)
atof	(stdlib.h, math.h)	frexpl	(math.h)
atoi	(stdlib.h)	gcvt	(stdlib.h)
atol	(stdlib.h)	hypot	(math.h)
_atold	(math.h)	hypotl	(math.h)
bcd	(bcd.h)	imag	(complex.h)
cabs	(math.h)	ltoa	(stdlib.h)
cabsl	(math.h)	labs	(stdlib.h)
ceil	(math.h)	ldexp	(math.h)
ceilf	(math.h)	ldexpl	(math.h)

© RMAC XII-2001

171

11. Bibliotecas de Funções (10/14)

- matemática -

_clear87	(float.h)	ldiv	(math.h)
complex	(complex.h)	log	(complex.h, math.h)
conj	(complex.h)	logl	(math.h)
_control87	(float.h)	log10	(complex.h, math.h)
cos	(complex.h, math.h)	log10l	(math.h)
cosl	(math.h)	_rotl	(stdlib.h)
cosh	(complex.h, math.h)	_rotr	(stdlib.h)
coshl	(math.h)	ltoa	(stdlib.h)
div	(math.h)	matherr	(math.h)
ecvt	(stdlib.h)	_matherrl	(math.h)
exp	(complex.h, math.h)	modf	(math.h)
modfl	(math.h)	sinl	(math.h)
norm	(complex.h)	sinh	(complex.h, math.h)
polar	(complex.h)	sinhl	(math.h)
poly	(math.h)	sqrt	(complex.h, math.h)
polyl	(math.h)	sqrtil	(math.h)
pow	(complex.h, math.h)	srand	(stdlib.h)
powl	(math.h)	_status87	(float.h)
pow10	(math.h)	strtod	(stdlib.h)
pow10l	(math.h)	strtol	(stdlib.h)
rand	(stdlib.h)	_strtol	(stdlib.h)
random	(stdlib.h)	strtoul	(stdlib.h)
randomize	(stdlib.h)	tan	(complex.h, math.h)
real	(complex.h)	tanl	(math.h)
_rotl	(stdlib.h)	tanh	(complex.h, math.h)
_rotr	(stdlib.h)	tanhl	(complex.h, math.h)
sin	(complex.h, math.h)	ultoa	(stdlib.h)

© RMAC XII-2001

172

11. Bibliotecas de Funções (11/14)

- memória -

Memory routines

These routines provide dynamic memory allocation in the small-data and large-data models.

alloca	(malloc.h)	farheapfillfree	(alloc.h)
allocmem	(dos.h)	farheapwalk	(alloc.h)
_bios_memsiz	(bios.h)	farmalloc	(alloc.h)
brk	(alloc.h)	farrealloc	(alloc.h)
calloc	(alloc.h, stdlib.h)	free	(alloc.h, stdlib.h)
coreleft	(alloc.h, stdlib.h)	heapcheck	(alloc.h)
_dos_allocmem	(dos.h)	heapcheckfree	(alloc.h)
_dos_freemem	(dos.h)	heapchecknode	(alloc.h)
_dos_setblock	(dos.h)	heapwalk	(alloc.h)
farcalloc	(alloc.h)	malloc	(alloc.h, stdlib.h)
farcoreleft	(alloc.h)	realloc	(alloc.h, stdlib.h)
farfree	(alloc.h)	sbrk	(alloc.h)
farheapcheck	(alloc.h)	setblock	(dos.h)
farheapcheckfree	(alloc.h)	set_new_handler	(new.h)
farheapchecknode	(alloc.h)		

11. Bibliotecas de Funções (12/14)

- vários -

Miscellaneous routines

These routines provide nonlocal goto capabilities, sound effects, and locale.

delay	(dos.h)	setjmp	(setjmp.h)
localeconv	(locale.h)	setlocale	(locale.h)
longjmp	(setjmp.h)	sound	(dos.h)
nosound	(dos.h)		

11. Bibliotecas de Funções (13/14)

- processos -

Process control routines

These routines invoke and terminate new processes from within another.

abort	(process.h)	execve	(process.h)	spawnl	(process.h)
_c_exit	(process.h)	execvp	(process.h)	spawnle	(process.h)
_c_exit	(process.h)	execvpe	(process.h)	spawnlp	(process.h)
execl	(process.h)	_exit	(process.h)	spawnlpe	(process.h)
execle	(process.h)	exit	(process.h)	spawnv	(process.h)
execlep	(process.h)	getpid	(process.h)	spawnve	(process.h)
execlep	(process.h)	raise	(signal.h)	spawnvp	(process.h)
execv	(process.h)	signal	(signal.h)	spawnvpe	(process.h)

11. Bibliotecas de Funções (14/14)

- tempo -

Time and date routines

These are time conversion and time manipulation routines.

asctime	(time.h)	gettime	(dos.h)
_bios_timeofday	(bios.h)	gmtime	(time.h)
ctime	(time.h)	localtime	(time.h)
difftime	(time.h)	mktime	(time.h)
_dos_getdate	(dos.h)	setdate	(dos.h)
_dos_gettime	(dos.h)	settime	(dos.h)
_dos_setdate	(dos.h)	stime	(time.h)
_dos_settime	(dos.h)	strftime	(time.h)
dostounix	(dos.h)	time	(time.h)
ftime	(sys\timeb.h)	tzset	(time.h)
getdate	(dos.h)	unixtodos	(dos.h)

12. Erros Típicos (1/10)

■ Iniciação de uma variável

– incorrecto

```
int num;  
  
get_num (num);
```

– problemas

- *num* ainda não foi iniciada no momento em que é passada para *get_num ()*
- *num* continua não iniciada após a execução de *get_num ()*

– correcto

```
int num;  
  
num = get_num ( );
```

© RMAC XII-2001

177

12. Erros Típicos (2/10)

■ Captura do valor de retorno

– incorrecto

```
int get_num (void); /* prototype of getnum ( ) */  
  
void main (void)  
{  
    int num;  
  
    get_num ( );  
}
```

– problema

- o inteiro retornado por *get_num ()* nunca chega a ser capturado

© RMAC XII-2001

178

12. Erros Típicos (3/10)

■ Captura do valor de retorno (cont.)

– correcto

```
int get_num (void); /* prototype of getnum ( ) */

void main (void)
{
    int num;

    num = get_num ( );
}
```

© RMAC XII-2001

179

12. Erros Típicos (4/10)

■ Local de declaração de variáveis

– incorrecto

```
void main (void)
{ int num, value;
  num = multiply (num);
}

int multiply (int param)
{
    value = get_value ( );
    return (3*value*param);
}
```

© RMAC XII-2001

– problema

- variável *value* declarada no local errado e variável *num* não iniciada

180

12. Erros Típicos (5/10)

■ Local de declaração de variáveis

– correcto

```
void main (void)
{ int num;
  num = 12;
  num = multiply (num);
}

int multiply (int param)
{ int value;
  value = get_value ( );
  return (3*value*param);
}
```

© RMAC XII-2001

181

12. Erros Típicos (6/10)

■ Respeitar o protótipo

– incorrecto

```
int add (void); /* prototype of add ( ) */

int add (int n1, int n2) /* definition of add ( ) */
{
  return (n1+n2);
}
```

– problema

- o protótipo e a definição (implementação) de `add ()` não são coerentes

© RMAC XII-2001

182

12. Erros Típicos (7/10)

■ Respeitar o protótipo (cont.)

– correcto

```
int add (int n1, int n2); /* prototype of add ( ) */  
  
int add (int n1, int n2) /* definition of add ( ) */  
{  
    return (n1+n2);  
}
```

12. Erros Típicos (8/10)

■ Adequação dos nomes das variáveis

- os identificadores (nomes) das variáveis são importantes por duas razões
 - para distinguir as variáveis com *scopes* que se interceptam
 - para aumentar a legibilidade dos programas
- desta forma, é possível duas funções distintas possuírem duas variáveis (uma em cada função) locais (*scopes* não interceptados) com nomes iguais
- alterações numa das variáveis dentro de uma das funções não provoca (como *just a kind of magic*) alterações de conteúdo na variável com o mesmo nome na outra função
- um erro típico é julgar-se que uma função pode ter acesso (pode “ver”) a variáveis de outra função
- uma função só pode “ver” e alterar variáveis da sua lista de parâmetros ou variáveis declaradas localmente (atenção à declaração de variáveis globais)

12. Erros Típicos (9/10)

■ Respeitar a ordem dos parâmetros

– incorrecto

```
double calc (double amount, double interest); /* prototype */

void main (void)
{
    double total, amount, interest;

    amount = get_amount ();
    interest = get_interest ();
    total = calc (interest, amount);
}
```

– problema

- a ordem de passagem dos parâmetros para *calc ()* não está correcta

© RMAC XII-2001

185

12. Erros Típicos (10/10)

■ Respeitar a ordem dos parâmetros (cont.)

– correcto

```
double calc (double amount, double interest); /* prototype */

void main (void)
{
    double total, amount, interest;

    amount = get_amount ();
    interest = get_interest ();
    total = calc (amount, interest);
}
```

– observação

- as variáveis (ou apontadores) são passadas a uma função, tendo, somente, em conta a sua ordem na lista de parâmetros de entrada e nunca os seus nomes

© RMAC XII-2001

186