

Programação procedimental

Linguagem C

João Barroso (jbarroso@utad.pt)

Linguagem de Programação C

utad

Programação procedimental

Linguagens de programação:

- linguagens funcionais,

- Linguagens estruturadas.

Noções de programação estruturada:

- sequência, selecção, iteração.

Noções básicas de programação:

- constantes, variáveis,

- tipos de dados,

- expressões, operadores,

- funções pré-definidas.

Estruturas de decisão:

- if, if-else, switch.

Estruturas de repetição:

- for, while, do-while.

Conceitos de endereçamento de memória:

- apontadores,

- alocação dinâmica de memória,

- exemplos de aplicação.

Subprogramas:

- funções,

- correspondência argumento – parâmetro,

- passagem de parâmetros (por cópia de valor e por referência).

Vectores e matrizes - arrays de uma e de duas dimensões.

Caracteres e cadeias de caracteres - strings.

Linguagem de Programação C

utad

Programação procedimental

Algoritmos de pesquisa e de ordenação:

pesquisa linear e pesquisa binária,
ordenação pelos métodos de selection sort e
bubble sort,
exemplos de aplicação.

Ficheiros de texto:

leitura de ficheiros,
escrita em ficheiros.

Estruturas de dados:

vectores de estruturas.

Listas ligadas:

listas simplesmente ligadas,
listas duplamente ligadas,
exemplos de aplicação.

Programação orientada a objectos (linguagem de
programação C++)

Linguagem de Programação C

utad

Programação procedimental

As **linguagens de programação funcionais** são constituídas por funções, as quais são tratadas de uma forma muito flexível, podem funcionar como parâmetros de entrada para outras funções ou podem ser valores de saída de outras funções.

Mapeamento dos valores de entrada nos valores de saída através de funções.

A execução de uma função não produz efeitos laterais, apenas é calculado o resultado e devolvido.

Esta abordagem não implica a existência de uma ordem de execução desta forma é eliminado um grande conjunto de possibilidades de erro.

As linguagens de programação funcional são interessantes essencialmente pela sua simplicidade sintáctica e facilidade de resolver problemas de forma recursiva (não têm o conceito de “ciclo”).

Linguagem de Programação C

utad

Programação procedimental

Programação estruturada é uma forma de programação de computadores podendo os programas ser reduzidos a três estruturas essenciais:

**sequência,
decisão
iteração.**

Programação estruturada foi, na prática, transformada na Programação modular, orienta os programadores para a criação de estruturas simples nos seus programas, usando as subrotinas e as funções.

Este paradigma teve origem com as linguagens de programação PASCAL e C, que são linguagens de alto nível.

É usando este paradigma que serão leccionadas todas as matérias.

Linguagem de Programação C

utad **Identação**

Programação procedimental

Cabeçalho

```
/*
Ficheiro:
Título:
Autor:
Obs:
Data:
Revisões:
.....
*/
```

Corpo do Programa

```
/*
Nome_função
Assinatura: nome_função: parâmetros → resultados
Comentários
*/
```

ola.c

- O nome do ficheiro deve ser expressivo

Código fonte: alguns estilos....

utad

Programação procedimental

```
/*o programa mais pequeno*/  
main() {  
    return;  
}
```

```
/*o programa mais pequeno*/  
main()  
{  
    return;  
}
```

```
/*o programa mais pequeno*/  
main()  
    {  
        return;  
    }
```

Programas....

utad

Programação procedimental

```
1: main() {  
2:     return;  
3: }
```

```
1: #include <stdio.h>  
2: main() {  
3:     printf("Olá mundo\n"); // escreve no ecran "Olá mundo"  
4: return;  
5: }
```


Identificadores

32 palavras reservadas

utad

Programação procedimental

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

■ sempre em minúsculas: a linguagem C é “case sensitive”

Estrutura geral de um Programa

utad

Programação procedimental

```
Global declarations
return-type main(parameter list)
{
    local variables
    C statement sequence
}
return-type f1(parameter list)
{
    local variables
    C statement sequence
}
.....
return-type fN(parameter list)
{
    local variables
    C statement sequence
}
```

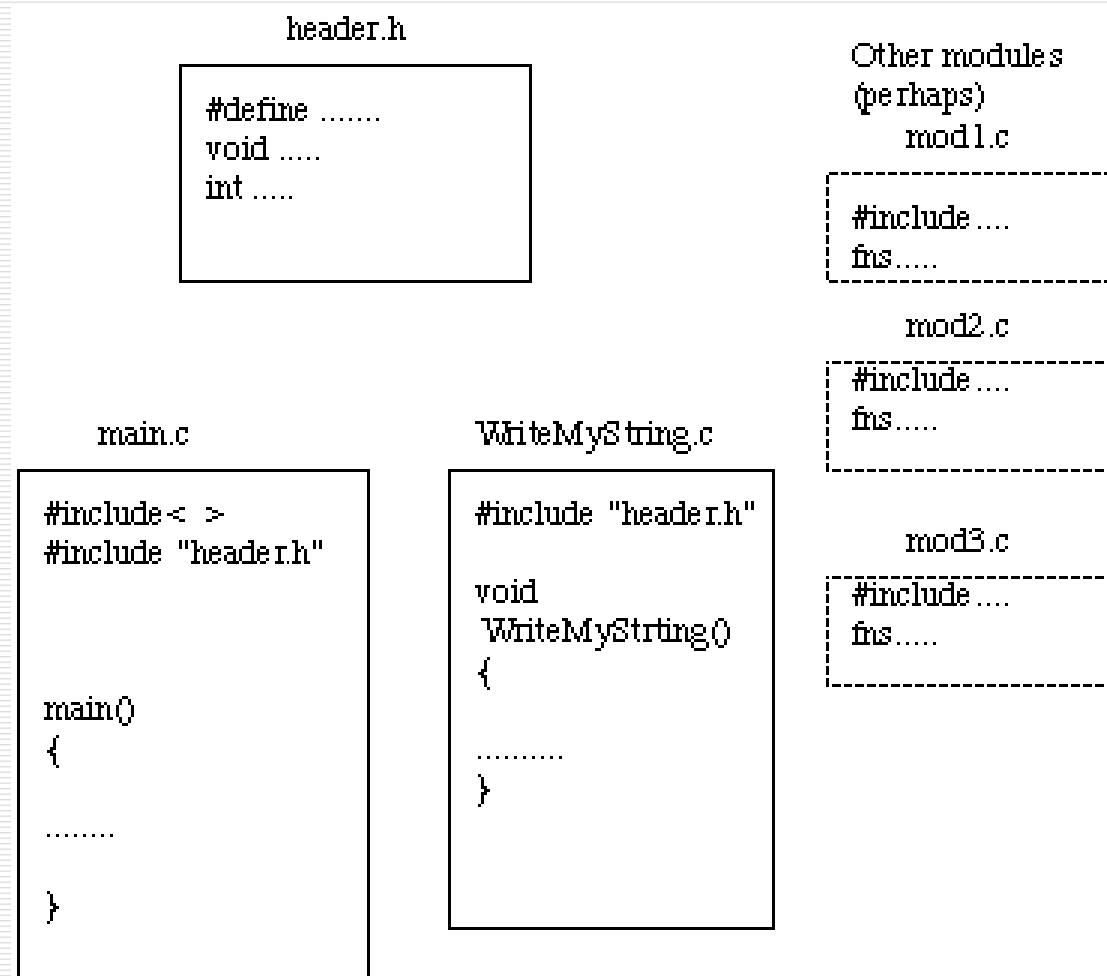
PROGRAMA =

- N funções
- main() é obrigatória
- main() corresponde à sequência de eventos a processar

Programa extenso...

utad

Programação procedimental



PROGRAMA =

- vários Módulos (*.c)
- Um ou mais Headers
- *#include*

NOTA : Um *header* file geralmente contém: Definição de tipos; Protótipos de funções; Comandos de Pré-processamento

Exemplo...

utad

Programação procedimental

```
/*
 * main.c
 */

#include "header.h"
#include <stdio.h>

main() {
    printf("Running...\n");
    WriteMyString(MY_STRING);
    printf("Finished.\n");
}
```

```
/*
 * header.h
 */
#define MY_STRING "Hello
World"

void
WriteMyString(ThisString)
char *ThisString;
{
    printf("%s\n", ThisString);
}
```

Constantes simbólicas

utad

Programação procedimental

define

abreviatura

texto substituído

■ Constantes enumeradas

enum tipo { id0[= valor0], id1 [= valor1], ... }

Exemplos :

```
# define  MAXLINE  50
# define  BELL      '\x7'
# define  TRUE      1
```

.....

```
enum      boolean  { NO, YES};
enum      especiais { BELL = '\a', BACKSPACE = '\b', TAB = '\t' };
enum      meses    { JAN = 1, FEV = 2 , MAR = 3};
```

Variáveis

utad

Programação procedimental

- “contentores” de valores em memória
- nomes expressivos
- nomes não podem começar por dígitos
- deve ser declarada (definida) antes de ser utilizada
- Declaração: **tipo nome_variável [=valor_inicial];**

Exemplos :

```
int a;  
char val = 'a';  
int a, b=5, c=2*3;  
float pi, raio, perimetro;
```

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

Operadores aritméticos

utad

Programação procedimental

+	Soma
-	Subtracção
*	Multiplicação
/	Divisão
%	Resto da divisão inteira

NOTA :

- *O operador % não pode ser aplicado a float e double*
- *+ e - têm a mesma precedência*
- *+ e - têm menor precedência que *, / e %*
- *Associatividade da esquerda para a direita*

Operadores relacionais e lógicos

utad

Programação procedimental

Relacionais:

> >= < <= == !=

Lógicos

&& || !

NOTA :

Operadores relacionais têm menor precedência que os aritméticos

Ex: a expressão $i < lim - 1$ é vista como $i < (lim - 1)$

Expressões avaliadas da esquerda para a direita

&& tem maior precedência que ||; ambos têm menor precedência que os operadores relacionais e de igualdade

Ex: $i < lim - 1 \ \&\& \ lim > 12$

O operador ! converte em zero (0) um operador diferente de zero e em um (1) um operador igual a zero

Ex: $if (! valido)$ é idêntico a $if (valido == 0)$

Operadores de incremento e decremento

++ --

utad

Programação procedimental

- *Operadores unários*
- *O operador ++ adiciona um (1) ao seu operando*
- *O operador -- diminui um (1) ao seu operando*
- *Podem ser usados antes ou depois do operando:*

<i>++ n</i>	<i>-- n</i>	<i>antes</i>
<i>n ++</i>	<i>n --</i>	<i>depois</i>

Efeito:

<i>++ n :</i>	<i>primeiro incrementa a variável e depois usa-a</i>
<i>n ++ :</i>	<i>primeiro usa a variável e só depois a incrementa</i>

O mesmo acontece com o operador -- .

Operadores sobre Bits

utad

Programação procedimental

~	Complemento para 1
<<	Deslocamento à esquerda (multiplicar por múltiplos de 2)
>>	Deslocamento à direita (dividir por múltiplos de 2)
&	AND bit a bit
^	XOR bit a bit
	Or bit a bit

Exemplo:


x	0 1 0 1 1 0 1 0
y	0 0 1 0 1 0 1 0
<hr/>	
x ^ y	
~x	
x & y	
x y	
x << 1	
y >> 3	

```
#include <stdio.h>
main() {
    printf("8>>2: %d\n", 8>>2);
    printf("8<<5: %d\n", 8<<5);
    return;
}
```

Exercício:

utad

Programação procedimental



x	0 1 0 1 1 0 1 0
y	0 0 1 0 1 0 1 0

$x \wedge y$	0 1 1 1 0 0 0 0
--------------	-----------------

$\sim x$	1 0 1 0 0 1 0 1
----------	-----------------

$x \& y$	0 0 0 0 1 0 1 0
----------	-----------------

x / y	0 1 1 1 1 0 1 0
---------	-----------------

$x \ll 1$	1 0 1 1 0 1 0 0
-----------	-----------------

$y \gg 3$	0 0 0 0 0 1 0 1
-----------	-----------------



Operador atribuição

= op =

utad

Programação procedimental

Syntax

var = *exp*

var *op* = *exp*

- A atribuição dá um resultado do tipo de *var*
- Se a variável do lado esquerdo também aparece no lado direito $e1 = e1 \text{ op } e2$, podemos utilizar o operador com atribuição;

$e1 \text{ op} = e2$	op =		+	-
			*	/
			%	&
			>>	<<
			^	

Exemplo:

$n += 2$

\Leftrightarrow

$n = n + 2$

Operador de condição

e1 ? ac1 : ac2

utad

Programação procedimental

- Associa da esquerda para a direita
- É avaliada $e1$ em primeiro lugar. Caso seja verdadeira, é executada a acção $ac1$; caso contrário executa-se a acção $ac2$
- Em determinadas condições pode ser atribuído a uma variável

Exemplo:

$$z = (a > b) ? a : b;$$

```
/* z = max ( a, b ) */
```

```
printf ( " Valor %s ", n > 0 ? "Positivo" : "Negativo" );
```

Operador ,

utad

Programação procedimental

- Separa os argumentos de uma função
- É usado como operador em expressões com “,”

func (*i*, (*j* = 1, *j* += 4), *k*); Trata-se de uma chamada da função *func* com 3 argumentos. Os argumentos são *i*, **5** e *k*.

E1, E2 A expressão *E1* é executada como uma expressão *void*, depois *E2* é executada, sendo o seu resultado e tipo devolvido na expressão como um todo.

Exemplos:

a = (k=2 , ++k , k+=1)

a tem o valor 4

printf (“ %d\n”, (b=1, a=2, c= b+a));

escreve 3 no écran

Tipos de Dados (básicos)

utad

Programação procedimental

É de salientar que alguns tipos não têm tamanho (em bytes) igual para todos os compiladores, como é o caso do inteiro que tem 4 bytes em *unix* e em outros sistemas operativos tem 2 bytes (nas versões mais antigas do *windows*, por exemplo).

char

float

int

double

long

short

Void

(não especifica nenhum tipo)

Tipos de Dados (básicos)

utad

Programação procedimental

Variáveis

Tipo	bytes	limite mínimo	limite máximo
char	1	-128	127
unsigned char	1	0	255
short int	2	-32768	32767
unsigned short int	2	0	65535
int	4	-2147483648	2147483647
Long int	4	-2147483648	2147483647
float	4	$-3.2 \times 10^{\pm 38}$	$+3.2 \times 10^{\pm 38}$
double	8	$-1.7 \times 10^{\pm 308}$	$+1.7 \times 10^{\pm 308}$

Conversão de tipos

utad

Programação procedimental

- Em expressões aritméticas, *char* e *short* são convertidos para *int*
- *unsigned char* e *unsigned short* são convertidos para *unsigned*
- Se tiver tipos diferentes, o tipo inferior é convertido para o tipo hierarquicamente superior

$\text{int} < \text{unsigned} < \text{long} < \text{unsigned long} < \text{float} < \text{double}$

- Numa atribuição, converte-se para o tipo da variável à esquerda com truncagem ou arredondamentos. Pode-se explicitar a conversão com a utilização do *cast*:

Ex: $(\text{int}) 2.3$

Conversão de tipos (cont.)

utad

Programação procedimental

Exemplos:

E1 = E2

Float	⇐	Double	(Arredondamento)
Int	⇐	Float	(Truncagem)
Short	⇐	Long	(Perde bits significativos)
Char	⇐	Long	(Perde bits significativos)

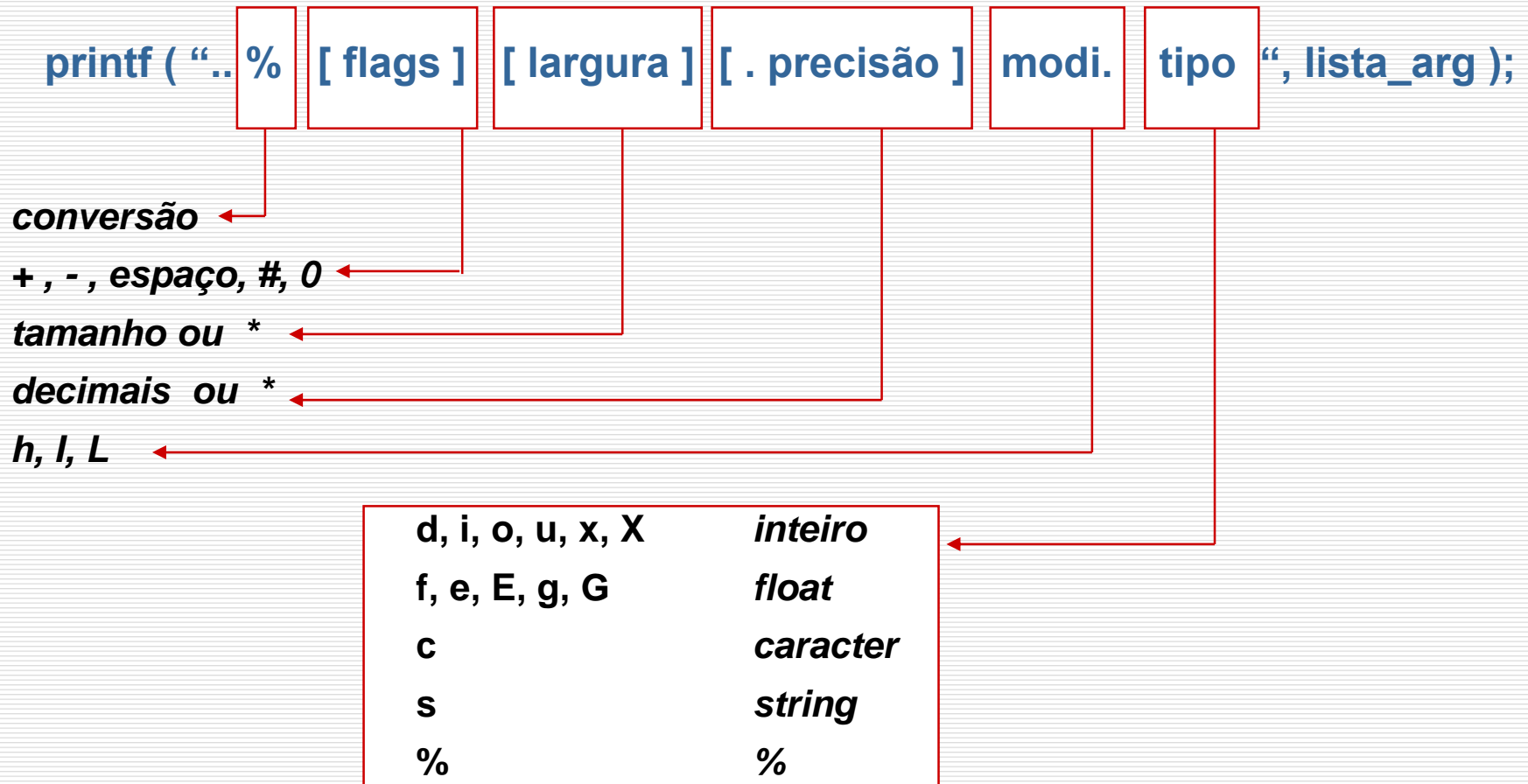
<code>int a;</code>	<code>a = 2.345;</code>	a tem o valor 2
<code>float b;</code>	<code>b = 2;</code>	b tem o valor 2.00...
<code>short c;</code>	<code>c = 3276801;</code>	c tem o valor 1

Funções de I/O Formatado

printf

utad

Programação procedimental



printf

utad

Programação procedimental

% Inicia o bloco de conversão

Flags:

sinal menos (-) significa alinhamento à esquerda do argumento convertido. Por defeito o alinhamento é feito à direita.

sinal mais (+) especifica que o número deve ser mostrado com sinal explícito.

caracter espaço (' ') especifica que se o primeiro caracter não for um sinal, deve ser guardado um espaço suplementar.

caracter zero ('0') em conversão numérica significa preencher com zeros à esquerda do número, toda a largura do campo

Largura:

especifica o número mínimo de caracteres que o argumento precisa (sem *truncar*). Se o número for escrito antecedido de zero, o espaço sobranete é preenchido com zeros.

printf

utad

Programação procedimental

Precisão:

Diferentes significados consoante o tipo dos argumentos:

String: número máximo de caracteres a apresentar
(com *truncagem*);

Float e Double: número de casas decimais à direita
do ponto decimal num valor em vírgula flutuante;

Inteiro: número mínimo de dígitos (acrescentando se
necessário com zeros à esquerda:

Modificador:

Opcional, para converter tipos em *short* (**h**), em *long int* (**l**) ou *long double* (**L**);

Tipo:

Especifica a conversão que deve ocorrer ao argumento

printf

utad

Programação procedimental

NOTA: Em vez de números, podem-se especificar os campos *tamanho* e *precisão*, com um *. Por exemplo, (**%*.*c**), indica que deve ser usado o primeiro argumento para o tamanho, o segundo para a precisão e o terceiro. para o caracter.

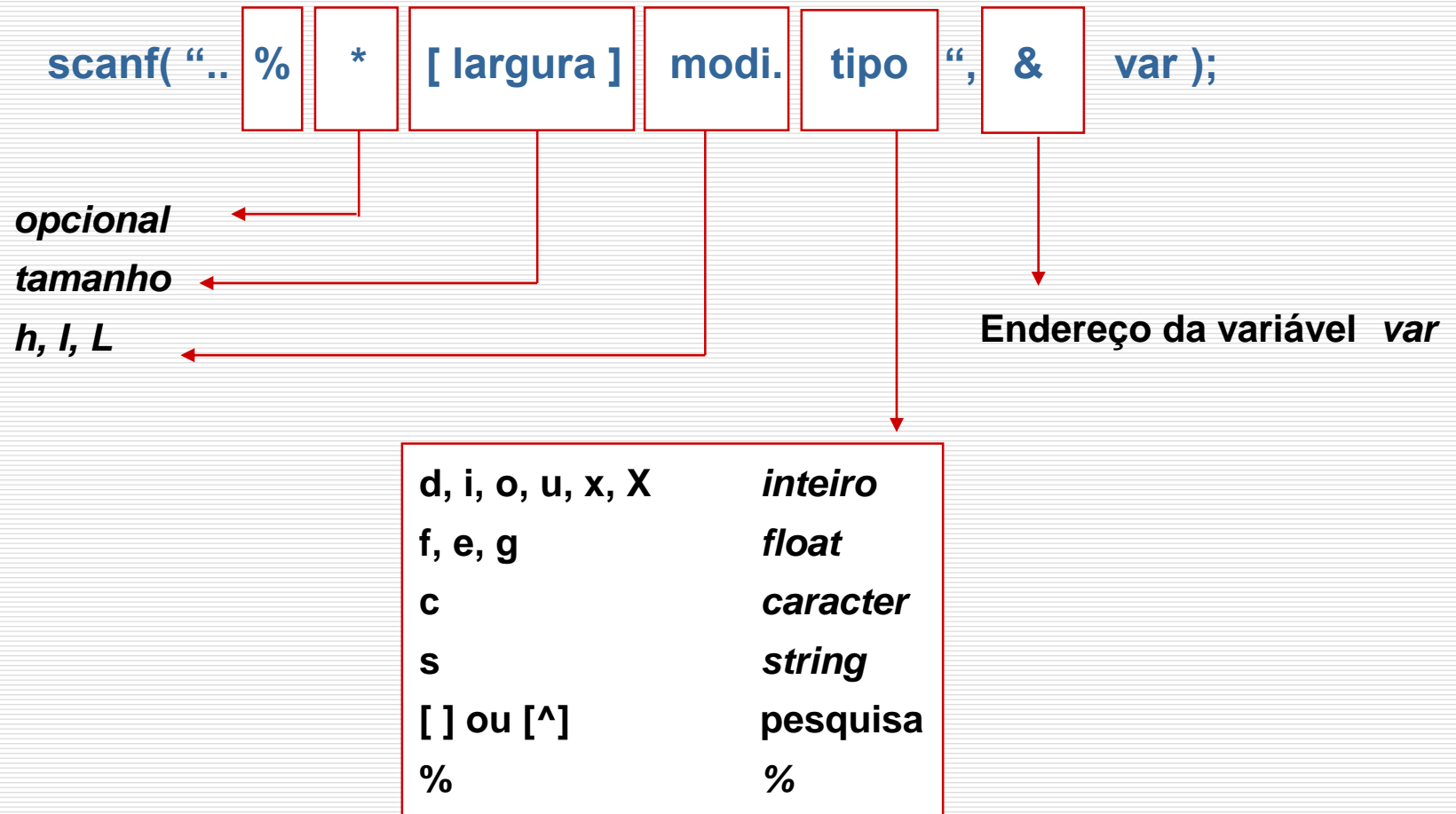
Exemplo : `printf (“\n %*d”, 20, 10);` irá escrever o 10 a partir da coluna 20

Funções de I/O Formatado

scanf

utad

Programação procedimental



scanf

utad

Programação procedimental

- * significa que a entrada é consumida mas não é afectada a nenhum argumento

Largura:

decimal positivo que determina o número máximo de caracteres a ler do *stdin*

Modificador :

Especifica o tipo de dados a recolher; **(h)** short e **(l)** long.
Por exemplo, lf, significa double.

Tipo:

%c : Ler o próximo caracter, mesmo que se trate de um caracter *branco* (' ' , '\t', '\n'). Para saltar estes caracteres e ler o próximo diferente deles, usar **%1s**;

%s: Ler uma sequência de caracteres

%[conjunto-de-pesquisa]: Definem os caracteres que podem ser lidos. Se o primeiro caracter for (^), o conjunto de pesquisa é o inverso.

scanf

utad

Programação procedimental

NOTA:

Um caracter espaço (‘ ‘) no inicio ou entre os especificadores do formato, elimina todos os caracteres *brancos* existentes no buffer.

Um caracter diferente de espaço, provoca que esse caracter seja tomado como separador de dados, pelo que será desprezado.

Por exemplo:

```
scanf (“ %d,%d”, &x, &y);
```

impõe a scanf() ler primeiro um inteiro para x, em seguida ler e desprezar uma vírgula e, finalmente, ler outro inteiro para a variável y. Se o separador especificado não for encontrado, scanf() termina.

Exercícios

utad

Programação procedimental

- Experimentar `printf` e `scanf` formatado
 - Não esquecer dos *header*

Comandos de decisão

utad

Programação procedimental

if (***cond***)

acção 1

else

acção 2

Notas:

- A condição ***cond*** tem um valor lógico (F ou V)
- A acção 1 executa-se caso a condição ***cond*** seja verdadeira
- A acção 2 executa-se caso a condição ***cond*** seja falsa
- Sempre que as acções a executar sejam compostas por mais que uma instrução é necessário usar as chavetas ({ })
- Um ***else*** está sempre associado ao ***if*** mais recente que não tenha um ***else*** ...

Exemplos

utad

Programação procedimental

① **if (a > 20)**
printf (“ Número ...\\n”);
printf (“ Maior...\\n”);



② **if (a > 20)**
{
printf (“ Número ...\\n”);
printf (“ Maior...\\n”);
}



Exemplos

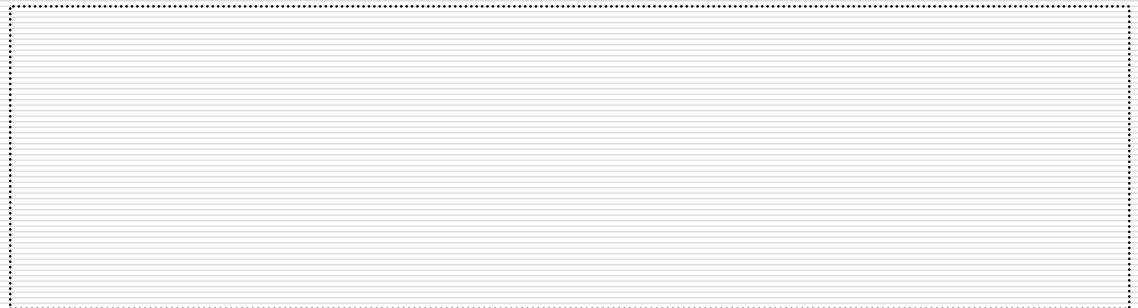
utad

Programação procedimental

```
③ if ( N_teórica > 15 )  
    if ( N_prática > 14 )  
        printf ( " Bom ...\n" );  
    else  
        printf ( " Xau ...\n" );
```



```
④ if ( N_teórica > 15 )  
    {  
        if ( N_prática > 14 )  
            printf ( " Bom ...\n" );  
    }  
else  
    printf ( " Xau ...\n" );
```



Comandos de decisão (cont.)

break;

utad

Programação procedimental

- O comando **break** permite sair de um ciclo, sendo executada a instrução imediatamente a seguir
- Aplica-se aos ciclos **do**, **while** e **for** e à instrução de decisão **switch**

Notas:

```
tot = 0;
while ( (ch = getchar()) != '\n' )
{
    if ( ch == '0' )
        break;
    putchar (ch);
    tot++;
}
printf ("Total de caracteres : %d\n", tot);
```

- Executa o ciclo até que encontre o fim de linha (\n) ou encontre o caracter zero (0);

Exemplo :

nº 123 Lisboa 1000 <return>

Total de caracteres : 15

Comandos de decisão (cont.)

switch

utad

Programação procedimental

```
switch ( expressão_inteira )  
  {  
    case conts_int1:  
      ..... /* opcional */  
      break;  
  
    case conts_int2:  
      ..... /* opcional */  
      break;  
    .....  
  
    default : ..... /* opcional */  
  }
```

O comando *switch* permite escolher uma entre várias alternativas

switch

utad

Programação procedimental

- O comando **switch** avalia a expressão entre parênteses e compara o seu valor com as hipóteses **case**.
- A instrução a executar é a do **case** cujo valor é igual ao valor da expressão. Todas as instruções até ao **break** *mais próximo* são então executadas.
- A instrução **break** causa a saída do **switch**.
- Se nenhum **case** satisfizer a expressão, é executada a alternativa **default** (caso exista).

Exemplo:

```
.....  
switch ( ch = getchar() )  
{  
    case '+':    res = num1 + num2;  
                break;  
    .....  
    default :    res = num1 * num2;  
}  
printf ( ".....\n", res );
```




utad

Programação procedimental

```
main()
{
    float num1, num2, resultado=0;
    char op;

    printf (" Numero operador Numero \n");
    scanf ("%f %c %f", &num1, &op, &num2);
    switch ( op )
    {
        case '+':
            resultado = num1 + num2;
            break;
        case '-':
            resultado = num1 - num2;
            break;
        case '*':
            resultado = num1 * num2;
            break;
        case '/':
            resultado = num1 / num2;
            break;
        default :
            printf(" Operador errado ...\n");
    }
    printf ( " %f %c %f = %f \n", num1, op, num2, resultado)
}
```

O que faz este código?

Controlo de fluxos (ciclos)

while

utad

Programação procedimental

```
while ( cond )  
{  
    acção  
}
```

- **cond** é uma expressão que tem valor booleano (V ou F)
- Se a condição **cond** for verdadeira, i.e, diferente de zero (0), o corpo do ciclo (**acção**) é executada.
- O teste da condição **cond** é sempre feita após a execução da **acção**, com a excepção da primeira vez, que é testada à entrada do ciclo
- O ciclo repete-se até que condição seja falsa

Controlo de fluxos (ciclos)

do

utad

Programação procedimental

```
do  
{  
    acção  
}  
while ( cond );
```

- Tanto o ciclo **while** como o ciclo **do** usam-se em situações em que o nº de vezes que o ciclo vai ser repetido é desconhecido, dependendo das acções que ocorrem dentro dele.
- O ciclo **do** difere do ciclo **while** na forma como os testes são feitos. No **while** testa-se à entrada, enquanto que no **do** testa-se à saída. O ciclo **do** executa-se pelo menos uma vez.

Exemplo

utad

Programação procedimental

/* Adivinhar letras */

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
main()
{
    char aux, c;
    int tentativas;
        aux = rand() % 26 + 'a';
        tentativas=1;
        printf("Introduza uma letra:");
        while ( (c = getch()) != aux && tentativas < 10)
        {
            printf ("Essa letra não é a correcta. Teste de novo. \n");
            tentativas++;
        }
        if (tentativas < 10)
        {
            printf (" %c  é correcto ....",c);
            printf (" Precisou de %d tentativas \n", tentativas);
        }
        else
            printf("Paciência!");
return 0;
}
```

Controlo de fluxos (ciclos)

for

utad

Programação procedimental

```
for ( inicialização ; teste ; incremento )  
{  
    acção  
}
```

- O ciclo *for* deve ser utilizado quando é conhecido o número de vezes que o ciclo vai executar;
- Cada uma das expressões que compõem o ciclo *for* podem ser formadas por várias instruções, separadas por virgulas.
- Podem-se omitir qualquer uma das expressões que compõem o ciclo embora devam permanecer os pontos-e-virgula.

Exemplo

utad

Programação procedimental

```
/* Números de 0 a 9 e totais */

# include <stdio.h>

main()
{
    int conta, total;

    for ( conta=0, total=0 ; conta < 10 ; conta ++ )
    {
        total += conta;
        printf (" conta = %d, total = %d\n", conta, total);
    }
}
```

Exercício

utad

Programação procedimental

- Apresentar a tabela da tabuada

Exemplo

utad

Programação procedimental

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
main()
{
    int i, j;
    printf("      *      ");
    for (j=1; j<10; j++)
        printf("%d\t", j);

    printf("\n");
    for (i=1; i<10; i++)
    {
        printf("\n");
        printf("%d\t", i);
        for (j=1; j<10; j++)
            printf("%d\t", i*j);
    }
    return 0;
}
```

Tabela da tabuada

Metodologias de Programação I

Pointers

(Apontadores e Endereços)

Pointers (Apontadores e Endereços)

utad

Programação procedimental

Variáveis

são entidades com: **nome**, **tipo**, **valor**, **localização**

o **nome** e o **tipo** são estabelecidos na sua declaração

o **valor** é usado (acedido) através do nome, este valor é modificado pelas instruções de afectação

a **localização** é estabelecida pelo compilador, e é materializada pelo endereço da posição de memória

Em **C**, as variáveis podem ser acedidas pelo seu endereço, o mecanismo que permite fazer isto são os **apontadores**.

Em **C**, os **apontadores** são usados em muitas circunstancias, como por exemplo, no processamento de vectores, passagem de parâmetros (saída) em funções, estruturas de dados, listas,...

Pointers (Apontadores e Endereços)

utad

Programação procedimental

operadores **&** e *****

os apontadores surgem através de destes dois operadores

o operador **&** quando aplicado a uma variável devolve o seu endereço

o operador ***** quando aplicado a um endereço devolve a variável,
localizada nesse endereço

EXEMPLO

considere a seguinte declaração: `int n;`

a expressão `&n` representa o endereço de `n`

a expressão `*&n` representa a variável `n`

Pointers (Apontadores e Endereços)

utad

Programação procedimental

tipos de apontadores

variáveis cujos os valores são endereços de outras variáveis

para cada tipo de variável existe também um tipo de apontador

exemplo da declaração de uma variável de *x* cujo tipo é apontador para inteiro:

```
int *p;
```

```
p = &x; /* o valor de p é o endereço de x */
```

```
*p
```

```
/* designa a variável x */  
/* diz-se que aponta para x */
```

```
x++;
```

```
(*p)++;
```

```
/* têm o mesmo significado */
```

Pointers (Apontadores e Endereços)

utad

Programação procedimental

Apontador = variável que contém um endereço de memória

Declaração:

*tipo *nome_variável;*

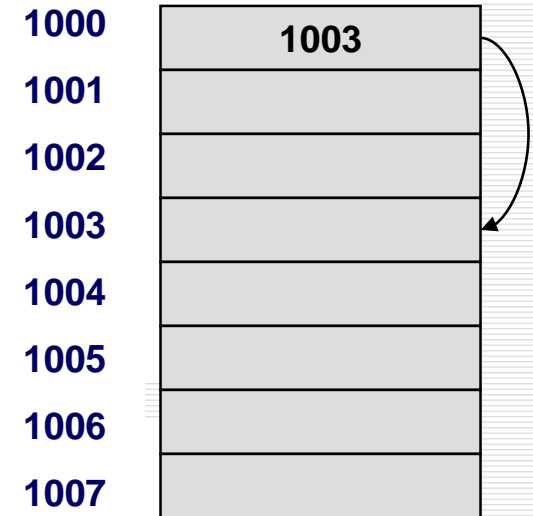
Operadores:

& - endereço $x = \&y$
 * - apontador $x = *y$

Exemplo 1:

```
int aux = 200, aux2;  
int *p, *q;
```

```
p = &aux;    /* p possui o endereço de aux ou p aponta para aux */  
q = &aux2;   /* q possui o endereço de aux2 ou q aponta para aux2 */  
aux2 = *p;   /* aux2 fica com o valor de aux ou aux2 toma o valor do apontado por p */
```



Memória

Pointers (Apontadores e Endereços)

utad

Programação procedimental

Exemplo 2:

```
int  x = 1, y = 2, z[10];
int *ip, *iq; /* ip é apontador para inteiros */
ip = &x;      /* ip agora aponta para x */
y = *ip;      /* y tem o valor 1 */
*iq = 0;      /* o conteúdo de iq passa a ter o
                                valor 0 */
iq = &z[0];   /* ip aponta agora para z[0] */
```

Operações:

```
*ip = *ip + 10; /* x tem o valor 11 */
y = *ip + 1;    /* y tem o valor 12 */
*ip += 1        /* incrementa o valor de x */
++*ip;         /* incrementa o valor de x */
(*ip)++;       /* incrementa o valor de x */
iq = ip;       /* iq aponta para x */
```

Metodologias de Programação I

Subprogramas

Subprogramas

utad

Programação procedimental

Um problema complexo é mais fácil de resolver se primeiro o dividirmos em pequenos sub-problemas.

Os subprogramas destinam-se principalmente a:

- Permitir a criação de rotinas ou partes de código que podem ser usadas mais do que uma vez no programa;
- Estruturar melhor o programa;
- Escrever determinadas porções de código de uma forma mais autónoma;
- Construir programas de leitura, compreensão e correcção mais fáceis.

Funções embutidas ou pré-definidas

utad

Programação procedimental

São funções já definidas e implementadas no compilador (*ex.: sin, cos, tan, abs, sqrt, sqr...*). São utilizadas directamente em expressões como se fossem variáveis regulares.

Exemplo 1:

Valor \leftarrow *sin(angulo) + cos(angulo)*

É da responsabilidade do programador fornecer o(s) argumento(s) (no exemplo acima *angulo*), sobre os quais a função opera para o cálculo do resultado de saída.

Exemplo 2:

x \leftarrow 3
y \leftarrow 5
res1 \leftarrow *sqrt*(*x*)
res2 \leftarrow *sqrt*(*y*)

argumento de entrada

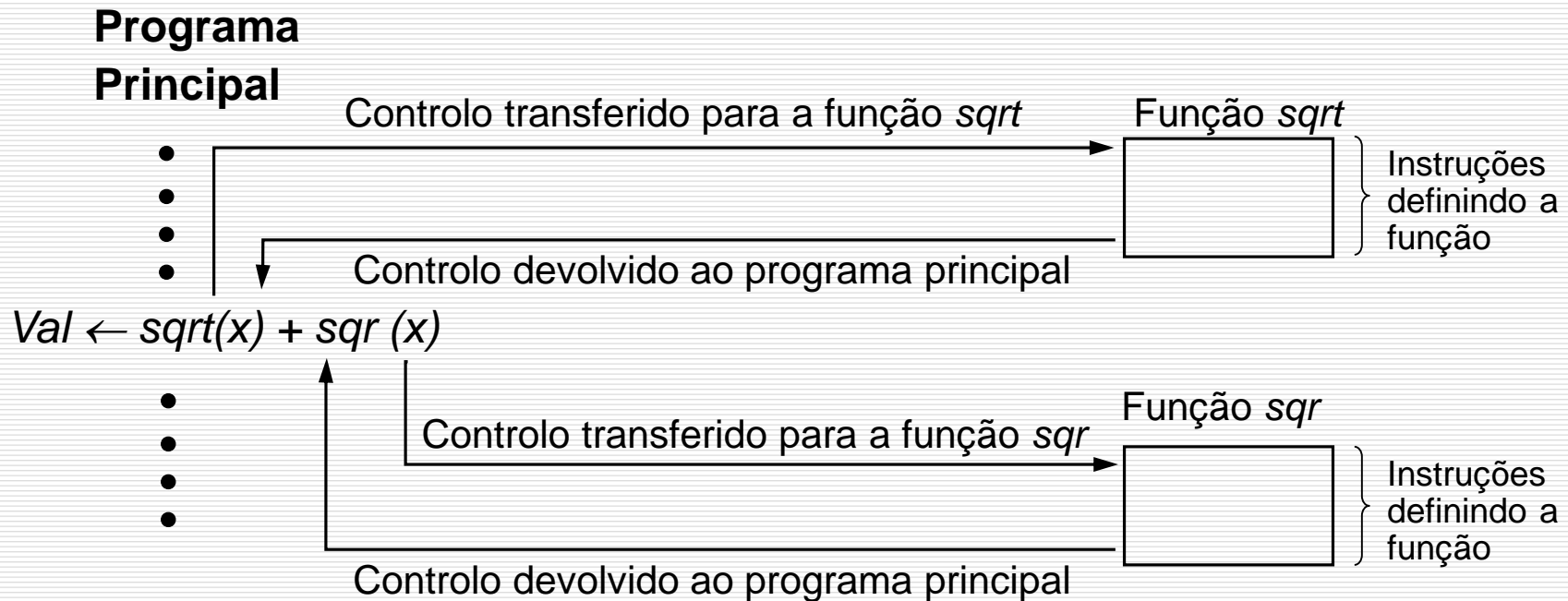
Nome da função

Efeito do uso de subprogramas no fluxo de controlo de um programa

utad

Programação procedimental

A utilização de subprogramas afecta o fluxo de controlo num algoritmo. Quando se dá a chamada a um subprograma, a sequência de operações do programa principal passa para o subprograma em questão até que este termine. Em seguida, a sequência de operações é retomada no ponto logo a seguir onde essa chamada foi feita.



Sintaxe algorítmica para funções

utad

Programação procedimental

Para além de utilizar as funções pré-definidas o programador pode definir as suas próprias funções. A forma de definir em linguagem algorítmica uma função é:

```
Função nome (parâmetro1, parâmetro2, ..., parâmetron)
  Tipo dos parâmetros
    Variáveis locais
    Conjunto de instruções
Devolver (valor de saída)
```

O valor que a função retorna é dado como uma expressão entre parêntesis a seguir à palavra chave **Devolver**.

Devolver indica o ponto em que o controlo regressa ao ponto de partida (ao algoritmo principal, na maioria dos casos), onde o valor devolvido pela função é então utilizado como parte da expressão na qual a função aparece.

A sintaxe geral para a definição de uma função, em linguagem C, é a seguinte:

```
(tipo de retorno) nome_da_função (parâmetro_1, ... parâmetro_n,)
{
    variáveis locais
    Conjunto de instruções
    return (tipo de variável/valor)
}
```

Variáveis globais/variáveis locais

utad

Programação procedimental

Quando uma variável é declarada antes de todos os subprogramas, diz-se que é uma variável global ou que tem um alcance ou raio de acção global - querendo isto dizer que é reconhecida em todo o programa, não só na parte operativa do programa principal , mas também dentro de qualquer subprograma (são visíveis a partir do local da declaração).

Quando uma variável é declarada na parte declarativa de um subprograma então diz-se que se trata de uma variável local ou que tem um raio de acção local - o que implica que a variável só é reconhecida dentro desse mesmo subprograma em que foi declarada

exemplo:

```
char a,b;
int n;
double numero;

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

exemplo:

```
void main(void)
{
    char a,b;
    int n;
    double numero;
    ...
    ...
}
```

Parâmetros e argumentos

utad

Programação procedimental

Em muitos casos, os subprogramas aceitam parâmetros - elementos que permitem passar dados ou valores do programa ao subprograma e, eventualmente, também no sentido contrário - o que lhes confere uma grande flexibilidade de utilização em situações diferenciadas.

Parâmetros

- São identificadores que são inseridos no cabeçalho de definição dos subprogramas;
- São usados para passar dados do programa principal para o subprograma e/ou vice versa;
- Podem ser usados nas instruções operativas dos subprogramas;
- Tal como acontece com as variáveis, cada parâmetro tem um tipo associado;
- Um subprograma recebe uma lista de parâmetros, podendo esta ser nula.

Parâmetros e argumentos

utad

Programação procedimental

Argumentos

- São valores passados aos subprogramas em correspondência com os parâmetros.
- Na altura em que é feita a chamada ao subprograma, é tida em consideração a ordem dos argumentos, bem como o tipo a que pertencem.
- Os argumentos utilizados nas chamadas dos subprogramas podem ser valores directos, variáveis ou expressões, desde que os valores sejam compatíveis no tipo, com os correspondentes parâmetros .

Subprogramas

utad

Programação procedimental

A função **gdexy(x,y)** possui dois parâmetros de entrada e para cada avaliação da função é necessário o mesmo número de argumentos. É estabelecida uma correspondência entre os parâmetros e os argumentos fornecidos à função.

Exemplo:

$$res \leftarrow gdexy(a, b)$$

Neste exemplo a e b são os argumentos. Ao parâmetro x corresponde o argumento a e ao parâmetro y corresponde o argumento b .

A ordem é crucial; **gdexy(a,b)** tem um resultado diferente de **gdexy(b, a)**.

Tipos de passagem de parâmetros

utad

Programação procedimental

Existem dois métodos de associação entre **argumentos** e **parâmetros**:

- **Passagem por cópia de valor** (parâmetros de entrada)
- **Passagem por referência** (parâmetros de entrada/saída)

No primeiro caso, sempre que são feitas alterações nos parâmetros, dentro do subprograma, essas alterações não resultam em mudanças nos argumentos correspondentes. Isto porque os parâmetros não estão ligados aos argumentos; **é feita uma cópia de valores**.

No segundo caso, as alterações efectuadas nos parâmetros, dentro do subprograma, reflectem-se nos respectivos argumentos. Isto porque é passado o **endereço** de memória da variável.

Subprogramas - passagem de parâmetros (exemplo)

Pointers (Apontadores e Endereços)

utad

Programação procedimental

passagem de parâmetros em subprogramas:

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

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

neste exemplo, pretendemos que os efeitos laterais (alterações) aplicados aos parâmetros `x` e `y` da função `troca` sejam transmitidos aos argumentos correspondentes

chamada da função:

```
int  x=10, y=5;
int  *p, *q;
p = &x;
q = &y;
troca (&x, &y);    /* x passa a ter o valor de y e viceversa */
troca (p, q);      /* x passa a ter o valor de y e viceversa */
troca (x,y);       /* errado */
```

Subprogramas - passagem de parâmetros (exemplo)

Pointers (Apontadores e Endereços)

utad

Programação procedimental

Fica fácil perceber, agora, a razão pela qual se faz a passagem de endereços em vez de variáveis quando se utiliza a função predefinida ***scanf***, por exemplo:

```
int a;
```

```
float b;
```

```
scanf("%d %f", &a, &b);
```

as alterações (valores introduzidos pelo teclado) devem ser reflectidos nos argumentos a e b.

Se fossem utilizadas variáveis (caso a sintaxe da função *scanf* o permitisse) os valores introduzidos não eram reflectidos nos argumentos, pelo que apenas eram conhecidos dentro da função *scanf*.

Sintaxe algorítmica dos subprogramas (FUNÇÕES)

utad

Programação procedimental

Um programa, em linguagem **C**, é composto por um conjunto de funções. Estes subprogramas (funções) são semelhantes aos subprogramas possíveis de definir noutras linguagens (PASCAL). A função *main()* (programa principal) é também um subprograma.

No caso da linguagem **C** vamos passar a ver os subprogramas todos como funções, não existe o conceito de procedimento tão bem definido como no Pascal.

O número de tipos/valores a devolver (retornar), que distingue os conceitos entre funções e procedimentos em PASCAL, podem ser usados conjuntamente na linguagem **C**.

Um subprograma, em **C**, pode devolver:

- um tipo/valor através da palavra chave *return*;
- nenhum tipo/valor;
- um conjunto de tipos/valores, através da lista de argumentos e conjuntamente também pode ser utilizada a palavra chave *return* para devolver outro valor.

Subprogramas (exemplo)

utad

Programação procedimental

Exemplo: Vamos supor que queremos construir uma função para calcular $g(x,y)=x^2+y^2$.

ALGORITMO TESTE DA FUNÇÃO *gdexy*

função *gdexy*(*x*,*y*)

Parâmetros de entrada *x*,*y* do tipo real

variáveis locais *res* do tipo real

res \leftarrow *sqr*(*x*) + *sqr*(*y*)

Devolver (*res*)

Variáveis *a*,*b*,*res* do tipo real

a \leftarrow 5

b \leftarrow 3

res \leftarrow ***gdexy***(*a*,*b*)

Visualizar(*res*)

res \leftarrow ***gdexy***(*a* * *b*+3 , 2)

Visualizar(*res*)

fim de algoritmo

Subprogramas (exemplos)

utad

Programação procedimental

```
PASCAL
program TESTA_FUNC_gdexy;
    function
    gdexy(x,y:real):real
    var res:real;
    begin
        res := sqr(x) +
        sqr(y);
        gdexy := res
    end;
    var a,b,res:real;
begin
    a := 5;
    b := 3;
    res := gdexy(a,b);
    write(res);
    res := gdexy(a * b+3 , 2);
    write(res)
end.
```

```
C // TESTA_FUNC_gdexy
#include <stdio.h>
#include <math.h>
double gdexy(double,double);
int main()
{
    double a,b,res;
    a = 5;
    b = 3;
    res = gdexy(a,b);
    printf("%lf\n",res);
    res = gdexy(a * b+3 , 2);
    printf("%lf\n",res);
    return 0;
}
double gdexy(double x, double y)
{
    double res;
    res = pow(x,2) + pow(y,2);
    return res;
}
```

Subprogramas (exemplos)

utad

Programação procedimental

PASCAL

progra

fu

go

va

be

sq

en

va

begin

a

b

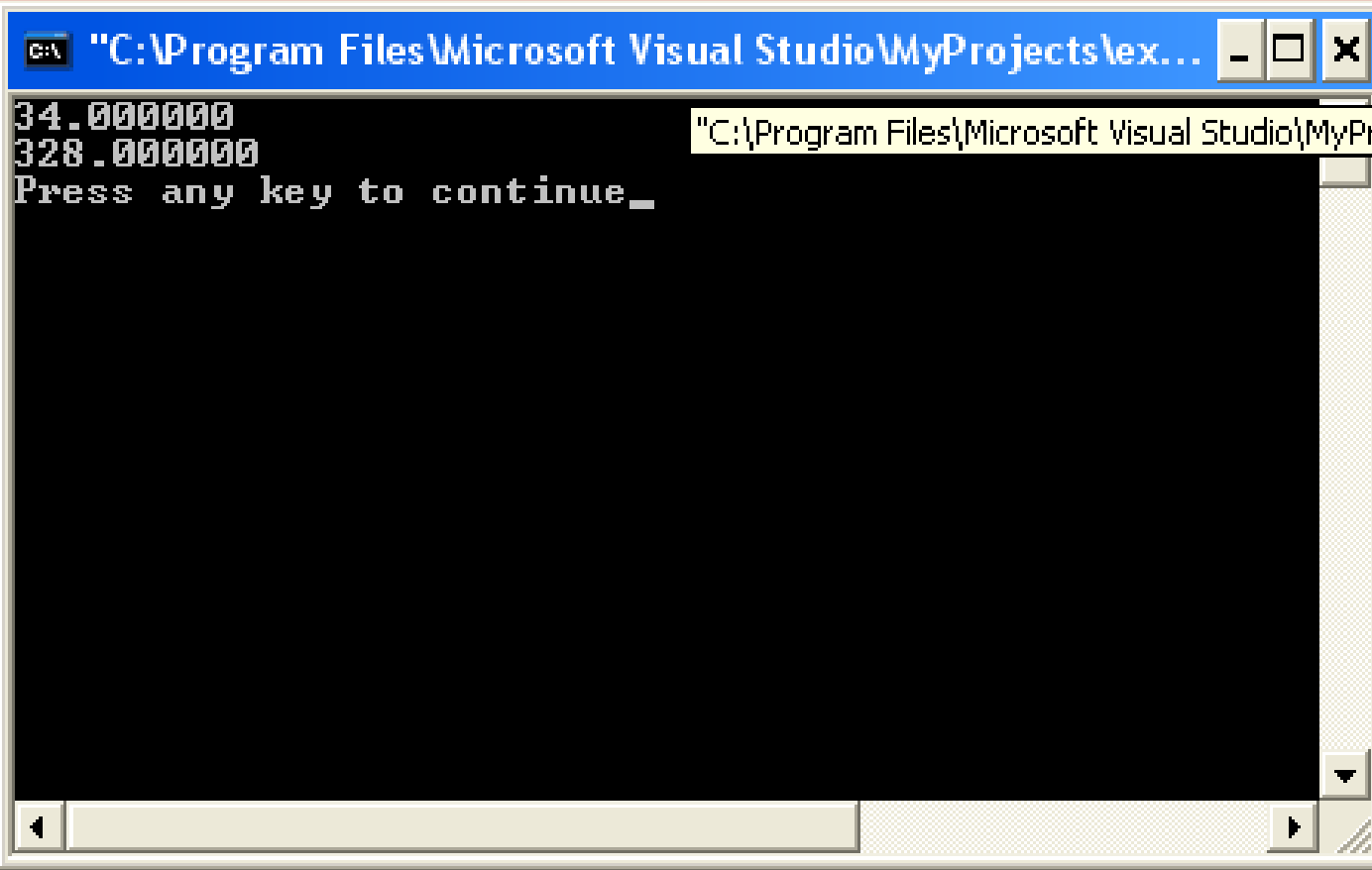
re

wr

res := **gdexy**(a * b+3 , 2);

write(res)

end.

C // TESTA FUNC *gdexy*

double);

es);

+3 , 2);

es);

double y)

```
res = pow(x,2) + pow(y,2);
return res;
```

}

Subprogramas (exemplo)

utad

Programação procedimental

Exemplo: Pretende-se desenvolver um procedimento que execute a divisão de dois valores inteiros e devolva, como resultados, o quociente e o resto da divisão inteira.

ALGORITMO TESTA PROCEDIMENTO divide

Procedimento divide (dividendo, divisor, quociente, resto)

Parâmetros de entrada: dividendo, divisor do tipo inteiro

Parâmetros de entrada/saída: quociente, resto do tipo inteiro

Variáveis locais: ...

quociente \leftarrow dividendo div divisor

resto \leftarrow dividendo mod divisor

Devolver

Variáveis locais: a, b, c, d do tipo inteiro

a \leftarrow 5

b \leftarrow 3

divide (a, b, c, d)

Visualizar(c, d)

divide (a*b , 2 , c, d)

Visualizar(c, d)

fim de algoritmo

Subprogramas (exemplo)

Linguagem C

Para exemplificar o conceito de variável local e global

```
// testa_proc
#include <stdio.h>
#include <conio.h>
int a; // variável global
void divisao(int , int , int *, int *);
int main()
{
    int dividendo,divisor,quociente,resto; // variáveis locais

    dividendo = 8;
    divisor = 3;
    divisao(dividendo,divisor,&quociente,&resto); // chamada do procedimento
    printf("\nQuociente = %d \nResto = %d",quociente,resto);
    getch();
    return 0;
}

void divisao(int x,int y, int *z,int *k)
// x,y passagem por copia de valor. z,k passagem por referência
{
    int b; // variável local
    b = 3;
    a = b+2;
    *z = (int) (x / y);
    *k = x % y;
}
```

subprograma

Programa principal

Subprogramas

Exemplo

utad

Programação procedimental

Dados a margem de lucro e o custo de um artigo, calcular o seu preço de venda ao público sabendo que a taxa de IVA a aplicar é de 19%.

Subprogramas

Notação Algorítmica (exemplo)

utad

Programação procedimental

ALGORITMO Etiquetagem

Constante IVA=17

Variáveis lucro, quantia do tipo
Real

Função pvp(custo do tipo real)

devolve pvp do tipo Real

Variável quantia do tipo Real

[Calcular o preco acrescido do lucro]

quantia \leftarrow custo*(100+lucro)/100

[Calcular o P.V.P.]

pvp \leftarrow pcIVA(quantia)

devolve pvp

Função pcIVA(preco do tipo Real)

devolve pcIVA do tipo Real

Variáveis imposto do tipo Real

[Cálculo do valor do IVA]

Imposto \leftarrow preco*IVA/100

[Calcular o P.V.P.]

pcIVA \leftarrow preco+imposto

devolve pcIVA

[Ler lucro associado a cada produto]

Visualizar('Percentagem de lucro (%): ')

Aceitar(lucro)

[Ler custo dos artigos e escrever o seu
p.v.p. até este ser nulo]

Repetir

Visualizar('Custo do artigo: ')

Aceitar(quantia)

Se (quantia > 0) **então**

Visualizar('P.V.P. = ', pvp(quantia))

Fim de Se

até quantia=0

Fim do algoritmo

Subprogramas

Linguagem C (Programa Etiquetagem)

utad

Programação procedimental

```
#include <stdio.h>
#include <conio.h>
#define IVA 19
double lucro;
double pvp(double);
double pcIVA(double);
main()
{
    double quantia;
    // Ler lucro associado a cada produto
    printf("Percentagem de lucro (%%): ");
    scanf("%lf",&lucro);
    //Ler custo dos artigos e escrever o seu p.v.p. até ler custo nulo
    do{
        printf("Custo do artigo: ");
        scanf("%lf",&quantia);
        if (quantia > 0)
            printf("P.V.P. = %lf",pvp(quantia));
    }while (quantia!=0);
    getch();
    return(0);
}
```

```
double pvp(double custo)
{
    double quantia;
    // Calcular o preço acrescido do lucro
    quantia = custo*(100+lucro)/100;
    // Calcular o P.V.P.
    return (pcIVA(quantia));
}

double pcIVA(double preco)
{
    double imposto;
    // Calcular o valor do IVA
    imposto = preco*IVA/100;
    // Calcular o P.V.P.
    return (preco+imposto);
}
```

Metodologias de Programação I

Arrays

- *Vectores*
- *Strings*
- *Matrizes*

ARRAY

utad

Programação procedimental

Um tipo de dados estruturado, composto por um número fixo de elementos do mesmo tipo.

- Uma variável deste tipo é designada por um identificador, mas cada elemento tem um ou mais índices associados.
- Pode-se aceder a qualquer componente de um **array** através do índice.
- Os arrays podem ser:
 - Unidimensionais (Vectores)
 - Multidimensionais (Matrizes)
- O limite inferior de um **array** em **C** é sempre zero.

Declaração :

tipo nome_array [num_elementos]

ARRAY

utad

Programação procedimental

Exemplo :

```
int valores [ 20 ]
```



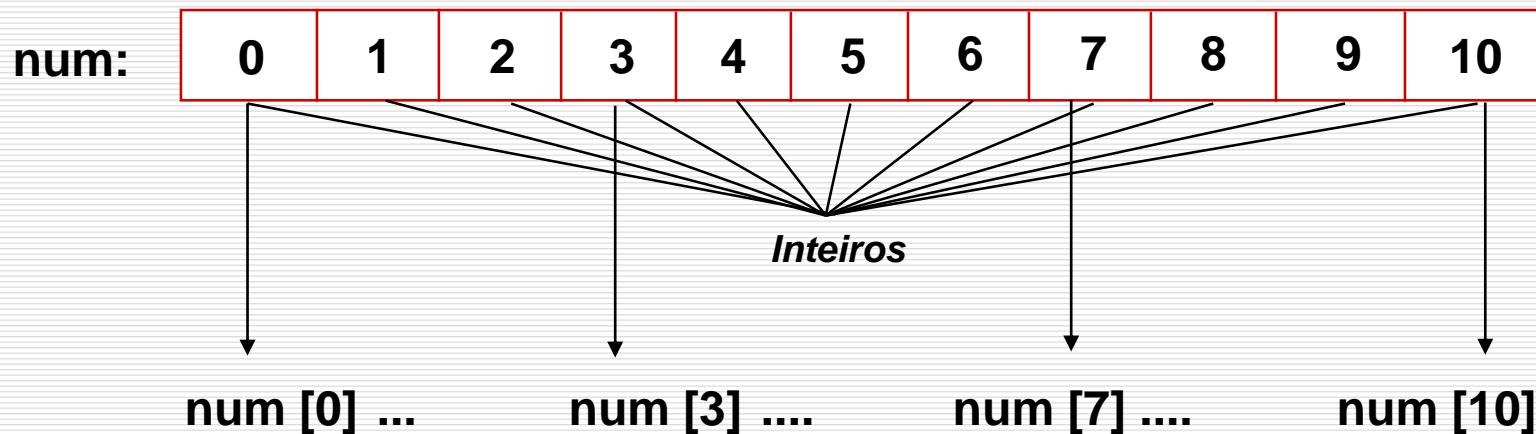
Array **valores** de inteiros

Nº de elementos : 20

Limite inferior : 0

Limite superior: 19

```
int num [11] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }
```



ARRAY

utad

Programação procedimental

- **num [0]** refere a primeira posição do *array* num
- **num [1]** refere a posição 1 do *array* num
-
- **num [i]** refere a iésima posição do *array* num
- O nome do *array* é o endereço (apontador) do início duma zona de memória que é alocada quando o *array* é declarado (suficiente para a dimensão do *array*).

Inicialização:

```
int          a [4] = { 3, 4, 5, -9};
```

```
int          a [ ] = { 3, 4, 5, -9};
```

```
char  letras [10] = {'a', 'b', 'c'};
```

```
char  nome [ ] = "Benfica" /* nome passa a ser um array de 8 elementos */
```

ARRAY - Exemplos

utad

Programação procedimental

1

```
int a = num [3];  
.....  
printf(" %d\n", a);
```

2

```
for (i=0; i<MAX; i++)  
    printf(" %d : ", num [i]);
```

3

```
i=0;  
while ( 1 )  
{  
    if ( isdigit ( ( ch=getche() ) ) )  
        num [i++] = ch;  
    if ( ch == '\n' )  
        break;  
}
```


ARRAY - Exemplos

utad

Programação procedimental

- Imaginem que queríamos fazer um programa para calcular a média de uma lista de 10 números e o desvio de cada número em relação à média. Com a matéria que aprendemos até agora, teríamos de fazer qualquer coisa deste estilo:

```
float x1, x2, x3, x4, x5, x6, x7, x8, x9, x10;  
float d1, d2, d3, d4, d5, d6, d7, d8, d9, d10;  
float media;  
...  
media = (x1+x2+x3+x4+x5+x6+x7+x8+x9+x10) / 10;  
d1 = x1 - media;  
d2 = x2 - media;  
...  
d10 = x10 - media;
```

- Se em vez de 10 números fossem 1000, teríamos de declarar 1000 variáveis e isso seria incomportável. É aqui que surge o conceito de *array*. A definição seguinte resolve este problema,

```
float x[10];
```

ARRAY - Exemplos

utad

Programação procedimental

```
float x[10];
```

- Define um array de **nome x** com **10 posições**, cada uma correspondendo a uma variável do **tipo float**. Um *array* é como se fosse uma lista ou sequência de variáveis. Na linguagem C, os *arrays* começam sempre na posição 0. Por isso, a declaração **float x[10]** define as variáveis:

`x[0], x[1], x[2], ..., x[9]`

- Depois de definido o array, os seus elementos podem ser acedidos e modificados individualmente, tal e qual como nas variáveis que vimos até agora. Por exemplo:

```
x[7] = 54;  
a = x[7];
```

ARRAY - Exemplos

```

#include <stdio.h>
main()
{
    float x[10], desvio[10]; /* lista de números e desvios em relação à média */
    float soma, media;
    int    i;
    /* introdução dos números e cálculo da média */
    soma=0;
    printf("Introduz 10 números: ");
    for(i=0; i<10; i++ )
    {
        scanf("%f", &x[i] );
        soma = soma + x[i];
    }
    media = soma / 10;

    /* calculo dos desvios */
    for( i=0; i<10; i++ )
        desvio[i] = x[i] - media;

    /* escreve a média, os números, e os respectivos desvios */
    printf("A média é %f\n", media );
    for( i=0; i<10; i++ )
        printf("x[%d] = %f    desvio[%d] = %f\n", i, x[i], i, desvio[i] );
}

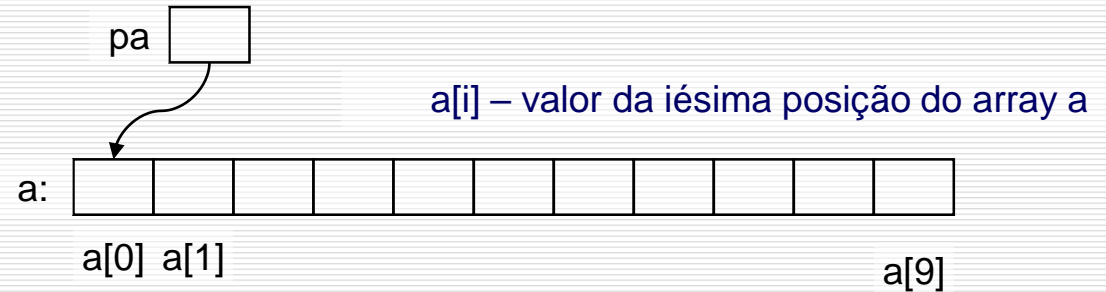
```

ARRAY

Pointers (Apontadores e Endereços)

utad

Programação procedimental



Arrays e Apontadores:

```
int a[10];           /* a é um array de 10 inteiros */
```

```
int *pa;             /* pa apontador para int */
```

```
pa = &a[0]           /* pa aponta para a primeira posição do array a */
```

```
pa = a               /* pa aponta para a primeira posição do array a */
```

```
aux = *pa            /* aux tem o valor de a[0] */
```

```
*(pa+1)              /* valor da 2ª posição do array a */
```

```
*(pa+4) == a[4];     /* equivalente */
```

ARRAY

Pointers (Apontadores e Endereços)

utad

Programação procedimental

Aritmética de apontadores

```
int *p, V[500];  
int i;
```

p = V+i;

p aponta para $V[i]$ – tal que: o valor de p é $\&V[i]$

Então: $V[i]$ pode ser acedido por $*p$

Tendo em conta que: $p+1$ aponta para $V[i+1]$ e $p-1$ aponta para $V[i-1]$

Então: $V[i+1]$ pode ser acedido por $*(p+1)$ e $V[i-1]$ pode ser acedido por $*(p-1)$

O vector V pode ser percorrido utilizando os operadores de incremento de decremento associados aos apontadores: $p++$ ou $p--$ equivalente a $p=p+1$ ou $p=p-1$ respectivamente

É necessário ter em conta que o resultado seja um apontador para um elemento que exista, isto é, os índices devem encontrar-se dentro dos limites do índice do vector definido quando da sua declaração

ARRAY

Pointers (Apontadores e Endereços)

utad

Programação procedimental

```
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#define N 15
int main(int argc, char* argv[])
{
    int *a,*b,V[N],c;

    for(a=V;a<V+N;a++)
        *a = rand() % 500;

    b = a-1;
    for(a=V;a<V+N;a++,b--)
        printf("%d\t%d\n",*a,*b);

    b = V+N-1;
    for(a=V;a<V+N/2;a++,b--) {
        c = *a;
        *a = *b;
        *b = c;
    }

    printf("\n\n#####\n\n");

    for(a=V;a<V+N;a++,b--)
        printf("%d\n",*a);
    return 0;
}
```

descreva o
output deste
programa

ARRAY

Pointers (Apontadores e Endereços)

utad

Programação procedimental

```
#include "
#include <
#include <
#define N
int main(i
{
int *a,*b,

for(a=

b = a-
for(a=

=====

b = V+
for(a=

}

printf

for(a=

return 0;
}
```

```
41      461
467      327
334      281
0        145
169      205
224      464
478      462
358      358
462      478
464      224
205      169
145      0
281      334
327      467
461      41

=====

461
327
281
145
205
464
462
358
478
224
169
0
334
467
41
Press any key to continue
```

a o
este
a

ARRAY

Pointers (Apontadores e Endereços)

utad

Programação procedimental

Arrays de Apontadores - *tipo nome_array [dimensão_array]*

exemplo:

```
#include "stdafx.h"
#include <stdio.h>
void mostra_array ( int *aux[] , int size );

int main(int argc, char* argv[])
{
    int *valores[10];          /* valores é um array de 10 apontadores para inteiros */
    int aux;

    valores [2] = &aux;        /* posição 2 contém o endereço de aux */
    (*valores[2])++;           /* incrementa aux */
    mostra_array ( valores, 10 );
    return 0;
}

void mostra_array ( int *aux[] , int size )
{
    int i;
    for (i=0; i<size; i++)
        printf ( " %p ",aux[i] );
}
```


ARRAY

Pointers (Memória dinâmica)

utad

Programação procedimental

Alocação Dinâmica

■ malloc()

```
void *malloc( size_t size );
```

Valor de retorno

Retorna um ponteiro *void ** para o espaço reservado ou *NULL* caso tenha ocorrido um erro.

Parâmetros

size - número de *bytes* a reservar (alocar)

■ free

```
void free( void *mемblock );
```

Valor de retorno

Nenhum

Parâmetros

mемblock - bloco de memória previamente reservado, deve ser libertado

ARRAY

Pointers (Memória dinâmica)

utad

Programação procedimental

Exemplo

```
int *x;    /* x é um apontador para int */
/* reserva 10 posições de memória do tipo int */
x = (int *) malloc(10*sizeof(int));

if(x ==NULL) {

    printf("Não foi possível reservar memória para x");

    return;

}
:
:
/* liberta a memória reservada pela função malloc para x */
free(x);
```

ARRAY

Pointers (Apontadores e Endereços)

utad

Programação procedimental

```
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#define N 15
int main(int argc, char* argv[])
{
    int *a,*b,*V,c;
    if((V = (int *)malloc(N*sizeof(int)))==NULL)
        return -1;
    for(a=V;a<V+N;a++)
        *a = rand() % 500;
    b = a-1;
    for(a=V;a<V+N;a++,b--)
        printf("%d\t%d\n",*a,*b);

    b = V+N-1;
    for(a=V;a<V+N/2;a++,b--) {
        c = *a;
        *a = *b;
        *b = c;
    }
    printf("\n\n#####\n\n");
    for(a=V;a<V+N;a++,b--)
        printf("%d\n",*a);

    free(V);
    return 0;
}
```

O mesmo programa do slide nº85 com memória dinâmica para o vector V

String

utad

Programação procedimental

- **String** é uma cadeia de caracteres.
- Em **C**, **string** é um *array* de caracteres, terminado pelo caracter **null** ('\0')
- Assim, cada caracter de uma **string** pode ser acedido como um elemento de um *array*, com a utilização do índice da posição.
- O caracter **null** ('\0') tem o código 0 (zero) decimal

```
char    texto [ ] = "Tripeiros."
```

T	r	i	p	e	i	r	o	s	.	'\0'
0	1	2	3	4	5	6	7	8	9	10

String

Declaração de variáveis

utad

Programação procedimental

Como **strings** são arrays de caracteres, são declarados como qualquer array. O operador `[]` contém o tamanho da **string** e o tipo é, claro está, **char** ou **unsigned char**

```
char text [81];  
char comando [30];  
unsigned char letras [24];
```

⇒ Inicialização de variáveis do tipo **String**

❶ `char nome [] = { 'P', 'o', 'r', 't', 'o', '\0' };`

❷ `char nome [] = "Porto";`

❷ Neste caso insere-se de forma automática o caracter `'\0'`

String

Manipulação de *strings*

utad

Programação procedimental

- Inicializar
- Ler
- Escrever
- Copiar
- Comparar
- Tamanho

String

Ler de *strings*

utad

Programação procedimental

Ler uma *string* consiste em dois passos: reservar espaço para armazenar a *string* e utilizar alguma função que permita a sua leitura.



Funções:

scanf com o formato **%s**

```
Ex:    char nome[20];  
        .....  
        scanf (" %s", nome);
```

É preciso ter cuidado com os espaços entre palavras. A leitura termina quando for encontrado um espaço ou o fim de linha.

Não é preciso o caracter **&** na leitura pois o nome da *string* é o endereço inicial do espaço reservado na memória.

String

Ler strings

utad

Programação procedimental

⇒ **gets** (*nome_string*);

gets lê caracteres do teclado até encontrar o caracter de nova linha (**\n**), que é gerado pressionando-se a tecla **RETURN**.

A string lida é automaticamente concluída com o caracter **'\0'**.

O caracter **'\n'** não faz parte da string lida.

```
main()
{
    char  nome [81];

    gets (nome);
    printf ("\n Ola %s", nome);
}
```


String

Escrever strings

utad

Programação procedimental

➡ **printf** com o formato `%s`

```
Ex:      char nome[20];  
  
         gets (nome);  
         printf ("%s", nome);
```

➡ **puts** (*nome_string*);

puts imprime uma *string* de cada vez. É o complemento da função *gets*.

Cada *string* impressa por *puts* termina por um caracter de nova linha.

puts (*nome*) é equivalente a **printf** ("%s\n",*nome*)

String

Manipulação de *strings*

utad

Programação procedimental

Exemplo:

```
main()  
{  
    char  nome [81];  
  
    puts(" Digite o seu nome ...");  
    gets (nome);  
    puts ("Como está, ");  
    puts (nome)  
}
```

String

Outras funções de *strings*

utad

Programação procedimental

size_t	<i>strlen</i> (string);	Tamanho da <i>string</i> sem '\0'
char	* <i>strcat</i> (string1, string2);	Concatena as <i>strings</i>
int	<i>strcmp</i> (string1,string2);	Compara as <i>strings</i>
char	* <i>strcpy</i> (string1, string2);	Copia <i>strings</i>
char	* <i>strlwr</i> (string);	Converte para minúsculas
char	* <i>strupr</i> (string);	Converte para maiúsculas

size_t é um tipo usado para tamanhos de objectos em memória (*unsigned*)

String

Pointers (Apontadores e Endereços)

utad

Programação procedimental

Exemplo :

```
/* strlen : devolve o número de caracteres de uma string */  
  
int strlen (char *s)  
{  
    int i;  
    for (i=0; *s != '\0'; s++)  
        i++;  
    return i;  
}
```

Evocação:

```
strlen ("Ola mundo"); /* constante String */  
strlen ( array );      /* char array[20] */  
strlen (ptr);          /* char *ptr; */
```

String

Pointers (Apontadores e Endereços)

utad

Programação procedimental

Arrays de Apontadores : Exemplo

```
void error ( int error_number )
{
    static char *error_msg [ ] = {
        "Não pode aceder\n",
        "Disquete não disponível ou com problemas\n",
        "Máquina Desligada\n" }

    puts (error_msg [error_number] );
}

main() {
    FILE *fp;

    if (fp=fopen("a:\ola.txt", "r")) { /* abre o ficheiro */
        error (1);
        exit (0);
    }
}
```

String

Exercícios

utad

Programação procedimental

- ❑ Experimentar as funções de manipulação de strings

Arrays bidimensionais

Matrizes

utad

Programação procedimental

- ❑ É possível definir um *array* de *arrays*, o que equivale a falar de *arrays* multidimensionais.
- ❑ Se considerarmos um *array*, ou matriz, bidimensional, podemos facilmente representá-la como uma tabela ou quadro de 2 entradas, em que se atribui um índice às colunas e outro índice às linhas.

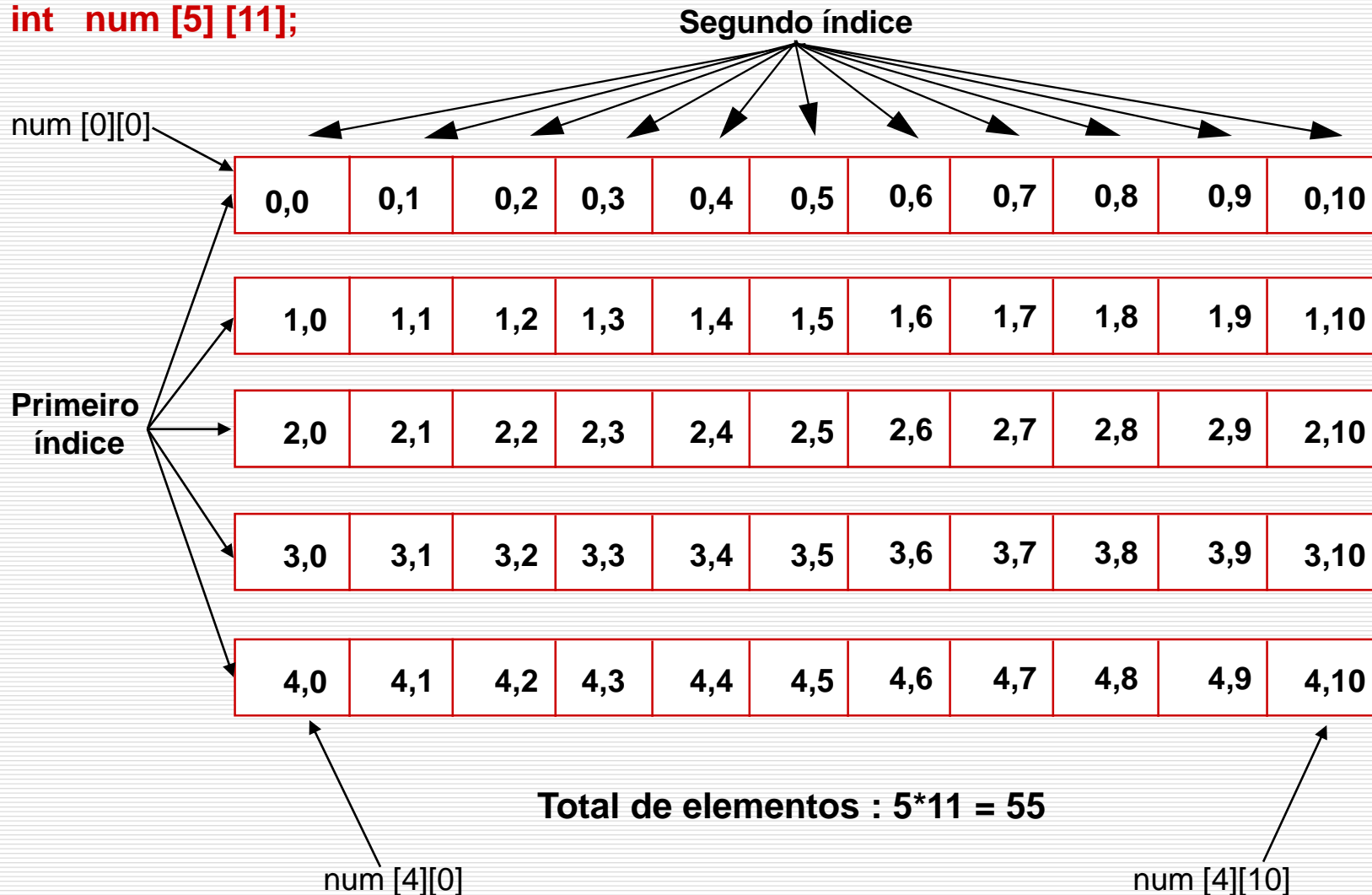
Arrays bidimensionais

Matrizes

utad

Programação procedimental

```
int num [5] [11];
```



Arrays bidimensionais

Matrizes

utad

Programação procedimental

□ Em C

<tipo> <nome> [nº de linhas, nº de colunas]

Exemplos

- `float v[3][4];`
- `Char u[6][6];`

Numa estrutura deste tipo, os dados podem ser carregados utilizando uma estrutura de ciclos encaixados

□ Utilizando notação algorítmica

```
Repetir para  $i \leftarrow 1$  até  $n_{linhas}$ 
  Repetir para  $j \leftarrow 1$  até  $n_{colunas}$ 
    Aceitar( $A[i, j]$ )
  fim repetir
fim repetir
```

- A variável i é usada para índice das linhas
- A variável j é usada para índice das colunas

Arrays bidimensionais

Matrizes

utad

Programação procedimental

Apresenta no ecrã os valores e as respectivas posições da matriz **A** (codificação em **C**)

```
:  
float A[10][20];  
:  
for (i=0; i<10; i++) /* 10 é o número de linhas */  
{  
    for (j=0; j<20; j++) /* 20 é o número de colunas */  
    {  
        printf("A[%d][%d]=", i, j);  
        scanf("%f ", &A[i][j]);  
    }  
    printf("\n");  
}  
:
```

Arrays bidimensionais

Matrizes

utad

Programação procedimental

- ❑ Existem aplicações onde se pode utilizar estruturas tridimensionais, que são representadas por conjuntos de 3 índices.
- ❑ Embora seja possível ter conjuntos com mais de 3 índices, é difícil ilustrá-los com exemplos reais, sendo apenas utilizados em aplicações muito específicas.

Arrays bidimensionais

Exercícios

utad

Programação procedimental

1. Implementar um programa em '**C**', utilizando uma **matriz**, que resolva o exercício proposto no slide 68.

... fazer um programa para calcular a média de uma lista de 10 números e o desvio de cada número em relação à média ...

2. Carregar a tabela da tabuada num *array* bidimensional

Arrays bidimensionais

Exercício 1

utad

Programação procedimental

```
main()
{
    int    i;
    float  x[2][10]; // lista de números e desvios em relação à média
    float  soma, media;
    printf("Introduz 10 números:\n");
    for(soma=0, i=0; i<10; i++ ) // introdução dos números e cálculo da média
    {
        printf("X[0][%i] =", i);
        scanf("%f", &x[0][i] );
        soma = soma + x[0][i];
    }
    media = soma / 10;
    for( i=0; i<10; i++ ) // calculo dos desvios
        x[1][i] = x[0][i] - media;
    // escreve a média, os números, e os respectivos desvios
    printf("A média é %f\n Valor de X\t\t desvio", media );
    for( i=0; i<10; i++ ){
        printf("\n");
        for(int j=0; j<2; j++ ){
            printf("%f\t\t", x[j][i]);
        }
    }
    return 0;
}
```

Pesquisa e ordenação com vectores

utad

Programação procedimental

- ❑ **Ordenar ou classificar** um vector é a operação de arranjar os seus elementos numa ordem sequencial, utilizando um critério de ordenação.
- ❑ **Pesquisar** um vector consiste basicamente em percorrer um conjunto de elementos a fim de localizar um item desejado.
- ❑ Existem vários métodos de pesquisa e ordenação vamos apenas abordar dois deles.

Ordenação ou classificação

utad

Programação procedimental

Vamos supor que temos um vector de N elementos e pretendemos ordená-lo por ordem ascendente.

Vamos estudar a classificação pelos métodos por:

- Selecção (*selection sort*)
- Bolha (*bubble sort*)
- Inserção sequencial

Ordenação por selecção

utad

Programação procedimental

O algoritmo de ordenação/classificação por selecção consiste nos seguintes passos:

- 1) Iniciando com o primeiro elemento do vector faz-se uma pesquisa para localizar o elemento com menor valor;
- 2) Este elemento é permutado com o primeiro elemento do vector (esta troca coloca o elemento com o menor valor na primeira posição do vector);
- 3) Executa-se uma pesquisa para o segundo menor valor, isto é feito pelo exame dos valores de cada elemento a partir da segunda posição;
- 4) O elemento encontrado é permutado com o localizado na segunda posição do vector;

Ordenação por selecção

utad

Programação procedimental

5) Repetem-se os passos 3) e 4) até que todos os elementos tenham sido classificados em ordem ascendente.

- Se o vector V possui N elementos são necessárias $N-1$ passagens para o ordenar. A última passagem não é necessária.
- O algoritmo da próxima transparência ordena um vector V de N elementos por ordem ascendente utilizando este algoritmo.

Ordenação por selecção

utad

Programação procedimental

```

inicio algoritmo
variáveis
    V[N] do tipo vector de reais
    indice_min, passo, l do tipo inteiro

    repetir para passo ← 1 até N-1    [Executa N-1 passagens]
        indice_min ← passo            [Inicializa o índice do mínimo]
                                        [Executa uma passagem e obtém o
                                        índice do elemento com menor valor]
        repetir para l ← passo+1 até N
            se V[l] < V[indice_min] então
                indice_min ← l
            fim se
        fim repetir

        se indice_min ≠ passo então
            Troca(V[passo], V[indice_min])
        fim se

    fim repetir
fim algoritmo
  
```

Ordenação por selecção

utad

Programação procedimental

```
#include <stdio.h>
#define N 10
void Troca(float *, float*);
void CarregaVector(int, float *);
void VisualizaVector(int, float *);
void SelectionSort(int, float *);

int main()
{
    float V[N];
    /*carrega o vector V com os valores introduzidos pelo utilizador*/
    CarregaVector(N, V);

    /*Ordena o vector V de forma crescente pelo método de selecção*/
    SelectionSort(N, V);

    printf("\n");
    /*Visualiza os valores do vector V*/
    VisualizaVector(N, V);
    return 0;
}
```

Ordenação por bolha

utad

Programação procedimental

- Ao contrário do que acontece com o método de classificação por selecção, em que primeiro se encontra o elemento com menor valor e só depois se efectua a troca, neste método, dois elementos são trocados imediatamente após a descoberta de que eles estão fora de ordem. São requeridas, no máximo, $N-1$ passagens.
- O funcionamento do método é o seguinte:
 - Durante a primeira passagem , $V[1]$ e $V[2]$ são comparados e se estão fora de ordem, são permutados;

Ordenação por bolha

utad

Programação procedimental

- Isto é repetido para os pares de elementos $V[2]$ — $V[3]$, (...), $V[i]$ — $V[i+1]$ (O método leva os elementos com menor valor a "*borbulharem para cima*"). Após a primeira passagem, o elemento com maior valor ocupará a posição N .
 - Em cada passagem sucessiva, os elementos com os valores imediatamente mais baixos serão colocados na posição $N-1$, $N-2$, $N-3$, ..., 3 , 2 , 1 .
 - É feita uma verificação para ver se houve alguma troca de valores durante cada passagem. Se não houve permutas durante a passagem, o vector encontra-se classificado, não sendo necessário mais passagens.
- O algoritmo seguinte traduz este método.

Ordenação por bolha

utad

Programação procedimental

```
inicio algoritmo
variáveis
    V[N] do tipo vector de reais
    flag do tipo lógico
    i,j do tipo inteiro
i ← 1
flag ← verdade
repetir enquanto i<N e flag=verdade
    flag ← falso
    repetir para j ← 1 até N-i
        se V[j+1]<V[j] então
            Troca(V[j+1],V[j])
            flag ← verdade
        fim se
    fim repetir
    i ← i+1
fim repetir
fim algoritmo
```

Ordenação por bolha

utad

Programação procedimental

```
#include <stdio.h>
#define N 10
void Troca(float *, float*);
void CarregaVector(int, float *);
void VisualizaVector(int, float *);
void BubbleSort(int, float *);

int main()
{
    float V[N];
    /*carrega o vector V com os valores introduzidos pelo utilizador*/
    CarregaVector(N, V);

    /*Ordena o vector V de forma crescente pelo método de selecção*/
    BubbleSort(N, V);

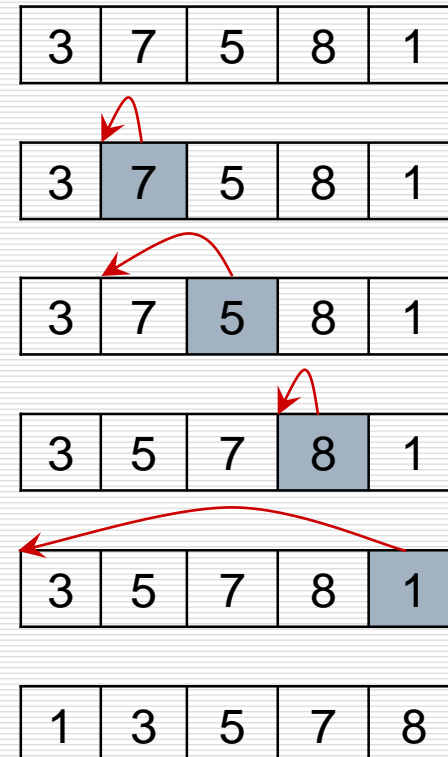
    printf("\n");
    /*Visualiza os valores do vector V*/
    VisualizaVector(N, V);
    return 0;
}
```

Inserção sequencial

utad

Programação procedimental

Se um sub-vector (formado pelos primeiros elementos do vector dado) já estiver ordenado, basta inserir ordenadamente o elemento seguinte no sub-vector.



Inserção sequencial – código C

utad

Programação procedimental

```
/* dado o Vector V[1..n], ordena-lo por ordem crescente */
/* O sub-vector V[1..1] está ordenado */

void OrdenaInsercao(int n, float *V)
{
    int i,k;
    float x;

    for(i=0;i<n-1;i++){
        if(V[i+1]<V[i]){ /*inserir V[i+1] no sub-vector ordenado V[1..i]*/
            k = i;
            x = V[i+1];
            while((k>=0) && (V[k]>x)){ /*procurar lugar em V[1..i]*/
                V[k+1] = V[k];
                k--;
            }
            V[k+1] = x;
        }
    }
}

/* resultado: vector V[1..n] ordenado por ordem crescente */
```

Exercícios

utad

Programação procedimental

Implementar os subprogramas seguintes

```
void CarregaVector(int, float*)
```

```
void VisualizaVector(int, float*)
```

```
void BubbleSort(int, float*)
```

```
void SelectionSort(int, float*)
```

```
void Troca(float*, float*)
```

Pesquisa

utad

Programação procedimental

- Iremos abordar duas técnicas de pesquisa:
 - linear
 - binária
- A pesquisa linear envolve a verificação sequencial de cada elemento do vector até que o elemento desejado seja encontrado.

Dado um vector V não ordenado com N elementos ($N \geq 1$) o próximo algoritmo procura no vector um elemento particular cujo valor é x .

Pesquisa linear

utad

Programação procedimental

```
Inicio algoritmo
Variáveis
    V[N] do tipo vector de inteiros
    flag do tipo lógico
    i,x do tipo inteiro
i ← 1
flag ← falso
Repetir enquanto i ≤ N e flag=falso
    Se V[i]=x então
        flag ← verdade
    fim se
    i ← i+1
fim repetir
Se flag = verdade então
    Visualizar('Valor encontrado na posição', i-1)
Senão
    Visualizar('Valor não encontrado')
fim se
fim algoritmo
```

Pesquisa binária

utad

Programação procedimental

- Vamos assumir que:
 - os N elementos de um vector V estão armazenados por ordem ascendente.
- Algoritmo:
 - 1) O meio do vector pode ser calculado e o seu valor é examinado.
 - 2) Se o valor for muito alto, então outro elemento do meio da primeira metade é examinado;
Repete-se 1) até que o valor desejado seja encontrado.
 - 3) Se o valor é muito baixo então um elemento médio da segunda metade é tentado e o procedimento termina quando for encontrado o elemento pretendido ou o intervalo de pesquisa seja vazio.

Dado um vector V de N elementos ordenado por ordem crescente, este algoritmo pesquisa o vector para um dado elemento x .

Pesquisa binária

utad

Programação procedimental

```
Variáveis
  V[N] do tipo vector de reais
  flag do tipo booleano
  alto, medio, baixo do tipo inteiro
baixo ← 1
alto ← n
flag ← falso
Repetir enquanto baixo < alto e flag=falso
  medio ← (alto-baixo)/2 + baixo
  Se x < V[medio] então
    alto ← medio -1
  Senão
    Se x > V[medio] então
      baixo← medio+1
    Senão
      flag ← verdade
    fim se
  fim se
fim repetir
```

Pesquisa binária

utad

Programação procedimental

```
Se baixo=alto e x=v[baixo] então
    flag <- verdade
    medio <- baixo
fim se
Se flag=verdade então
    Visualizar('Pesquisa bem sucedida na posição', medio)
Senão
    Visualizar('Pesquisa mal sucedida')
fim se
fim algoritmo
```

Exercícios

utad

Programação procedimental

Implementar os subprogramas seguintes

```
void PesquisaLinear(int, float*, float)
```

```
void PesquisaBinaria(int, float*, float)
```


Ficheiros de Texto

utad

Programação procedimental

Um ficheiro de texto é uma sequência de caracteres que ficam armazenados no disco do computador. Esta sequência de caracteres termina com um carácter especial denominado fim-de-ficheiro, **EOF** do inglês *End-Of-File*.

Para criar um ficheiro de texto usamos um editor de texto, como por exemplo o “Bloco de Notas” do Windows.

Em linguagem de programação C, podemos programar o computador para ler e escrever em ficheiros de texto, em vez de ler do teclado e escrever no ecrã. Tem a vantagem de guardar os dados para uma próxima utilização.

Ficheiros de Texto - Abrir ficheiros

utad

Programação procedimental

Em linguagem de programação **C**, existem algumas funções que permitem abrir um ficheiro, são exemplo disso as funções, definidas em <stdio.h>:

```
int _open( const char *filename, int oflag [, int pmode] );
```

Filename – nome do ficheiro

Oflag – tipo de operações que são permitidas

Pmode – modo de permissão

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

Filename – nome do ficheiro

Pmode – tipo de acesso permitido

A função **fopen** tem 2 argumentos, o 1º é o nome do ficheiro e o 2º indica se o ficheiro vai ser aberto para leitura ou escrita ("**r**" ou "**w**" respectivamente. Estes nomes vêm do inglês, *r-read*, *w-write*).

Depois de abrir o ficheiro deve-se testar se a operação foi bem sucedida. Para tal, compara-se o valor retornado com a constante **NULL** que está definida em stdio.h. Caso haja um erro, podemos escrever no ecrã uma mensagem de erro e terminar o programa.

Ficheiros de Texto

utad

Programação procedimental

Leitura e Escrita

Se for utilizada a função `_open` para abrir os ficheiros, para ler e escrever, podemos usar, as funções `read` e `write`.

`int _read(int handle, void *buffer, unsigned int count);`

`int _write(int handle, const void *buffer, unsigned int count);`

handle – variável associada ao ficheiro

buffer – array de dados

count – número de *bytes*

Retorna o nº de bytes transferidos com sucesso (lidos ou escritos) ou 0 se encontrou o fim do ficheiro ou -1 se ocorreu um erro.

Exemplo:

```
if( (fh = _open( "read.c", _O_RDONLY )) == -1 ) {
    perror( "Problemas com a abertura do ficheiro" );
    exit( 1 );
}
if( ( bytesread = _read( fh, buffer, nbytes ) ) <= 0 )
    perror( "Problemas com a leitura do ficheiro"
```

Ficheiros de Texto

utad

Programação procedimental

Leitura e Escrita

Se for utilizada a função *fopen* para abrir os ficheiros, para ler e escrever em ficheiros de texto, podemos usar, as funções *fscanf* e *fprintf*, entre outras. Estas funções têm um funcionamento e uma sintaxe muito semelhantes ao *scanf* e *printf*. A única diferença é que as funções têm um argumento adicional que indica o ficheiro em que se pretende ler ou escrever.

```
fscanf( f, "%d", &n );
```

A variável *f* é uma variável que tem de ser associada previamente a um ficheiro, e que se declara como sendo do tipo `FILE *`.

Outras funções de leitura e escrita:

```
size_t fread( void *buffer, size_t size, size_t count, FILE *stream );
```

```
size_t fwrite( const void *buffer, size_t size, size_t count, FILE *stream );
```

Ver detalhes no *help*
do Visual Studio

Ficheiros de Texto

utad

Programação procedimental

Outras funções

```
int fclose( FILE *stream ); /* fecha o ficheiro */
```

```
int feof( FILE *stream ); /* testa o fim do ficheiro */
```

```
int fgetc( FILE *stream ); /* lê um caracter do ficheiro */
```

```
char *fgets( char *string, int n, FILE *stream ); /* lê uma string do ficheiro */
```

```
int fputc( int c, FILE *stream ); /* escreve um caracter no ficheiro */
```

```
int fputs( const char *string, FILE *stream ); /* escreve uma string no ficheiro */
```

```
int fseek( FILE *stream, long offset, int origin ); /* Move o apontador (stream) para uma  
posição específica do ficheiro */
```

O exemplo que se segue é um programa que lê dois números de um ficheiro chamado "dados_ent.txt" e escreve o quadrado desses números num ficheiro novo chamado "dados_saida.txt"

Ficheiros de Texto

utad

Programação procedimental

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    FILE *f1, *f2;
    int a, b;
    /* abre o ficheiro dados_ent.txt no modo de leitura e associa-o a f1 */
    if(( f1 = fopen( "dados_ent.txt", "r" )) == NULL ){
        printf("ERRO: não é possível abrir o ficheiro dados_ent.txt\n");
        exit(1);
    }
    /* abre o ficheiro dados_saida.txt no modo de escrita e associa-o a f2 */
    if(( f2 = fopen( "dados_saida.txt", "w" ))== NULL ){
        printf("ERRO: não é possível abrir o ficheiro dados_saida.txt\n");
        exit(1);
    }
    fscanf( f1, "%d", &a ); /* leitura e escrita */
    fscanf( f1, "%d", &b );
    fprintf( f2, "Este ficheiro foi alterado na aula N° ... .\n" );
    fprintf( f2, "%d\n", a*a );
    fprintf( f2, "%d\n", b*b );
    fclose( f1 ); /* fecha o ficheiro associado a f1 */
    fclose( f2 ); /* fecha o ficheiro associado a f2*/
}
```

Estruturas de dados

utad

Programação procedimental

Definição de novos tipos de dados

Até agora vimos que podemos declarar variáveis do tipo inteiro (int), real (float ou double) e character (char). E depois podemos declarar vectores (arrays) de variáveis de um determinado tipo.

Além dos 4 tipos mais comuns, int, float, double e char, podemos definir novos tipos de dados, como já foi referido anteriormente.

Estrutura de dados

Uma estrutura de dados é caracterizada por uma organização coerente dos dados. A organização apropriada dos dados numa estrutura de dados pode tornar um problema complicado numa solução simples.

Estruturas de dados

utad

Programação procedimental

A declaração de uma estrutura em C

```
struct _nome_  
{  
    tipo1 nome1;  
    tipo2 nome2;  
    ...  
}
```

} Campos da estrutura

Exemplo:

```
struct atleta  
{  
    char nome[50];  
    int idade;  
    int altura;  
    int numero;  
    ...  
}
```

} Campos da estrutura

Estruturas de dados

utad

Programação procedimental

Declaração de variáveis aquando da definição da estrutura

```
struct nome
{
    campo1;
    campo2;
    ...
}var1, var2, ...;
```

Declaração numa fase posterior à declaração

```
struct nome var;
```

Exemplo:

```
struct atleta atl;
```

Estruturas de dados

utad

Programação procedimental

Definição de um tipo aquando da definição da estrutura

```
typedef struct nome
{
    campo1;
    campo2;
    ...
} NomeTipo;
```

Declaração de variáveis com recurso ao tipo definido:

```
NomeTipo var;
```

Exemplo:

```
typedef struct atleta
{
    char nome[50];
    int idade;
    int altura;
    int numero;
} Atletas;
```

Declaração de
variáveis com recurso
ao novo tipo

```
Atletas atl;
```

Estruturas de dados

utad

Programação procedimental

Referenciar os campos de uma estrutura de dados

Tendo sido declarada uma variável:

```
struct nome var;
```

Os campos são referenciados da seguinte:

```
var.campo1; // var é uma variável "normal"  
var->campo1; // var é um apontador
```

Exemplo:

```
Atletas atl;  
atl.idade = 22;
```

```
Atletas *atl; // necessita de reservar memória dinamicamente  
atl->idade = 22;
```

Vectorres de estruturas de dados

utad

Programação procedimental

Considerando a estrutura de dados atleta e a definição do novo tipo *Atletas*

```
typedef struct atleta
{
    char nome[50];
    int idade;
    int altura;
    int numero;
}Atletas
```

Um array de estruturas do atleta pode ser declarado da seguinte forma:

```
struct atleta vatl[N]; //usando o nome da estrutura
ou
Atletas vatl[N];      // usando o novo tipo
```

Onde:

<i>vatl</i>	– é o nome do vector
<i>N</i>	– é o número de elementos

Vector de estruturas de dados

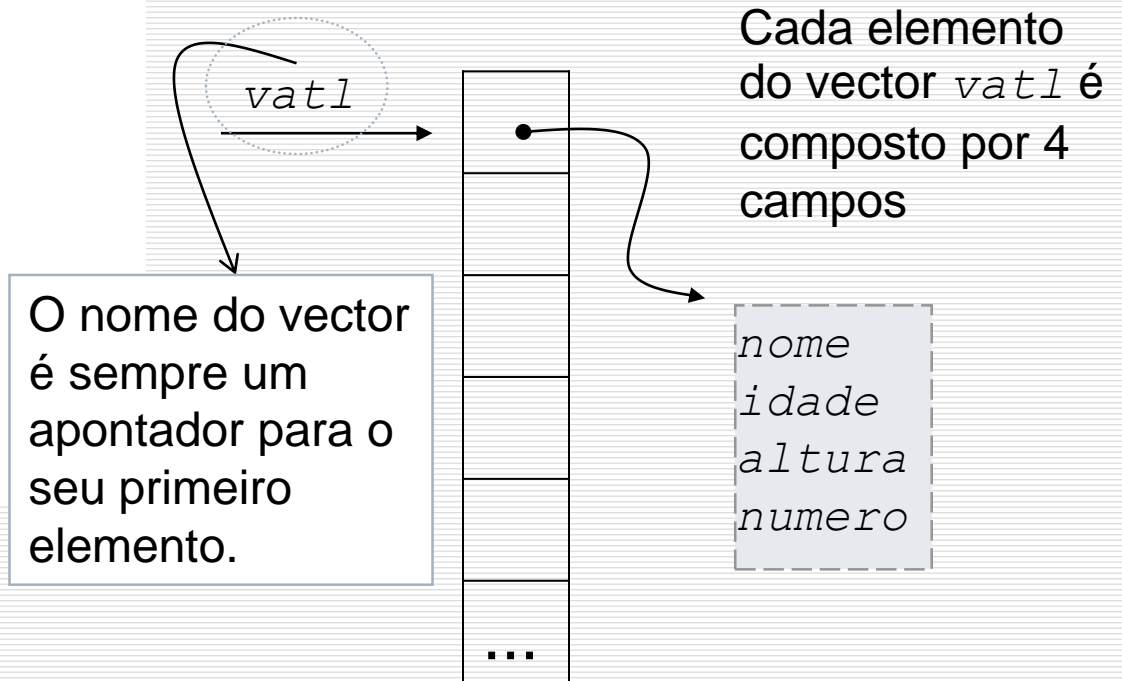
utad

Programação procedimental

Considerando a estrutura de dados atleta e a definição do novo tipo *Atletas*

```
typedef struct atleta  
{  
    char nome[50];  
    int idade;  
    int altura;  
    int numero;  
}Atletas
```

Atletas *vat1*[N];



Vectores de estruturas de dados

utad

Programação procedimental

Considerando a estrutura de dados atleta e a definição do novo tipo *Atletas*

```
typedef struct atleta
{
    char nome[50];
    int idade;
    int altura;
    int numero;
}Atletas
```

Acesso aos campos de um array de estruturas:

```
Atletas vatl[N];
```

Acesso ao 1º elemento do vector de estruturas atleta

```
vatl[0].nome = "António";
vatl[0].idade = 18;
vatl[0].altura = 185;
vatl[0].numero = 7;
```

Acesso ao 2º elemento do vector de estruturas atleta

```
vatl[1].nome = "António";
vatl[1].idade = 18;
```

...

Estruturas de dados (exemplo de revisão)

utad

Programação procedimental

Definição do novo tipo de dados ponto

Em C, o **tipo ponto** ponto é definido do seguinte modo:

```
typedef struct {  
    float x;  
    float y;  
} ponto;
```

Esta declaração indica que um ponto tem 2 campos, x e y que são dois números reais.

Declaração de variáveis do tipo ponto:

```
ponto p;
```

A variável **p** é uma estrutura.

Estruturas de dados (exemplo de revisão)

utad

Programação procedimental

Acesso aos campos da estrutura ponto

Uma estrutura é uma colecção de uma ou mais variáveis agrupadas num único nome. No exemplo anterior, *x* e *y* dizem-se campos da estrutura. Um campo de uma estrutura é acedido da seguinte forma:

nome_da_estrutura . nome_do_campo

Através da variável *p* podemos aceder aos seus membros (campos) *x* e *y*.

Por exemplo:

```
p.x = 3;
```

```
p.y = 2;
```


Estruturas de dados encadeadas

utad

Programação procedimental

Mais exemplos de estruturas: pontos, triângulos e círculos

Uma vez definido o tipo ponto, pode-se definir outros tipos fazendo uso do tipo ponto. Por exemplo, pode criar-se o tipo de dados triângulo e círculo. Um triângulo é definido por 3 pontos. Um círculo é definido pelo centro (um ponto) e pelo raio (um número real).

```
typedef struct P      typedef struct T      typedef struct C
{
    float x;           ponto a;             ponto centro;
    float y;           ponto b;             float raio;
} ponto;              ponto c;             } circulo;
                      } triangulo;
```

Estruturas de dados encadeadas

utad

Programação procedimental

```
typedef struct P      typedef struct T      typedef struct C
{
    float x;           ponto a;             ponto centro;
    float y;           ponto b;             float raio;
} ponto;              ponto c;
                       } triangulo;         } circulo;
```

Acesso aos campos em estruturas de dados encadeadas

```
circulo c;           //declaração da variável c
//acesso ao campo x do campo centro da estrutura circulo
c.centro.x = 1;
//acesso ao campo x do campo centro da estrutura circulo
c.centro.y = 2;
c.raio = 5;          //acesso ao campo raio do circulo
```

Estruturas de dados encadeadas

utad

Programação procedimental

```
typedef struct P
{
    float x;
    float y;
} ponto;

typedef struct T
{
    ponto a;
    ponto b;
    ponto c;
} triangulo;

typedef struct C
{
    ponto centro;
    float raio;
} circulo;
```

Declaração de vetores (arrays) de estruturas de dados encaixadas e acesso aos respectivos campos.

```
triangulo t[20]; // vector t com 20 elementos do tipo triângulo
circulo c[10];   // vector c com 10 elementos do tipo circulo
// acesso aos campos ponto a do 1º elemento do vector de triângulos t
    t[0].a.x = 10
    t[0].a.y = 5;
// acesso aos campos ponto centro do 2º elemento do vector circulos c
    c[1].centro.x = 7;
    c[1].centro.y = 7;
```

Estruturas de dados encadeadas

utad

Programação procedimental

```
typedef struct P
{
    float x;
    float y;
} ponto;

typedef struct T
{
    ponto a;
    ponto b;
    ponto c;
} triangulo;

typedef struct C
{
    ponto centro;
    float raio;
} circulo;
```

Implementação, em linguagem C, de um programa de ilustração das 3 estruturas de dados apresentadas e definição dos novos tipos *ponto*, *triangulo* e *circulo*.

Estruturas de dados encadeadas

utad

Programação procedimental

Programa exemplificativo do uso de estruturas

```
#include <stdio.h>

typedef struct P
{
    float x;
    float y;
} ponto;

typedef struct T
{
    ponto a;
    ponto b;
    ponto c;
} triangulo;

typedef struct C
{
    ponto centro;
    float raio;
} circulo;

void CarregaCirculo(circulo *cir);
void ApresentaCirculo(circulo cir);
```

```
void main()
{
    circulo C;

    CarregaCirculo (&C);
    ApresentaCirculo (C);
}

void CarregaCirculo (circulo *cir)
{
    printf("\nintroduza as coordenadas do centro");
    printf("\nx=");
    scanf("%f",&(cir->centro.x));
    printf("\ny=");
    scanf("%f",&(cir->centro.y));
    printf("\nintroduza o raio");
    printf("\nraio=");
    scanf("%f",&(cir->raio));
}

void ApresentaCirculo (circulo cir)
{
    printf("\n\ncordenadas do centro do circulo");
    printf("\nx = %f",cir.centro.x);
    printf("\ny = %f",cir.centro.y);
    printf("\n\nraio do circulo");
    printf("\nraio = %f\n",cir.raio);
}
```

Estruturas de dados - memória dinâmica

utad


Programação procedimental

Consideremos novamente a estrutura *Atleta*

Versão 1 da estrutura Atleta

```
typedef struct atleta
{
    char nome[50];
    int idade;
    int altura;
    int numero;
}Atletas
```


Vector de
caracteres estático



Versão 2 da estrutura Atleta

```
typedef struct atleta
{
    char *nome;
    int idade;
    int altura;
    int numero;
}Atletas
```

Apontador para caracter (necessita
de alocação de memória dinâmica)



Estruturas de dados - memória dinâmica

utad

Programação procedimental

Consideremos a estrutura *Atletas* – **versão 1**

Consideremos também um apontador *a1* para estrutura *Atletas*

```
Atletas *a1;
```

Alocação dinâmica de memória de um vector de estruturas de dados complexas, numa primeira abordagem, é em tudo idêntica a alocação dinâmica de memória de vectores do tipo *int* ou *char* (estruturas de dados simples) já estudados anteriormente, fazendo uso das funções *malloc* e *free*.

A alocação dinâmica de memória de um vector de estruturas *Atleta* de *N* elementos seria implementada da seguinte forma:

```
a1 = (Atletas *) malloc( N * sizeof(Atletas) );
```

apontador devolvido

número de elementos

tamanho de cada elemento em bytes

Estruturas de dados - memória dinâmica

utad

Programação procedimental

Consideremos a estrutura *Atletas* – **versão 2**

Consideremos também um apontador *a1* para estrutura *Atletas*

```
Atletas *a1;
```

```
a1 = (Atletas *) malloc( N * sizeof(Atletas) );
```

apontador devolvido

número de elementos

tamanho de cada elemento em bytes

Alocação de memória dinamicamente para o campo nome da estrutura *Atletas*

```
for(i=0; i<N; i++)
```

```
    a1[i].nome = (char *) malloc(50 * sizeof(char));
```


Estruturas de dados - memória dinâmica

utad

Programação procedimental

Consideremos também um apontador *a1* para estrutura *Atletas* – **versão 1**

```
Atletas *a1;
```

```
free(a1); // liberta a memória alocada para a estrutura
```

Consideremos também um apontador *a1* para estrutura *Atletas* – **versão 2**

```
Atletas *a1;
```

```
(Em primeiro lugar liberta a memória reservada para os  
campos da estrutura)
```

```
for(i=0; i<N; i++)
```

```
    free(a1[i].nome); //liberta a memória alocada para o campo nome
```

```
free(a1); // liberta a memória alocada para a estrutura
```

Listas ligadas

utad

Programação procedimental

Uma lista ligada não é mais que um conjunto de estruturas de dados, são uma forma de armazenar ou organizar informação.

Este armazenamento é independente do tipo de dados.

Cada elemento da lista é constituído por duas zona: a primeira é de armazenamento de dados e a segunda é um apontador para o próximo elemento (no caso de listas simplesmente ligadas).

Os elementos de uma lista podem ser manipulados individualmente sem afectar os restantes elementos, como por exemplo: retirar, eliminar ou alterar elementos.

Listas ligadas

utad

Programação procedimental

- A relação entre elementos é estabelecida através de apontadores.
- O nome da lista é um apontador para o primeiro elemento.
- Ao primeiro elemento da lista dá-se o nome de “cabeça da lista” (*head*).
- Ao último elemento da lista dá-se o nome de “cauda da lista” (*tail*).
- Uma lista diz-se vazia quando o nome da lista aponta para *NULL*;
- Uma lista é percorrida de uma forma sequencial passando de elemento em elemento, tendo início no primeiro elemento e fim no último elemento (quando o apontador para o elemento seguinte = *NULL*).

Listas simplesmente ligadas

utad

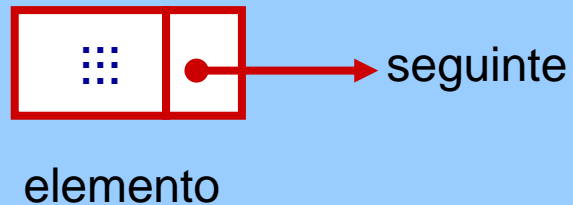
Programação procedimental

Numa lista simplesmente ligada cada elemento tem apenas uma ligação (apontador) com o elemento seguinte da lista.

Só é possível percorrer a lista num sentido, a partir do primeiro elemento (head) até ao ultimo (tail).

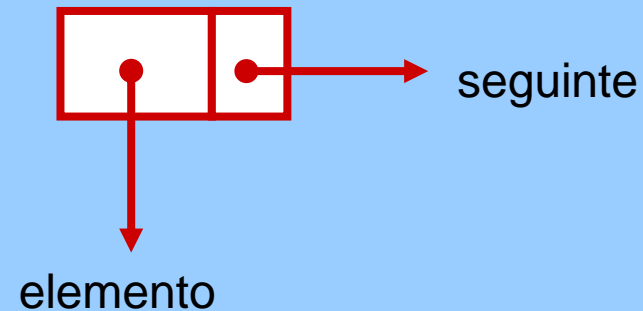
versão 1

nó



versão 2

nó



Listas simplesmente ligadas

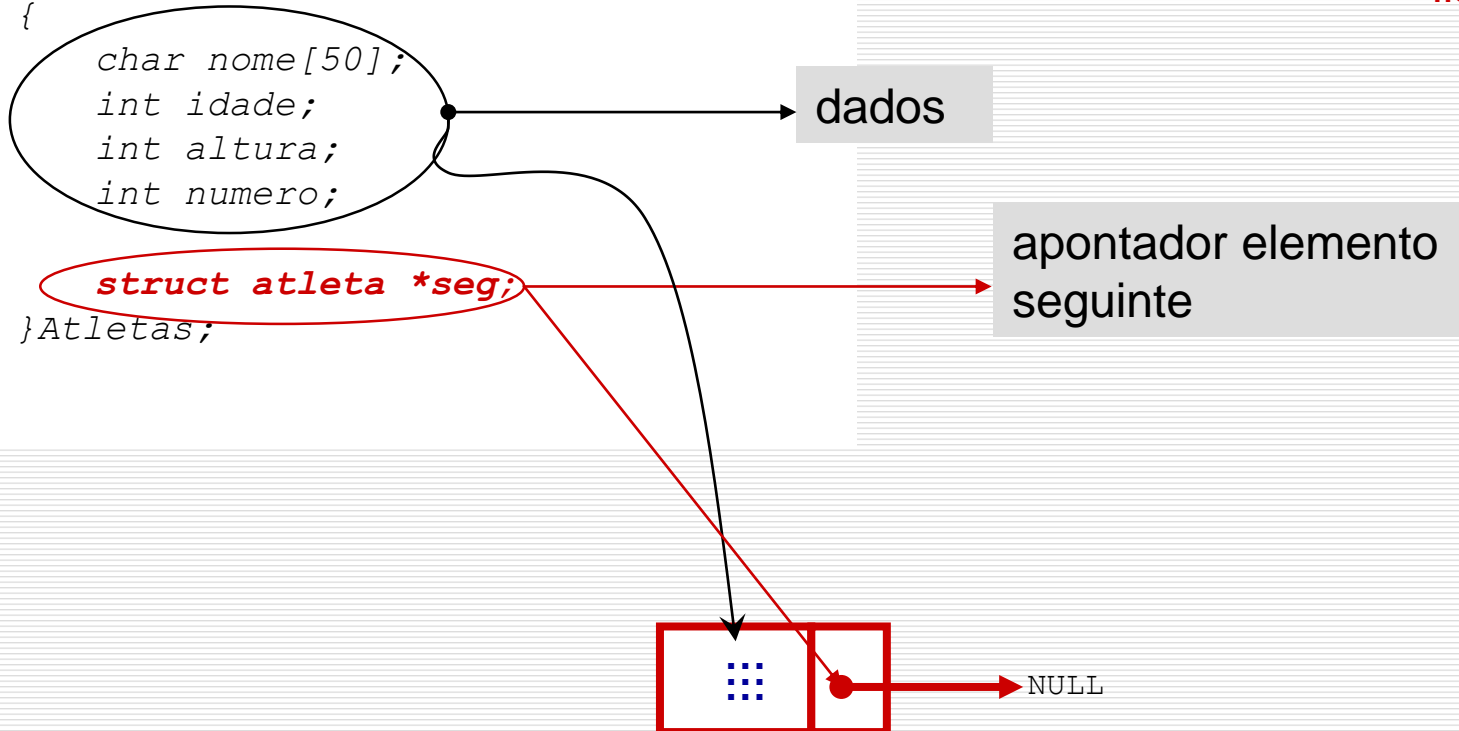
utad

Programação procedimental

Consideremos a estrutura *Atletas*

```
typedef struct atleta
{
    char nome[50];
    int idade;
    int altura;
    int numero;
    struct atleta *seg;
}Atletas;
```

lista versão 1



Listas simplesmente ligadas

utad

Programação procedimental

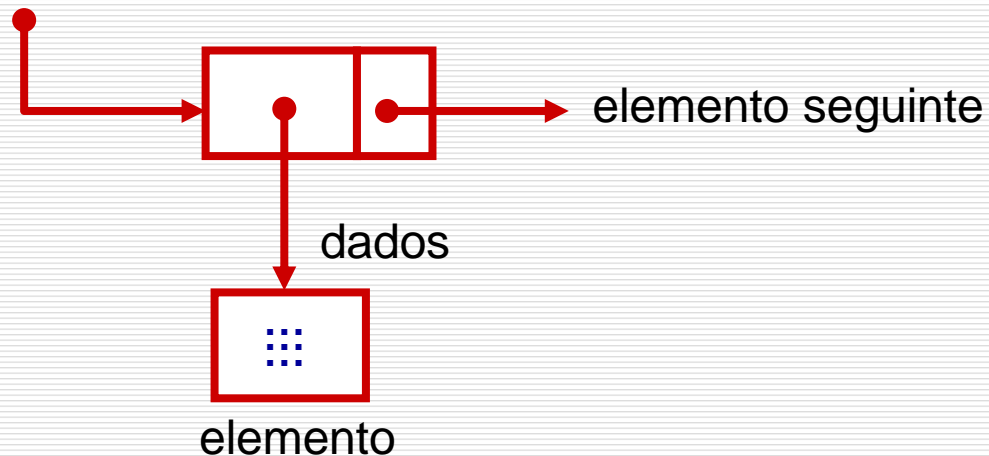
Consideremos a estrutura *Atletas*

```
typedef struct NO
{
    struct NO *seg;
    struct atleta *dados;
}no;
```

```
typedef struct atleta
{
    char nome[50];
    int idade;
    int altura;
    int numero;

}Atletas;
```

início da lista



lista versão 2

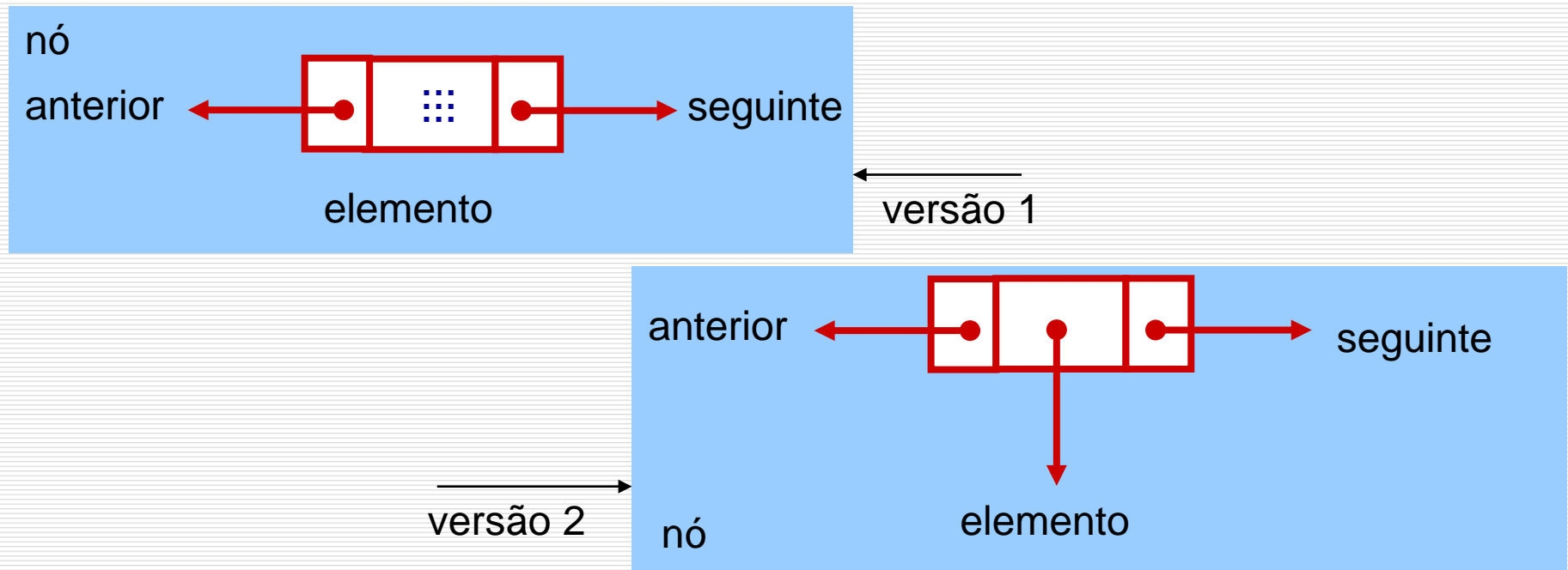
Listas duplamente ligadas

utad

Programação procedimental

Numa lista simplesmente ligada cada elemento tem uma ligação (apontador) com o elemento seguinte da lista e uma ligação com o elemento anterior da lista.

É possível percorrer a lista nos dois sentidos, a partir do primeiro elemento (head) até ao último (tail) ou a partir do último elemento até ao primeiro.



Listas duplamente ligadas

utad

Programação procedimental

Consideremos a estrutura *Atletas*

```
typedef struct atleta
```

```
{  
    char nome[50];  
    int idade;  
    int altura;  
    int numero;  
}
```

```
    struct atleta *seg;  
    struct atleta *ant;  
}Atletas;
```

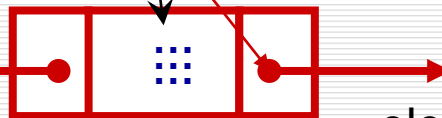
lista versão 1

dados

apontador elemento
seguinteapontador elemento
anterior

elemento anterior

elemento seguinte



Listas duplamente ligadas

utad

Programação procedimental

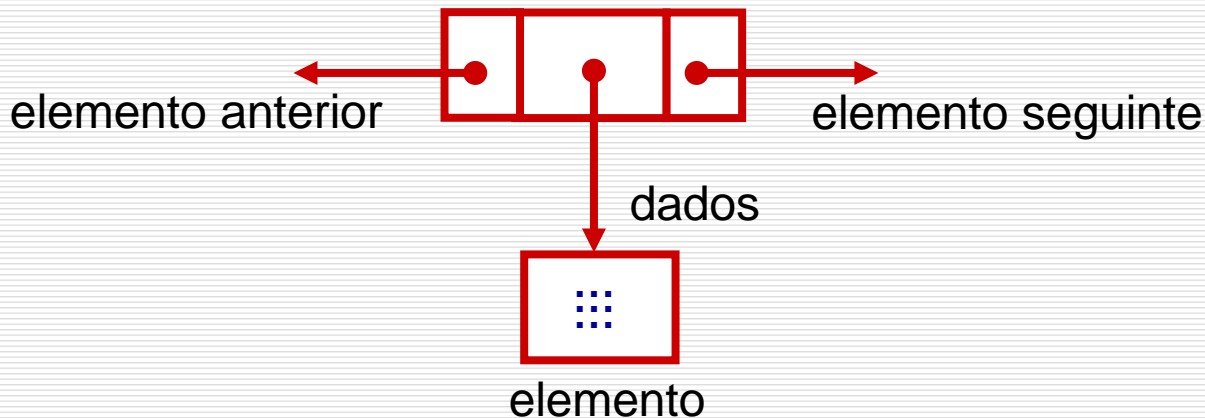
Consideremos a estrutura *Atletas*

```
typedef struct NO
{
    struct NO *seg;
    struct NO *ant;

    struct atleta *dados;
}no;
```

```
typedef struct atleta
{
    char nome[50];
    int idade;
    int altura;
    int numero;

}Atletas;
```



lista versão 2

Listas ligadas

utad

Programação procedimental

Conjunto de funções (ferramentas) para manipular uma lista simplesmente ligada:

1. inserir elemento na lista (início, meio, fim);
2. remover elemento da lista (início, meio, fim);
3. percorrer a lista;
4. libertar a lista;
5. ordenar a lista;
6. pesquisar um dado na lista.

Listas ligadas

utad

Programação procedimental

Consideremos a seguinte estrutura *Atletas* e o Subprograma “*NewNode()*”

```
typedef struct atleta
{
    char nome[50];
    int idade;
    int altura;
    int numero;
    struct atleta *next;
}LIST_NODE,LIST
```

```
LIST_NODE * NewNode()
{
    LIST_NODE * new_node;
    if((new_node = (LIST_NODE *)malloc(sizeof(LIST_NODE)))==NULL)
        return NULL;
    new_node->next = NULL;
    new_node->idade = var++;
    ...
    return new_node;
}
```

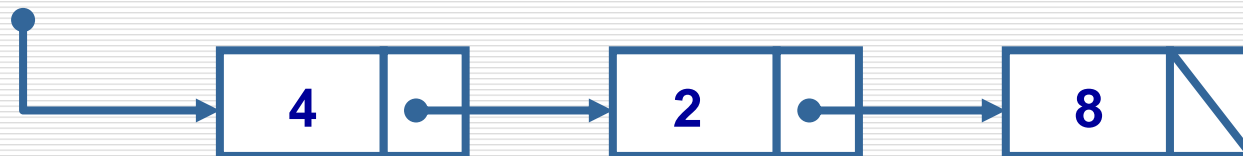
Inserir novo elemento no início da lista

utad

Programação procedimental

```
LIST_NODE * InsertIni(LIST *list)
{
    LIST_NODE * new_node;
    if ((new_node = NewNode()) != NULL)
    {
        new_node->next = list;
        list = new_node;
    }
    return(new_node);
}
```

*list



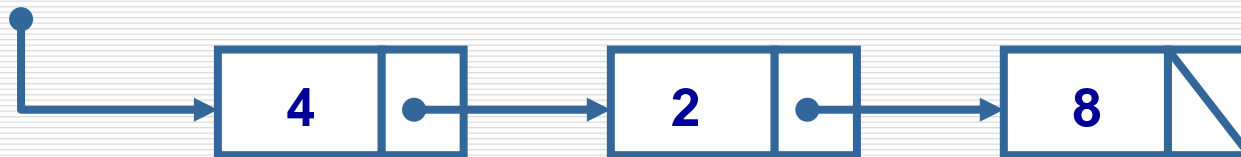
Inserir novo elemento no início da lista

utad

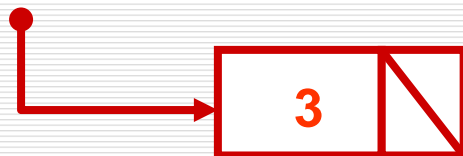
Programação procedimental

```
LIST_NODE * InsertIni(LIST *list)
{
    LIST_NODE * new_node;
    if ((new_node = NewNode()) != NULL)
    {
        new_node->next = list;
        list = new_node;
    }
    return(new_node);
}
```

*list



new_node



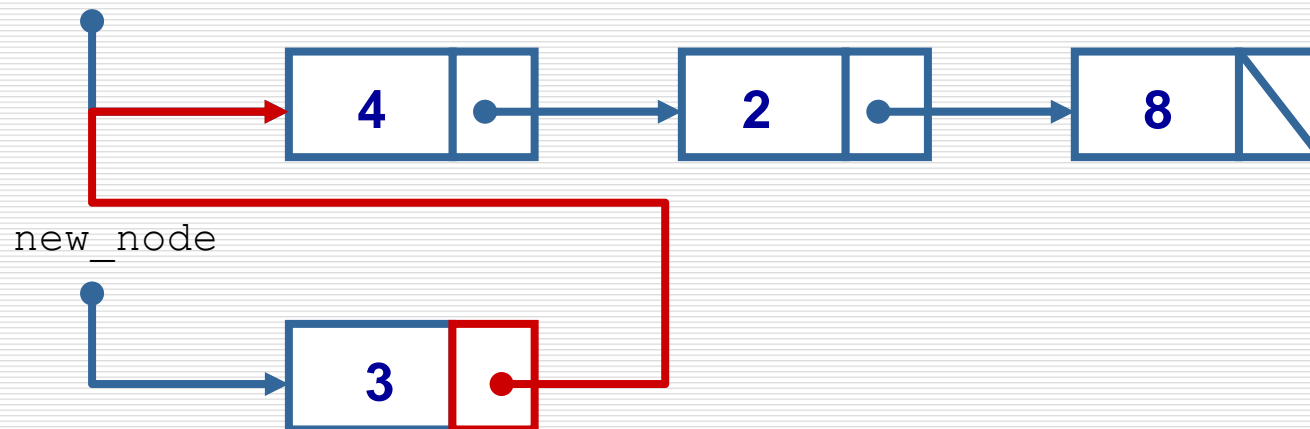
Inserir novo elemento no início da lista

utad

Programação procedimental

```
LIST_NODE * InsertIni(LIST *list)
{
    LIST_NODE * new_node;
    if ((new_node = NewNode()) != NULL)
    {
        new_node->next = list;
        list = new_node;
    }
    return(new_node);
}
```

*list

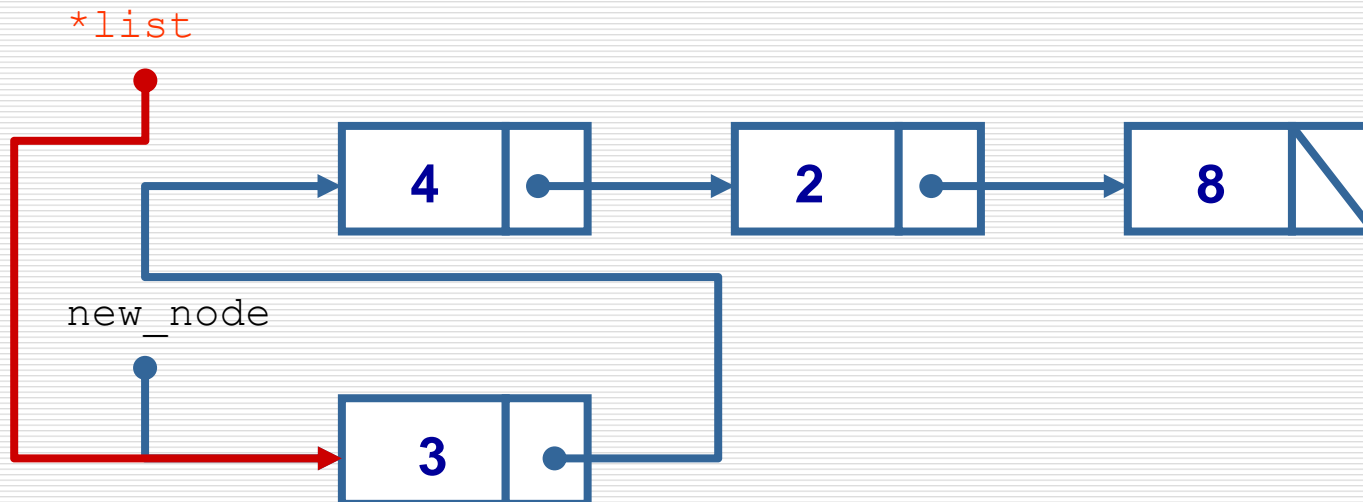


Inserir novo elemento no início da lista

utad

Programação procedimental

```
LIST_NODE * InsertIni(LIST *list)
{
    LIST_NODE * new_node;
    if ((new_node = NewNode()) != NULL)
    {
        new_node->next = list;
        list = new_node;
    }
    return(new_node);
}
```



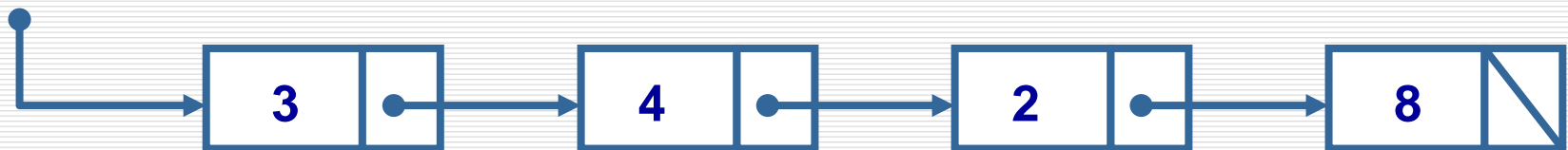
Inserir novo elemento no início da lista

utad

Programação procedimental

```
LIST_NODE * InsertIni(LIST *list)
{
    LIST_NODE * new_node;
    if ((new_node = NewNode()) != NULL)
    {
        new_node->next = list;
        list = new_node;
    }
    return(new_node);
}
```

new_node
*list



Inserir novo elemento no início da lista

utad

Programação procedimental

Exemplo, codificado em C, utilizando a lista Atleta:

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

typedef struct atleta
{
    char nome[50];
    int idade;
    int altura;
    int numero;

    struct atleta *next;
} LIST_NODE, LIST;

LIST_NODE * InsertIni(LIST *list);
LIST_NODE * NEXT(LIST_NODE *new_node);
LIST_NODE * NewNode();

int var = 20;
```

```
void main()
{
    LIST *lst=NULL;

    lst = InsertIni(lst);
    lst = InsertIni(lst);
    lst = InsertIni(lst);
}
```

Inserir novo elemento no início da lista

utad

Programação procedimental

```
LIST_NODE * NewNode()
{
    LIST_NODE * new_node;
    if((new_node = (LIST_NODE *)malloc(sizeof(LIST_NODE)))==NULL))
        return NULL;
    new_node->next = NULL;
    new_node->idade = var++;
    ...
    return new_node;
}

LIST_NODE * InsertIni(LIST *list)
{
    LIST_NODE * new_node;
    if ((new_node = NewNode()) != NULL)
    {
        new_node->next = list;
        list = new_node;
    }
    return(new_node);
}
```

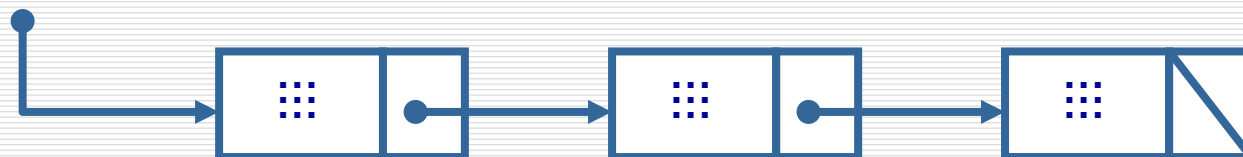
Inserir novo elemento no fim da lista

utad

Programação procedimental

```
LIST_NODE * InsertValueEnd(LIST *list)
{
    LIST_NODE * temp, * new_node = NULL;
    if ((new_node = NewNode()) != NULL)
        if (*list == NULL) list = new_node;
        else
        {
            temp = list;
            while (temp->next != NULL)
                temp = temp->next;
            NEXT(temp) = new_node;
        }
    return(list);
}
```

*list



Inserir novo elemento no fim da lista

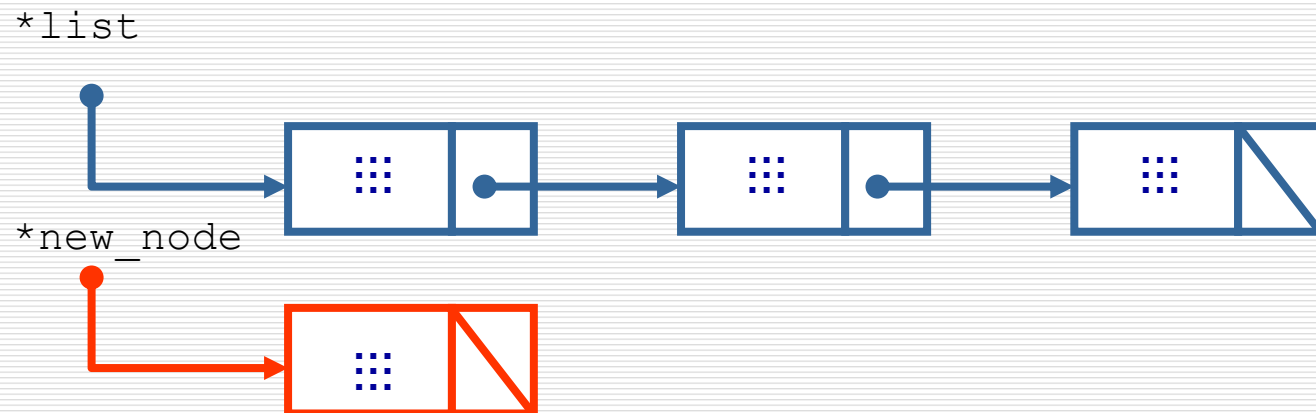
utad

Programação procedimental

```

LIST_NODE * InsertValueEnd(LIST *list)
{
    LIST_NODE * temp, * new_node = NULL;
    if ((new_node = NewNode()) != NULL)
        if (list == NULL) list = new_node;
        else
        {
            temp = list;
            while (temp->next != NULL)
                temp = temp->next;
            temp->next = new_node;
        }
    return(list);
}

```



Inserir novo elemento no fim da lista

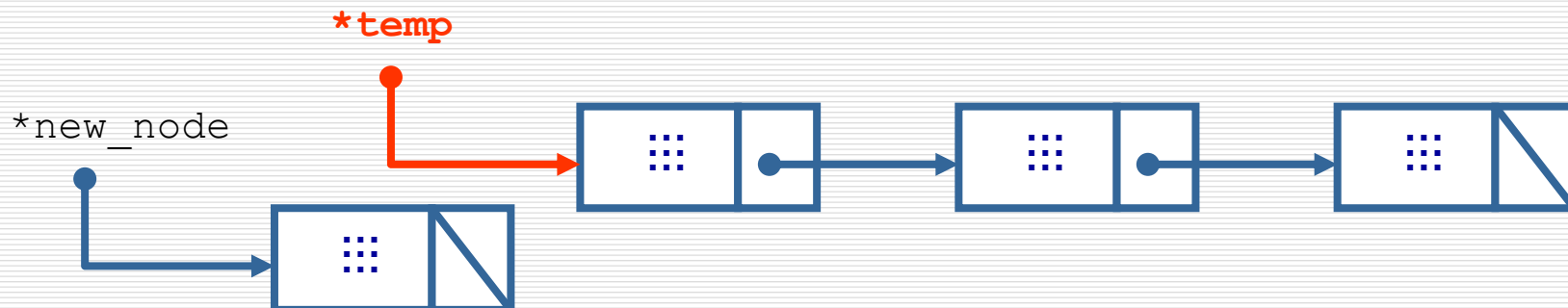
utad

Programação procedimental

```

LIST_NODE * InsertValueEnd(LIST *list)
{
    LIST_NODE * temp, * new_node = NULL;
    if ((new_node = NewNode()) != NULL)
        if (list == NULL) list = new_node;
        else
        {
            temp = list;
            while (temp->next != NULL)
                temp = temp->next;
            temp->next = new_node;
        }
    return(list);
}

```



Inserir novo elemento no fim da lista

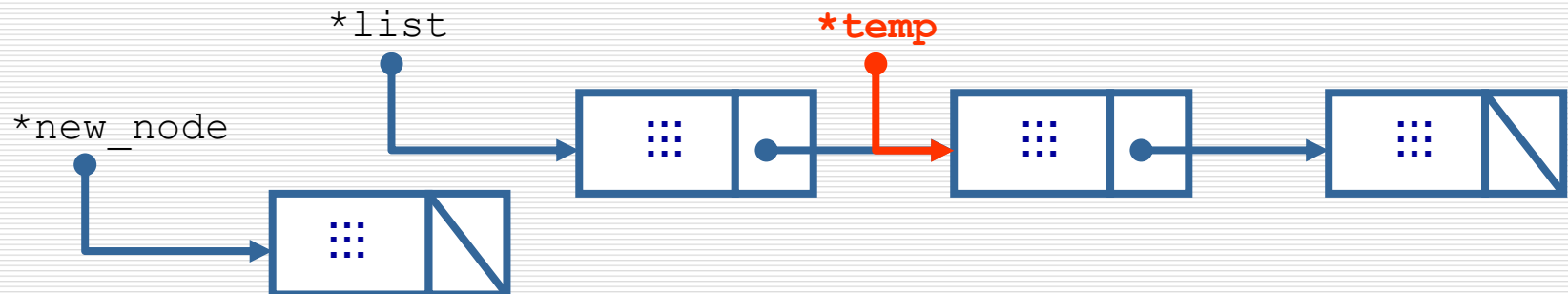
utad

Programação procedimental

```

LIST_NODE * InsertValueEnd(LIST *list)
{
    LIST_NODE * temp, * new_node = NULL;
    if ((new_node = NewNode()) != NULL)
        if (list == NULL) list = new_node;
        else
        {
            temp = list;
            while (temp->next != NULL)
                temp = temp->next;
            tem->next = new_node;
        }
    return(list);
}

```



Inserir novo elemento no fim da lista

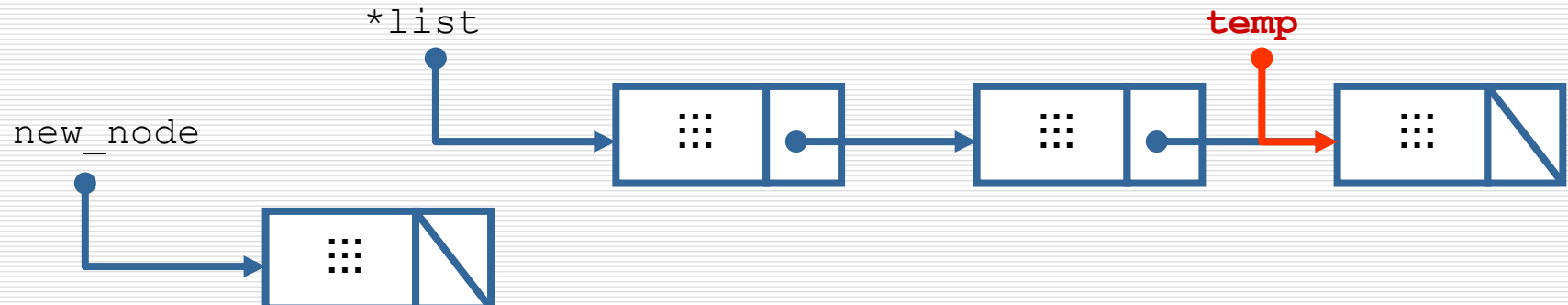
utad

Programação procedimental

```

LIST_NODE * InsertValueEnd(LIST *list)
{
    LIST_NODE * temp, * new_node = NULL;
    if ((new_node = NewNode()) != NULL)
        if (*list == NULL) *list = new_node;
        else
        {
            temp = list;
            while (temp->next != NULL)
                temp = temp->next;
            temp->next = new_node;
        }
    return(list);
}

```



Inserir novo elemento no fim da lista

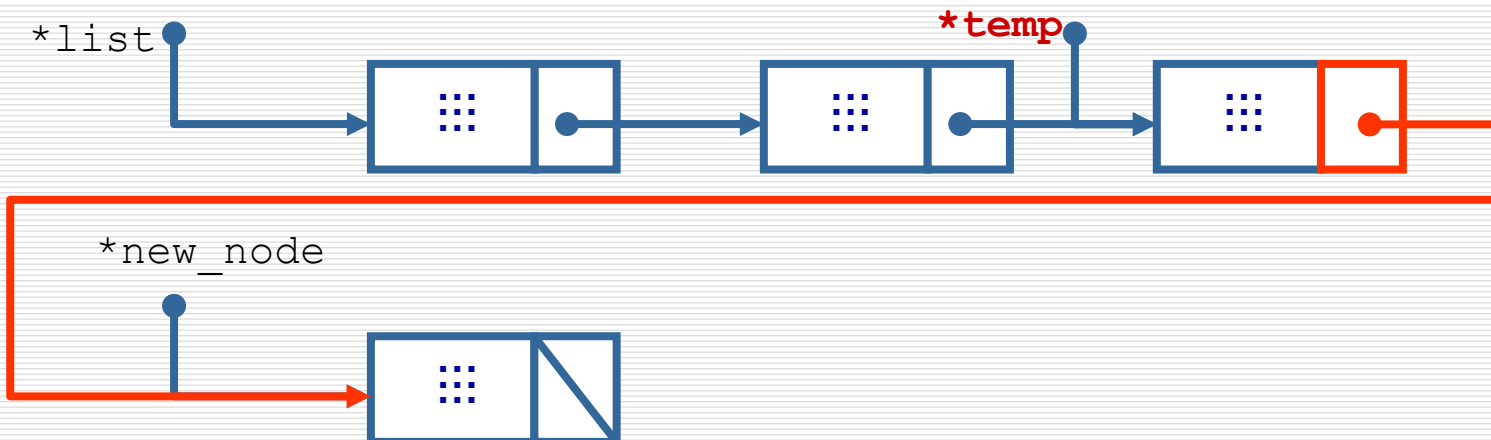
utad

Programação procedimental

```

LIST_NODE * InsertValueEnd(LIST *list)
{
    LIST_NODE * temp, * new_node = NULL;
    if ((new_node = NewNode()) != NULL)
        if (list == NULL) list = new_node;
        else
        {
            temp = list;
            while (temp->next != NULL)
                temp = temp->next;
            temp->next = new_node;
        }
    return(list);
}

```



Inserir novo elemento no fim da lista

utad

Programação procedimental

```

LIST_NODE * InsertValueEnd(LIST *list)
{
    LIST_NODE * temp, * new_node = NULL;
    if ((new_node = NewNode()) != NULL)
        if (list == NULL) list = new_node;
        else
        {
            temp = list;
            while (temp->next != NULL)
                temp = temp->next;
            temp->next = new_node;
        }
    return(list);
}

```

