

Aula 7

- Introdução à utilização de ponteiros em linguagem C
- Acesso sequencial aos elementos de um *array*:
 - acesso indexado
 - acesso com ponteiros
- Tradução para *assembly* do MIPS

Bernardo Cunha, José Luís Azevedo, Arnaldo Oliveira

Introdução

- Em linguagem C, em muitas situações é necessário saber qual o endereço de memória onde reside uma variável, para então aceder ao seu conteúdo
- Por exemplo, para imprimir os caracteres de uma *string* (array de caracteres) no ecrã é necessário ler sequencialmente cada uma das posições de memória onde a *string* se encontra alojada (que são identificadas pelo seu endereço)
- O acesso a cada um dos caracteres é feito indiretamente através do endereço onde residem; conhecido o endereço inicial da *string* em memória, o acesso sequencial é garantido pelo incremento sucessivo do endereço
- A linguagem C providencia um mecanismo para acesso a variáveis residentes na memória externa através da utilização de ponteiros

Linguagem C: ponteiros e endereços – o operador &

- Um **ponteiro** é uma **variável que contém o endereço de outra variável** – o acesso à 2ª variável pode fazer-se indiretamente através do ponteiro
- Se **var** é uma variável, então **&var** dá-nos o seu endereço
- Exemplo:
 - **x** é uma variável (por ex. um inteiro) e **px** é um ponteiro. O **endereço da variável x** pode ser obtido através do **operador &**, do seguinte modo:

```
px = &x; // Atribui o endereço de "x" a "px"
```
 - Diz-se que **px é um ponteiro que aponta para x**
- O operador **&** apenas pode ser utilizado com variáveis e elementos de *arrays*.
 - Exemplos de utilizações **erradas**:

```
&5;      &(x+1);
```

Ponteiros e endereços – o operador *

- O operador "*":
 - trata o seu operando como um endereço
 - permite o acesso ao endereço para obter o respetivo conteúdo

- Exemplo:

```
y = *px;    // Atribui o conteúdo da variável  
            // apontada por "px" a "y"
```

- A sequência:

```
px = &x;    // px é um ponteiro para x
```

```
y = *px;    // *px é o valor de x
```

Atribui a **y** o mesmo valor que: **y = x;**

Ponteiros e endereços – declaração de variáveis

- As variáveis envolvidas têm que ser declaradas
- Para o exemplo anterior, supondo que se tratava de variáveis inteiras:

```
int  x, y; // x, y - variáveis do tipo inteiro
int  *px;  // ou int* px; (ponteiro para inteiro)
           // Esta declaração apenas reserva o
           // espaço para o ponteiro, ou seja,
           // não há qualquer inicialização
```

- A declaração do ponteiro (`int *px;` ou `int* px;`) deve ser entendida como uma mnemónica e significa que **px é um ponteiro** e que o conjunto ***px é do tipo inteiro**
- Exemplos de **declarações de ponteiros**:

```
char  *p;  // p é um ponteiro para caracter
double *v; // v é um ponteiro para double
```

Manipulação de ponteiros em expressões

- Exemplo: supondo que `px` aponta para `x` (`px = &x;`), a expressão `y = *px + 1;` atribui a `y` o valor de `x` acrescido de 1
- Os ponteiros podem igualmente ser utilizados na parte esquerda de uma expressão. Por exemplo, (supondo que `px = &x;`):

```
*px = 0;           // equivalente a x=0
```

ou

```
*px = *px + 1; // equivalente a x = x + 1
```

```
*px += 1;       // o mesmo que *px = *px + 1
```

```
(*px)++;       // o mesmo que *px = *px + 1
```

Ponteiros como argumentos de funções

- Em C os argumentos das funções são passados por valor (cópia do conteúdo da variável original)
- Assim, uma função chamada não pode alterar diretamente o valor de uma variável da função chamadora
- Então, no código seguinte:

```
void change(int a)
{
    a = 10;
}

void main(void)
{
    int b = 25;
    change(b);           // Para a função é passado o valor
                        // da variável (passagem por valor)
    printf("%d", b);    // Imprime o valor de b
}
```

- Qual o valor de "b" após a chamada à função "change()"?
- A função alterou o valor da cópia da variável, pelo que, "b" mantém o valor original, ou seja, 25

Ponteiros como argumentos de funções

- Se pretendermos que a função altere o valor da variável da função chamadora, então teremos que passar como argumento da função o endereço da variável, ou seja, um ponteiro para a variável

```
void change(int *a)      // argumento de entrada é um
{                          // ponteiro para um inteiro
    *a = 10;
}

void main(void)
{
    int b = 25;
    change(&b); // Para a função é passado o endereço
                // da variável (passagem por referência)
    printf("%d", b);
}
```

- Qual o valor de "b" após a chamada à função "change()"?
- A função acedeu à variável da função chamadora através do seu endereço, pelo que "b" passa a ter o valor 10

Ponteiros e *arrays*

- Sejam as declarações

```
int a[10]; // array de inteiros "a" com
           // 10 elementos

int *pa;   // ponteiro para um inteiro

int v;     // variável do tipo inteiro
```

- A expressão `pa = &a[0];` atribui a `pa` o endereço do 1º elemento do *array*; então, a expressão `v = *pa;` atribui a `v` o valor de `a[0]`
- Se `pa` aponta para um dado elemento do *array*, `pa+1` aponta para o seguinte
- Se `pa` aponta para o primeiro elemento do *array*, então `(pa+i)` aponta para o elemento `i` e `*(pa+i)` refere-se ao seu conteúdo
- A expressão `pa = &a[0];` pode também ser escrita como `pa=a;` isto é, o nome do *array* é o endereço do seu primeiro elemento

Aritmética de Ponteiros

- Se `pa` é um ponteiro, então a expressão `pa++;` incrementa `pa` de modo a apontar para o elemento seguinte (seja qual for o tipo de variável para o qual `pa` aponta)
- Do mesmo modo `pa = pa + i;` incrementa `pa` para apontar para `i` elementos à frente do elemento atual
- **A tradução das expressões anteriores para *Assembly* tem que ter em conta o tipo de variável para o qual o ponteiro aponta**
- Por exemplo, se um inteiro for definido com 4 bytes (32 bits), então a expressão `pa++;` implica adicionar 4 ao valor atual do endereço correspondente

Exemplo 1

- Analise o código C deste e dos slides seguintes e determine o resultado produzido

```
void main(void)
{
    char s[]="Hello";
```

```
    int i = 0;
```

```
    while(s[i] != '\0')
    {
```

```
        printf("%c", s[i]); // imprime carater
                             // print_char(s[i])
```

```
        i++;
```

```
    }
}
```

H	e	l	l	o	\0
---	---	---	---	---	----

```
// "s" é um array de
// carateres (string)
// terminado com o
// carater '\0' (0x00)
```

Exemplo 2

```
void main(void)
{
    char s[] = "Hello";
    char *p;    // Declara um ponteiro para
                // carater (reserva espaço)
    p = s;      // Inicializa o ponteiro com o
                // endereço inicial do array
    while(*p != '\0')
    {
        printf("%c", *p); // imprime carater
        p++;              // incrementa o ponteiro
    }
}
```

- O ponteiro "p" é usado pelo "printf()" para aceder ao carater a imprimir (*p)
- O ponteiro é depois incrementado, i.e., fica a apontar para o carater seguinte do *array*

Exemplo 3

```
void main(void)
{
    char s[] = "Hello";
    char *p1 = s;    // p1 = &s[0]
    char *p2 = s;    // p2 = &s[0]

    while(*p2 != '\0')
        p2++;
    while(p1 < p2)
    {
        printf("%c", *p1);
        p1++;
    }
}
```

- Após o primeiro "while" o ponteiro "p2" aponta para o fim da *string* (i.e., para o carater '\0')
- O ponteiro "p1" é usado pelo "printf()" para aceder ao carater a imprimir (*p1); o ponteiro "p1" é incrementado na linha seguinte

Exemplo 4

```
void main(void)
{
    char s[] = "Hello";
    char *p = s;

    while(*p != '\0')
    {
        printf("%c", *p++);
    }
}
```

- O ponteiro "p" é usado pelo "printf()" para aceder ao carater a imprimir (*p)
- O ponteiro "p" é incrementado após o acesso ao conteúdo (pós-incremento)
- Esta construção é equivalente à do exemplo 3

Exemplo 5

```
void main(void)
{
    char s[] = "Hello";
    char *p = s;
    int i;

    for(i = 0; i < 5; i++)
    {
        printf("%c", (*p)++);
    }
}
```

- O ponteiro "p" é usado pelo "printf()" para aceder ao carater a imprimir (*p)
- A operação de incremento está a ser aplicada à variável apontada pelo ponteiro
- Neste exemplo o ponteiro "p" nunca é incrementado
- Qual a sequência de caracteres impressa?

Acesso sequencial a elementos de um *array*

- O acesso sequencial a elementos de um *array* apoia-se em uma de duas estratégias:
 1. Acesso indexado, isto é, endereçamento a partir do nome do *array* e de um índice que identifica o elemento a que se pretende aceder:
f = a[i];
 2. Utilização de um ponteiro (endereço armazenado num registo) que identifica em cada instante o endereço do elemento a que se pretende aceder:
f = *pt; // com pt = endereço de a[i] (i.e. **pt = &a[i]**)
- Estas 2 formas de acesso traduzem-se em **implementações distintas** em *assembly*

Acesso sequencial a elementos de um *array*

- **Acesso indexado**

- **$f = a[i];$** // Com $i \geq 0$
- Para aceder ao elemento "**i**" do *array* "**a**", o programa começa por calcular o respetivo endereço, a partir do endereço inicial do *array*:

endereço do elemento a aceder = endereço inicial do *array* +
(índice * dimensão em bytes de cada posição do *array*)

- **Acesso por ponteiro**

- **$f = *pt;$**
- O endereço do elemento a aceder está armazenado num registo

endereço do elemento seguinte = endereço actual +
dimensão em bytes de cada posição do *array*

Exemplos de acesso sequencial a *arrays*

```
// Exemplo 1
int i;
static int array[size];

for(i = 0; i < size; i++){
    array[i] = 0;
}
```

Acesso indexado

```
// Exemplo 2
int *p;
static int array[size];

for(p=&array[0]; p < &array[size]; p++)
{
    *p = 0;
}
```

Acesso por ponteiro

Também pode ser escrito como: `for(p=array; p < array+size; p++)`

Acesso sequencial a arrays – exemplo 1

```
#define SIZE 10
void main(void) {
    int i;
    static int array[SIZE];
    for (i = 0; i < SIZE; i++)
        array[i] = 0;
}
```

\$t0 > i
\$t1 > temp
\$t2 > &(array[0])
\$a0 > SIZE

```
.data
array: .space 40          # SIZE*4 não é suportado pelo MARS
      .eqv    SIZE, 10
.text
.globl main
main:  li      $t0, 0      # i = 0;
loop:  bge     $t0, SIZE, endf # while (i < size) {
      la      $t2, array  # $t2 = &(array[0]);
      sll     $t1, $t0, 2  # temp = i * 4;
      addu    $t1, $t2, $t1 # temp = &(array[i])
      sw      $0, 0($t1)  # array[i] = 0;
      addi    $t0, $t0, 1  # i = i + 1;
      j      loop        # }
endf:  ...
```

Acesso sequencial a arrays – exemplo 2

```

#define SIZE 10
void main(void){
    int *p;
    static int array[size];
    for (p=&array[0]; p < &array[size]; p++){
        *p = 0;
    }
}

```



```

.data
array: .space 40          #SIZE * 4      $t0 > p
      .eqv    SIZE, 10    $t1 > &(array[size])
      .text
      .globl  main        $a0 > size
main:  li     $a0, SIZE    #
      la     $t0, array    # $t0 = &(array[0]);
      sll    $t1, $a0, 2    # $t1 = size * 4;
      addu   $t1, $t1, $t0  # $t1 = &(array[size]);
loop:  bgeu   $t0, $t1, endf # while (p < &array[size]) {
      sw     $0, 0($t0)    #      *p = 0;
      addiu  $t0, $t0, 4    #      p = p + 1;
      j      loop         # }
endf:  ...

```

Questões

- O que significa a declaração `int *ac;`? Qual a diferença entre essa declaração e `int ac`?

O que significa a declaração `char *ac;`?

- A partir das declarações de `a` e `b`:

```
int a;
```

```
int *b;
```

identifique quais das seguintes atribuições são válidas:

```
a=b;      b=*a;      b=&(a+1);  a=&b;      b=&a;
b=*a+1;  b=*(a+1);  a=*b;      a=*(b+1);  a=*b+1;
```

- Identifique as operações, e respetiva sequência, realizadas nas seguintes instruções C:

```
a=*b++;  a=*(b)++;  a=*(++b);
```

- Suponha que `p` está declarado como `int *p;`. Supondo que a organização da memória é do tipo *byte-addressable*, qual o incremento no endereço que é obtido pela operação `p=p+2;` ?

Questões

- Suponha que "b" é um *array* declarado como `int b[25];`. Como é obtido o endereço inicial do *array*, i.e., o endereço da sua primeira posição? Supondo uma memória "*byte-addressable*", como é obtido o endereço do elemento "b[6]"?
- Dada a seguinte sequência de declarações:

```
int b[25];  
int a;  
int *p = b;
```

Identifique qual ou quais das seguintes atribuições permitem aceder ao elemento de índice 5 do *array* "b":

```
a = b[5];          a = *p + 5;  
a = *(p + 5);      a = *(p + 20);
```

Exercício

- Pretende-se escrever uma função para a troca do conteúdo de duas variáveis (**troca(a, b);**). Isto é, se, antes da chamada à função, a=2 e b=5, então, após a chamada à função, os valores de a e b devem ser: a=5 e b= 2

Uma solução incorreta para o problema é a seguinte:

```
void troca(int x, int y)
{
    int aux;

    aux = x;
    x = y;
    y = aux;
}
```

- Identifique o erro presente no trecho de código e faça as necessárias correções para que a função tenha o comportamento pretendido