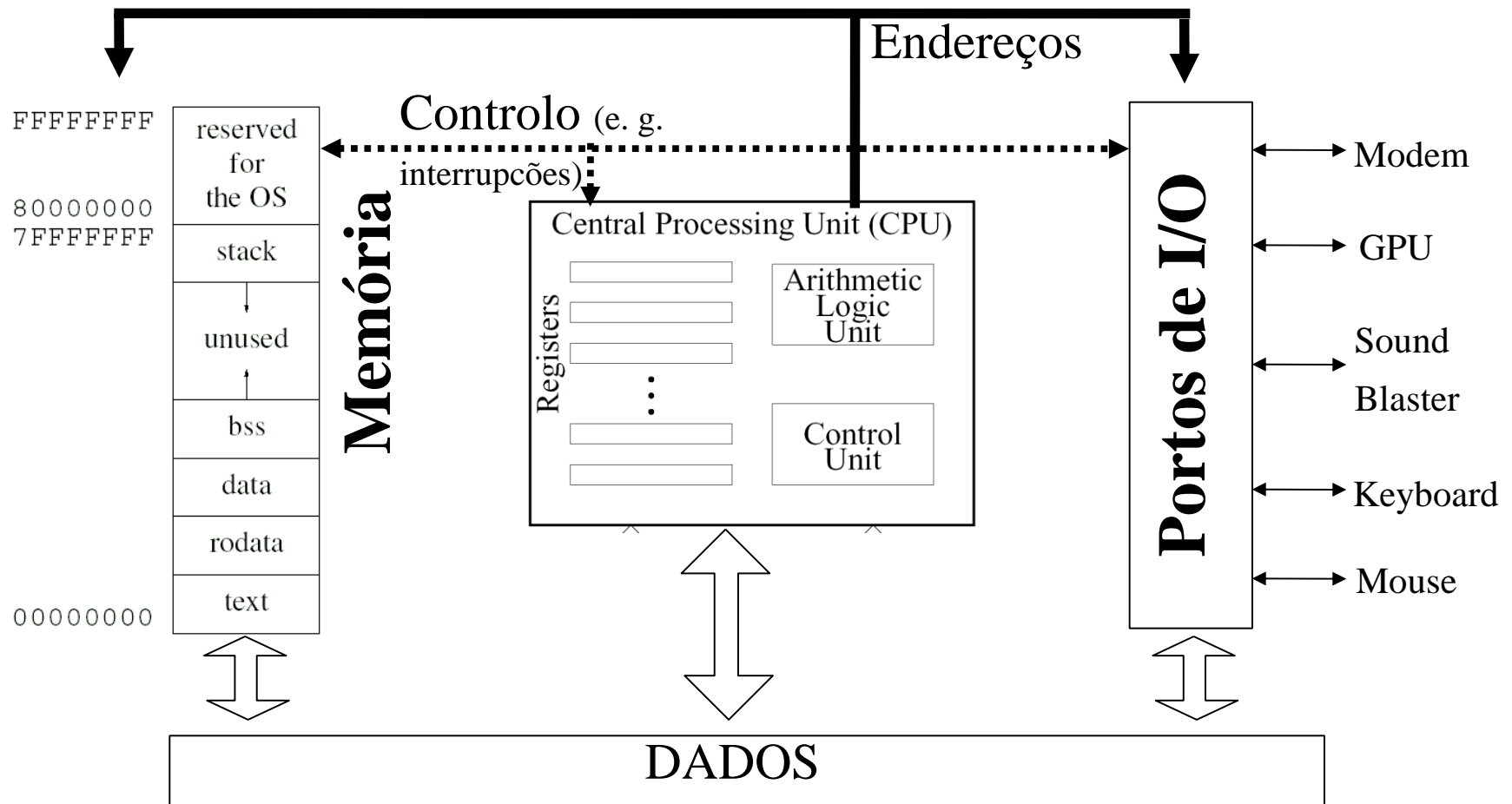


Linguagem C

- Ponteiros e Tabelas -

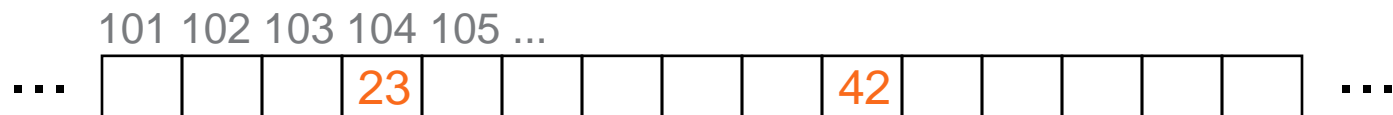
Arquitetura de Computadores 2024/2025

Anatomia de um Computador



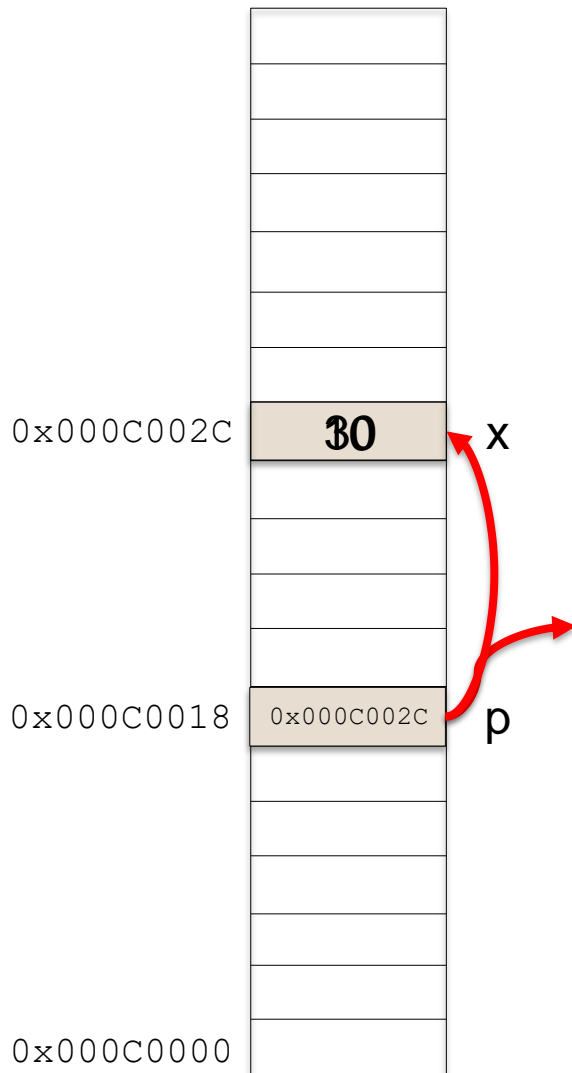
Endereço vs. Valor

- Considere a memória como sendo uma grande tabela:
 - Cada célula da tabela tem um endereço associado
 - Cada célula da tabela contém um valor
- Não confundir o endereço, que referencia uma determinada célula de memória, com o valor armazenado nessa célula de memória.
- Seria ridículo dizer que vocês e o vosso endereço de correio são a mesma coisa !





Variáveis – Conceito de Ponteiro



```
int x;
```

```
x = 10;
```

```
int *p;
```

```
*p = 30;
```

```
p = &x;
```

```
*p = 30;
```

Um endereço referencia uma determinada zona da memória. Por outras palavras, aponta para essa zona de memória.

Ponteiro: uma variável que contém um endereço de memória

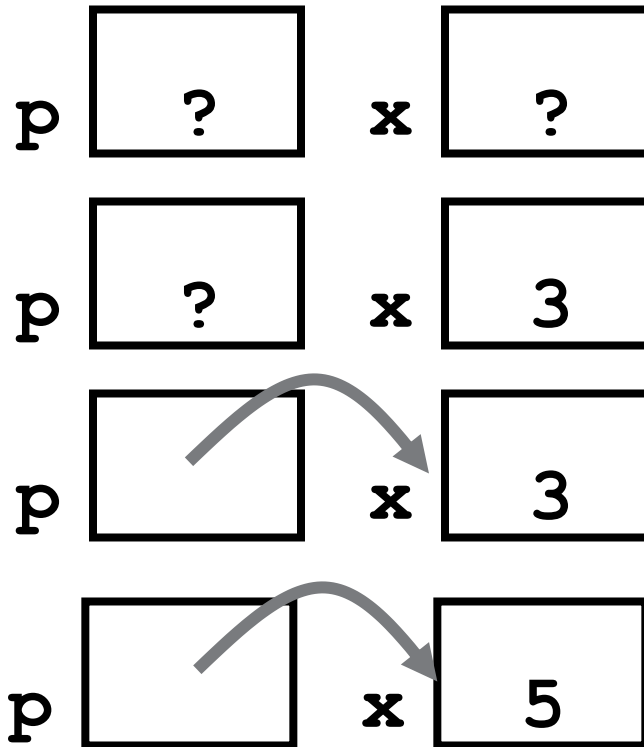
Ponteiros

```
int *p, x;
```

```
x = 3;
```

```
p = &x;
```

```
*p = 5;
```



- Operador `&` : obtém o endereço da variável
- Operador `*`: dá acesso ao valor apontado, tanto para fins de leitura, como de escrita

```
printf("p points to %d\n", *p);
```

Ponteiros e Passagem de Parâmetros

- Em C a passagem de parâmetros é sempre feita “por valor”

```
void addOne (int x) {  
    x = x + 1;  
}  
int y = 3;  
addOne(y);
```

y é ainda = 3

```
void addOne (int *p) {  
    *p = *p + 1;  
}  
int y = 3;  
  
addOne(&y);
```

y é agora = 4

Sintaxe do C: Função `main`

- Para a função `main` aceitar parâmetros de entrada passados pela linha de comando, utilize o seguinte:

```
int main (int argc, char *argv[])
```

- O que é que isto significa?
 - `argc` indica o número de strings na linha de comando (o executável conta um, mais um por cada argumento adicional).
 - Example: `unix% sort myFile`
 - `argv` é um ponteiro para um array que contém as strings da linha de comando (ver adiante).

Concluindo ...

- As declarações são feitas no início de cada função/bloco.
- Só o 0 e o NULL são avaliados como FALSO.
- Os dados estão todos em memória. Cada célula/zona de memória tem um endereço para ser referenciada e um valor armazenado. (não confundir endereço com valor).
- Um ponteiro é a "versão C" de um endereço .
 - * “segue” um ponteiro para obter o valor apontado
 - & obtém o endereço de uma variável
- Os ponteiros podem referenciar qualquer tipo de dados (`int`, `char`, uma `struct`, etc.).

Ponteiros e Alocação (1/2)

- Depois de declararmos um ponteiro:

```
int *ptr;
```

`ptr` não aponta ainda para nada (*na realidade aponta para algo ... só não sabemos o quê!*).

Podemos:

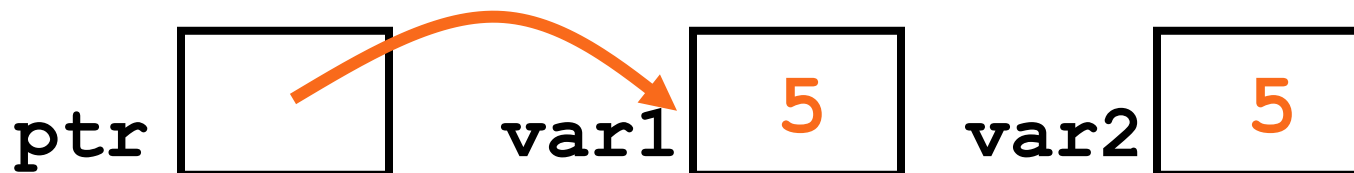
- Fazê-lo apontar para algo que já existe (operador `&`), ou
- Alocar espaço em memória e pô-lo a apontar para algo novo ... (veremos isto mais à frente)

Ponteiros & Alocação (2/2)

- Apontar algo que já existe:

```
int *ptr, var1, var2;  
var1 = 5;  
ptr = &var1;  
var2 = *ptr;
```

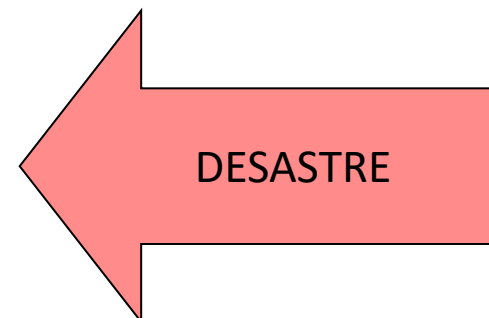
- `var1` e `var2` possuem espaço que foi implicitamente alocado (neste caso 4 bytes)



Atenção aos Ponteiros !!!

- Declarar um ponteiro somente aloca espaço para guardar um endereço de memória - não aloca nenhum espaço a ser apontado.
- **As variáveis em C não são inicializadas**, elas podem conter qualquer coisa.
- O que fará a seguinte função?

```
void f()  
{  
    int *ptr;  
    *ptr = 5;  
}
```





Tabelas/Arrays (1/5)

- Declaração:

```
int ar[2];
```

declara uma tabela de inteiros com 2 elementos. *Uma tabela/array é só um bloco de memória (neste caso de 8 bytes).*

- Declaração:

```
int ar[] = {795, 635};
```

declara e preenche uma tabela de inteiros de 2 elementos.

- Acesso a elementos:

```
ar[num];
```

devolve o $(num+1)^{\circ}$ elemento (atenção: o primeiro elemento é acedido com $num=0!$).

Tabelas/Arrays (2/5)

- As tabelas são (quase) idênticas aos ponteiros
 - `char *string` e `char string[]` são declarações muito semelhantes
 - As diferenças são subtis: incremento, declaração de preenchimento de células, etc
- **Conceito Chave:** Uma variável do tipo tabela (o "nome da tabela") é um ponteiro para o primeiro elemento..

Tabelas/Arrays (3/5)

- Consequências:
 - `ar` é uma variável tabela mas em muitos aspectos comporta-se como um ponteiro
 - `ar[0]` é o mesmo que `*ar`
 - `ar[2]` é o mesmo que `*(ar+2)`
 - Podemos utilizar aritmética de ponteiros para aceder aos elementos de uma tabela de forma mais conveniente.
- O que está errado na seguinte função?

```
char *foo() {  
    char string[32]; ...;  
    return string;  
}
```

Tabelas/Arrays (4/5)

- Tabela de dimensão n ; queremos aceder aos elementos de 0 a $n-1$, usando como teste de saída a comparação com o endereço da “célula de memória” depois do fim da tabela.

```
int ar[10], *p, *q, sum = 0;
...
p = &ar[0]; q = &ar[10];
while (p != q)
    sum += *p++; /* sum = sum + *p; p = p + 1; */
```

- O C assume que depois da tabela **continua a ser um endereço válido**, i.e., não causa um erro de *bus* ou um *segmentation fault*
- O que aconteceria se acrescentássemos a seguinte instrução?
`*q=20;`

Tabelas/Arrays (5/5)

- **Erro Frequente:** Uma tabela em C NÃO sabe a sua própria dimensão, e os seus limites não são verificados automaticamente!
 - Consequência: Podemos acidentalmente transpor os limites da tabela. *É necessário evitar isto de forma explícita*
 - Consequência: Uma função que percorra uma tabela tem de receber a variável da tabela e a respectiva dimensão
- Segmentation faults e bus errors:
 - Isto são "*runtime errors*" muito difíceis de detectar. É preciso ser cuidadoso! (Nas práticas veremos como fazer o debug usando gdb...)

Segmentation Fault vs Bus Error?

- Retirado de
<http://www.hyperdictionary.com/>
- **Bus Error**
 - A fatal failure in the execution of a machine language instruction resulting from the processor detecting an anomalous condition on its bus. Such conditions include **invalid address alignment** (accessing a multi-byte number at an odd address), accessing a physical address that does not correspond to any device, or some other device-specific hardware error. A bus error triggers a processor-level exception which Unix translates into a “SIGBUS” signal which, if not caught, will terminate the current process.
- **Segmentation Fault**
 - An error in which a running Unix program attempts to **access memory not allocated** to it and terminates with a segmentation violation error and usually a core dump.

Boas e Más Práticas

- Má Prática

```
int i, ar[10];  
for(i = 0; i < 10; i++){ ... }
```

- Boa Prática

```
#define ARRAY_SIZE 10  
int i, a[ARRAY_SIZE];  
for(i = 0; i < ARRAY_SIZE; i++){ ... }
```

- Porquê? **SINGLE SOURCE OF TRUTH**

–Evitar ter múltiplas cópias do número 10.

Aritmética de Ponteiros (1/4)

- Um ponteiro é simplesmente um endereço de memória. Podemos adicionar-lhe valores de forma a percorrermos uma tabela/array.
- $p+1$ é um ponteiro para o próximo elemento da tabela.
- $*p++$ vs $(*p)++$?
 - $x = *p++ \Rightarrow x = *p ; p = p + 1 ;$
 - $x = (*p)++ \Rightarrow x = *p ; *p = *p + 1 ;$
- O que acontece se cada célula da tabela tiver uma dimensão superior a 1 byte?
 - O C trata disto automaticamente. Na realidade $p+1$ não adiciona 1 ao endereço de memória. Adiciona antes o tamanho de cada elemento da tabela (por isso é que associamos tipos aos ponteiros).

Aritmética de Ponteiros (2/4)

- Quais são as operações válidas?
 - Adicionar inteiros a ponteiros.
 - Subtrair 2 ponteiros no mesmo array (para saber a sua distância relativa).
 - Comparar ponteiros ($<$, $<=$, $=$, $!=$, $>$, $>=$)
 - Comparar o ponteiro com `NULL` (indica que o ponteiro não aponta para nada).
- ... tudo o resto é inválido por não fazer sentido
 - Adicionar 2 ponteiros
 - Multiplicar 2 ponteiros
 - Subtrair um ponteiro de um inteiro

Aritmética de Ponteiros (3/4)

- O C sabe o tamanho daquilo que o ponteiro aponta (definido implicitamente na declaração) – assim uma adição/subtracção move o ponteiro um número adequado de bytes.
 - 1 byte para char, 4 bytes para int, etc.
- As seguintes instruções são equivalentes:

```
int get(int array[], int n)
{
    return  (array[n]);
    /* OR */
    return *(array + n);
}
```

Aritmética de Ponteiros (4/4)

- Podemos utilizar a aritmética de ponteiros para "caminhar" ao longo da memória:

```
void copy(int *from, int *to, int n) {  
    int i;  
    for (i=0; i<n; i++) {  
        *to++ = *from++;  
    }  
}
```

Representação ASCII de caracteres

◆ Os caracteres são representados através de bytes

◆ Existem várias codificações: ASCII, unicode, etc

◆ É tudo um questão de interpretação ...

```
char a='A' ;
```

```
a=a+3;
```

```
puts (&a) ;
```

O que aparece?

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	nul			0	@	P	`	p	Ç	É	á	▯	⌞	⌚	α	≡
1			!	1	A	Q	a	q	ü	æ	í	▯	⌞	⌚	β	±
2			"	2	B	R	b	r	é	Æ	ó	▯	⌞	⌚	Γ	≥
3			#	3	C	S	c	s	â	ô	ú		⌞	⌚	π	≤
4			\$	4	D	T	d	t	ä	ö	ñ	⌞	⌚	⌞	Σ	∫
5			%	5	E	U	e	u	à	ò	Ñ	⌞	⌚	⌞	σ	∫
6			&	6	F	V	f	v	â	û	z	⌞	⌚	⌞	μ	÷
7	bel		'	7	G	W	g	w	ç	ù	z	⌞	⌚	⌞	τ	≈
8	bs		(8	H	X	h	x	ê	ÿ	ı	⌞	⌚	⌞	Φ	°
9	tab)	9	I	Y	i	y	ë	Ö	ı	⌞	⌚	⌞	θ	°
A	lf		*	:	J	Z	j	z	è	Ü	ı	⌞	⌚	⌞	Ω	.
B	vt	esc	+	;	K	[k	{	ı	ı	ı	⌞	⌚	⌞	δ	√
C	ff		.	<	L	\	l		î	é	ı	⌞	⌚	⌞	∞	ⁿ
D	cr		-	=	M]	m	}	ı	ı	ı	⌞	⌚	⌞	∅	²
E			.	>	N	^	n	~	Ä	ı	ı	⌞	⌚	⌞	ε	■
F			/	?	O	_	o		Å	f	ı	⌞	⌚	⌞	n	

C Strings

- Uma *string* em C é uma tabela de caracteres.

```
char string[] = "abc";
```

- Como é que sabemos quando uma *string* termina?
 - O último carácter é seguido de um byte com o valor '`\0`' (*null terminator*)

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

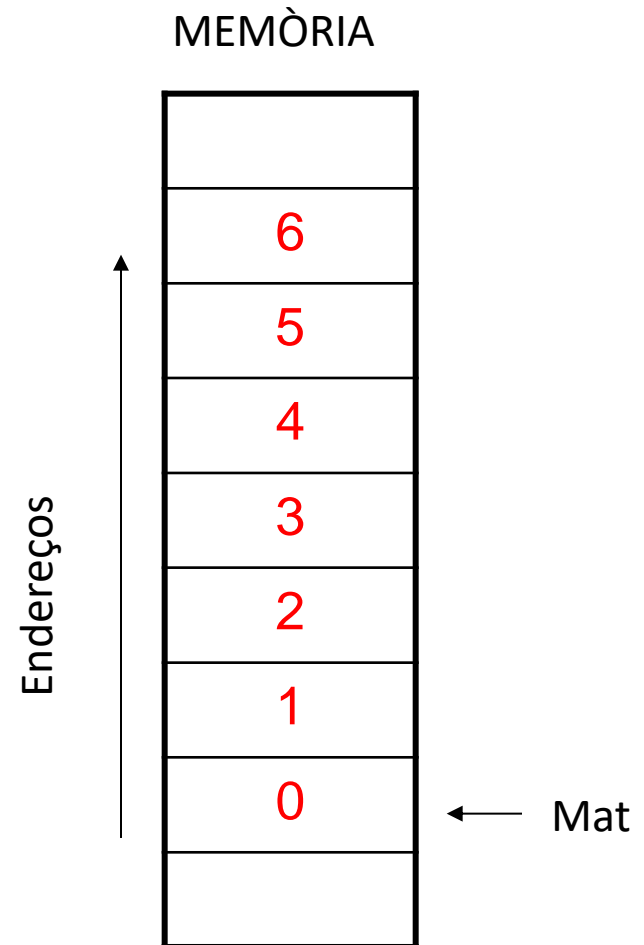
- Um erro comum é esquecermos de alocar um byte para o terminador

Tabelas bidimensionais (1/2)

```
#define ROW_SIZE 3
#define COL_SIZE 2

...
char Mat[ROW_SIZE][COL_SIZE];
char aux=0;
int i, j;
for ( i=0; i<ROW_SIZE; i++)
    for ( j=0; j<COL_SIZE; j++) {
        Mat[i][j]=aux;
        aux++;
    }
...
```

$$\text{Mat} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \\ 4 & 5 \end{pmatrix}$$



Tabelas bidimensionais (2/2)

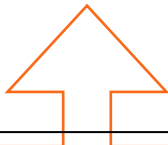
- O C arruma uma tabela bidimensional empilhando as linhas umas a seguir às outras.
- O espaço total de memória ocupado é $\text{ROW_SIZE} \times \text{COL_SIZE}$
- Sendo assim:
 $\text{Mat}[2][1]$ é o mesmo que $\text{Mat}[2 * \text{COL_SIZE} + 1]$

Tabelas vs. Ponteiros

- O nome de uma tabela é um ponteiro para o seu primeiro elemento (índice 0).
- Um parâmetro tabela pode ser declarado como um *array* ou como um ponteiro.

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

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



Pode ser escrito:
while (s[n])



QUIZ - Aritmética de Ponteiros

◆ How many of the following are invalid?

- I. pointer + integer
- II. integer + pointer
- III. pointer + pointer
- IV. pointer – integer
- V. integer – pointer
- VI. pointer – pointer
- VII. compare pointer to pointer
- VIII. compare pointer to integer
- IX. compare pointer to 0
- X. compare pointer to NULL

ptr + 1
1 + ptr
ptr + ptr
ptr - 1
1 - ptr
ptr - ptr
ptr1 == ptr2
ptr == 1
ptr == 0
ptr == NULL

#invalid
1
2
3
4
5
6
7
8
9
10

Concluindo ...

- Ponteiros e tabelas são **virtualmente o mesmo**
- O C sabe como **incrementar ponteiros**
- O C é uma linguagem eficiente com muito poucas proteções
 - Os limites das tabelas **não são verificados**
 - As variáveis **não** são automaticamente inicializadas
- (Atenção) O custo da eficiência é um "*overhead*" adicional para o programador
 - “C gives you a lot of extra rope but be careful not to hang with it!” (tirado de K&R)

Linguagem C

- Alocação Dinâmica de Memória -

Arquitetura de Computadores 2024/2025

Alocação dinâmica de memória (1/4)

- Em C existe a função `sizeof()` que dá a dimensão em bytes do tipo ou variável que é passada como parâmetro.
- Partir do princípio que conhecemos o tamanho dos objectos pode dar origem a erros e é uma má prática, por isso utilize `sizeof(type)`
 - Há muitos anos o tamanho de um `int` eram 16 bits, e muitos programas foram escritos com este pressuposto.
 - Qual é o tamanho actual de um `int`?

Alocação dinâmica de memória (2/4)

- Para alocar memória para algo novo utilize a função `malloc()` com a ajuda de `typedef` e `sizeof`:

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

- `ptr` aponta para um espaço alíquo na memória com tamanho `(sizeof(int))` bytes.
- `(int *)` indica ao compilador o tipo de objectos que irá ser guardado naquele espaço (**chama-se um `typedef` ou simplesmente `cast`**).

- `malloc` é raramente utilizado para uma única variável

```
ptr = (int *) malloc (n*sizeof(int));
```

- Isto é, aloca espaço para **uma tabela** de `n` inteiros.

Alocação dinâmica de memória (3/4)

- Depois do `malloc()` ser chamado, a memória alocada contém só lixo, portanto não a utilize até ter definido os valores aí guardados.
- Depois de alocar dinamicamente espaço, deverá libertá-lo de forma também dinâmica:

```
free(ptr);
```
- Utilize a função `free()` para fazer a limpeza
 - Embora o programa `liberte` toda a memória na saída (ou quando o `main` termina), não seja preguiçoso!
 - Nunca sabe quando o seu código será reaproveitado e o `main` transformado numa sub-rotina!

Alocação dinâmica de memória (4/4)

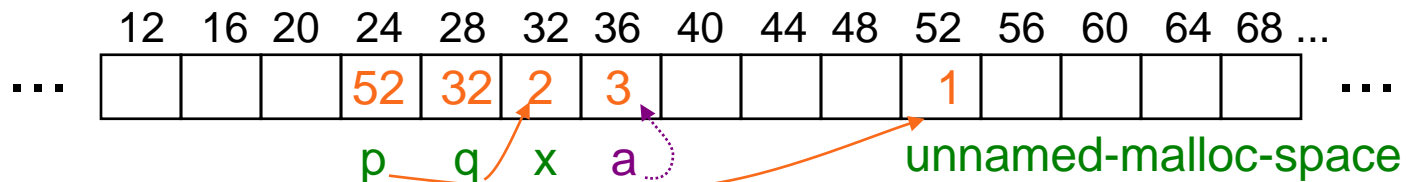
- As seguintes acções fazem com que o seu programa "crash" ou se comporte estranhamente mais à frente. Estes dois erros são bugs MUITO MUITO difíceis de detetar, portanto tenha cuidado:
 - `free()` ing a mesma zona de memória mais do que uma vez
 - chamar `free()` sobre algo que não foi devolvido por `malloc()`
- O runtime não verifica este tipo de erros
 - A alocação de memória é tão crítica para o desempenho que simplesmente não há tempo para fazer estas verificações
 - Assim, este tipo de erros faz com que as estruturas internas de gestão de memória sejam corrompidas
 - E o problema só se manifesta mais tarde numa zona de código que não tem nada a ver ...!

Diferença subtil entre tabelas e ponteiros

```
void foo() {
    int *p, *q, x, a[1]; // a[] = {3} also works here
    p = (int *) malloc (sizeof(int));
    q = &x;

    *p = 1; // p[0] would also work here
    *q = 2; // q[0] would also work here
    *a = 3; // a[0] would also work here

    printf("*p:%u, p:%u, &p:%u\n", *p, p, &p);
    printf("*q:%u, q:%u, &q:%u\n", *q, q, &q);
    printf("*a:%u, a:%u, &a:%u\n", *a, a, &a);
}
```



```
*p:1, p:52, &p:24
*q:2, q:32, &q:28
*a:3, a:36, &a:36
```



QUIZ

Considere o seguinte programa em C em que a tabela "*tab*" começa no endereço 0x62FE20. Com base nisso, indique qual das seguintes opções é VERDADEIRA:

- a) As instruções I1 e I2 imprimem 0X62FE20 no ecrã, enquanto que a instrução I3 imprime 0x6 e a instrução I4 imprime 0x5 no ecrã.
- b) A instrução I1 imprime 0X62FE20, a instrução I2 imprime 0 no ecrã, a instrução I3 imprime 0x6 e a instrução I4 imprime 0x5 no ecrã.
- c) As instruções I1 e I2 imprimem 0X62FE20 no ecrã enquanto que as instruções I3 e I4 imprimem 0x5 no ecrã.
- d) As instruções I1 e I2 imprimem 0 no ecrã enquanto que as instruções I3 e I4 imprimem 0x3 no ecrã.

```
#include <stdio.h>
int main(){
int tab[] = {0,2,4,6,8,10,12};
int *p1, **p2;
p1 = tab+2;
p2 = &p1;
// Instrução I1
printf("%#X \n", &tab );
// Instrução I2
printf("%#X \n", tab );
// Instrução I3
printf("%#X \n", *(*(p2)+1) );
// Instrução I4
printf("%#X \n", *(p1)+1 );
return 0;
}
```

Para saber mais ...

- K&R - The C Programming Language
 - Capítulo 5
- Tutorial de Nick Parlante
- Links úteis para Introdução ao C
 - <http://man.he.net/> (man pages de Unix)
 - <http://linux.die.net/man/> (man pages de Unix)
 - <http://www.lysator.liu.se/c/bwk-tutor.html>
 - <http://www.allfreetutorials.com/content/view/16/33/>
(vários tutoriais)

