



Introdução



Sistemas de Microprocessadores 2021/2022

Os Computadores são Inteligentes?

♦ Na perspetiva do programador:

□ Operações/Funções muito complexas:

- `(map (lambda (x) (* x x)) '(1 2 3 4))`

□ Gestão automática de memória:

- `List l = new List;`

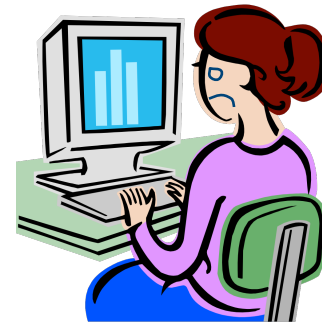
□ Estruturas "básicas" pré-definidas:

- Integers, floats, caracteres, operadores, print commands



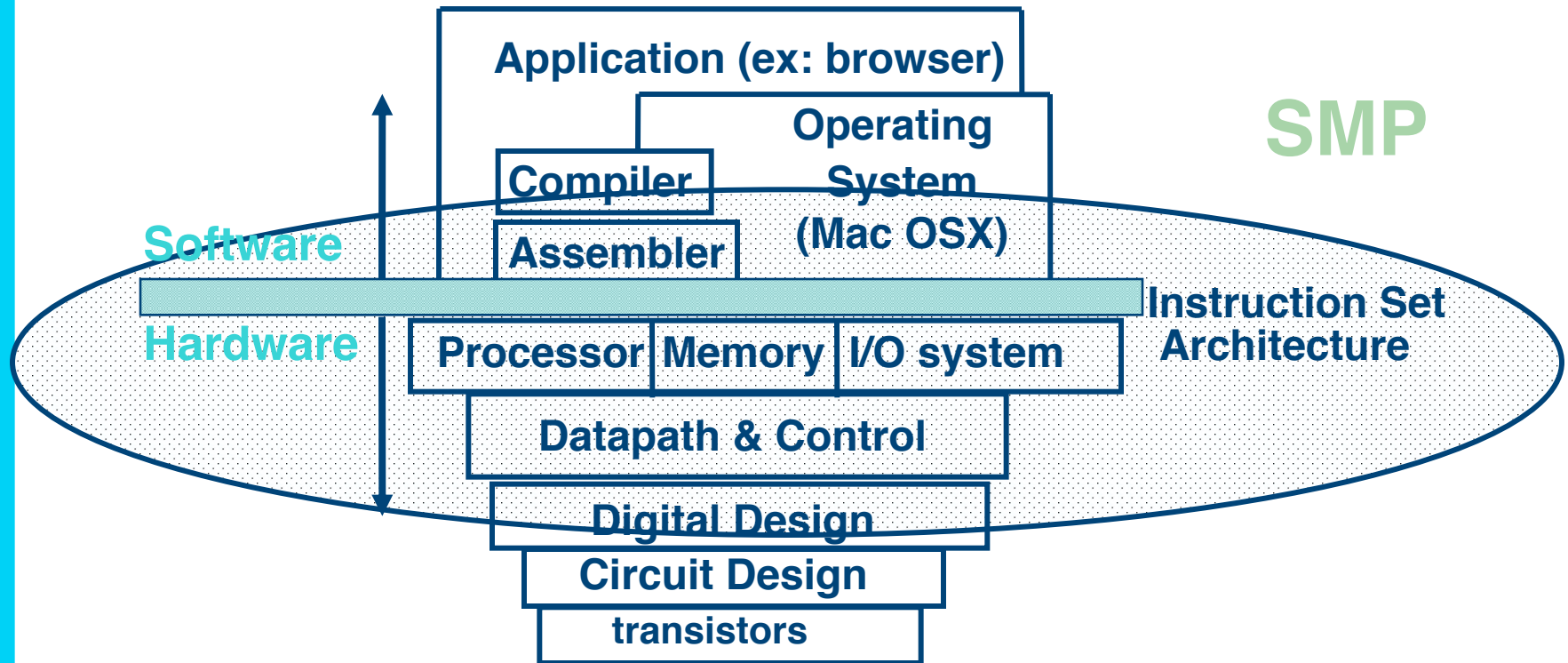
Os Computadores são Inteligentes?

- ◆ No mundo "real" do hardware:
 - Meia dúzia de operações lógicas:
 - {and, or, not}
 - A memória não se gere sozinha
 - Só dois valores possíveis:
 - {0, 1} ou {low, high} ou {off, on}



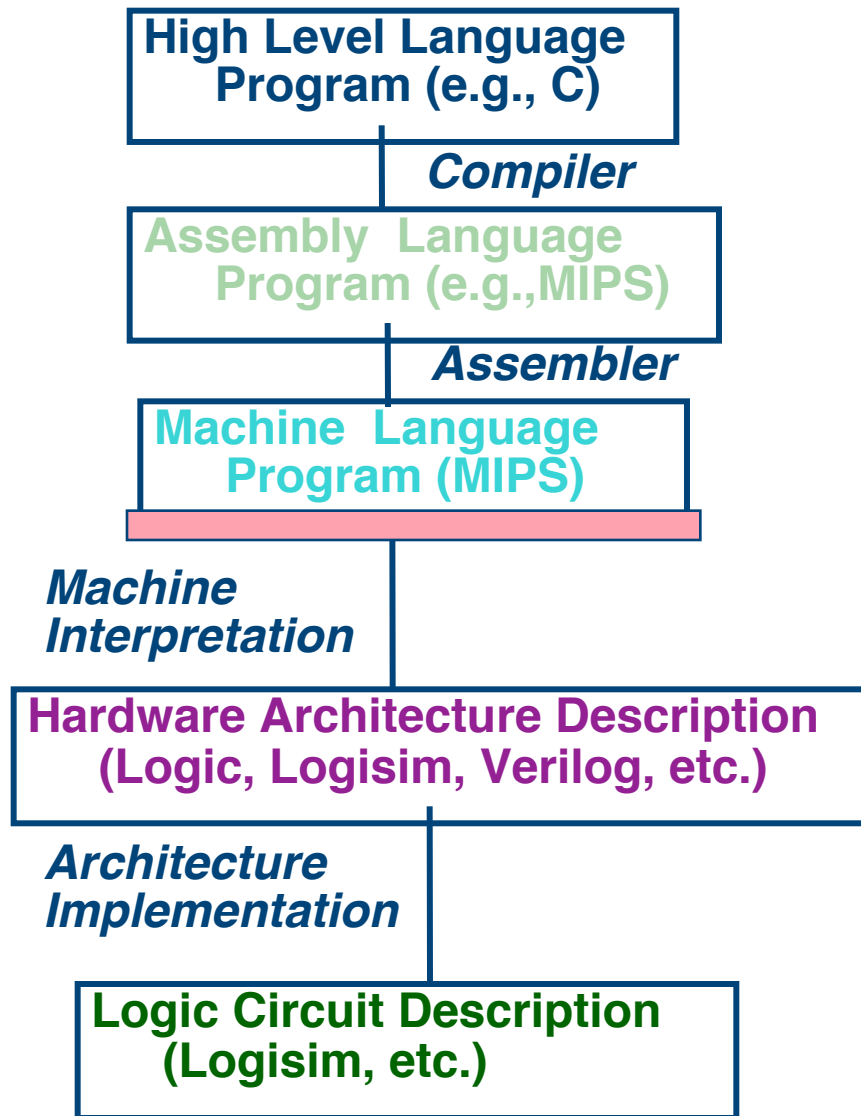
Computers are
dumb !

SMP na "Big Picture" ...



- ◆ Coordenação de muitos
níveis (layers) de abstração

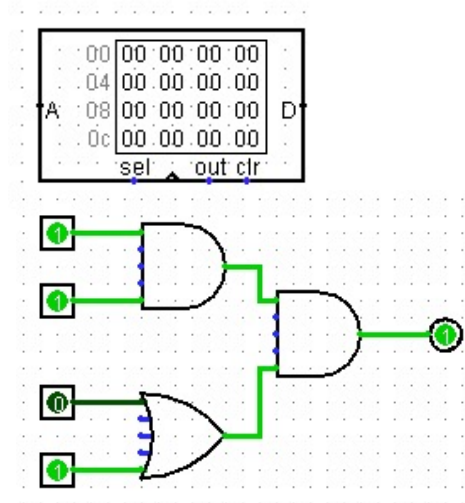
Vamos fazer a ponte entre PC e LSD ...



```
temp = v[k];  
v[k] = v[k+1];  
v[k+1] = temp;
```

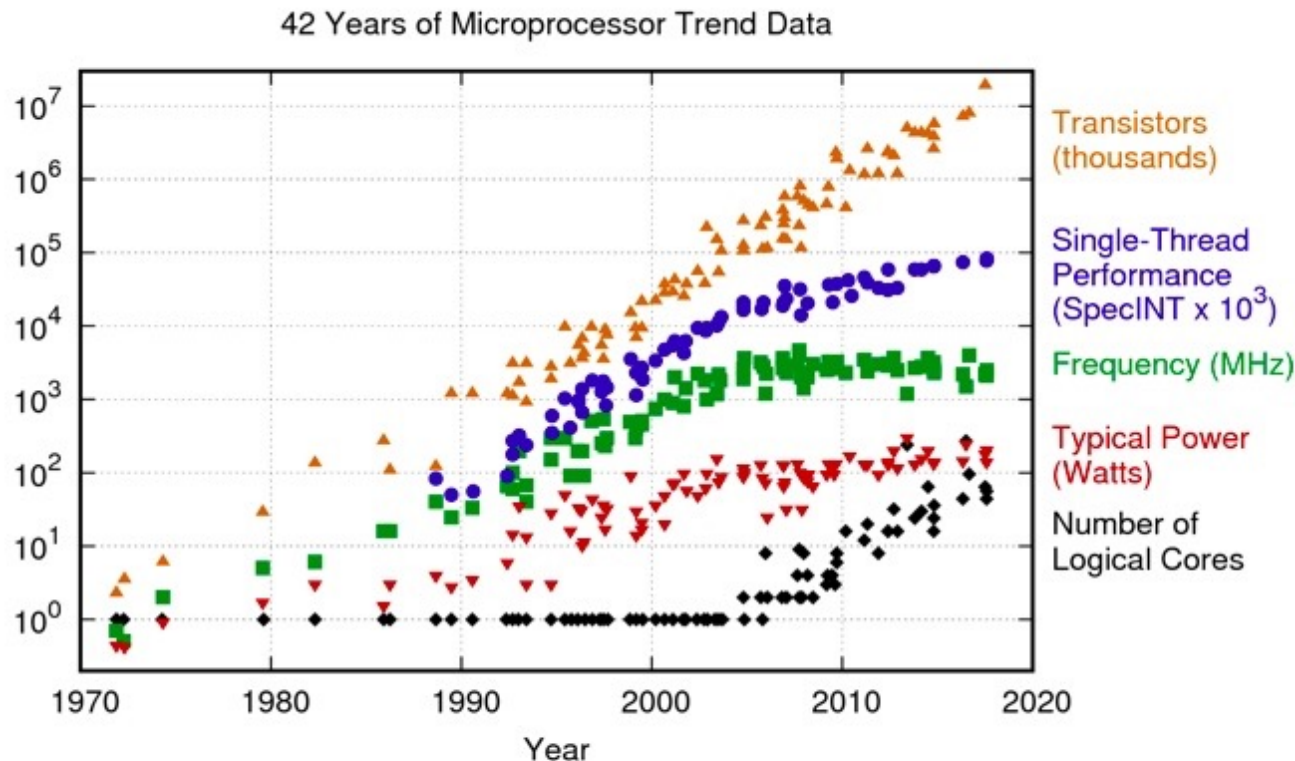
```
lw    $t0, 0($2)  
lw    $t1, 4($2)  
sw    $t1, 0($2)  
sw    $t0, 4($2)
```

```
0000 1001 1100 0110 1010 1111 0101 1000  
1010 1111 0101 1000 0000 1001 1100 0110  
1100 0110 1010 1111 0101 1000 0000 1001  
0101 1000 0000 1001 1100 0110 1010 1111
```



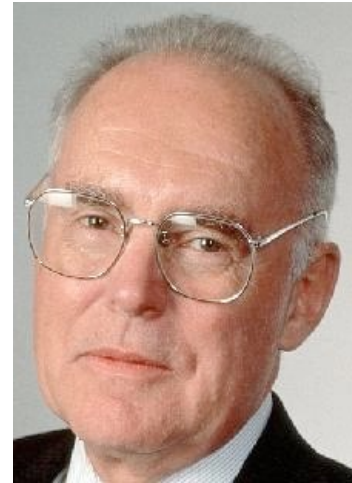
Complexidade dos μ Ps

de transístores num CI



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

<https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/>



Gordon Moore
Co-fundador da Intel

“Lei de Moore”

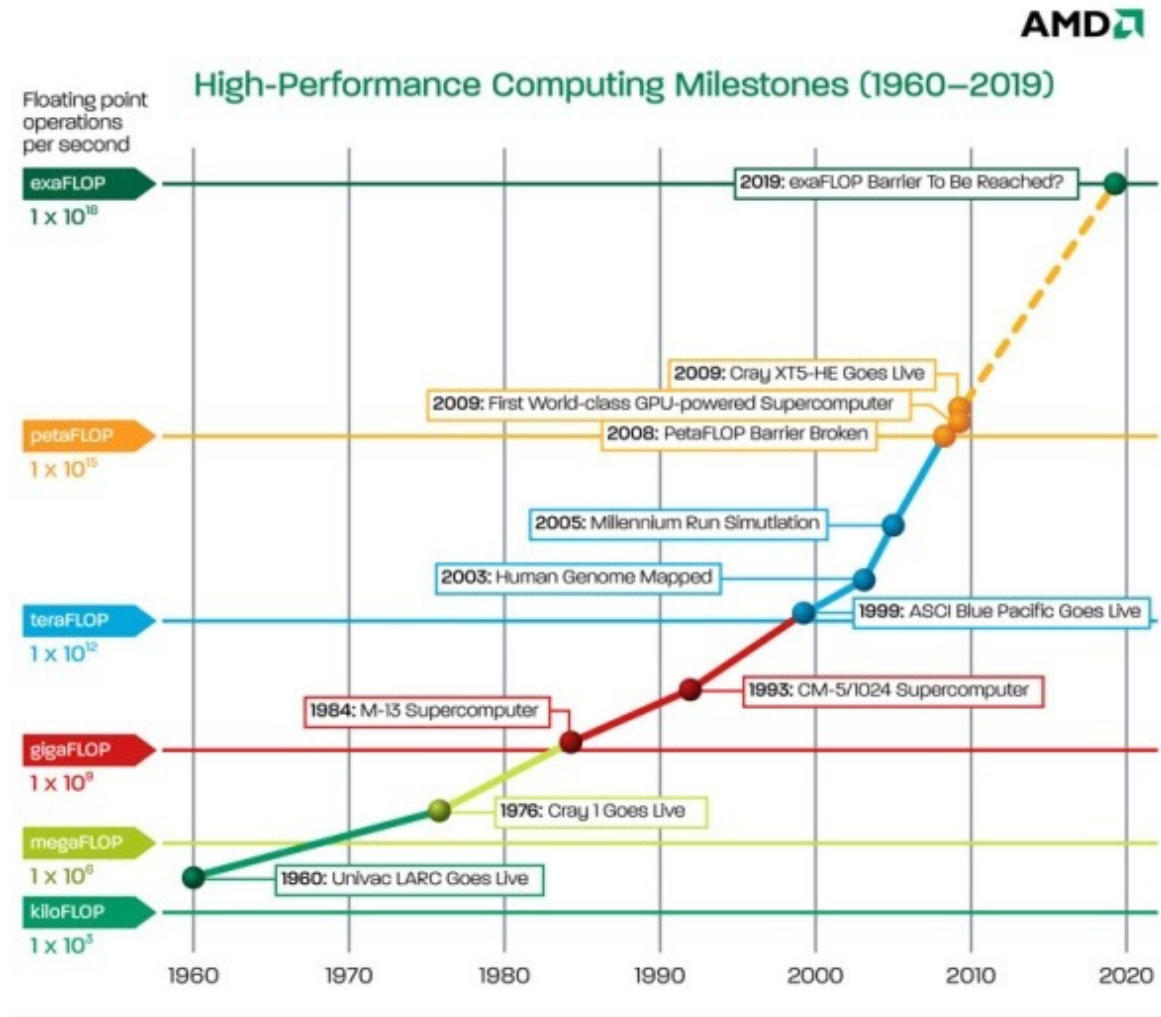
O número de transístores por chip duplica cada 1.5 anos

Capacidade de memória (Single-Chip D(D)RAM)

Ano	Capacidade (Mbit)	Ano	Capacidade (Gbit)
1980	0.0625	2004	1
1983	0.25	2006	2
1986	1	2008	4
1989	4	2010	8
1992	16	2012	16
1996	64	2014	32
1998	128	2016	64
2000	256	2018	128
2002	512	2020	256
2004	1024 (1Gbit)	2022	512
		2024	1024 (1Tbit)

- Agora 1.4X/ano, ou 2X cada 2 anos.
- 4 milhões de vezes desde 1980!

Desempenho de um μP

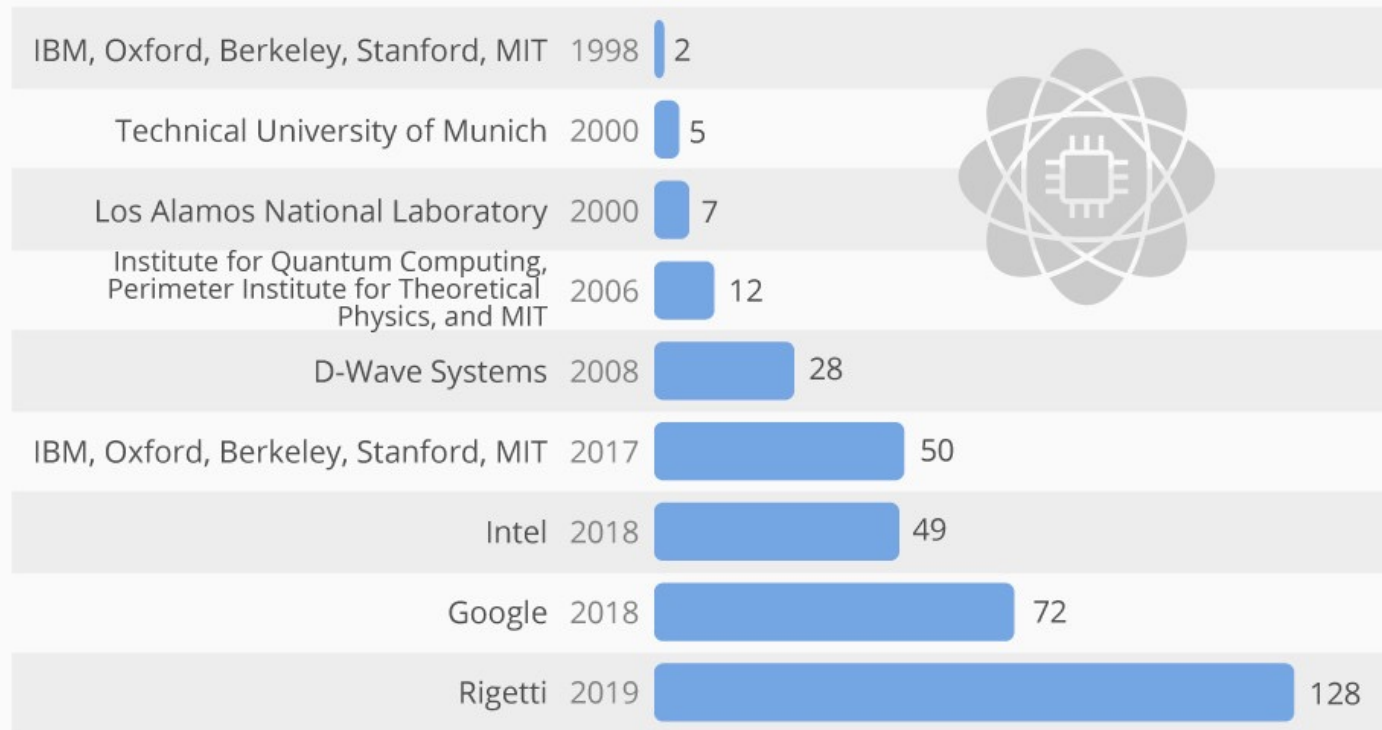


©2010 Advanced Micro Devices, Inc. All rights reserved. AMD, the AMD Arrow logo, combinations thereof, are trademarks of Advanced Micro Devices, Inc. All other trademarks are the property of their respective owners.

Próxima geração de processadores

20 Years of Quantum Computing Growth

Quantum computing systems produced by organization(s) in qubits, between 1998 to 2019*



* Rigetti announced in August 2018 that it would release a 128-qubit quantum computer system within the next 12 months.



@StatistaCharts

Source: CB Insights

statista

Pondo as coisas em perspetiva ...

“If the automobile had followed the same development cycle as the computer,
a Rolls-Royce would today cost \$100,
get a million miles per gallon,
and explode once a year,
killing everyone inside.”

– *Robert X. Cringely*



Objetivos

- ◆ Perceber os princípios e ideias dominantes que estão por detrás da computação e engenharia:
 - Princípios de abstração usados para construir as diferentes camadas dos sistemas
 - Dados são bytes em memória: o seu tipo (integers, floating point, characters) é uma interpretação determinada pelo programa
 - Armazenamento de programas: instruções são bytes na memória, a diferença entre instruções e dados é a forma como são interpretados
 - Princípios de localidade usados na hierarquia de memória
 - Aumento de desempenho tirando partido do paralelismo
 - Compilação v. Interpretação

Competências Adicionais

◆ Programação em C

- Quem sabe uma linguagem (Python/C) deve ser capaz de aprender outra de forma autónoma
- Consolidação das competências de programação
- Compreensão da razão de ser de muitas das regras de sintaxe
- No final serão programadores muito mais "hardware aware"

◆ Programação em Assembly

- Competência adquirida como efeito "colateral" de compreender os grandes princípios que regem uma máquina-computador

◆ Desenho e Arquitetura de Computadores

- Introdução ao desenho de hardware
- Poderão continuar a aprender em Arquitetura de Computadores e Projeto de Sistemas Digitais (4º ano do Ramo de Computadores)

Tópicos que vamos abordar ...

- ♦ Módulo 1: A Linguagem C e o Hardware
 - Linguagem C (básico + ponteiros)
 - Gestão de Memória (alocação dinâmica, estática, etc)
 - Portos de I/O e programação de hardware
- ♦ Módulo 2: Programação em Assembly para o MIPS
 - Instruções Aritméticas Básicas
 - Leitura e escrita da memória
 - Controlo de Fluxo
 - Codificação de instruções
 - make-ing an Executable (compilação, assemblagem, etc)
- ♦ Módulo 3: Introdução à Arquitetura de Computadores
 - Organização da CPU
 - Pipelining
 - Caches e Hierarquia de Memória
 - Polling e interrupções

Funcionamento: Pré - Requisitos

- Domínio de pelo menos uma linguagem de programação
Definição de variáveis, Operadores, Ciclos, Rotinas e Procedimentos, princípios de algoritmia, etc
- Conhecimentos básicos de Sistemas Digitais
Portas lógicas, Mux, Demux, Flip-flops, circuitos combinacionais, circuitos sequenciais/máquinas de estado, etc.
- Representação de números inteiros positivos e negativos
Binário, hexadecimal, complementos de 2, overflow, bit, Kbit, Mbit, Byte, KByte, MByte, etc

Funcionamento: Aulas & Laboratórios

- Uma aula semanal de 2 horas para exposição e discussão teórica
 - ▶ 1 hora de preparação através da leitura prévia dos slides
 - ▶ 1 a 2 horas de estudo posterior para consolidar os conhecimentos
- Uma aula semanal de 2 horas para a realização de práticas laboratoriais
 - ▶ 1 trabalho por semana (total de ~ 12 trabalhos)
 - ▶ 1 a 2 horas de preparação prévia
 - ▶ Instalar "tools" no computador pessoal a partir da primeira aula
- Notas:
 - ▶ O tempo de preparação e estudo são valores mínimos aconselhados
 - ▶ Não há picos de trabalho (carga média semanal à volta de 8 horas)

Funcionamento: Atitude e Método

- Ser participativo nas aulas, não hesitando em interromper o instrutor sempre que algo não é claro.
 - ▶ Já que estou na aula vou aproveitar para não ter que estudar tanto em casa.
 - ▶ O instrutor só pode saber que está a ir muito depressa se alguém lhe disser
 - ▶ A aula passa a ser bem mais interessante para toda a gente
- Ler os slides antes da aula (1 hora) e preparar os trabalhos (1 a 2 horas)
- Tirar notas para apoiar o estudo
 - ▶ As notas permitem-me recordar o que foi dito na aula (que pode não estar nos livros)
 - ▶ Vou saber aquilo a que o professor dá mais importância (útil para o exame)

Funcionamento: Avaliação

A avaliação de Época Normal consiste em:

- ♦ 2 valores para o desempenho nas aulas laboratoriais (obrigatórias)
- ♦ 6 valores para um miniteste individual
- ♦ 6 valores para uma frequência a realizar a meio do semestre
- ♦ 6 valores para uma segunda frequência a realizar na época normal

A avaliação nas restantes épocas consiste em

- ♦ 2 valores para o desempenho nas aulas laboratoriais (obrigatórias)
- ♦ 6 valores para o miniteste (não repetível)
- ♦ 12 valores para um exame de recurso final

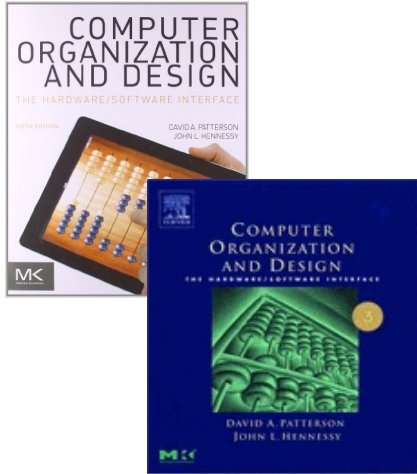
Funcionamento: Avaliação

Notas Importantes:

É obrigatória a frequência de um mínimo de 82.5% das aulas laboratoriais.

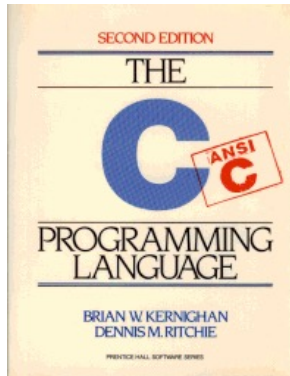
Os estudantes trabalhadores têm de cumprir a componente laboratorial. No caso de haver dificuldades de horários deverão contactar o docente das aulas teóricas imediatamente.

Não é permitido os alunos frequentarem regularmente turmas práticas em que não estejam inscritos (situações pontuais deverão merecer anuência prévia do docente responsável).



- ♦ **P&H** - *"Computer Organization and Design: The Hardware/Software Interface", Fifth Edition (2013)*, Patterson and Hennessy.

- ♦ **P&H** - *"Computer Organization and Design: The Hardware/Software Interface", Fifth Edition (2014)*, Patterson and Hennessy.



- ♦ **K&R** - *"The C Programming Language", Kernighan and Ritchie, 2nd edition*

- ♦ Slides

- ♦ Textos Fornecidos na página InfoEstudante

Recursos online da disciplina



As aulas teóricas dos anos letivos anteriores foram gravadas e estão disponíveis neste link do canal educast:

2019/2020 - <https://educast.fccn.pt/vod/channels/7xsgmoltz?locale=pt>.

2020/2021 - <https://educast.fccn.pt/vod/channels/2gm9lws14o?locale=pt>



Linguagem C

- Ponteiros e Arrays -



Sistemas de Microprocessadores 2021/2022

Programa em C espalhado por ficheiros

Ficheiro main.c

```
#include<stdio.h>
int IntroduzFaltas();

int main(){
    int total=15, faltas;
    faltas=IntroduzFaltas();
    printf("Vai entao assistir a %d aulas \n",total-faltas);
}
```

Ficheiro intro.c

```
#include<stdio.h>

int IntroduzFaltas(){
    int tmp;
    printf("Quantas faltas vai dar? ");
    scanf("%d",&tmp);
    return(tmp);
}
```

Compilação : Overview

◆ O compilador converte C em código máquina (string the 0s e 1s) que é específico da arquitetura.

- Diferente do Java que converte para um bytecode independente da arquitetura (máquinas virtuais).

- Diferente do Python que interpreta o código permitindo interatividade.

- Para o C a geração do executável passa normalmente por duas etapas principais:

- A **compilação**, que converte ficheiros .c (código fonte) em ficheiros .o (código objeto).

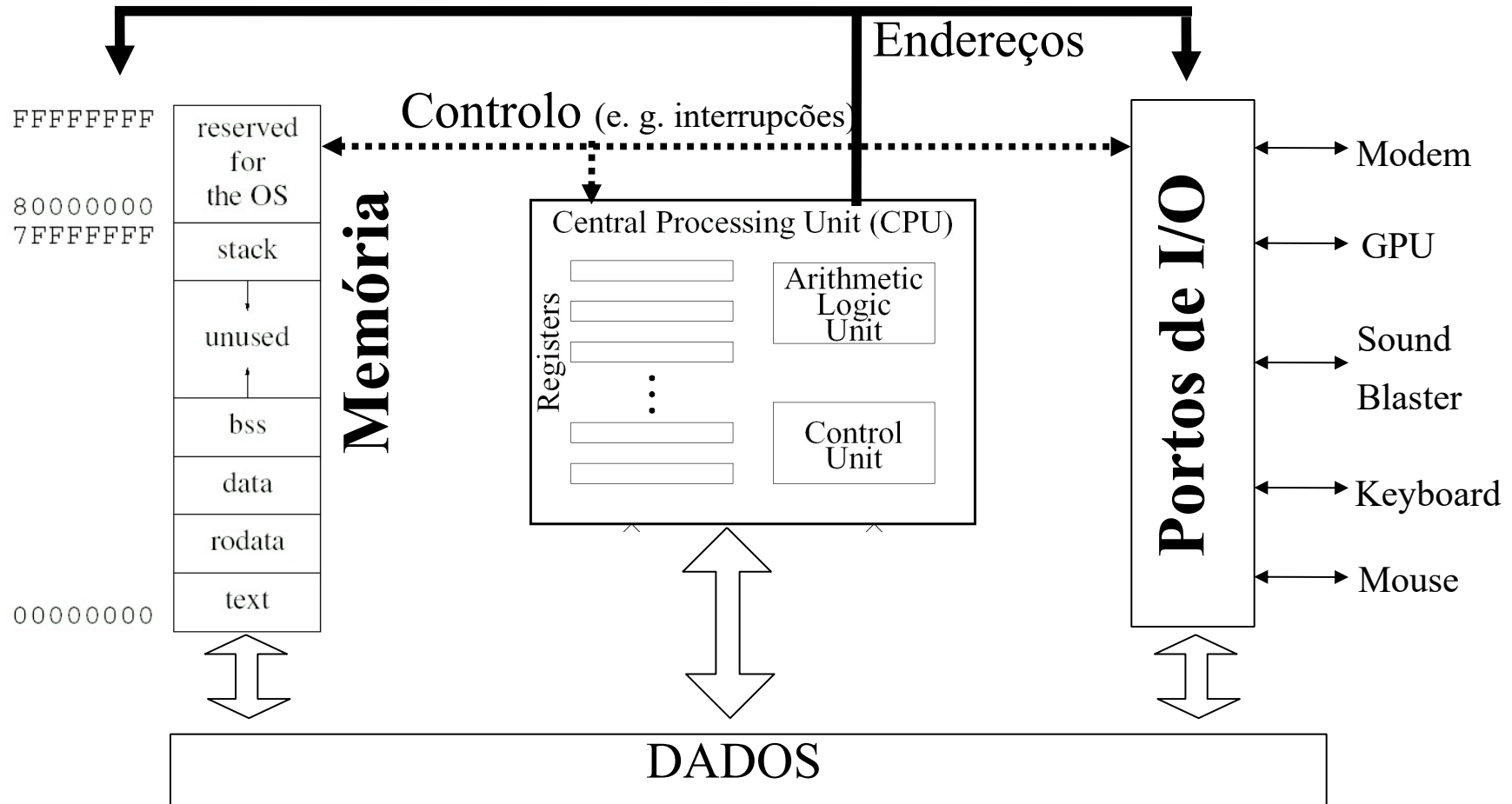
```
gcc -c main.c
```

```
gcc -c intro.c
```

- A **linkagem**, que junta os ficheiros .o num executável final

```
gcc -o final.exe main.o intro.o
```

Anatomia de um Computador



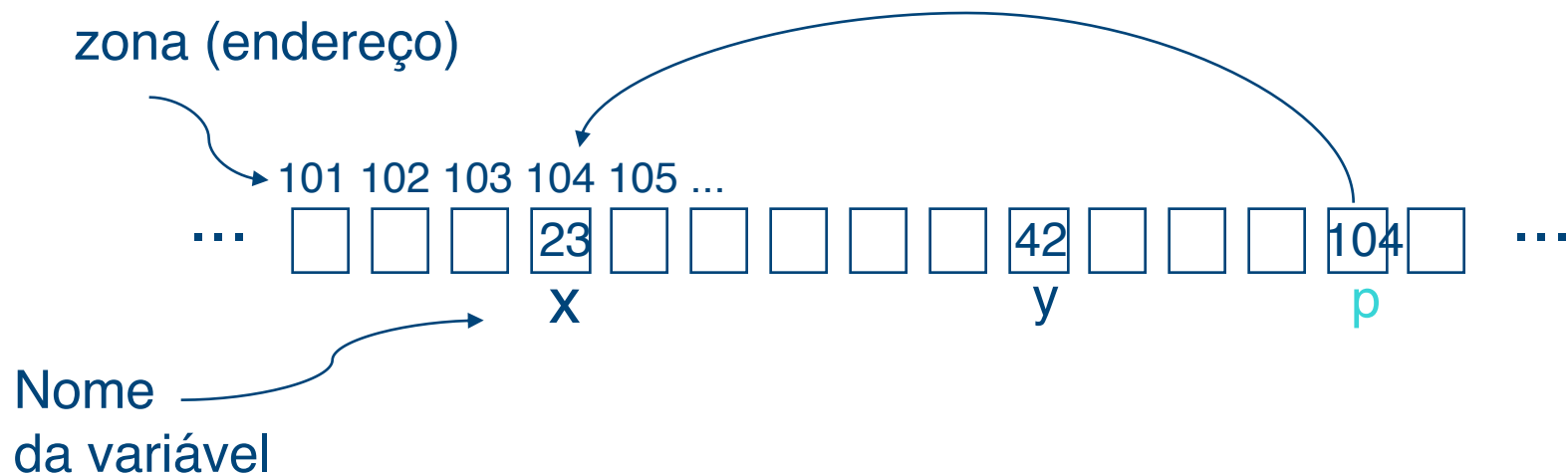
Endereço vs. Valor

- ◆ Considere a memória como sendo um grande array:
 - Cada célula do array tem um endereço associado
 - Cada célula do array contém um valor
- ◆ Não confundir o **endereço**, que referencia uma determinada célula de memória, com o **valor** armazenado nessa célula de memória.
- ◆ É ridículo dizer que uma pessoa e o seu endereço de correio são a mesma coisa !



Ponteiros (revisão)

- ◆ 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 (revisão)

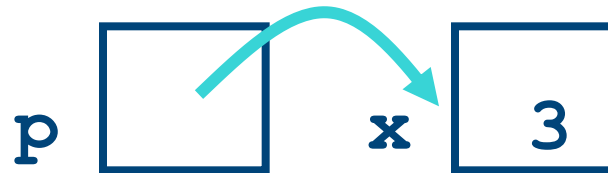
```
int *p, x;
```



```
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 (revisão)

◆ 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` (revisão)

- ◆ 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 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.).

Trabalho para Casa ...

P&H - Computer Organization and Design

Capítulo 1 (ler) Secções 3.1, 3.2 e 3.3 (ignorar referências ao MIPS)

K&R - The C Programming Language

Capítulos 1 a 5 (revisão de programação em C)





Linguagem C

- Ponteiros e Arrays - (Continuação)



Sistemas de Microprocessadores 2021/2022

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 têm 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^{o} elemento (atenção o primeiro elemento é acedido com $\text{num}=0$).

Tabelas/Arrays (2/5)

◆ Arrays são (quase) idênticos a 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 array (o "nome da tabela") é um ponteiro para o primeiro elemento.

Tabelas/Arrays (3/5)

◆ Consequências:

- ☐ `ar` é uma variável array mas em muitos aspetos 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)

- ◆ Array de dimensão n ; queremos aceder aos elementos de 0 a $n-1$, usando como teste de saída a comparação com o endereço da "casa" depois do fim do array.

```
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 transpôr os limites da tabela. *É necessário evitar isto de forma explícita*
- Consequência: Uma função que percorra uma tabela tem que receber a variável array e a respetiva dimensão.

◆ **Segmentation faults e bus errors:**

- Isto são "runtime errors" muito difíceis de detetar. É preciso ser cuidadoso! (Nas aulas laboratoriais 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 do array.

◆ $*p++$ 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 sim 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 dois 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 dois ponteiros
- ☐ Multiplicar dois 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/subtração move o ponteiro o 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-1]);
    /* OR */
    return *(array + n-1);
}
```

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 the caracteres

- ◆ Os caracteres são representados através de bytes
- ◆ Existem várias codificações: ASCII, unicode, etc
- ◆ É tudo uma 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		L	⌞	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	ƒ
5			%	5	E	U	e	u	à	ò	Ñ	⌞	⌞	⌞	σ	J
6			&	6	F	V	f	v	â	û	ä	⌞	⌞	⌞	μ	÷
7	bel		'	7	G	W	g	w	ç	ù	é	⌞	⌞	⌞	τ	≈
8	bs		(8	H	X	h	x	ê	ÿ	í	⌞	⌞	⌞	Φ	°
9	tab)	9	I	Y	i	y	ë	Ö	⌞	⌞	⌞	⌞	θ	°
A	lf		*	:	J	Z	j	z	è	Ü	⌞	⌞	⌞	⌞	Ω	.
B	vt	esc	+	;	K	[k	{	ï	¢	½	⌞	⌞	⌞	ó	√
C	ff		.	<	L	\	l		î	£	¼	⌞	⌞	⌞	∞	ⁿ
D	cr		-	=	M]	m	}	ì	¥	ì	⌞	⌞	⌞	∅	²
E			.	>	N	^	n	~	Ä	℞	«	⌞	⌞	⌞	ε	■
F			/	?	O	_	o		Å	f	»	⌞	⌞	⌞	n	

C Strings

- ◆ Uma **string** em C é um array de caracteres.

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

- ◆ Como é que sabemos quando uma string termina?
 - O último carácter é seguido de um byte a 0 (null terminator)

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

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

Arrays bi-dimensionais (1/2)

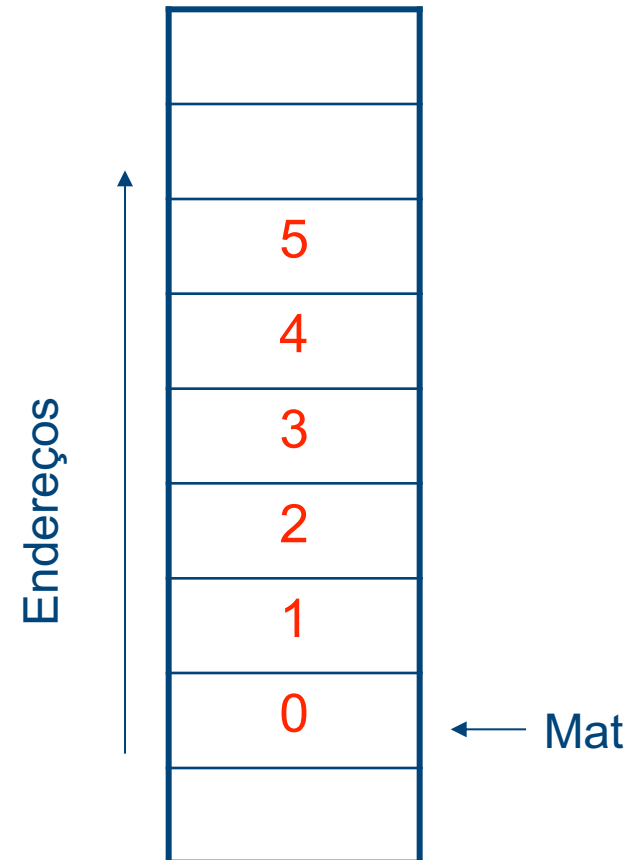
```
#define ROW_SIZE 3  
#define COL_SIZE 2
```

```
...  
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++;  
    }  
...
```

Mat =

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

MEMÓRIA



Arrays bi-dimensionais (2/2)

- ◆ O C arruma um array bi-dimensional empilhando as linhas umas a seguir às outras.
- ◆ O espaço total de memória ocupado é $\text{ROW_SIZE} \times \text{COL_SIZE}$
- ◆ Temos que:
 $\text{Mat}[2][1]$ é o mesmo que $\text{Mat}[2 * \text{COL_SIZE} + 1]$

Arrays vs. Ponteiros

- ◆ O nome de um array é um ponteiro para o primeiro elemento da tabela (índice 0).
- ◆ Um parâmetro tabela pode ser declarado como um array ou 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 | $\text{ptr} + 1$ |
| II. | integer + pointer | $1 + \text{ptr}$ |
| III. | pointer + pointer | $\text{ptr} + \text{ptr}$ |
| IV. | pointer – integer | $\text{ptr} - 1$ |
| V. | integer – pointer | $1 - \text{ptr}$ |
| VI. | pointer – pointer | $\text{ptr} - \text{ptr}$ |
| VII. | compare pointer to pointer | $\text{ptr1} == \text{ptr2}$ |
| VIII. | compare pointer to integer | $\text{ptr} == 1$ |
| IX. | compare pointer to 0 | $\text{ptr} == \text{NULL}$ |
| X. | compare pointer to NULL | $\text{ptr} == \text{NULL}$ |

#invalid
1
2
3
4
5
6
7
8
9
(1) 0

Concluindo ...

- ◆ Ponteiros e arrays são **virtualmente o mesmo**
- ◆ O C sabe como **incrementar ponteiros**
- ◆ O C é uma linguagem eficiente com muito poucas proteções
 - Os limites dos arrays **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 yourself with it!" (extraído de K&R)



Linguagem C

- Alocação Dinâmica -



Sistemas de Microprocessadores 2021/2022

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 objetos pode dar origem a erros e é uma má prática, por isso utilize `sizeof(type)`
 - Há muitos anos o tamanho de um `int` era 16 bits, e muitos programas foram escritos com este pressuposto.
 - Qual é o tamanho atual de um `int`?

- ◆ “`sizeof`” determina o tamanho para arrays:

```
int ar[3]; // Or: int ar[] = {54, 47, 99}
sizeof(ar) ⇒ 12
```

- ...bem como para arrays cujo tamanho é definido em run-time:

```
int n = 3;
int ar[n]; // or: int ar[fun_that_returns_3()];
sizeof(ar) ⇒ 12
```

Alocação dinâmica de memória (2/4)

- ◆ Para alocar memória para algo novo utilize a função `malloc()` com a ajuda de `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 objetos que irá ser guardado naquele espaço (chama-se um **typecast** ou **simplesmente cast**).

- ◆ `malloc()` é raramente utilizado para uma única variável

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

- Isto é um **array** 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 re-aproveitado e o `main` transformado numa sub-rotina!

Alocação dinâmica de memória (4/4)

◆ As seguintes ações fazem com que o seu programa "crashe" ou se comporte estranhamente mais à frente. Estes dois erros são bugs MUITO MUITO difíceis de se apanhar, portanto atenção:

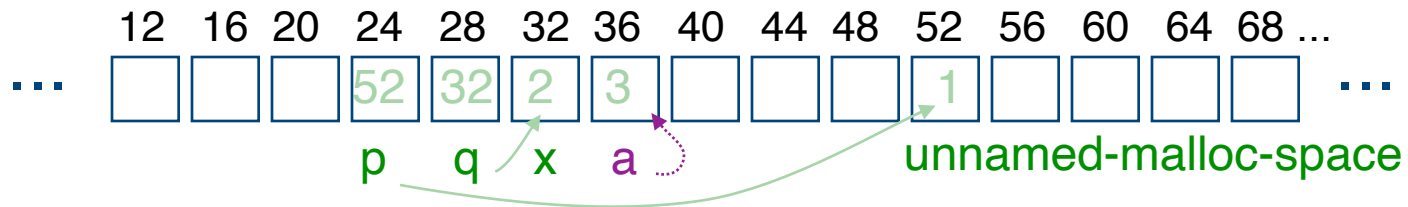
- `free()` ing a mesma zona de memória mais do que uma vez
- chamar `free()` sobre algo que não foi devolvido por `malloc()`

◆ 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 arrays 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);  
  
    free(p);  
}
```



***p:1, p:52, &p:24**

***q:2, q:32, &q:28**

***a:3, a:3, &a:36**

Binky Video

Pointer Fun with

Binky



by Nick Parlante

This is document 104 in the Stanford CS
Education Library — please see
cslibrary.stanford.edu
for this video, its associated documents,
and other free educational materials.

Copyright © 1999 Nick Parlante. See copyright
panel for redistribution terms.
Carpe Post Meridiem!

QUIZ

Which are guaranteed to print out 5?

I: `main() {
 int *a_ptr; *a_ptr = 5; printf("%d", *a_ptr); }`

II: `main() {
 int *p, a = 5;
 p = &a; ...
 /* code; a & p NEVER on LHS of = */
 printf("%d", a); }`

III: `main() {
 int *ptr;
 ptr = (int *) malloc (sizeof(int));
 *ptr = 5;
 printf("%d", *ptr); }`

	I	II	III
0:	-	-	-
1:	-	-	YES
2:	-	YES	-
3:	-	YES	YES
4:	YES	-	-
5:	YES	-	YES
6:	YES	YES	-
7:	YES	YES	YES

Nota: LHS significa "Left Hand Side"

QUIZ

```
{ char a= 0xFF;  
  unsigned char b=0xFF;  
  printf(" %d %d \n", a, b);  
  
  ...
```

☐ O que é que aparece no ecrã?

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>





Linguagem C

- Zonas de Memória -



Sistemas de Microprocessadores 2021/2022

Variáveis Globais

- ◆ A declaração de ponteiros não aloca memória em frente do ponteiro
- ◆ Até agora falámos de duas maneiras diferentes de alocar memória:
 - Declaração de variáveis locais
`int i; char *string; int ar[n];`
 - Alocação dinâmica em runtime usando "malloc"
`ptr = (struct Node *) malloc(sizeof(struct Node)*n);`
- ◆ Existe uma terceira possibilidade ...
 - Declaração de variáveis fora de uma função (i.e. antes do `main`)
 - É similar às variáveis locais mas tem um âmbito global, podendo ser lida e escrita de qualquer ponto do programa

```
int myGlobal;  
main() {  
}
```

Gestão de Memória em C (1/2)

◆ Um programa em C define três zonas de memória distintas para o armazenamento de dados

- Static Storage: onde ficam as variáveis globais que podem ser lidas/escritas por qualquer função do programa. Este espaço está alocado permanentemente durante todo o tempo em que o programa corre (daí o nome estático)
- A Pilha/Stack: armazenamento de variáveis locais, parâmetros, endereços de retorno, etc.
- A Heap (dynamic malloc storage): os dados são válidos até ao instante em que o programador faz a desalocação manual com `free()`.

◆ O C precisa de saber a localização dos objetos na memória, senão as coisas não funcionam como devem.

Gestão de Memória em C (2/2)

◆ O *espaço de endereçamento* de um programa contém 4 regiões:

- **stack**: variáveis locais, *cresce "para baixo"*
- **heap**: espaço requisitado via `malloc()` ; *cresce "para cima"*.
- **dados estáticos**: variáveis globais declaradas fora do `main()`, *tamanho constante durante a execução*.
- **código**: Carregado quando o programa começa, o *tamanho não se modifica*

$\sim \text{FFFF FFFF}_{\text{hex}}$



$\sim 0_{\text{hex}}$

O Sistema Operativo evita a sobreposição da Stack com a Heap

Onde é que as variáveis são alocadas?

- ◆ Se são declaradas fora de qualquer função/procedimento, então são alocadas na zona estática.
- ◆ Se são declaradas dentro da função, então são alocadas na “stack” sendo o espaço liberto quando o procedimento termina.

□ Nota: `main()` is a procedure

```
int myGlobal;  
main() {  
    int myTemp;  
}
```

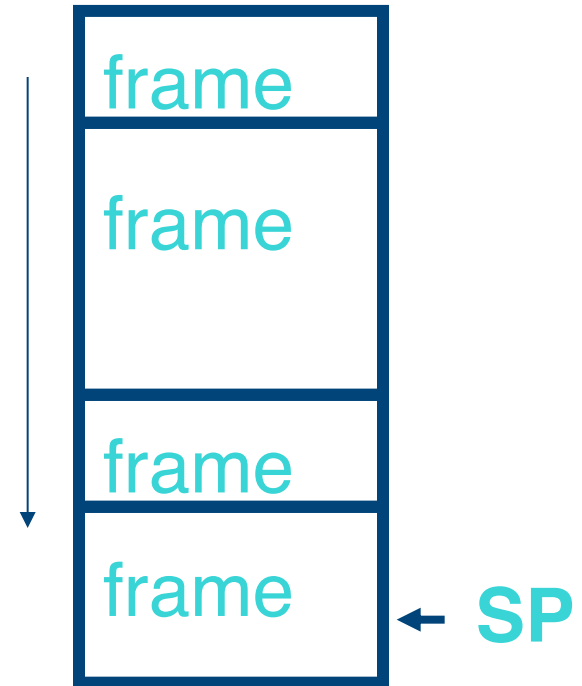
A Pilha/Stack (1/2)

◆ Um "Stack Frame" inclui:

- Endereços de retorno
- Parâmetros
- Espaço para variáveis locais

◆ Os "Stack frames" são blocos contíguos de memória; o "stack pointer" indica qual é o "frame" no topo da pilha (ver FILO)

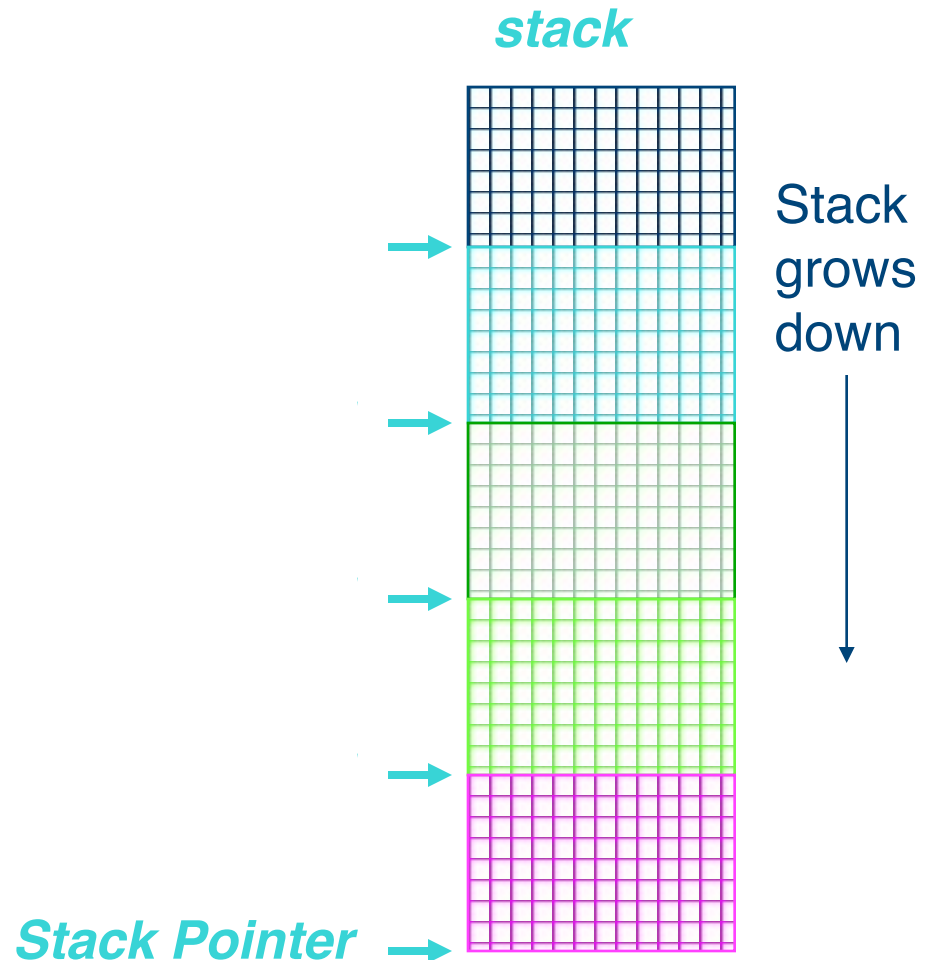
◆ Quando uma rotina termina, o seu "stack frame" é descartado (não explicitamente apagado). Isto permite libertar memória para futuras utilizações



A Pilha/Stack (2/2)

◆ Last In, First Out (LIFO) data structure

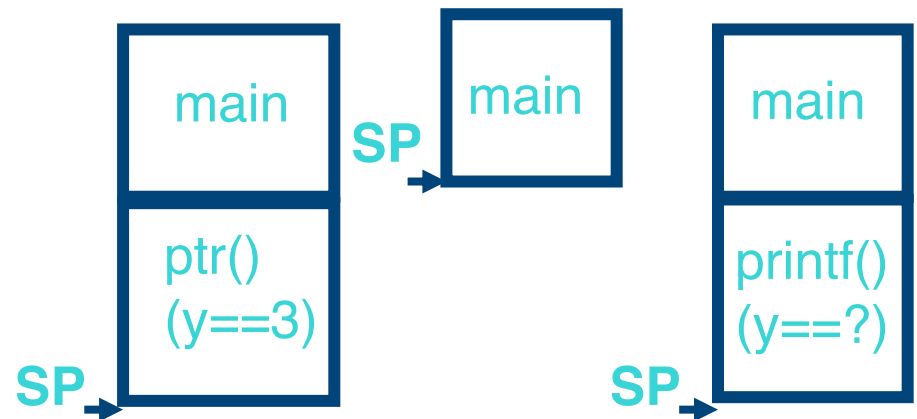
```
main ()
{ a(0);
}
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)
{ d(3);
}
void d (int p)
{
}
```



Quem gere a pilha ?

- ◆ Os ponteiros em C permitem-nos aceder a zonas de memória que foram entretanto desalocadas. Isto pode levar a problemas de consistência e bugs difíceis de encontrar !

```
int *ptr () {  
    int y;  
    y = 3;  
    return &y;  
};  
  
main () {  
    int *stackAddr, content;  
    stackAddr = ptr();  
    content = *stackAddr;  
    printf("%d", content); /* 3 */  
    content = *stackAddr;  
    printf("%d", content); /*13451514 */  
};
```



A Heap (Memória Dinâmica)

- ◆ Grande bloco de memória, onde a alocação não é feita de forma contígua. É uma espécie de "espaço comunal" do programa.
- ◆ Em C, é necessário especificar o número exato de bytes que se pretende alocar

```
int *ptr;  
ptr = (int *) malloc(sizeof(int));  
/* malloc returns type (void *),  
so need to cast to right type */
```

□ `malloc()`: aloca memória não inicializada na área da heap

Características das diferentes zonas de memória

◆ Variáveis estáticas

- ❑ Espaço de memória acessível a partir de qualquer zona do programa
- ❑ O espaço de memória permanece alocado durante todo o "runtime" (pouco eficiente)

◆ Pilha/Stack

- ❑ Guarda variáveis locais, endereços de retorno, etc.
- ❑ A memória é desalocada sempre que uma rotina termina, permitindo a re-utilização por um novo procedimento.
- ❑ Funciona como o "bloco de notas" das funções/procedimentos
- ❑ Não é adequada para armazenar dados de grandes dimensões (stack overflow)
- ❑ Não permite a partilha de dados entre diferentes procedimentos

Características das diferentes zonas de memória

◆ Heap / Alocação dinâmica

- Alocação em "runtime" de blocos de memória
- A alocação não é contígua, e os blocos podem ficar muito distantes no espaço de endereçamento
- Em C, a desalocação tem que ser feita de forma explícita pelo programador (no Garbage Collector)
- Os mecanismos de gestão de memória são complexos de forma a evitar a fragmentação

Gestão de Memória

◆ Como é feita a gestão de memória?

- Zona do código e variáveis estáticas é fácil:

estas zonas nunca aumentam ou diminuem

- O espaço da pilha também é fácil:

As "stack frames" são criadas e destruídas usando uma ordem last-in, first-out (LIFO)

- Gerir a heap já é mais complicado:

a memória pode ser alocada / desalocada em qualquer instante

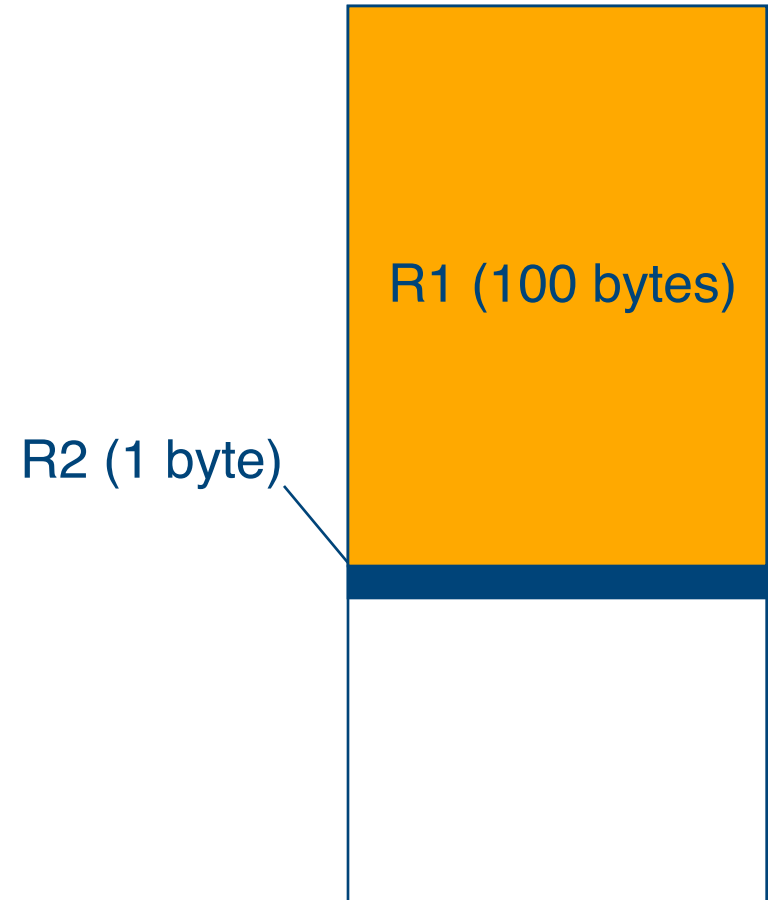
Requisitos da Gestão da Heap

- ◆ As funções `malloc()` e `free()` devem executar rapidamente.
- ◆ Pretende-se o mínimo de overhead na gestão de memória
- ◆ Queremos evitar *fragmentação (externa)** – quando a maior parte da memória está dividida em vários blocos pequenos
 - Neste caso podemos ter muito bytes disponíveis mas não sermos capazes de dar resposta a uma solicitação de espaço porque os bytes livres não são contíguos.

Gestão da Heap (1/2)

◆ Exemplo

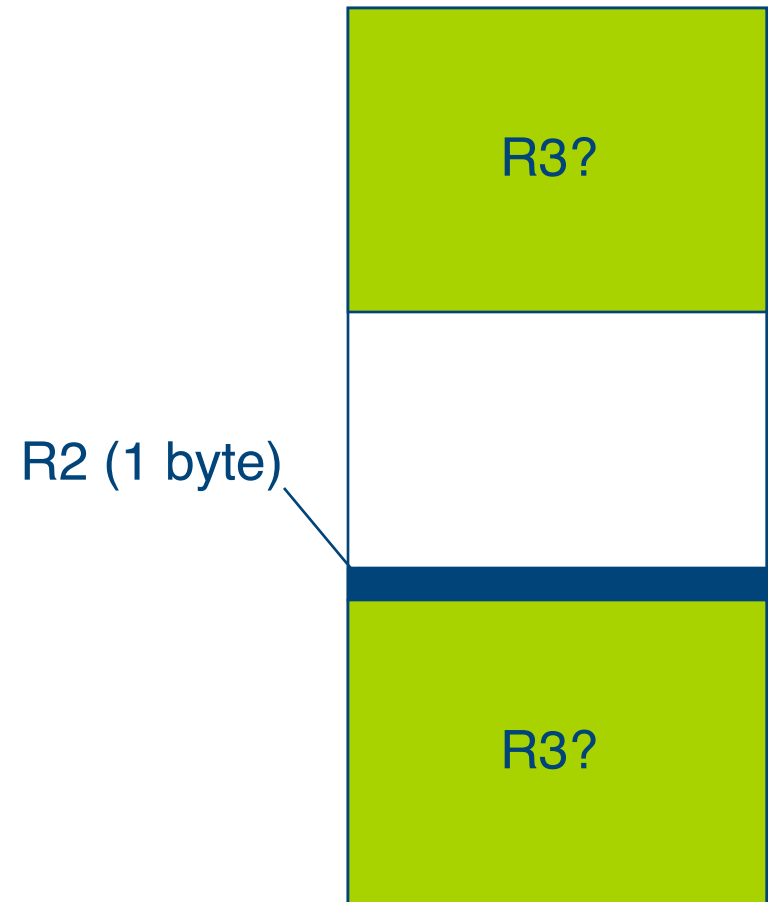
- Request R1 for 100 bytes
- Request R2 for 1 byte
- Memory from R1 is freed
- Request R3 for 50 bytes



Gestão da Heap (2/2)

◆ Exemplo

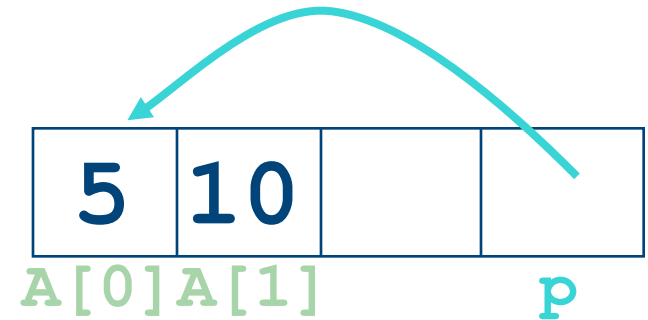
- Request R1 for 100 bytes
- Request R2 for 1 byte
- Memory from R1 is freed
- Request R3 for 50 bytes



QUIZ

```
int main(void){
    int A[] = {5,10};
    int *p = A;

    printf("%u %d %d %d\n",p,*p,A[0],A[1]);
    p = p + 1;
    printf("%u %d %d %d\n",p,*p,A[0],A[1]);
    *p = *p + 1;
    printf("%u %d %d %d\n",p,*p,A[0],A[1]);
}
```



Se o primeiro printf mostrar 100 5 5 10, qual será o output dos outros dois printf ?

- 1: 101 10 5 10 then 101 11 5 11
- 2: 104 10 5 10 then 104 11 5 11
- 3: 101 <other> 5 10 then 101 <3-others>
- 4: 104 <other> 5 10 then 104 <3-others>
- 5: Um dos dois printf's causa um ERROR
- 6: Rendo-me!



Linguagem C

- Gestão da Memória Dinâmica-



Sistemas de Microprocessadores 2021/2022

Mecanismos de Gestão da Heap

- ◆ Alocação Dinâmica "Manual" - Caso do C, em que o programador é responsável por alocar e libertar os blocos de memória
 - malloc()/free() implementação do K&R Sec 8.7 (ler só introdução)
 - Slab Allocators
 - Buddy System
- ◆ Alocação "Automática" / Garbage Collectors - O sistema mantém registo de forma automática das zonas da heap que estão alocadas e em uso, reclamando todas as restantes*
- Contagem de referências
- Mark and Sweep
- Copying Garbage Collection

* O overhead com Garbage Collectors é obviamente maior

Implementação do malloc/free (K&R Sec. 8.7)

- ◆ Cada bloco de memória na heap tem um cabeçalho com dois campos:
 - **tamanho** do bloco e
 - um **ponteiro** para o bloco livre seguinte
- ◆ Todos os **blocos livres** são mantidos numa lista ligada circular (a "free list").
- ◆ Normalmente os blocos da "free list" estão por ordem crescente de endereços no espaço de endereçamento
- ◆ No caso de um bloco ser alocado, o seu ponteiro fica NULL.

Implementação do malloc/free (K&R Sec. 8.7)

- ◆ `malloc()` procura na "free list" um bloco que seja suficientemente grande para satisfazer o pedido.
 - Se existir, então bloco é partido de forma a satisfazer o pedido, e a "sobra" é mantida na lista.
 - Se não existir então é feito um pedido ao sistema operativo de mais áreas de memória.
- ◆ `free()` verifica se os blocos adjacentes ao bloco libertado também estão livres.
 - Se sim, então os blocos adjacentes são juntos (**coalesced**) num único bloco de maiores dimensões (evitar fragmentação)
 - Se não, o bloco é simplesmente adicionado à "free list".

Qual é o bloco que o `malloc()` escolhe?

◆ Se existirem vários blocos na "free list" que satisfaçam os requisitos, qual deles é escolhido?

- **best-fit**: escolhe o bloco mais pequeno que satisfaça os requisitos de espaço
- **first-fit**: Escolhe o primeiro bloco que satisfaça os requisitos
- **next-fit**: semelhante ao first-fit, mas lembra-se onde terminou a pesquisa da última vez, e retoma-a a partir desse ponto (não volta ao início)

QUIZ - Prós e Contras dos "Fit"

- A. Um contra do **first-fit** é que resulta em vários **pequenos blocos** no início da free list
- B. Um contra do **next-fit** é que é **mais lento** do que o **first-fit**, dado que demora mais tempo à procura de um bloco adequado
- C. Um contra do **best-fit** é que **gera muitos blocos** de pequenas **dimensões** na free list




	ABC
0 :	FFF
1 :	FFT
2 :	FTF
3 :	FTT
4 :	TFF
5 :	TFT
6 :	TTF
7 :	TTT

Slab Allocator (1/2)

- ◆ Um sistema alternativo utilizado na GNU `libc`
- ◆ Divide os blocos que formam a heap em "grandes" e "pequenos". Os "grandes" são geridos através de uma freelist como anteriormente
- ◆ Para blocos pequenos, a alocação é feita em blocos que são múltiplos de potências de 2
 - e.g., se o programa quiser alocar 20 bytes, dá-se-lhe 32 bytes.

Slab Allocator (2/2)

- ◆ A gestão dos pequenos blocos é fácil; basta usar um *bitmap* para cada gama de blocos do mesmo tamanho

16 byte blocks:		16 byte block bitmap: 11011000
32 byte blocks:		32 byte block bitmap: 0111
64 byte blocks:		64 byte block bitmap: 00

- ◆ Os bitmaps permitem minimizar os overheads na alocação de blocos pequenos (mais frequentes)
- ◆ As desvantagens do esquema são
 - ❑ Existem zonas alocadas que não são utilizadas (caso dos 32 bytes para 20 pedidos)
 - ❑ A alocação de blocos grandes é lenta

Fragmentação Externa vs Interna

- ◆ Com o slab allocator, a diferença entre o tamanho requisitado e a potência de 2 mais próxima faz com que se desperdice muito espaço
 - e.g., se o programa quer alocar 20 bytes e nós damos 32 bytes, então há 12 bytes que não são utilizados
- ◆ Repare que isto não é *fragmentação externa*. A fragmentação externa refere-se ao espaço desperdiçado entre blocos alocados.
- ◆ Este problema é conhecido por *fragmentação interna*. Trata-se de espaço desperdiçado dentro de um bloco já alocado.

Buddy System (1/2)

- ◆ Outro sistema de gestão de memória usado no kernel do Linux.
- ◆ É semelhante ao “slab allocator”, mas só aloca blocos em tamanhos que são potência de 2 (fragmentação interna é ainda possível)
- ◆ Mantém free-lists separadas para cada tamanho
 - e.g., listas separadas para 16 byte, 32 byte, 64 byte, etc.

Buddy System (2/2)

- ◆ Se não há um bloco de tamanho n disponível, então procura um bloco de tamanho $2n$ e divide-o em dois blocos de tamanho n
- ◆ Quando o bloco de tamanho n é liberto, então, se o vizinho (buddy) estiver também livre, os dois são combinados num bloco de $2n$



- ◆ Tem as mesmas vantagens de velocidade que o slab

Esquemas de Alocação

◆ Qual destes sistemas é o melhor?

- ❑ Não existe um esquema que seja melhor para toda e qualquer aplicação
- ❑ As aplicações têm diferentes padrões de alocação/desalocação.
- ❑ Um esquema que funcione bem para uma aplicação, poderá não funcionar bem para outra.

Gestão automática de memória

- ◆ É difícil gerir e manter registos das alocação/desaloções de memória – porque não tentar fazê-lo de forma automática?
- ◆ Se conseguirmos saber em cada instante de runtime os blocos da heap que estão a ser usados, então todo o espaço restante está livre para alocação.
 - A memória que não está a ser apontada chama-se *garbage* (é impossível aceder-lhe). O processo de a recuperar chama-se *garbage collection*. No C a recuperação/libertação de memória tem que ser feita manualmente
- ◆ Como conseguimos saber o que está a ser usado?

Manter Registo da Memória Utilizada

- ◆ As técnicas dependem da linguagem de programação utilizada e precisam da ajuda do compilador.
- ◆ Pode começar-se por manter registo de todos os ponteiros, definidos tanto como variáveis globais ou locais (root set). (para isto o compilador tem de colaborar)
- ◆ **Ideia Chave:** Durante o runtime mantém-se registo dos objetos dinâmicos apontados por esses ponteiros.
 - À partida um objeto que não seja apontado por ninguém é garbage e pode ser desalocado.

Manter Registo da Memória Utilizada

- ◆ Mas o problema não é assim tão simples ...
 - O que é que acontece se houver um type cast daquilo que é apontado pelo ponteiro? (permitido pelo C)
 - O que acontece se são definidas variáveis ponteiro na zona alocada?
- ◆ A pesquisa de garbage tem de ser sempre feita de forma recursiva.
- ◆ Não é um mecanismo simples e envolve sempre maiores overheads do que a gestão manual
- ◆ Os "Garbage Collectors" estão fora do nosso programa, mas os alunos interessados poderão consultar o material suplementar fornecido no InfoEstudante.

Concluindo ...

◆ O C tem 3 zonas de memória:

- Armazenamento estático: variáveis globais
- A Pilha: variáveis locais, parâmetros, etc
- A heap (alocação dinâmica): `malloc()` aloca espaço, `free()` liberta espaço.

◆ Várias técnicas para gerir a heap via malloc e free: best-, first-, next-fit

- 2 tipos de fragmentação de memória: interna e externa; todas as técnicas sofrem com pelo menos uma delas
- Cada técnica tem pontos fortes e fracos, e nenhuma é melhor para todos os casos

◆ A gestão automática de memória liberta o programador da responsabilidade de gerir a memória. O preço é um maior overhead durante a execução.

Para saber mais ...



- ◆ Hilfiger Notes (fornecidas na InfoEstudante)
- ◆ Artigo a explicar a divisão de memória no C (atenção dividem a zona estática em inicializada e não inicializada)
 - <http://www.informit.com/articles/article.aspx?p=173438>
- ◆ A Wikipedia ao nosso serviço
 - http://en.wikipedia.org/wiki/Dynamic_memory_allocation
 - [http://en.wikipedia.org/wiki/Garbage_collection_\(computer_science\)](http://en.wikipedia.org/wiki/Garbage_collection_(computer_science))