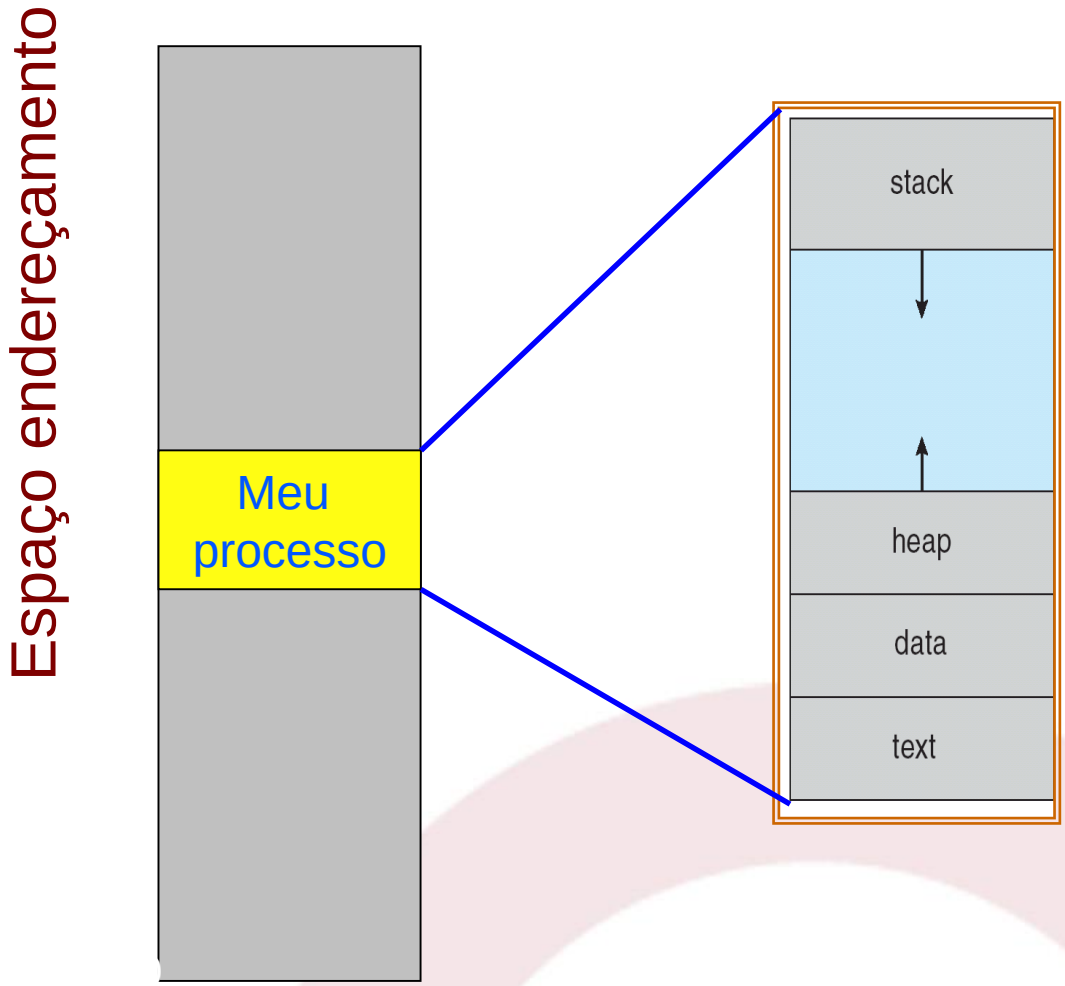
Sistemas Operativos
Capítulo 4
Ponteiros e Memória

Patrício Domingues
ESTG/IPLeiria
2019

Imagem de um processo em memória

- ✓ A imagem de um processo em memória é composta por vários segmentos
  - Segmento de texto
  - Segmento de dados
  - Segmento "heap"
  - Segmento "stack"
- ✓ Designa-se por imagem *virtual*
  - Associada ao mecanismo de memória virtual



Segmento de *heap*

- Segmento "heap"
  - Zona da memória dinâmica
    - De onde provém a memória alocada
    - Programador obtém memória através de funções apropriadas
    - API da linguagem C
      - malloc() , calloc(), realloc() para reservar memória
      - free() para libertar (devolver ao SO)
    - Segmento de tamanho variável
      - Varia consoante os pedidos de alocação /libertaçãode memória
- Significado de *heap*
  - *"an untidy collection of objects placed haphazardly on top of each other"*

Ponteiros na linguagem C

- A linguagem C fica a meio caminho entre:
  - Assembler (linguagem de baixo nível)
  - Linguagens de alto nível (C#, JAVA, etc.)

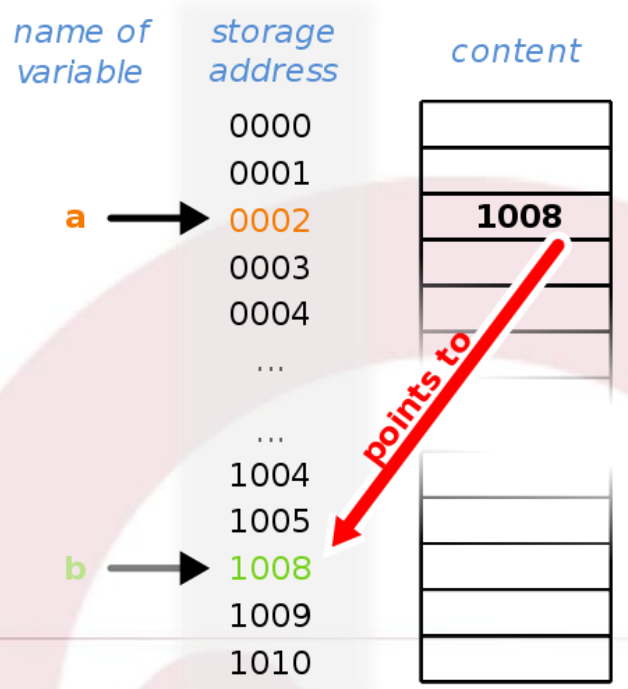
  NOTA: algumas linguagens são designadas por linguagens de muito alto nível (PERL, PYTHON, RUBY)

- Os ponteiros são uma parte importante da linguagem C
  - Saber programar em C é também perceber de ponteiros
  - Sem ponteiros... NADA FEITO!

https://xkcd.com/371/

O que é um ponteiro? (1)

- *"A pointer is a variable that contains the address of a variable"* [Kernighan and Ritchie, 1988]
- Ponteiro
  - Mecanismo para a manipulação direta da memória
  - Uma variável ponteiro contém o endereço de memória onde se encontra um determinado dado ou estrutura
  - Ponteiro aponta para um determinado dado/estrutura
    - significa que a variável ponteiro contém o endereço de memória da memória onde está guardado o dado/estrutura apontada



https://commons.wikimedia.org/wiki/File:Pointers.svg

O que é um ponteiro? (2)

- Cada ponteiro aponta para um determinado tipo de dado (int, double, char, struct, etc.)
- A declaração de um ponteiro é feita precedendo-se o nome da variável ponteiro por *

```
int *Ptr1;
char *Ptr2;
double *Ptr3;
...
```

- Declaração de um ponteiro apenas reserva o **espaço para a variável ponteiro**
  - ERRADO
    - Não existe espaço reservado
    - ptr não aponta para nenhum espaço de memória reservado

```
char *ptr;
strcpy(ptr, "errado!");
```

  - CERTO
    - O vetor *ptr* tem um tamanho de 16 octetos
      - Um octeto deve ser empregue para o '\n'
      - Octeto = byte

```
char ptr[16];
strcpy(ptr, "certo!");
```

O que é um ponteiro? (3)

- Como colocar um ponteiro a apontar para uma variável?
  - Uso do operador &
  - **&**XPTO devolve o endereço de memória da variável XPTO
- Exemplo

```
int *Ptr1;
int A = 20;
Ptr1 = &A;
```

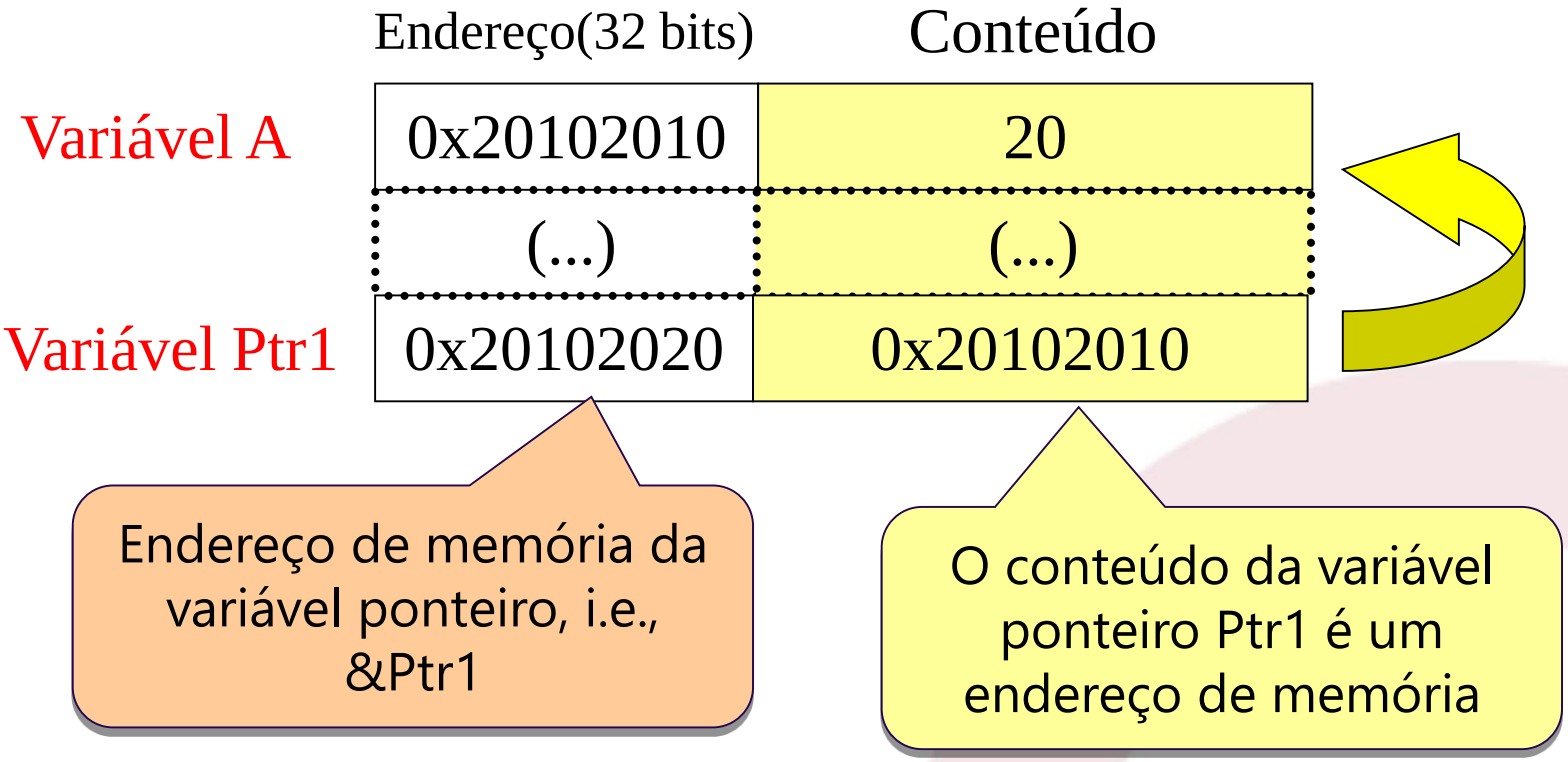
- **Pergunta**
  - Qual é o conteúdo da variável Ptr1?
- **Resposta**
  - o endereço de memória da variável A

```
/* uso do formato "%p" do
   printf para mostrar o
   endereço guardado em Ptr1
   */
printf("Valor da
   variável ponteiro:
   %p, endereço da
   variável A: %p",
   Ptr1, &A);
```

O que é um ponteiro? (4)

✓ Exemplo

```
int *Ptr1;
int A = 20;
Ptr1 = &A;
```



Representação na memória

- ✓ A memória RAM é endereçada por byte
  - A memória pode ser vista como uma sequência de bytes, cada byte tendo um endereço diferente

Endereços crescentes

0x20102010	B1
0x20102011	B2
0x20102012	B3
0x20102013	B4

Um valor de 32 bits composto por 4 bytes (B1, B2, B3 e B4)

- Se um "int" tiver 32 bits, será representado através de 4 bytes

B4	B3	B2	B1

- Um "double" é representado através de 8 bytes (sizeof(double)=8)

B8	B7	B6	B5	B4	B3	B2	B1

Que tamanho tem uma variável ponteiro?

- ✓ Uma variável ponteiro guarda endereços

  - O tamanho do ponteiro deve ser suficiente para guardar endereços
  - Um endereço é um valor inteiro
  - O tamanho de um endereço depende da arquitetura e do SO
    - Se os endereços tiverem 32 bits, então uma variável ponteiro deverá ter 32 bits (e.g. X86)
    - Se os endereços tiverem 64 bits, então uma variável ponteiro deverá ter 64 bits (e.g., AMD64)
  - Como saber?

```
char *Ptr;
```

B8	B7	B6	B5	B4	B3	B2	B1

sizeof(Ptr): devolve o número de bytes da variável ponteiro Ptr, isto é, devolve o número de bytes de um endereço na máquina local

- Qual é o resultado de sizeof(*Ptr)?

O que é um ponteiro? (5)

- Ao aceder-se diretamente a uma variável ponteiro, obtém-se o valor que ela contém, i.e., o endereço para onde aponta
- Como obter o valor apontado, isto é, o valor do endereço de memória que é apontado pelo ponteiro?
  - Operador de dereferenciação *
    - Também designado por operador de indireção pelo facto de requerer um segundo acesso (acesso indireto)

Exemplo

```
double pi = 3.1415; double Tmp;
double *Ptr2;
// Ptr2 aponta para pi
Ptr2 = &pi;
printf("Valor apontado por
    Ptr2=%f\n", *Ptr2);
// Tmp passa a ter o valor 3.1415
Tmp = *Ptr2;
// A variável "pi" passa a ter o
// valor 2. Porquê?
*Ptr2 = 2.0;
```

Lembrete

- Um ponteiro é uma variável que contém o endereço de outra variável
- Um ponteiro está associado a um tipo de dado, só podendo apontar para variáveis desse tipo
  - Exceção é void*
    - Ponteiro sem tipo
    - Requer um *cast* para ser empregue numa operação de indireção
- O operador **&** devolve o endereço de uma variável
- O operador * permite aceder ao valor apontado pelo ponteiro
  - O valor é interpretado como sendo do tipo de dado do ponteiro

Ponteiros e passagem de parâmetros

- Por omissão, na linguagem C, os parâmetros de funções são passados por valor
- Ponteiros podem ser empregues para passagem de parâmetros por referência
- Exemplo – função **Triple** e função main

- Resultado da execução:

```
void Triple(double param1, double *paramPtr){
        *paramPtr = param1 * 3.0;
        param1 = param1 * 3.0;
}
int main(void){
        double A = 10.0;
        double Result;
        Triple(A, &Result);
        printf("A=%lf, Result=%lf\n", A, Result);
        return 0;
}
```

- Resultado da execução:

```
A=10.0, Result = 30.0
```

Vetores (arrays) e ponteiros

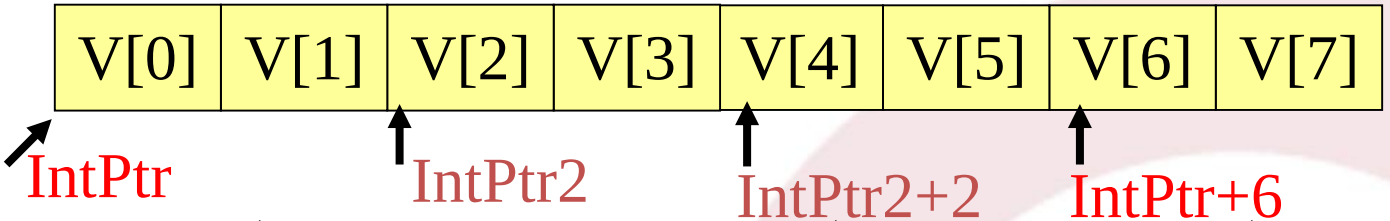
- Na linguagem C, os *arrays* estão ligados aos ponteiros
  - O nome de um *array* é um ponteiro para o primeiro elemento do *array*
    - i.e., o nome de um *array* contém o endereço do 1º elemento do array
  - O *array* é guardado sequencialmente na memória

Exemplo

```
int Array1[24];
// Ptr p/ inteiro pq array
// é int
int *FirstElemPtr;
// FirstElemPtr aponta
// p/ 1° elem
FirstElemPtr = &Array1[0];
// O mesmo que a linha
//anterior
FirstElemPtr = Array1;
```

Aritmética de ponteiros (1)

- A linguagem C permite a denominada **aritmética** de ponteiros
  - Somar um valor inteiro **N** a um ponteiro resulta num endereço que corresponde ao deslocamento em **N** posições para a frente do endereço original
  - O deslocamento corresponde a somar **N * sizeof(tipo_de_dado)** ao endereço do ponteiro
- **Exemplo**

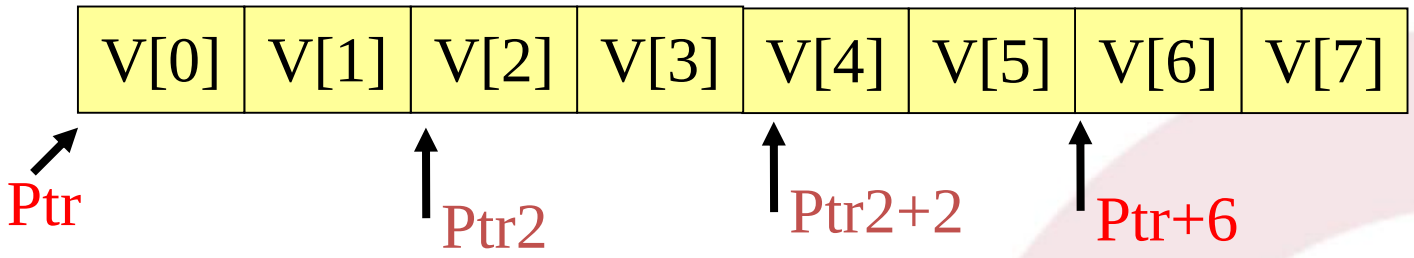



```
int V[8];
int *IntPtr = &V[0]; /* IntPtr aponta p/ 1° end. V */
int *IntPtr2 = IntPtr+2; /*IntPtr2 aponta para IntPtr[2]*/
```

Aritmética de ponteiros (2)

- ✓ E se forem ponteiros de “double”?
  - – Funciona de forma similar
- ✓ **Exemplo**

```
double V[8];
double *Ptr = &V[0]; /* Ptr aponta p/ 1° end. V */
double *Ptr2 = Ptr + 2;/* O mesmo que Ptr2=&V[2] */
```



Slide seguinte: exercício

Artimética de ponteiros - exercício

- **O que faz este código?**

```
double V[8];
double *Ptr = V; /* Ptr aponta p/ 1° end. V */
while( Ptr < &V[8] ){
   printf("%f\n", *Ptr);
  Ptr++;
}
```

O que faz este código?

Strings e ponteiros

- Uma string é um vetor de caracteres (“char”)
- O nome de uma variável string é um ponteiro para o 1º elemento da string (char *)
- O fim de string é assinalado através do byte 0 (todos os 8 bits estão a zero)
  - Pode ser representado pelo caracter ‘\0’
- No C, um **char** tem tamanho de um byte.

Exemplo

```
// Str: max. 23 caract. + '\0'
char Str[24];
Str[0]='a';
Str[1]='b';
Str[2]='c';
Str[3]='\0';
// o mesmo que a linha
// anterior
strcpy(Str, "abc");
char *StrPtr;
StrPtr = Str;
// alias para a string Str
printf("StrPtr='%s'\n",
  StrPtr);
/* o mesmo que
  printf("Str:'%s'\n",
  Str); */
```

Carácter vs. String (#1)

- Distinção entre ‘a’ e “a” na linguagem C
  - ‘a’ representa um caracter
- Exemplo
  - `char var_c = 'a'`
- Qual é o valor de `var_c`?
  - `var_c` fica com o valor numérico correspondente ao código ASCII da letra ‘a’
    - Valor 97 (base 10) ou 60 (base hexadecimal)

- Tabela ASCII
  - Caracteres e respetivo código numérico
  - Utilitário `ascii` do Unix

ASCII TABLE

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

http://bit.ly/2mARsC9

Carácter vs. String (#2)

```
#include <stdio.h>
int main(void){
        unsigned char c;
        c = 32;
        while( c <= 127 ){
            printf("'%c':'%u'\n",
              c,(unsigned int)c);
            c++;
        }
        return 0;
}
```

Português Europeu
Carácter / caracteres
OU
caráter / carateres
Fonte: http://bit.ly/2ntBNSN

- Código mostra tabela ASCII de 32 a 127
- Código 32: 1º caracter "printável"
- < 32 – caracteres de controlo
  - 8 ou 0x8: '\t'
  - 10 ou 0xA: '\n'
  - 13 ou 0xC: '\r'
  - …

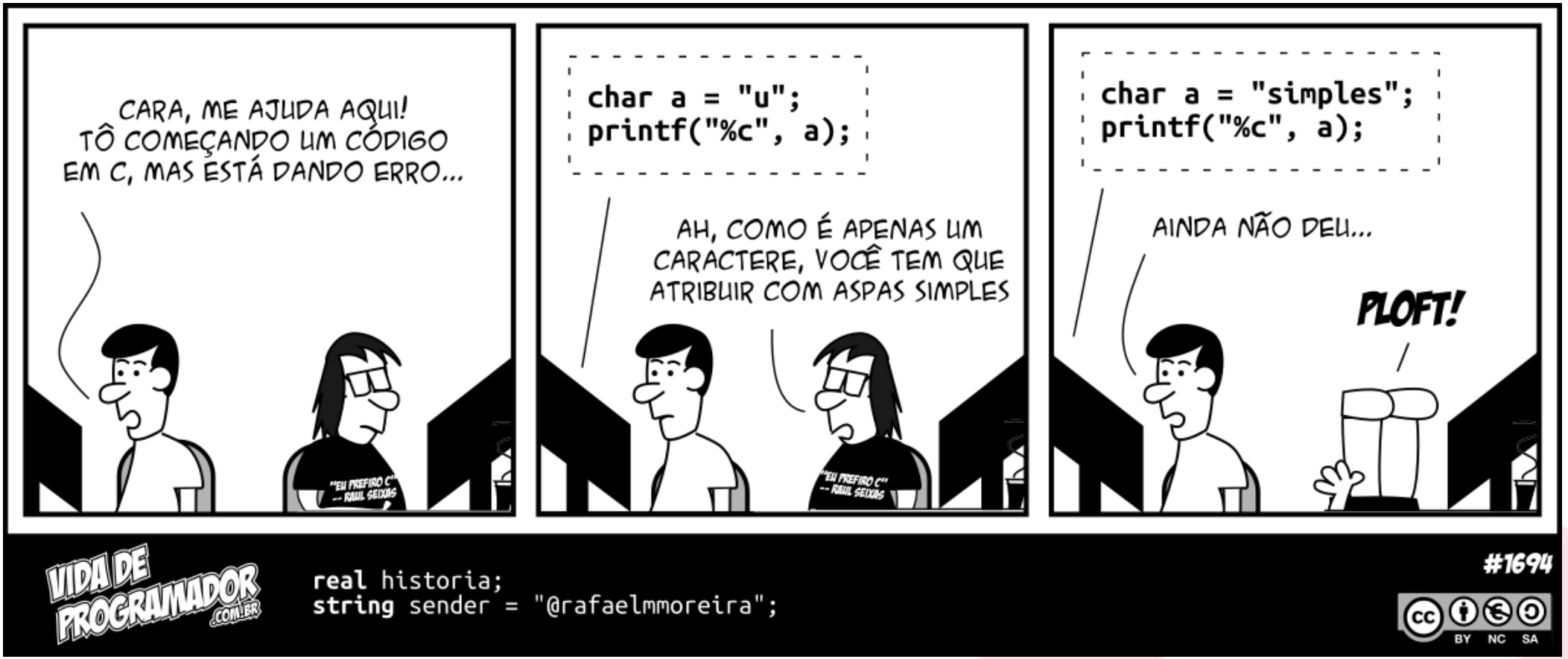
Carácter vs. String (#3)

- String “a”
  - Vetor com dois elementos
    - caracter ‘a’ e caracter fim de string ‘\0’
- Exemplo
  - `char *str1 = "abc";`
  - String constante str com o conteúdo ‘a’,’b’,’c’,’\0’

```
char str2[4];
str2 = “abc”; // errado!

strcpy(str2,”abc”);  // certo!
```

- Comparação do **conteúdo** de duas strings
  - `strcmp(str1,str2)`
    - 0 se as strings forem iguais
    - -1 se str1 < str2 (ordem alfabética)
    - 1 se str1 > str2 (ordem alfabética)
- Não é: `str1==str2` //errado
  - Compara os endereços para onde aponta os ponteiros



http://vidadeprogramador.com.br/2017/05/09/aspas-simples/

Strings e aritmética de ponteiros (1)

- ✓Somar uma unidade a um vetor de caracteres desloca o ponteiro para o caracter (e byte) seguinte
- ✓**Exemplo**

```
/*Calcular o tamanho de uma string (strlen) com aritmética de ponteiros */
int MyStrLen(char *str){
        char *Ptr = str;
        int Len = 0;
        while( *Ptr != '\0'){
                Len++;
                Ptr++;      /* Ptr = Ptr +1   Vai para o caracter seguinte */
        }
        return Len;
}
```

Strings e aritmética de ponteiros (2)

✓**Exemplo** – converter uma string para maiúsculas

```
// Converte a string str para maiúscula.  Devolve ptr para a string
char* ParaUpper(char *str){
  char *Ptr = str;
            while( *Ptr != '\0'){
                        *Ptr = (char)toupper(*Ptr);
                        Ptr++;      /* Ptr = Ptr +1   Vai para o caracter seguinte */
            }
  return str;
}
...
char S[24];
sprintf(S,"%s","teste!\n");
printf("Upper: '%s'\n", toupper(S));
```

Vetor de ponteiros

- Um vetor de ponteiros contém... ponteiros
  - Um ponteiro está associado a um tipo de dados
  - Assim, um vetor de ponteiros contém ponteiros para um determinado tipo de dados

Exemplo

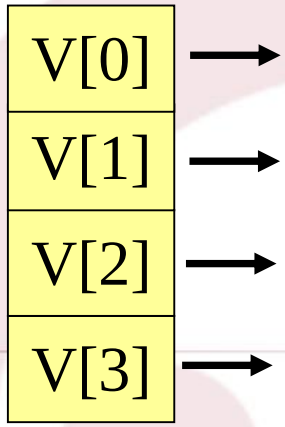
- Vetor com 10 ponteiros para double

double *VecPtr1[10];

- Vetor com 4 ponteiros para string

char *VecStr[4];

Nota: após a declaração os ponteiros **não** apontam para nada!



Vetor de strings “argv”

- Passagem de parâmetros da linha de comando
  - No C, os parâmetros passados pela linha de comando estão disponíveis através de um vector para strings definido pelo C e acessível como parâmetro da função main.
    - char *argv[ ]

- `int main(int argc, char *argv[])`
  - **`int argc`** `indica o número de elementos do vector de strings`
- O que faz o seguinte código?

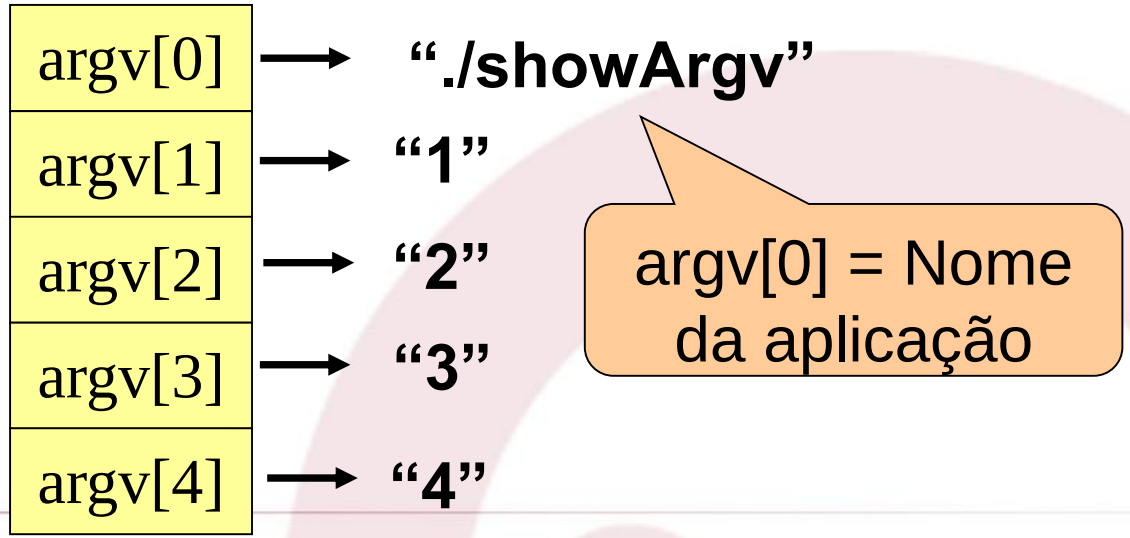
```
int main(int argc, char *argv[]){
    int i;
    for(i=0; i<argc; i++){
        printf(“argv[%d]:’%s’\n”, i, argv[i]);
    }
}
```

Resposta >>

char *argv[]

```
int main(int argc, char *argv[]){
        int i;
        for(i=0; i<argc; i++){
                printf("argv[%d]:'%s'\n", i, argv[i]);
        }
}
```

```
./showArgv 1 2 3 4
argv[0]:'./showArgv'
argv[1]:'1'
argv[2]:'2'
argv[3]:'3'
argv[4]:'4'
```



Vetor de strings “argv”

- `int main(int argc, char *argv[])`
  - O parâmetro “char *argv[ ]” também pode ser escrito “char **argv”
  - O que significam os dois asteríscos? **
    - Ponteiro para...ponteiro de caracteres

  - Não esquecer
    - A passagem de um vetor via parâmetro numa função é feito via ponteiro
    - Nome de um vetor representa o endereço do vector
    - O nome de um vetor é um ponteiro para o primeiro elemento do vetor
      - Neste caso, “argv” é um ponteiro para ponteiro de caracteres
    - Assim...

`char **argv <=> char *argv[]`

Onde estão as variáveis?

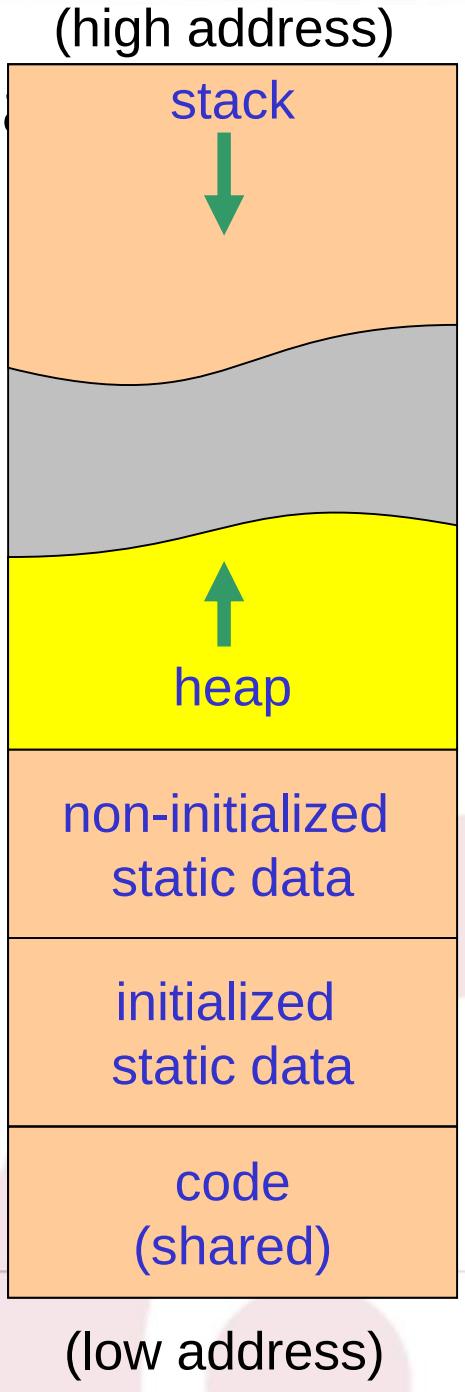
- Uma variável é um espaço de memória com *N* bits
  - char A;
    - Espaço de memória para 8 bits
  - int X;
    - Espaço de memória para sizeof(int)*8 bits
- Em que zona da memória é guardada uma variável?
- Variável global/static
  - Segmento DATA
- Variável local (ou automática)
  - Segmento *stack*
- Variável alocada
  - Segmento *heap*

Memória dinâmica (1)

- Principais características
  - É empregue consoante os pedidos da aplicação
    - A aplicação pode solicitar blocos de memória adequados às necessidades
    - Daí a designação "dinâmica"
  - É disponibilizada em blocos
    - Aplicação solicita memória para guardar um determinado tipo de dados *N* vezes
  - A linguagem C disponibiliza funções para alocar/realocar/libertar blocos de memória dinâmica
  - É manipulada através do endereço do bloco
    - Endereço do 1º octeto do bloco
    - Ao nível da linguagem C, o uso de endereços requer a utilização de ponteiros
    - Aplicação deve libertar memória quando já não precisa dela

Memória dinâmica (2)

- ✓ A gestão da memória dinâmica é feita pelo programador
  - Alocar memória: "**malloc**"
  - Libertar memória (previamente alocada): "**free**"
- ✓ A memória dinâmica está associada à secção de *heap* da imagem do processo



.stack segment

segmento .bss

segmento .data

segmento .text

Memória dinâmica (3)

- Importante
  - No C, a memória dinâmica obtida via “malloc” permanece afeta ao processo até que seja explicitamente libertada
    - Quando a aplicação termina, todos os recursos são devolvidos ao SO
  - Um bloco de memória alocado mas não libertado ocupa espaço (*memory leak)*
    - *Fuga de memória*
- Regra
  - Sempre que o programador usa o **malloc** deve pensar onde e quando fará uso do **free** para libertar a memória

Memória dinâmica - funções

- Alocar memória dinâmica

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

  - Devolve endereço de bloco de memória com *size* octetos
  - Memória não é "zerada"

```
void *calloc(size_t nmemb,
size_t size);
```

  - Devolve endereço de bloco de memória com *nmemb * size* octetos
  - Memória é "zerada" (todos os bits do bloco são colocados a zero)

```
void *realloc(void *ptr,
size_t size);
```

- Redimensiona bloco de memória dinâmica *ptr* para o novo tamanho *size*
  - N*ão é garantindo que a operação resulte (e.g., pode já não existir bloco de memória com tamanho suficiente)*
- O ponteiro devolvido deve ser sempre verificado
  - Se for NULL, ocorreu erro na alocação

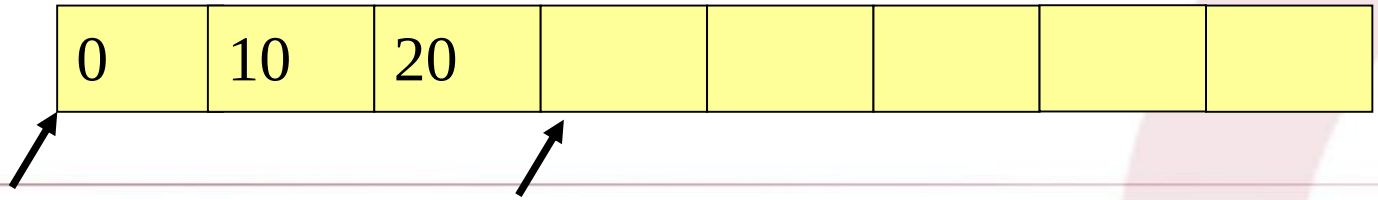
```
void free(void *ptr);
```

- Liberta bloco de memória ptr (previamente alocado)

Memória dinâmica – exemplo 1

✓ Exemplo

```
int *BlockPtr;
/* BlockPtr fica a apontar para início de bloco com
   capacidade para 8 inteiros */
BlockPtr = (int*) malloc (8*sizeof(int));
if( BlockPtr == NULL ){
   fprintf(stderr, "Can't alloc '%d' bytes\n", 8*sizeof(int));
   exit(1);
}
/* As duas próximas linhas fazem o mesmo de forma diferente */
*(BlockPtr+0) = 0; *(BlockPtr+1)=10;*(BlockPtr+2)=20;
BlockPtr[0]=0; BlockPtr[1]=10; BlockPtr[2]=20;
```



Memória dinâmica – exemplo 2

Exemplo – vetor de strings dinâmico

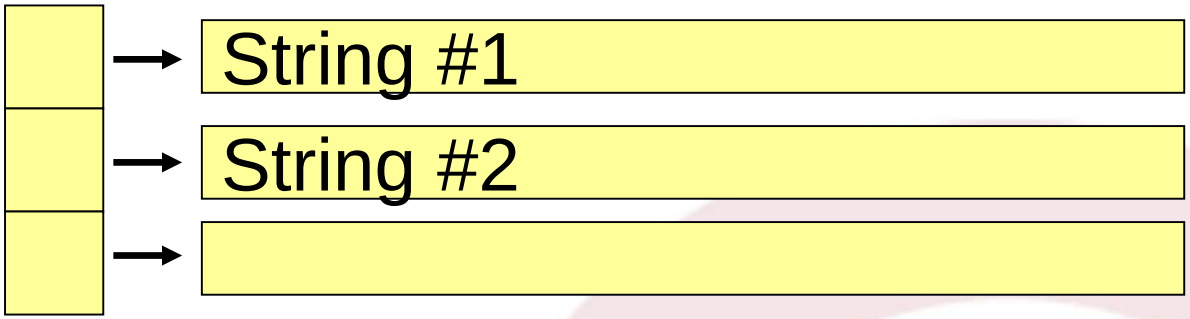
- Pretende-se um vetor de strings via memória dinâmica
  - Cada elemento do vector aponta para uma string
- Duas possíveis implementações
  - **A)** – alocar o vetor de ponteiro para strings + alocar memória para cada string separadamente
  - **B)** – alocar o vetor de ponteiro para strings + alocar bloco de memória para todas as strings

Próximo slide: implementação

Memória dinâmica – exemplo 2

- Solução #1 - alocar o vetor de ponteiro para strings + alocar memória para cada string separadamente

```
#define NUM_STRS          (3)
char **VecStr;
int i;
VecStr = (char**) malloc(NUM_STRS*sizeof(char*));
for(i=0; i<NUM_STRS;i++){
    VecStr[i] = (char*) malloc(64*sizeof(char));
}
strcpy(VecStr[0], “String #1”);
strcpy(VecStr[1], “String #2”);
```

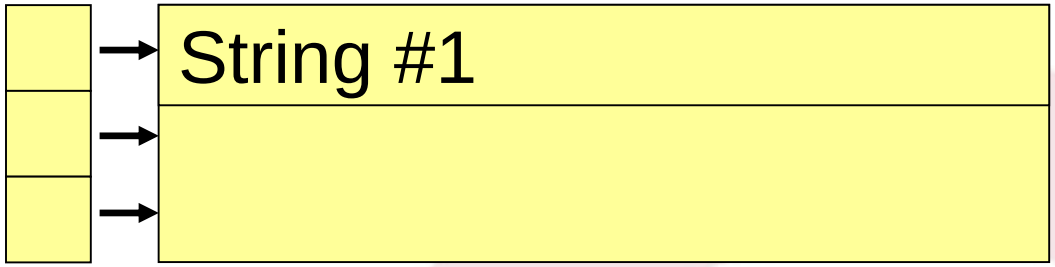


- Pergunta: como libertar a memória?

Memória dinâmica – exemplo 2

- Solução #2 - alocar o vetor de ponteiro para strings + alocar bloco de memória para todas as strings

```
#define NUM_STRS          (3)
char **VecStr;
int i;
VecStr = (char**) malloc(NUM_STRS*sizeof(char*));
BlockPtr = (char*) malloc(NUM_STRS*64*sizeof(char));
for(i=0; i<NUM_STRS;i++){
    VecStr[i] = (BlockPtr+i*64);
}
strcpy(VecStr[0], "String #1");
```



- Pergunta: como libertar a memória?

Memória dinâmica – exemplo 3

- ✓ Matriz dinâmica de inteiros com acesso [i][j]
- ✓ Código

```
int main(void){
        int **vector_2D;
        int *base_block_ptr;
        int n_cols = 10, n_rows = 5;
        int i,j;
        // aloca bloco base: n_cols * n_rows * sizeof(int)
        size_t len1 = n_cols*n_rows*sizeof(int);
        base_block_ptr = malloc(len1);
        if( base_block_ptr == NULL ){
                fprintf(stderr,"Can't alloc %zu bytes\n",len1);
                exit(1);
        }
```

(continua)

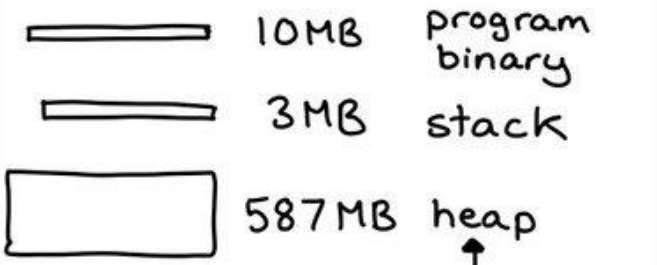
Memória dinâmica – exemplo 3

```
// aloca vetor linhas: n_rows * sizeof(int*)
size_t len2 = n_rows*sizeof(int*);
vector_2D = malloc(len2);
if( vector_2D == NULL ){
        fprintf(stderr,"Can't alloc %zu bytes\n",len2);
        exit(1);
}
for(i=0;i<n_rows;i++){
        vector_2D[i] = base_block_ptr + i * n_cols;
}
printf("vector2D: %d n_rows x %d n_cols\n",
                                        n_rows, n_cols);
for(i=0;i<n_rows;i++){
        for(j=0;j<n_cols;j++){
                vector_2D[i][j] = i*n_cols+j;
                printf("%02d ", vector_2D[i][j]);
        }
        printf("\n");
}
return 0;
}
```

memory allocation

Julia Evans
@b0rk

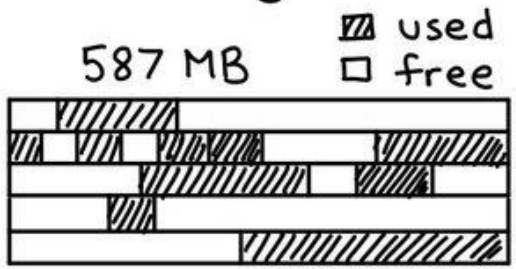
your program has memory



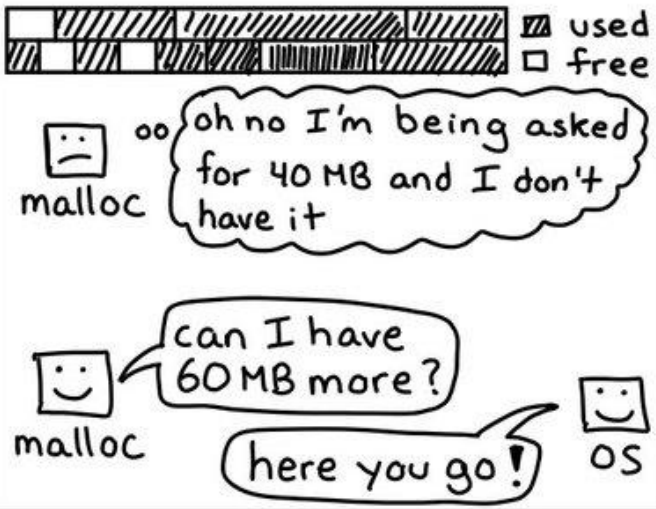
the heap is what your allocator manages

your memory allocator (malloc) is responsible for 2 things.

THING 1: keep track of what memory is used/free



THING 2: Ask the OS for more memory!



your memory allocator's interface

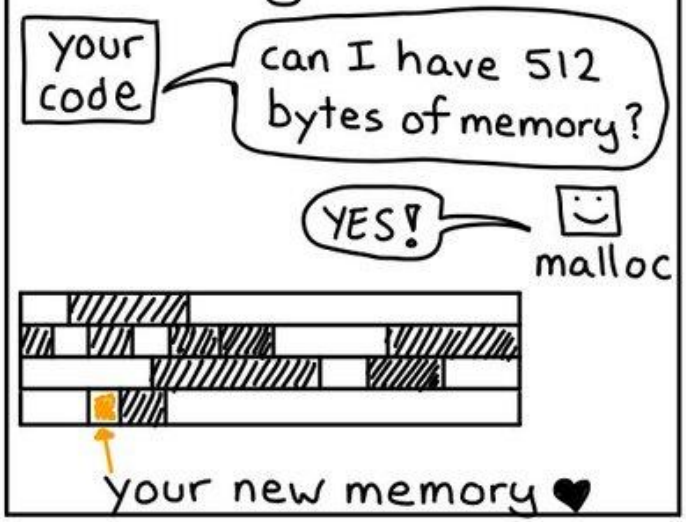
`malloc(size_t size)`
allocate size bytes of memory & return a pointer to it

`free(void* pointer)`
mark the memory as unused (and maybe give back to the OS)

`realloc(void* pointer, size_t size)`
ask for more/less memory for pointer

`calloc(size_t members, size_t size)`
allocate array + initialize to 0

malloc tries to fill in unused space when you ask for memory



malloc isn't magic

it's just a function!

you can always:

- use a different malloc library like jemalloc or tcmalloc (easy!)
- implement your own malloc (harder)

MALLOC_CHECK_

- Variável ambiente MALLOC_CHECK_
  - Permite ativação da depuração do gestor de memória dinâmica da **glibc**
- Valores possíveis
  - 0: sem depuração
  - 1: relatório para saída padrão
  - 2: programa terminado em caso de erro na memória dinâmica

- Exemplo

```
export MALLOC_CHECK_=1
./double_free.exe
*** Error in `./double_free.exe': free(): invalid pointer: 0x08066008 ***
Aborted (core dumped)
```

VLA – *Variable Length Arrays*

- C tradicional (C89) apenas suporta arrays declarados com tamanho que seja conhecido na compilação
- Exemplo
  - int vect_1[10]; // OK
  - int vect_2[n]; // not OK
- A norma C99 (1999) suporta VLA
  - *Variable Length Arrays*
  - Tamanho somente é conhecido durante a execução
- Na norma O C11 (2011), os VLAs passaram a...opcionais
  - Compilador pode ou não aceitá-los...
  - VLA pode comprometer a portabilidade
    - Evitar o uso de VLA

- O espaço de memória empregue pelos VLA fica a cargo do compilador/libc
  - GCC coloca os VLAs na pilha
  - VLA muito grande origina stack overflow....
- Exemplo
  - Suportado na norma C99, opcionalmente na norma C11
    - Valor de ***n*** apenas é conhecido durante a execução

```
void vla_example(int n){
    double vetor[n];
    int i;
    for(i=0;i<n;i++){
            vetor[i]=i*1.0;
    }
}
```

Transbordo de memória (1)

- O que sucede quando se escreve para além da memória que está reservada?
- Exemplo

```
int c[5];
c[5] = -1; // 5 é indice inválido [0]...[4]
```

- Ocorre transbordo de memória
  - Escrita para além dos limites reservados de uma dada zona de memória
  - A memória escrita pode pertencer a...outra variável
  - Uma das principais causas de falhas de segurança em aplicações
  - *Buffer overflow* ou *buffer overrun*

- Perigo quando o endereço de retorno da função é comprometido
  - Endereço de retorno é mantido na pilha
  - Transbordo de memória na pilha pode permitir que o endereço de retorno seja (maliciosamente) alterado
    - Ao terminar a função com endereço de retorno alterado, a execução salta para o endereço que foi alterado...

Exemplo >>

Transbordo de memória (2)

- Exemplo – 2016
  - Problema no código glibc para obter endereço IP a partir de um nome (serviço DNS)
    - www.ipleiria.pt ➔ 194.210.216.8
  - Qualquer software que usa código da *glibc* (v.2.9.x) para aceder ao DNS está vulnerável...
  - Falha permite a **execução remota** de código

Extremely severe bug leaves dizzying number of software and devices vulnerable

Since 2008, vulnerability has left apps and hardware open to remote hijacking.

by **Dan Goodin** - Feb 16, 2016 7:01pm GMT

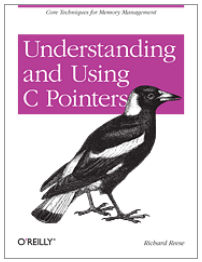
```
[[root@sandbox-3]$ gcc -o client client.c
[[root@sandbox-3]$
[[root@sandbox-3]$ ./client
Segmentation fault (core dumped)
[[root@sandbox-3]$
[[root@sandbox-3]$ wget https://google.com
--2016-02-16 15:51:51--  https://google.com/
Resolving google.com... Segmentation fault (core dumped)
[[root@sandbox-3]$
[[root@sandbox-3]$ curl https://google.com
Segmentation fault (core dumped)
[[root@sandbox-3]$
[root@sandbox-3]$
```

The vulnerability was introduced in 2008 in GNU C Library, a collection of open source code that powers thousands of standalone applications and most distributions of Linux, including those distributed with routers and other types of hardware. A function known as getaddrinfo() that performs domain-name lookups contains a buffer overflow bug that allows attackers to remotely execute malicious code.

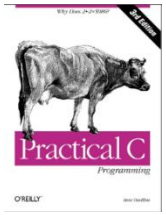
http://bit.ly/1QjZIur

Bibliografia

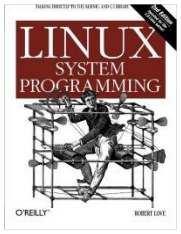
- "Understanding and Using C Pointers - Core Techniques for Memory Management", Richard M. Reese, O'Reilly, 2013.



- "Practical C Programming", Steve Oualline, O'Reilly, 3rd edition, 1998
  - Capítulo 13 e 17: ponteiros



- "Linux System Programming", Robert Love, cap. 9, O'Reilly, 2nd edition, 2013.



- "The Function Pointer Tutorials"
  - http://www.newty.de/fpt/index.html (2010)