

Alocação de Memória

Prof^a. Rose Yuri Shimizu

Roteiro

1 Memória

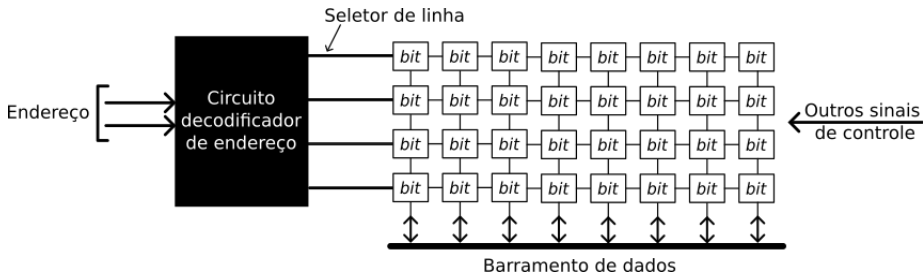
- Variáveis x Endereços
- Ponteiros - manipulação de endereços

2 Processo x Memória

- Alocação estática de memória
- Alocação automática de memória
- Alocação dinâmica de memória

Memória física

- Conjunto de componentes eletrônicos capazes de conservar estados
- Convencionou-se: 1 (alta tensão) e 0 (baixa tensão)
- Computador =
[sistema binário (dados) + álgebra booleana (lógica)] +
circuitos de comutação de estados
- Componente de armazenamento de dados: memória



Memória física : byte x endereço

Endereço

								.
								.
								.
byte	byte	byte	byte	byte	byte	byte	byte	6064
byte	byte	byte	byte	byte	byte	byte	byte	6056
byte	byte	byte	byte	byte	byte	byte	byte	6048
byte	byte	byte	byte	byte	byte	byte	byte	6040
byte	byte	byte	byte	byte	byte	byte	byte	6032
byte	byte	byte	byte	byte	byte	byte	byte	6024
byte	byte	byte	byte	byte	byte	byte	byte	6016
byte	byte	byte	byte	byte	byte	byte	byte	6008
byte	byte	byte	byte	byte	byte	byte	byte	6000
								.
								.
								.

Roteiro

1 Memória

- Variáveis x Endereços
- Ponteiros - manipulação de endereços

2 Processo x Memória

- Alocação estática de memória
- Alocação automática de memória
- Alocação dinâmica de memória

Variáveis x Endereços

- Cada variável possui um endereço na memória
- Endereço = byte menos significativo (início da alocação)

```
int x = 9;
```

$$9_{10} = 1 \cdot 2^3 + 0 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 1001_b$$

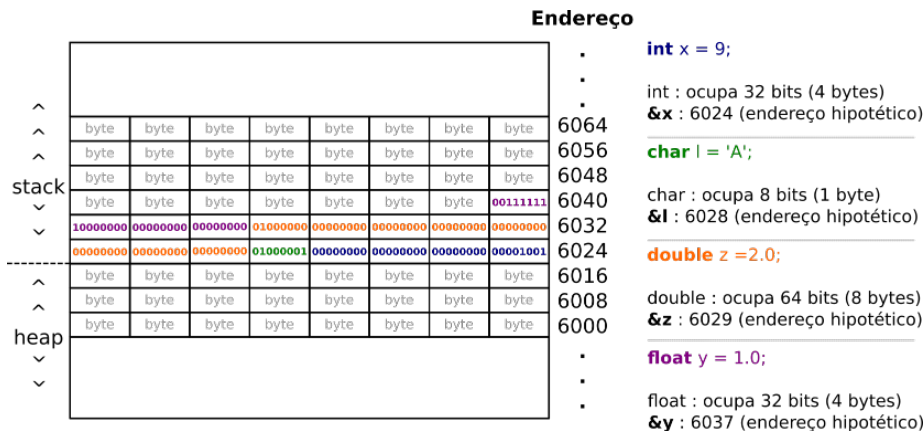
int: ocupa 32 bits (4 bytes)

&x : 6024 (endereço hipotético)

Memória	00000000	00000000	00000000	00001001
Endereço	6027	6026	6025	6024

Variáveis x Endereços

- Alocação contínua



Variáveis x Endereços

endereços	stack	
		int i = 1234;
		printf (" i = %d\n", i);
		//imprimir no formato decimal
		printf ("%i = %ld\n", (long int) &i);
		//imprimir no formato hexadecimal
		printf ("%i = %p\n", (void *) &i);

		Saída
		i = 1234
&i+4 = 0x7fffff2cb4b8		&i = 140737474507956
&i = 0x7fffff2cb4b4	i = 1234	&i = 0x7fffff2cb4b4

Vetor x Endereços

- Cada posição tem um endereço
- Cada posição é calculada a partir do endereço inicial
- Endereço inicial, é apontado pelo identificador (nome) do array

```
int v[2];  
v[0] = 3;  
v[1] = 7;  
  
printf (" endereco de v %ld\n", (long int)v);  
  
printf (" v[0] = %d\n", v[0]);  
printf ("%v[0] = %ld\n", (long int) &v[0]);  
printf ("%v[0] = %p\n", (void *) &v[0]);  
  
printf (" v[1] = %d\n", v[1]);  
printf ("%v[1] = %ld\n", (long int) &v[1]);  
printf ("%v[1] = %p\n", (void *) &v[1]);
```

Saída

endereco de v 140727096456912

v[0] = 3
&v[0] = 140727096456912
&v[0] = 0x7ffd949836d0

v[1] = 7
&v[1] = 140727096456916
&v[1] = 0x7ffd949836d4

endereços

&v[1] = 0x7ffd949836d4

v = &v[0] = 0x7ffd949836d0

stack

v[1] = 7

v[0] = 3

Matriz x Endereços

```
int v[2][2];
v[0][0] = 3;
v[1][0] = 7;

printf ("endereço de v %p\n", (void *)v);
//-----
//endereço da linha 0 (primeiro vetor)
printf ("endereço de v[0] %p\n", (void *)v[0]);

// conteúdo do primeiro elemento do primeiro vetor
//      linha 0 coluna 0
printf (" v[0][0] = %d\n", v[0][0]);

// endereço do primeiro elemento do primeiro vetor
printf ("%p\n", (void *) &v[0][0]);
//-----
//endereço da linha 1 (segundo vetor)
printf ("endereço de v[1] %p\n", (void *)v[1]);

// conteúdo do primeiro elemento do segundo vetor
//      linha 1 coluna 0
printf (" v[1][0] = %d\n", v[1][0]);

// endereço do primeiro elemento do segundo vetor
printf ("%p\n", (void *) &v[1][0]);
```

Saída

```
endereço de v 0x7ffc8d1ff960

endereço de v[0] 0x7ffc8d1ff960
    v[0][0] = 3
    &v[0][0] = 0x7ffc8d1ff960

endereço de v[1] 0x7ffc8d1ff968
    v[1][0] = 7
    &v[1][0] = 0x7ffc8d1ff968
```

endereços	stack
&v[1][1] = 0x7ffc8d1ff96C	v[1][1] = ?
v[1] = &v[1][0] = 0x7ffc8d1ff968	v[1][0] = 7
&v[0][1] = 0x7ffc8d1ff964	v[0][1] = ?
v = v[0] = &v[0][0] = 0x7ffc8d1ff960	v[0][0] = 3

Roteiro

1 Memória

- Variáveis x Endereços
- Ponteiros - manipulação de endereços

2 Processo x Memória

- Alocação estática de memória
- Alocação automática de memória
- Alocação dinâmica de memória

Ponteiros

- Variáveis capazes de armazenar e manipular endereços de memória
- Indicado na declaração da variável pelo **símbolo ***
- Sintaxe: TIPO *ponteiro;
 - ▶ TIPO: indica o tipo de dados da variável que o ponteiro irá apontar
 - ▶ int, float, double, char, struct
- Tamanho dos ponteiros: fixo, depende da arquitetura
- Tipo dos ponteiros: utilizado para desreferenciar (conteúdo) e operações aritméticas
- Pode ser NULL: indica endereço inválido; valor é 0 (zero)

Ponteiros

```
1  int i;  
2  int *p;  
3  p = NULL;  
4  p = &i; //diz-se:  
5         //    p aponta para i  
6         //    p referencia a variavel i  
7  
8  i = 5;  
9  // *p valor da variavel apontada por p, ou seja, valor de i  
10 printf("%d\n", *p); // *p eh igual a i  
11                        //saida: 5  
12 printf("%ld\n", sizeof(p));
```

Ponteiros

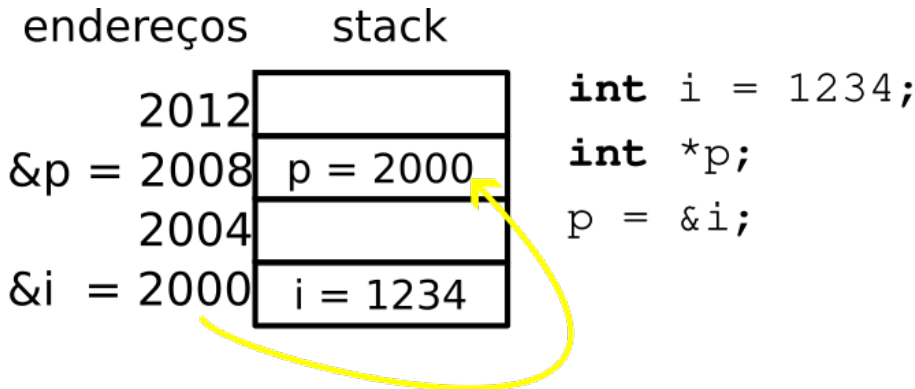
- Aloca i e p
- Conteúdo em i



```
int i = 1234;  
int *p;
```

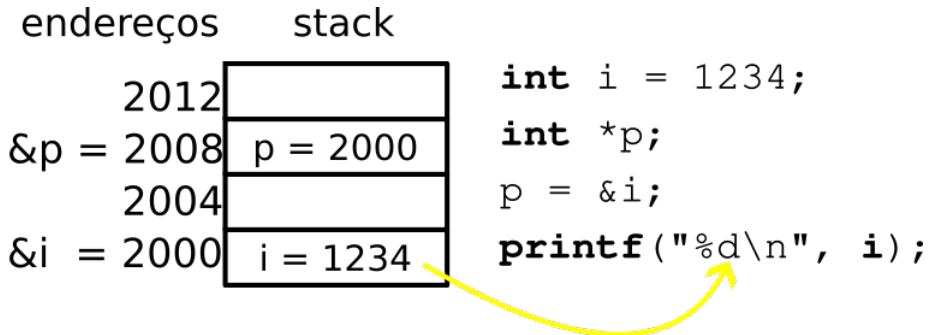
Ponteiros

- Conteúdo de `p` = endereço de `i`



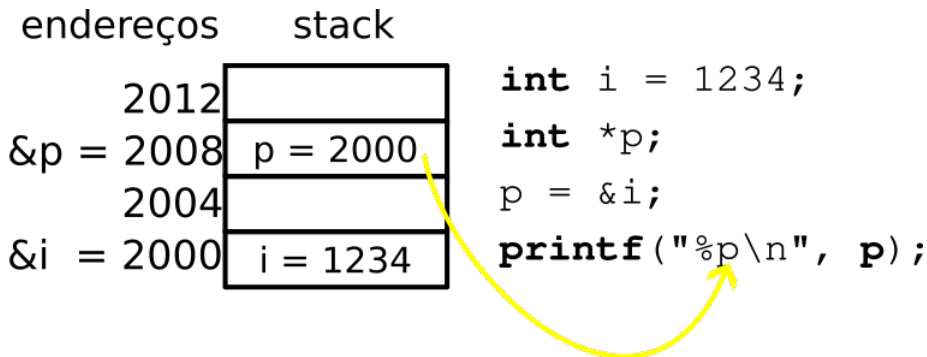
Ponteiros

- Mostra o conteúdo de i



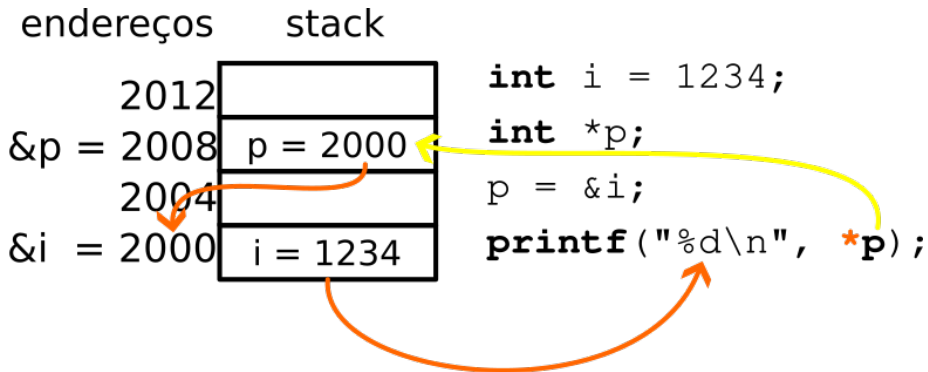
Ponteiros

- Endereço conteúdo de p



Ponteiros




- `*p` → mostra o conteúdo da variável apontado por `p`



Ponteiros

Ponteiro para ponteiro

- Mostra o conteúdo da variável apontada pelo ponteiro apontado por r

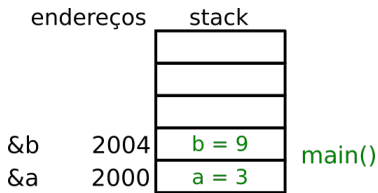
endereços	stack
 &r = 2012	r = 2008
 &p = 2008	p = 2000
2004	
 &i = 2000	i = 1234

```
int i = 1234;  
int *p;  
int **r;  
p = &i;  
r = &p;  
printf("%d\n", **r);
```

Ponteiros

Parâmetros de funções - passagem por cópia/valor

```
1 void troca (int a, int b) {  
2     int t;  
3     t = a;  
4     a = b;  
5     b = t;  
6 }  
7  
8 int main() {  
9     int a = 3;  
10    int b = 9;  
11    troca(a, b);  
12    //saida??  
13    printf("%d %d\n", a, b);  
14 }
```



Ponteiros

Parâmetros de funções - passagem por cópia/valor

```
1 void troca (int a, int b) {  
2     int t;  
3     t = a;  
4     a = b;  
5     b = t;  
6 }  
7  
8 int main() {  
9     int a = 3;  
10    int b = 9;  
11    troca(a, b);  
12    //saida??  
13    printf("%d %d\n", a, b);  
14 }
```

endereços		stack	
&b	2016	?	troca(3,9)
&a	2012	?	
&t	2008	?	
&b	2004	b = 9	main()
&a	2000	a = 3	

Ponteiros

Parâmetros de funções - passagem por cópia/valor

```
1 void troca (int a, int b) {  
2     int t;  
3     t = a;  
4     a = b;  
5     b = t;  
6 }  
7  
8 int main() {  
9     int a = 3;  
10    int b = 9;  
11    troca(a, b);  
12    //saida??  
13    printf("%d %d\n", a, b);  
14 }
```

endereços		stack	troca(3,9)
&b	2016	b = 9	
&a	2012	a = 3	
&t	2008	?	main()
&b	2004	b = 9	
&a	2000	a = 3	

Ponteiros

Parâmetros de funções - passagem por cópia/valor

```
1 void troca (int a, int b) {  
2     int t;  
3     t = a;  
4     a = b;  
5     b = t;  
6 }  
7  
8 int main() {  
9     int a = 3;  
10    int b = 9;  
11    troca(a, b);  
12    //saida??  
13    printf("%d %d\n", a, b);  
14 }
```

endereços		stack	troca(3,9)
&b	2016	b = t = 3	
&a	2012	a = b = 9	
&t	2008	t = a = 3	main()
&b	2004	b = 9	
&a	2000	a = 3	

Ponteiros

Parâmetros de funções - passagem por cópia/valor

```
1 void troca (int a, int b) {  
2     int t;  
3     t = a;  
4     a = b;  
5     b = t;  
6 }  
7  
8 int main() {  
9     int a = 3;  
10    int b = 9;  
11    troca(a, b);  
12    //saida??  
13    printf("%d %d\n", a, b);  
14 }
```

endereços	stack	
	2016	b = t = 3
	2012	a = b = 9
	2008	t = a = 3
&b	2004	b = 9
&a	2000	a = 3

~~troca(3,9)~~

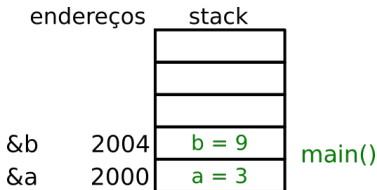
main()

Ponteiros

Parâmetros de funções - passagem por referência

- Passar o endereço de uma variável para salvar modificações

```
1 void troca (int *p, int *q) {  
2     int t;  
3     t = *p; //conteudo de temp = conteudo apontado por p  
4     *p = *q; //conteudo de p = conteudo apontado por q  
5     *q = t; //conteudo de q = conteudo de temp  
6 }  
7  
8 int main() {  
9     int a = 3, b = 9;  
10    troca(&a, &b);  
11    printf("%d %d\n", a, b); //saida??  
12 }
```



Ponteiros

Parâmetros de funções - passagem por referência

- Passar o endereço de uma variável para salvar modificações

```
1 void troca (int *p, int *q) {  
2     int t;  
3     t = *p; //conteudo de temp = conteudo apontado por p  
4     *p = *q; //conteudo de p = conteudo apontado por q  
5     *q = t; //conteudo de q = conteudo de temp  
6 }  
7  
8 int main() {  
9     int a = 3, b = 9;  
10    troca(&a, &b);  
11    printf("%d %d\n", a, b); //saida??  
12 }
```

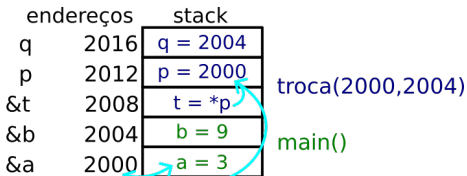
endereços		stack	troca(2000,2004)
q	2016	q = 2004	
p	2012	p = 2000	
&t	2008	?	
			main()
&b	2004	b = 9	
&a	2000	a = 3	

Ponteiros

Parâmetros de funções - passagem por referência

- Passar o endereço de uma variável para salvar modificações

```
1 void troca (int *p, int *q) {  
2     int t;  
3     t = *p; //conteudo de temp = conteudo apontado por p  
4     *p = *q; //conteudo de p = conteudo apontado por q  
5     *q = t; //conteudo de q = conteudo de temp  
6 }  
7  
8 int main() {  
9     int a = 3, b = 9;  
10    troca(&a, &b);  
11    printf("%d %d\n", a, b); //saida??  
12 }
```



Ponteiros

Parâmetros de funções - passagem por referência

- Passar o endereço de uma variável para salvar modificações

```
1 void troca (int *p, int *q) {  
2     int t;  
3     t = *p; //conteudo de temp = conteudo apontado por p  
4     *p = *q; //conteudo de p = conteudo apontado por q  
5     *q = t; //conteudo de q = conteudo de temp  
6 }  
7  
8 int main() {  
9     int a = 3, b = 9;  
10    troca(&a, &b);  
11    printf("%d %d\n", a, b); //saida??  
12 }
```

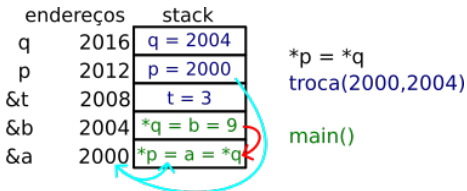
endereços		stack	
q	2016	q = 2004	*p = *q troca(2000,2004)
p	2012	p = 2000	
&t	2008	t = 3	main()
&b	2004	*q = b = 9	
&a	2000	a = 3	

Ponteiros

Parâmetros de funções - passagem por referência

- Passar o endereço de uma variável para salvar modificações

```
1 void troca (int *p, int *q) {  
2     int t;  
3     t = *p; //conteudo de temp = conteudo apontado por p  
4     *p = *q; //conteudo de p = conteudo apontado por q  
5     *q = t; //conteudo de q = conteudo de temp  
6 }  
7  
8 int main() {  
9     int a = 3, b = 9;  
10    troca(&a, &b);  
11    printf("%d %d\n", a, b); //saida??  
12 }
```



Ponteiros

Parâmetros de funções - passagem por referência

- Passar o endereço de uma variável para salvar modificações

```
1 void troca (int *p, int *q) {  
2     int t;  
3     t = *p; //conteudo de temp = conteudo apontado por p  
4     *p = *q; //conteudo de p = conteudo apontado por q  
5     *q = t; //conteudo de q = conteudo de temp  
6 }  
7  
8 int main() {  
9     int a = 3, b = 9;  
10    troca(&a, &b);  
11    printf("%d %d\n", a, b); //saida??  
12 }
```

endereços		stack	
q	2016	q = 2004	*p = *q troca(2000,2004)
p	2012	p = 2000	
&t	2008	t = 3	
&b	2004	*q = b = 9	main()
&a	2000	*p = a = 9	

Ponteiros

Parâmetros de funções - passagem por referência

- Passar o endereço de uma variável para salvar modificações

```
1 void troca (int *p, int *q) {  
2     int t;  
3     t = *p; //conteudo de temp = conteudo apontado por p  
4     *p = *q; //conteudo de p = conteudo apontado por q  
5     *q = t; //conteudo de q = conteudo de temp  
6 }  
7  
8 int main() {  
9     int a = 3, b = 9;  
10    troca(&a, &b);  
11    printf("%d %d\n", a, b); //saida??  
12 }
```

endereços		stack	
q	2016	q = 2004	*q = t troca(2000,2004)
p	2012	p = 2000	
&t	2008	t = 3	main()
&b	2004	*q = b = t	
&a	2000	a = 9	

Ponteiros

Parâmetros de funções - passagem por referência

- Passar o endereço de uma variável para salvar modificações

```
1 void troca (int *p, int *q) {  
2     int t;  
3     t = *p; //conteudo de temp = conteudo apontado por p  
4     *p = *q; //conteudo de p = conteudo apontado por q  
5     *q = t; //conteudo de q = conteudo de temp  
6 }  
7  
8 int main() {  
9     int a = 3, b = 9;  
10    troca(&a, &b);  
11    printf("%d %d\n", a, b); //saida??  
12 }
```

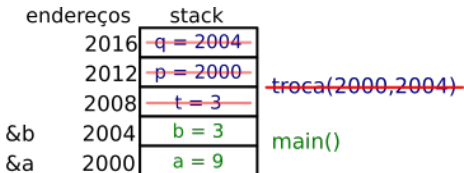
endereços		stack	
q	2016	q = 2004	*q = t troca(2000,2004)
p	2012	p = 2000	
&t	2008	t = 3	main()
&b	2004	b = 3	
&a	2000	a = 9	

Ponteiros

Parâmetros de funções - passagem por referência

- Passar o endereço de uma variável para salvar modificações

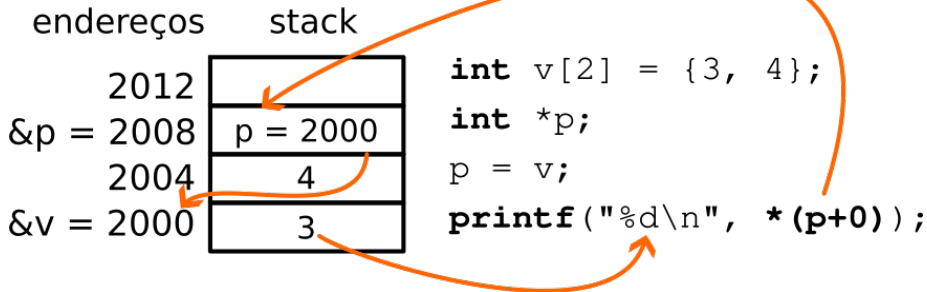
```
1 void troca (int *p, int *q) {  
2     int t;  
3     t = *p; //conteudo de temp = conteudo apontado por p  
4     *p = *q; //conteudo de p = conteudo apontado por q  
5     *q = t; //conteudo de q = conteudo de temp  
6 }  
7  
8 int main() {  
9     int a = 3, b = 9;  
10    troca(&a, &b);  
11    printf("%d %d\n", a, b); //saida??  
12 }
```



Ponteiros

Vetor x Ponteiro

- Se vetor aponta para um endereço
- E ponteiro é um apontador de endereços
- Então, um vetor pode ser manipulado por ponteiros



Ponteiros

```
1  int i[2]; //i eh o endereco do vetor
2
3  int *p;
4  p = i; //p armazena i,
5         //portanto, p armazena o endereco de vetor i
6
7  i[1] = 9; //alterando o valor da posicao 1 do endereco apontado
8           //por i
9
10 printf("%d\n", i[1]); //9
11 printf("%d\n", *(i+1)); //9
12
13 printf("%d\n", p[1]); //9
14 printf("%d\n", *(p+1)); //9
15
```

Ponteiros

Matriz x Ponteiro

- E como manipular matriz com ponteiros?

```
1  int a[2][2] = { {1, 2},  
2                  {3, 4} };  
3  int *m;  
4  m = a[0];  
5  
6  for(int i=0; i<4; i++)  
7      printf("%2d", *(m+i));  
8  
9  printf("\n");  
10
```

Ponteiros

Struct x Ponteiro

```
1  struct point {  
2      int value;  
3  };  
4  
5  struct point s;  
6  struct point *ptr = &s;  
7  
8  s.value = 20;  
9  (*ptr).value = 40;  
10 ptr->value = 30;  
11  
12 printf("%d\n", s.value);  
13
```

Ponteiros

Ponteiro de função

- Ponteiro para referenciar funções

```
1 #include <stdio.h>
2 void f(int a) {
3     printf("%d\n", a);
4 }
5 void f2(int a) {
6     printf("%d\n", a+1);
7 }
8 int main() {
9     void (*fp)(int);
10
11     fp = &f;
12     (*fp)(10);
13
14     fp = &f2;
15     (*fp)(10);
16
17     return 0;
18 }
19
```

Roteiro

1 Memória

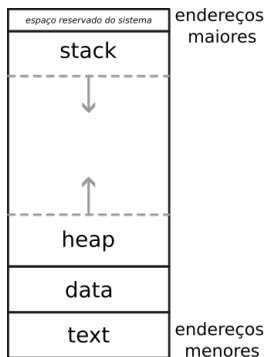
- Variáveis x Endereços
- Ponteiros - manipulação de endereços

2 Processo x Memória

- Alocação estática de memória
- Alocação automática de memória
- Alocação dinâmica de memória

Alocação de memória para os processos

- Programa em execução: processo
- Cada processo: possui uma porção da memória
- Cada porção: organizada por segmentos
- Layout geral dos segmentos:



- stack: variáveis locais, parâmetros de funções e endereços de retorno (instrução que chamou uma determinada função)
- heap: blocos de memória alocadas dinamicamente, a pedido do processo (gerenciado pelo sistema operacional)
- data: variáveis globais e estáticas
- text: código que está sendo executado
- Comando: **size executavel**
Lista os tamanhos de seção e tamanho total de arquivos binários

Alocação de memória

- Alocação estática
- Alocação automática
- Alocação dinâmica
- https://www.inf.ufpr.br/roberto/ci067/10_aloc.html

Roteiro

1 Memória

- Variáveis x Endereços
- Ponteiros - manipulação de endereços

2 Processo x Memória

- Alocação estática de memória
- Alocação automática de memória
- Alocação dinâmica de memória

Alocação estática de memória

Data

- Ocorre quando são declaradas
 - ▶ variáveis globais (alocadas fora de funções);
 - ▶ variáveis locais (internas a uma função) são alocadas usando o modificador "static";
 - ▶ Uma variável alocada estaticamente mantém seu valor durante toda a vida do programa, exceto quando explicitamente modificada.
- Geralmente alocadas em Data;

Alocação estática de memória

Data

```
1 int a = 0; //variável global, aloc. estática
2 static int b = 0; //variável estática global, aloc. estática
3 //acessível somente no arquivo
4
5 void incrementa(void) {
6     static int c = 0 ; //variável local, aloc. estática
7
8     printf ("a: %d, c: %d\n", a, c) ;
9     a++;
10    c++;
11 }
12
13 int main(void) {
14     for (int i = 0; i < 5; i++)
15         incrementa() ;
16     return 0 ;
17 }
18 //A execução desse código gera a seguinte saída:
19 // a: 0, c: 0
20 // a: 1, c: 1
21 // a: 2, c: 2
22 // a: 3, c: 3
23 // a: 4, c: 4
```

Roteiro

1 Memória

- Variáveis x Endereços
- Ponteiros - manipulação de endereços

2 Processo x Memória

- Alocação estática de memória
- **Alocação automática de memória**
- Alocação dinâmica de memória

Alocação automática de memória

Alocação de variáveis

- Cada tipo ocupa uma quantidade distinta
- Alocação automática:
 - ▶ Tamanho e quantidade reservada quando a função é invocada
 - ▶ Liberado quando a função termina
- Cada variável possui um **endereço na memória**
 - ▶ Byte menos significativo - início da alocação
 - ▶ Alocação contínua
- Alocação de tipos primitivos (int, float, double, char)
- Alocação de estruturas de dados (arrays) e registros (structs)

Alocação estática x automática

```
1 int *fa() {  
2     int a = -9;  
3     int *i = &a;  
4     return i;  
5 }  
6  
7 int *fb() {  
8     static int a = 1;  
9     int *i = &a;  
10    return i;  
11 }  
12  
13 int main() {  
14     int *b = fa();  
15     int *c = fb();  
16  
17     printf("%d %d\n", *b, *c);  
18  
19     return 0;  
20 }
```


Alocação estática x automática

```
1 int *fa() {
2     int a = -9;
3     int *i = &a;
4     return i;
5 }
6 int *fb() {
7     static int a = 1;
8     int *i = &a;
9     return i;
10 }
11
12 void fc() {
13     int s[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
14 }
15
16 int main() {
17     int *b = fa(), *c = fb();
18     fc();
19     printf("%d %d\n", *b, *c);
20
21     return 0;
22 }
```

Roteiro

1 Memória

- Variáveis x Endereços
- Ponteiros - manipulação de endereços

2 Processo x Memória

- Alocação estática de memória
- Alocação automática de memória
- Alocação dinâmica de memória

Alocação dinâmica de memória

Alocação dinâmica em C Funções malloc, realloc, calloc e free

- Biblioteca stdlib.h
- Protótipos das funções

```
1  #include <stdlib.h>
2
3  void *malloc(size_t size);
4  void free(void *ptr);
5  void *calloc(size_t nmemb, size_t size);
6  void *realloc(void *ptr, size_t size);
7
```

Alocação dinâmica de memória

Quanto espaço reservar? Operador “sizeof”

- Computar o tamanho dos operadores
 - ▶ Tipos primitivos (inteiros, ponto flutuante, ponteiros)
 - ▶ Tipos de dados (registros - structs)
- Retorna `size_t` (dados em bytes) - unsigned int - tamanho em bytes
- Sintaxe: **sizeof**(tipo_dado || variavel);

```
1 struct endereco {  
2     char rua[100];  
3     int numero;  
4 };  
5  
6 printf("%ld bytes\n", sizeof(int)); //4 bytes  
7 printf("%ld bytes\n", sizeof(float)); //4 bytes  
8 printf("%ld bytes\n", sizeof(double)); //8 bytes  
9 printf("%ld bytes\n", sizeof(char)); //1 bytes  
10 printf("%ld bytes\n", sizeof(struct endereco)); //104 bytes  
11
```

Alocação dinâmica de memória

Funções malloc

- Aloca uma quantidade de bytes
- Retorna um ponteiro da memória alocada
- A memória não é inicializada
- Retorna NULL em caso de erro
- Se a quantidade requerida for zero, retorna um valor que pode ser passado para a função que libera memória
- Estratégia otimista: não é garantido a real disponibilidade

Alocação dinâmica de memória

Exemplos - malloc

```
1 int *p = malloc(sizeof(int)); //1 inteiro
2 char *nome = malloc(sizeof(char)*50); //string 50 posicoes
3 float *f = malloc(sizeof(float)*10); //vetor float - 10 posicoes
4 int *i = (int *)malloc(5*sizeof(int)) //typecast dos retornos das
    funções:
5                                     //versão antigas de C, ou
    para C++
6 if(f){
7     f[1] = 4;
8     printf("%f\n", f[1]);
9 }
10
11 struct endereco {
12     char rua[100];
13     int numero;
14 };
15
16 struct endereco *end;
17 end = malloc(sizeof(struct endereco));
18
19 if(end){
20     end->numero = 324;
21 }
```

Alocação dinâmica de memória

Funções free

- Libera o espaço, **previamente alocado dinamicamente**, apontado por um ponteiro
- Porção livre para novas alocações
- Chamadas repetidas para o mesmo ponteiro: erros inesperados
- Não retorna valor

```
1  int *p = malloc(sizeof(int));  
2  free(p);  
3  
4  int b = 4;  
5  int *a;  
6  a = &b;  
7  //free(a) ?  
8
```


Alocação dinâmica de memória

Funções calloc

- Aloca memória para um array de A elementos de tamanho N bytes
calloc(A, N);
- Retorna um ponteiro da memória alocada
- Retorna NULL em caso de erro
- Se a quantidade requerida for zero, retorna um valor que pode ser passado para a função *free*
- A memória é inicializada com zero

```
1 int *p = calloc(5, sizeof(int));  
2  
3
```

Alocação dinâmica de memória

Funções realloc

- Altera o tamanho do bloco de memória apontado por um ponteiro
- Conteúdo anterior não é afetado
- Tamanho maior: memória adicionada não é inicializada
- Se o ponteiro for NULL, é alocado como uma nova porção de memória (malloc)
- Retorna um ponteiro para a nova área alocada (pode ser a mesma ou diferente da original)
- Retorna NULL
 - ▶ Em caso de erro: bloco original não é afetado, fica inalterado
 - ▶ Se o ponteiro não for NULL e for requisitado zero bytes: espaço apontado é liberado (free)

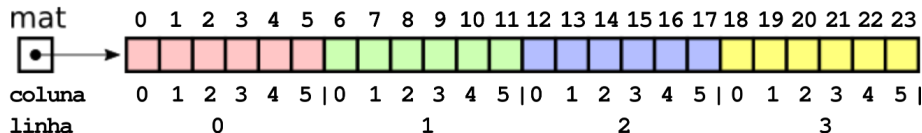
```
1 int *p = malloc(sizeof(int));  
2 p = realloc(p, 4*sizeof(int));  
3 free(p);  
4  
5
```

Alocação dinâmica de memória

Exemplos: Alocação dinâmica de uma Matriz (linear)

- Alocação linear: como um único vetor
- 1 ponteiro para o início do matriz

Linhas 4
Colunas 6
Posições $4 \times 6 = 24$



Alocação dinâmica de memória

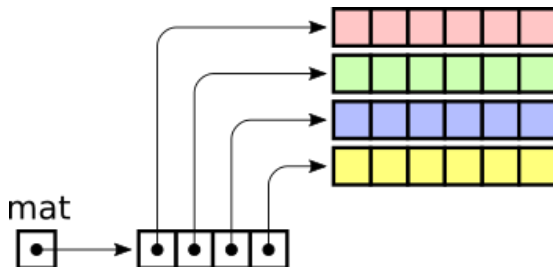
Exemplos: Alocação dinâmica de uma Matriz (linear)

```
1 //Elementos da matriz são alocados em um único vetor
2 #define LIN 4
3 #define COL 6
4 int *mat;
5 int lin , col;
6
7 //aloca um vetor com todos os elementos da matriz
8 mat = malloc (LIN * COL * sizeof (int)) ;
9
10 if(mat){
11     //percorre a matriz
12     for (lin = 0; lin < LIN; lin++)
13         for (col = 0; col < COL; col++)
14             //calcula a posição de cada elemento
15             mat[(lin*COL) + col] = 0 ;
16
17     //libera a memória alocada para a matriz
18     free(mat) ;
19 }
20
```

Alocação dinâmica de memória

Exemplos: Alocação dinâmica de uma Matriz (vetores)

- Alocação por vetores: cada vetor uma linha
- 1 ponteiro para ponteiros



Alocação dinâmica de memória

Exemplos: Alocação dinâmica de uma Matriz (vetores)

```
1 #define LIN 4
2 #define COL 6
3 int **mat, i, j;
4
5 //aloca um vetor de LIN ponteiros para linhas
6 mat = malloc (LIN * sizeof (int*)) ;
7
8 if(mat){
9     //aloca cada uma das linhas (vetores de COL inteiros)
10    for (i=0; i < LIN; i++)
11        mat[i] = malloc (COL * sizeof (int)) ;
12
13    //percorre a matriz
14    for (i=0; i < LIN; i++)
15        for (j=0; j < COL; j++)
16            mat[i][j] = 0 ; // acesso com sintaxe mais simples
17
18    //libera a memória da matriz
19    for (i=0; i < LIN; i++) free (mat[i]) ;
20    free(mat) ;
21 }
```