

Alocação de Memória

Prof^a. Rose Yuri Shimizu

Roteiro

1 Memória

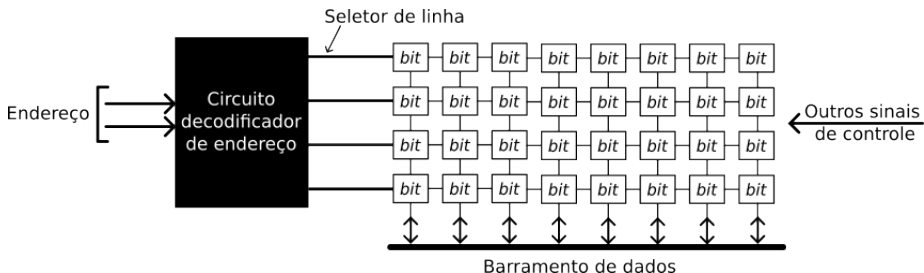
- Variáveis x Endereços
- Ponteiros - manipulação de endereços

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 e conversação 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
- Variáveis locais são alocadas na stack (pilha)
- Endereço = byte menos significativo (início da alocação)

```
1 #include <stdio.h>
2
3 int main(){
4     int x = 256; //2^8 = 00000000 00000000 00000001 00000000
5     char y = 'a';
6     int z = 0;
7
8     printf("%d\n", x); //saída?
9     printf("%ld\n", (long) &z); //140733520157276
10    printf("%ld\n", (long) &x); //140733520157272
11    printf("%ld\n", (long) &y); //140733520157271
12
13    //desloca dois endereços de tamanhos de char
14    printf("%ld\n", (long)(&y+2)); //140733520157273
15
16    return 0;
17 }
```

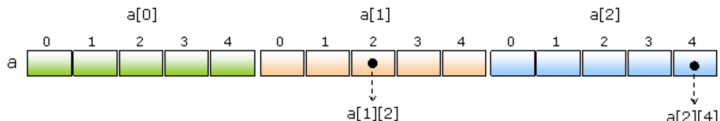
Vetor x Endereços

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

```
1 #include <stdio.h>
2
3 int main(){
4     int v[2] = {3, 7};
5
6     printf("%d %d\n", v[0], v[1]); //3 7
7
8     //140730478951952
9     printf("%ld\n", (long)v);
10
11    //140730478951952 140730478951956
12    printf("%ld %ld\n", (long)&v[0], (long)&v[1]);
13
14    return 0;
15 }
```

Matriz x Endereços

```
1 #include <stdio.h>
2 int main(){
3     int v[2][2] = {1, 2, 3, 4};
4     for(int i=0; i<2; i++){
5         for(int j=0; j<2; j++)
6             printf("%d ", v[i][j]);
7         printf("\n");
8     }
9
10    //alocação sequencial
11    //140730053083440
12    //140730053083440 140730053083444
13    //140730053083448 140730053083452
14    printf("%ld\n", (long)(v));
15    printf("%ld %ld\n", (long)&v[0][0], (long)&v[0][1]);
16    printf("%ld %ld\n", (long)&v[1][0], (long)&v[1][1]);
17
18    return 0;
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

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
 - ▶ Ex.: 3 bits representam até 2^3 números (linhas de uma tabela verdade)
 - ▶ x64: ponteiros de 8 bytes (armazenar $\approx 2^{64}$ endereços)
 - ▶ x86: ponteiros de 4 bytes
- Tipo dos ponteiros:
 - ▶ Utilizado para desreferenciar e operações aritméticas
 - ▶ Se **int** sabe-se que deverá ser considerado 4 bytes a partir do endereço inicial
- Pode ser NULL: indica endereço inválido; valor é 0 (zero)

Ponteiros

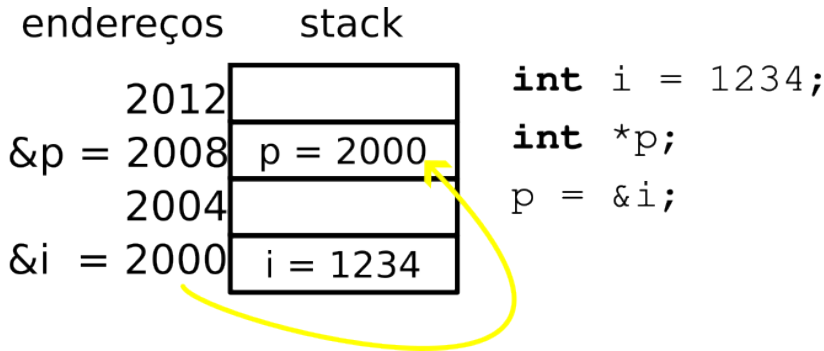
- Aloca i e p
- Conteúdo em i



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

Ponteiros

- Conteúdo de p = endereço de i




Ponteiros

- Mostra o conteúdo de `i`

endereços stack

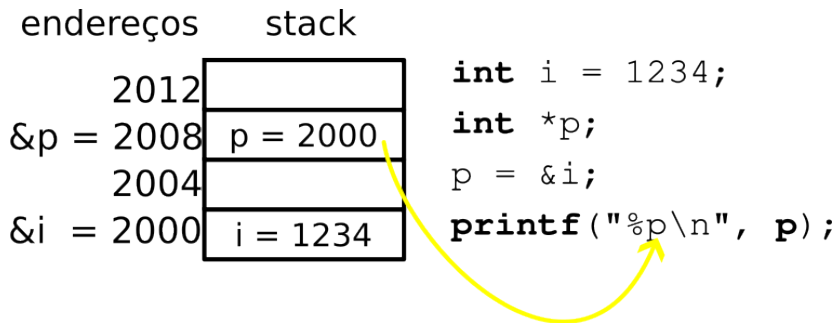
2012	
<code>&p = 2008</code>	<code>p = 2000</code>
2004	
<code>&i = 2000</code>	<code>i = 1234</code>

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



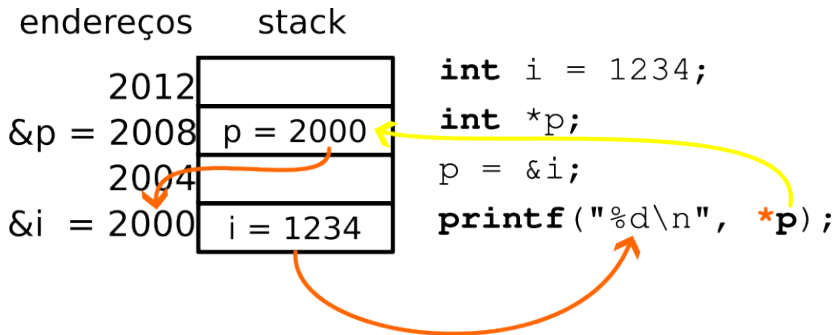
Ponteiros

- Endereço conteúdo de p



Ponteiros

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



```
1 #include <stdio.h>
2
3 int main(){
4     int i;
5     int *p;
6     p = NULL;
7     p = &i; //diz-se:
8             //    p aponta para i
9             //    p referencia a variavel i
10
11     i = 5;
12
13     /*p valor da variavel apontada por p, ou seja, valor de i
14     printf("%d\n", *p); /*p eh igual a i
15                        //saida: 5
16     printf("%ld\n", sizeof(p));
17
18     return 0;
19 }
```



```

1 #include <stdio.h>
2
3 int main(){
4     int x = 256;    //2^8 = 00000000 00000000 00000001 0000000
5     char y = 'a';  // 97 = 1100001
6     int z = 0;     // 0 = 00000000 00000000 00000000 0000000
7
8     printf("%d\n", x); //saída?
9     printf("%ld\n", (long) &z); //140733520157276
10    printf("%ld\n", (long) &x); //140733520157272
11    printf("%ld\n", (long) &y); //140733520157271
12
13    //memória
14    //00000000 00000000 00000000 00000000 -> ..76
15    //00000000 00000000 00000001 00000000 -> ..72
16    //1100001 -> ..71
17
18    //desloca dois endereços de tamanhos de char
19    int *p = (int *)(&y+2);
20    printf("%ld\n", (long)p); //140733520157273
21
22    printf("%d\n", *p); //saída?
23
24    return 0;
25 }

```

```

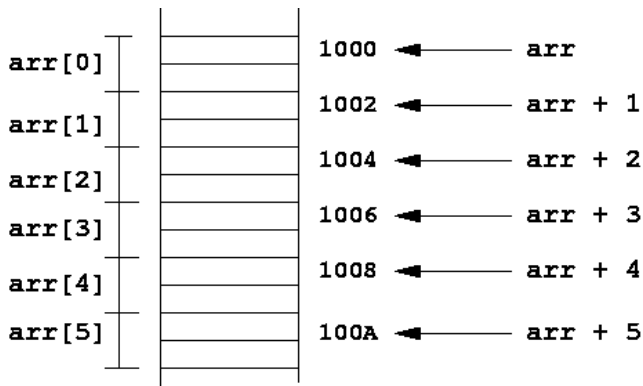
1 #include <stdio.h>
2
3 int main(){
4     int x = 256;    //2^8 = 00000000 00000000 00000001 00000000
5     char y = 'a';  // 97 = 1100001
6     int z = 0;     // 0 = 00000000 00000000 00000000 00000000
7
8     printf("%d\n", x); //saída?
9     printf("%ld\n", (long) &z); //140733520157276
10    printf("%ld\n", (long) &x); //140733520157272
11    printf("%ld\n", (long) &y); //140733520157271
12
13    //memória
14    //00000000 00000000 00000000 00000000 -> ..76
15    //00000000 00000000 00000001 00000000 -> ..72
16    //1100001 -> ..71
17
18    //desloca dois endereços de tamanhos de char
19    int *p = (int *)(&y+2);
20    printf("%ld\n", (long)p); //140733520157273
21
22    printf("%d\n", *p); //saída: 1
23
24    return 0;
25 }

```

```

1 #include <stdio.h>
2
3 int main(){
4
5     int v[2] = {3, 7};
6
7     printf("%d %d\n", v[0], v[1]);    //3 7
8     printf("%d %d\n", *(v+0), *(v+1)); //? ?
9
10    return 0;
11 }

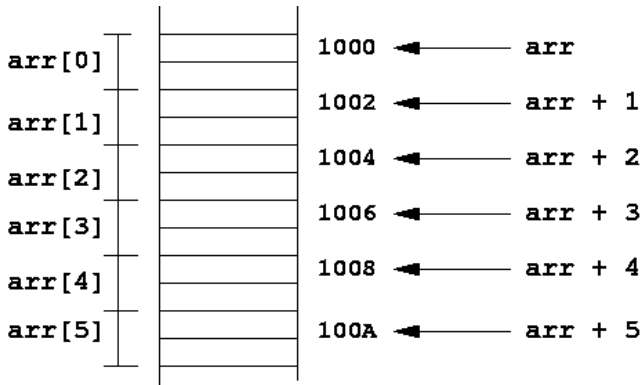
```



```

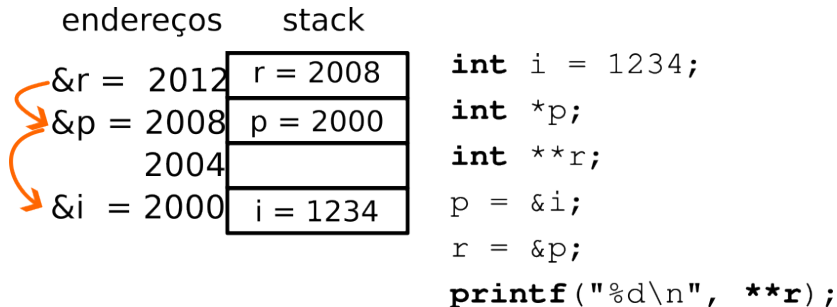
1 #include <stdio.h>
2
3 int main(){
4
5     int v[2] = {3, 7};
6
7     printf("%d %d\n", v[0], v[1]);    //3 7
8     printf("%d %d\n", *(v+0), *(v+1)); //3 7
9
10    return 0;
11 }

```



Ponteiro para ponteiro

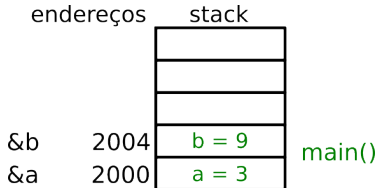
- Mostra o conteúdo da variável apontada pelo ponteiro apontado por r



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
12    troca(a, b);
13
14    printf("%d %d\n", a, b);
15 }
```



Ponteiros

```
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
12    troca(a, b); //<<
13
14    printf("%d %d\n", a, b);
15 }
```

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

troca(3,9)

main()

Ponteiros

```
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
12    troca(a, b);
13
14    printf("%d %d\n", a, b);
15 }
```

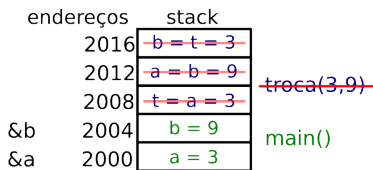
endereços	stack	
&b	2016	b = t = 3
&a	2012	a = b = 9
&t	2008	t = a = 3
&b	2004	b = 9
&a	2000	a = 3

troca(3,9)

main()

Ponteiros

```
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
12    troca(a, b); //<<
13
14    printf("%d %d\n", a, b); //<<
15 }
```



Saída??

Ponteiros

Parâmetros de funções - passagem por referência

- Passar o endereço de uma variável para salvar modificações

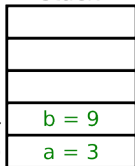
```
1 void troca (int *p, int *q) { //ponteiros recebem endereços
2     int t;
3     t = *p;
4     *p = *q;
5     *q = t;
6 }
7
8 int main() {
9     int a = 3, b = 9;
10
11     troca(&a, &b); //passando os endereços
12
13     printf("%d %d\n", a, b);
14 }
```

Ponteiros

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

endereços

stack



main()

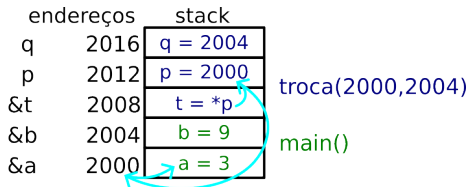
Ponteiros

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

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

Ponteiros

```
1 void troca (int *p, int *q) {
2     int t;
3     t = *p; //conteúdo de t = conteúdo do apontado por p
4     *p = *q;
5     *q = t;
6 }
7
8 int main() {
9     int a = 3, b = 9;
10
11     troca(&a, &b);
12
13     printf("%d %d\n", a, b);
14 }
```



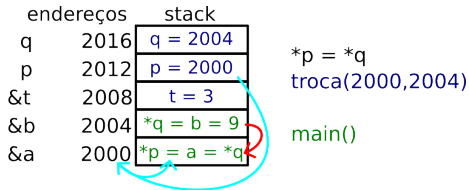
Ponteiros

```
1 void troca (int *p, int *q) {
2     int t;
3     t = *p;
4     *p = *q; /*q -> conteúdo do apontado por q
5     *q = t;
6 }
7
8 int main() {
9     int a = 3, b = 9;
10
11     troca(&a, &b);
12
13     printf("%d %d\n", a, b);
14 }
```

	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

```
1 void troca (int *p, int *q) {
2     int t;
3     t = *p;
4     *p = *q; // *p -> conteúdo do apontado por p
5             // cont. do apontado por p = cont. do apontado por q
6     *q = t;
7 }
8
9 int main() {
10     int a = 3, b = 9;
11
12     troca(&a, &b);
13
14     printf("%d %d\n", a, b);
15 }
```



Ponteiros

```
1 void troca (int *p, int *q) {
2     int t;
3     t = *p;
4     *p = *q; //cont. do apontado por p = cont. do apontado por q
5     *q = t;
6 }
7
8 int main() {
9     int a = 3, b = 9;
10
11     troca(&a, &b);
12
13     printf("%d %d\n", a, b);
14 }
```

	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

```
1 void troca (int *p, int *q) {
2     int t;
3     t = *p;
4     *p = *q;
5     *q = t; //conteúdo do apontado por q = conteúdo de t
6 }
7
8 int main() {
9     int a = 3, b = 9;
10
11     troca(&a, &b);
12
13     printf("%d %d\n", a, b);
14 }
```

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

Ponteiros

```
1 void troca (int *p, int *q) {
2     int t;
3     t = *p;
4     *p = *q;
5     *q = t; //conteúdo do apontado por q = conteúdo de t
6 }
7
8 int main() {
9     int a = 3, b = 9;
10
11     troca(&a, &b);
12
13     printf("%d %d\n", a, b);
14 }
```

	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

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

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

Ponteiro x Array

- Ponteiros podem referenciar arrays: apontar para o endereço inicial

```
1 #include <stdio.h>
2
3 int main(){
4     int v[2] = {1,2};
5     int m[2][2] = {1,2,3,4};
6
7     int *p;
8     p = v;
9     p[0] = 5;
10
11     printf("%d %d\n", p[0], p[1]);
12
13     p = m[0]; //(int *)m
14     for(int i=0; i<4; i++)
15         printf("%2d", p[i]);
16
17     return 0;
18 }
```

Ponteiro x Struct

```
1 #include <stdio.h>
2
3 typedef struct {
4     int value;
5 }Point;
6
7 int main(){
8
9     Point s;
10    Point *ptr = &s;
11
12    //variável simples
13    s.value = 20; //campo acessado por '.'
14
15    //conteúdo do apontado por ptr = variável simples
16    (*ptr).value = 40; //campo acessado por '.'
17
18    //acesso ao campo pelo ponteiro
19    ptr->value = 30; //setinha ->
20
21    printf("%d\n", s.value); //saída??
22    return 0;
23 }
```

Ponteiro x Função

```
1 void f(int a) {
2     printf("%d\n", a);
3 }
4
5 void f2(int a) {
6     printf("%d\n", a+1);
7 }
8
9 int main(int argc, char **argv) {
10     void (*fp)(int);
11
12     if(argc>1 && strcmp(argv[1], "f")==0)
13         fp = &f;
14     else
15         fp = &f2;
16
17     (*fp)(10);
18
19     void (*funcoes[2])(int) = { &f, &f2};
20     for(int i=0; i<2; i++)
21         (*funcoes[i])(i);
22
23     return 0;
24 }
```

Roteiro

1 Memória

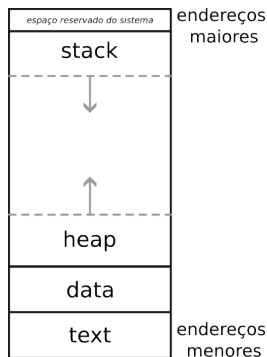
- Variáveis x Endereços
- Ponteiros - manipulação de endereços

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
- Layout geral:



- 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

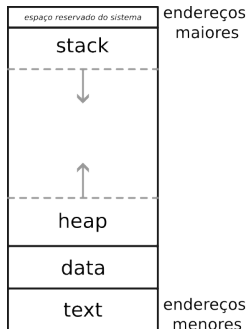
1 Memória

- 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”

```
1 int a; //global
2 static int b; //estática
3
4 int soma(){
5     static int c; //local estática
6 }
```

- Alocadas em Data

- Uma variável alocada estaticamente é válida por toda a vida do programa

Alocação estática de memória - Data - exemplo

```
1 //variaveis.h
2 #ifndef _VARIAVEIS_H_
3 #define _VARIAVEIS_H_
4 #include <stdio.h>
5 #include <stdlib.h>
6 int retornaVar1();
7 int retornaVar2();
8 #endif
```

```
1 //arquivo: variaveis.c
2 //compilar: gcc -c variaveis.c -o libvariaveis.o
3 #include "variaveis.h"
4
5 int var1 = 1;           //variável global, aloc. estática
6 static int var2 = 2;   //variável estática global, aloc. estática
7                       //acessível somente no arquivo
8
9 int retornaVar1(){
10     return var1;
11 }
12
13 int retornaVar2(){
14     return var2;
15 }
```

Alocação estática de memória - Data - exemplo

```
1 //arquivo: teste.c
2 //compilar: gcc teste.c libvariaveis.o
3 //executar: ./a.out
4 #include "variaveis.h"
5
6 extern int var1; //global de variaveis.c
7 int var2;
8
9 int main() {
10     printf("%d\n", var1); //1
11     printf("%d\n", retornaVar1()); //1
12     printf("%d\n", retornaVar2()); //2
13
14     var1 = 4;
15     printf("%d\n", var1); //4
16     printf("%d\n", retornaVar1()); //4
17
18     var2 = 5;
19     printf("%d\n", var2); //5
20     printf("%d\n", retornaVar2()); //2
21     return 0;
22 }
```

Alocação estática de memória - Data - exemplo

```
1 int a = 0;
2
3 void incrementa(void) {
4     static int c = 0 ; //variável local, alloc. estática
5                         //alocada uma única vez e
6                         //válida mesmo após o término da função
7
8     printf ("a: %d, c: %d\n", a, c) ;
9     a++;
10    c++;
11 }
12 int main(void) {
13     for (int i = 0; i < 5; i++)
14         incrementa() ;
15     return 0 ;
16 }
17 //A execução desse código gera a seguinte saída:
18 // a: 0, c: 0
19 // a: 1, c: 1
20 // a: 2, c: 2
21 // a: 3, c: 3
22 // a: 4, c: 4
```

Roteiro

1 Memória

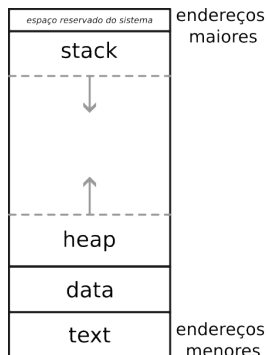
- Variáveis x Endereços
- Ponteiros - manipulação de endereços

2 Processo x Memória

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

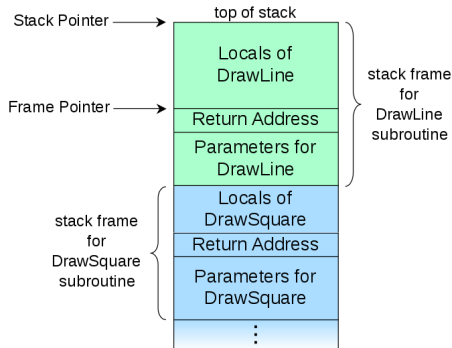
Alocação automática de memória - Stack

- Pilha de execução ou chamada
- Armazena
 - ▶ Variáveis locais e parâmetros de funções
 - ▶ Endereços de retorno (instrução que invocou a função)
- Alocação e desalocação: automática (SO - sistema operacional)
- Tempo de vida: enquanto a função existir (escopo local)
- Tamanho: limitado pelo SO
 - ▶ Linux: 8192 kB (ulimit -s)



Alocação automática de memória - Stack

- Alocação automática:
 - ▶ Tamanho e quantidade reservada quando a função é invocada
 - ▶ Liberado quando a função termina
- Exemplo: função `DrawLine` e `DrawSquare` na stack



- Alocação de variáveis: cada tipo ocupa uma quantidade distinta
 - ▶ Tipos primitivos (int, float, double, char), arrays, structs, ponteiros
- Alocação contínua (ordem: decisão do compilador)

Memória				Endereço	Variável	
stack	01000000	00000000	00000000	00000000	540	
	00000000	00000000	00000000	00000000	536	double c;
	01000000	00000000	00000000	00000000	532	float d;
	00000000	00000000	00000000	00000100	528	int b;
	00000000	00000000	00000000	00000011	524	int a;
	01000010	01000001	-	-	523 – 522	char f; char e;
heap		
data		
text	<pre> 1 int a=3; 2 int b=4; 3 double c=2; 4 float d=-2; 5 char e='A'; 6 char f='B'; 7 8 printf("%ld\n", (long) &a); 9 printf("%ld\n", (long) &b); 10 printf("%ld\n", (long) &c); 11 printf("%ld\n", (long) &d); 12 printf("%ld\n", (long) &e); 13 printf("%ld\n", (long) &f); </pre>				...	

Alocação estática x automática

```
1 #include <stdio.h>
2
3 int *fa()
4 {
5     int a = -9; //stack ou data?
6     int *i = &a;
7     return i;
8 }
9
10 int *fb()
11 {
12     static int a = 1; //stack ou data?
13     int *i = &a;
14     return i;
15 }
16
17 int main() {
18     int *b = fa();
19     int *c = fb();
20     printf("%d %d\n", *b, *c); //saída?
21
22     return 0;
23 }
```

Alocação estática x automática

```
1 #include <stdio.h>
2
3 int *fa()
4 {
5     int a = -9; //stack ou data?
6     int *i = &a;
7     return i;
8 }
9
10 int *fb()
11 {
12     static int a = 1; //stack ou data?
13     int *i = &a;
14     return i;
15 }
16
17 int main() {
18     int *b = fa();
19     int *c = fb();
20     printf("%d %d\n", *b, *c); //-9 1
21
22     return 0;
23 }
```

Alocação estática x automática

```
1 #include <stdio.h>
2 int *fa() {
3     int a = -9;
4     int *i = &a;
5     return i;
6 }
7
8 int *fb() {
9     static int a = 1;
10    int *i = &a;
11    return i;
12 }
13
14 void fc() {
15     int s[10] = {0}; //stack ou data?
16 }
17
18 int main() {
19     int *b = fa(), *c = fb();
20     fc();
21     printf("%d %d\n", *b, *c); //saídas??
22     return 0;
23 }
```

Alocação estática x automática

```
1 #include <stdio.h>
2 int *fa() {
3     int a = -9;
4     int *i = &a;
5     return i;
6 }
7
8 int *fb() {
9     static int a = 1;
10    int *i = &a;
11    return i;
12 }
13
14 void fc() {
15     int s[10] = {0}; //stack ou data?
16 }
17
18 int main() {
19     int *b = fa(), *c = fb();
20     fc();
21     printf("%d %d\n", *b, *c); //0 1
22     return 0;
23 }
```

Roteiro

1 Memória

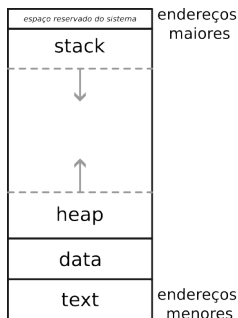
- Variáveis x Endereços
- Ponteiros - manipulação de endereços

2 Processo x Memória

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

Alocação dinâmica de memória

- Alocar memória em **durante a execução** do programa
- Alocar **tamanhos maiores** que a área reservada na stack
- Alocado no segmento **heap**
- Permite a alteração do tamanho alocado



Alocação dinâmica de memória - 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);
```

Alocação dinâmica de memória - Operador "sizeof"

- Auxilia na decisão de quanto espaço reservar
- Computa o tamanho
 - ▶ Tipos primitivos (inteiros, ponto flutuante, ponteiros)
 - ▶ Tipos de dados (registros - structs)
- Retorna `size_t` (dados em bytes) - long unsigned int - tamanho em bytes
- Sintaxe: `sizeof(<tipo_dado || variavel>);`

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

Alocação dinâmica de memória - Função malloc

- Aloca uma quantidade de bytes
- Retorna um ponteiro da memória alocada

```
1 #include <stdlib.h>
2
3 void *malloc(size_t size);
```

- 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 - Função malloc

Exemplos

```
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
5 //typedef: versão antigas de C, ou para C++
6 int *i = (int *)malloc(5*sizeof(int))
7 if(f){
8     f[1] = 4;
9     printf("%f\n", f[1]);
10 }
11
12 struct endereco {
13     char rua[100];
14     int numero;
15 };
16
17 struct endereco *end;
18 end = malloc(sizeof(struct endereco));
19
20 if(end){
21     end->numero = 324;
22 }
```

Alocação dinâmica de memória - Função 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 #include <stdlib.h>
2
3 void free(void *ptr);
```

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

Alocação dinâmica de memória - Função calloc

- Aloca memória para um array de A elementos de tamanho N bytes
calloc(A, N);

```
1 #include <stdlib.h>
2
3 void *calloc(size_t nmemb, size_t size);
```

- 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
- Exemplo: `int *p = calloc(5, sizeof(int));`

Alocação dinâmica de memória - Função realloc

- Altera o tamanho do bloco de memória apontado por um ponteiro

```
1 #include <stdlib.h>
2
3 void *realloc(void *ptr, size_t size);
```

- 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)
- Exemplo:

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

```
1 int *fa() {
2     int *v = malloc(10*sizeof(int)); //stack ou heap
3     for(int i=0; i<10; i++) v[i] = 1;
4     return v;
5 }
6 int *fb() {
7     int *v = malloc(10*sizeof(int)); //stack ou heap
8     for(int i=0; i<10; i++) v[i] = 2;
9     return v;
10 }
11 int main() {
12     int *a, *b;
13     a = fa();
14     b = a;
15     a = fb();
16
17     for(int i=0; i<10; i++)
18         printf("%d ", a[i]); //saída?
19     printf("\n");
20
21     for(int i=0; i<10; i++)
22         printf("%d ", b[i]); //saída?
23     printf("\n");
24
25     return 0;
26 }
```



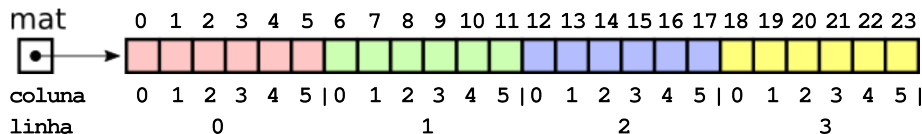
```
1 int *fa() {
2     int *v = malloc(10*sizeof(int)); //stack ou heap?
3     for(int i=0; i<10; i++) v[i] = 1;
4     return v;
5 }
6 int *fb() {
7     int *v = malloc(10*sizeof(int)); //stack ou heap?
8     for(int i=0; i<10; i++) v[i] = 2;
9     return v;
10 }
11
12
13 int main() {
14     int *a, *b;
15     a = fa();
16     b = a;
17     free(a);
18     a = fb();
19
20     for(int i=0; i<10; i++)
21         printf("%d ", a[i]); //saída?
22     printf("\n");
23
24     for(int i=0; i<10; i++)
25         printf("%d ", b[i]); //saída?
26     printf("\n");
27
```

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 $4*6 = 24$



Alocação dinâmica de memória

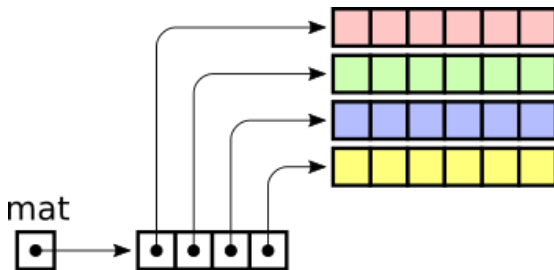
Exemplos: Alocação dinâmica de uma Matriz (linear)

```
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                                     // ^ ponteiro
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 }
```