Alocação de Memória

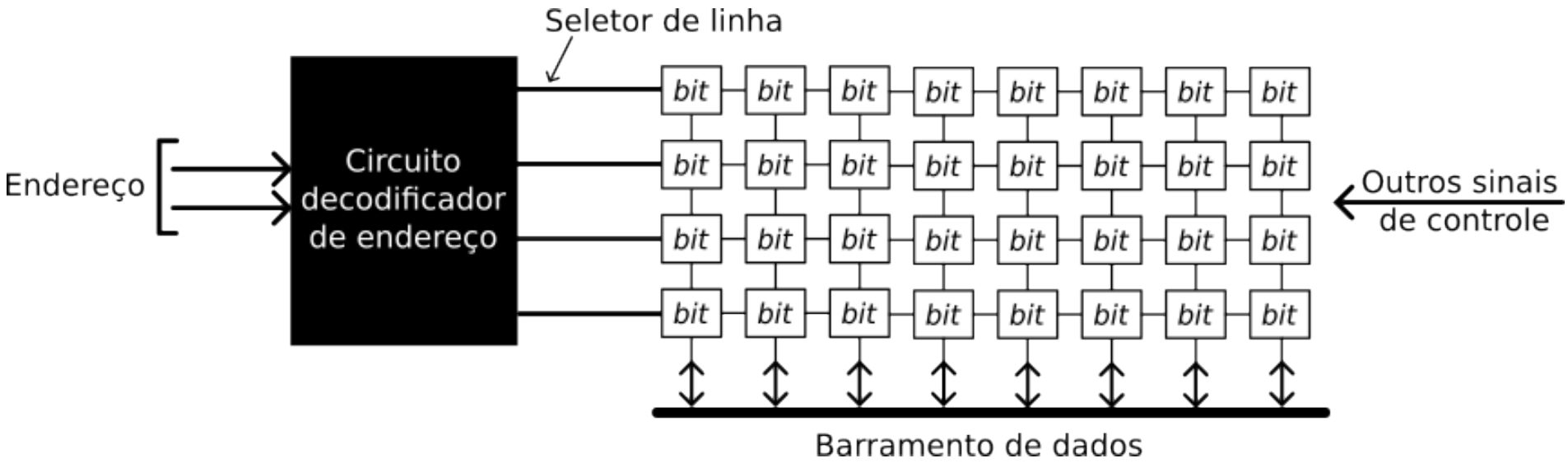
Profª. Rose Yuri Shimizu

Roteiro

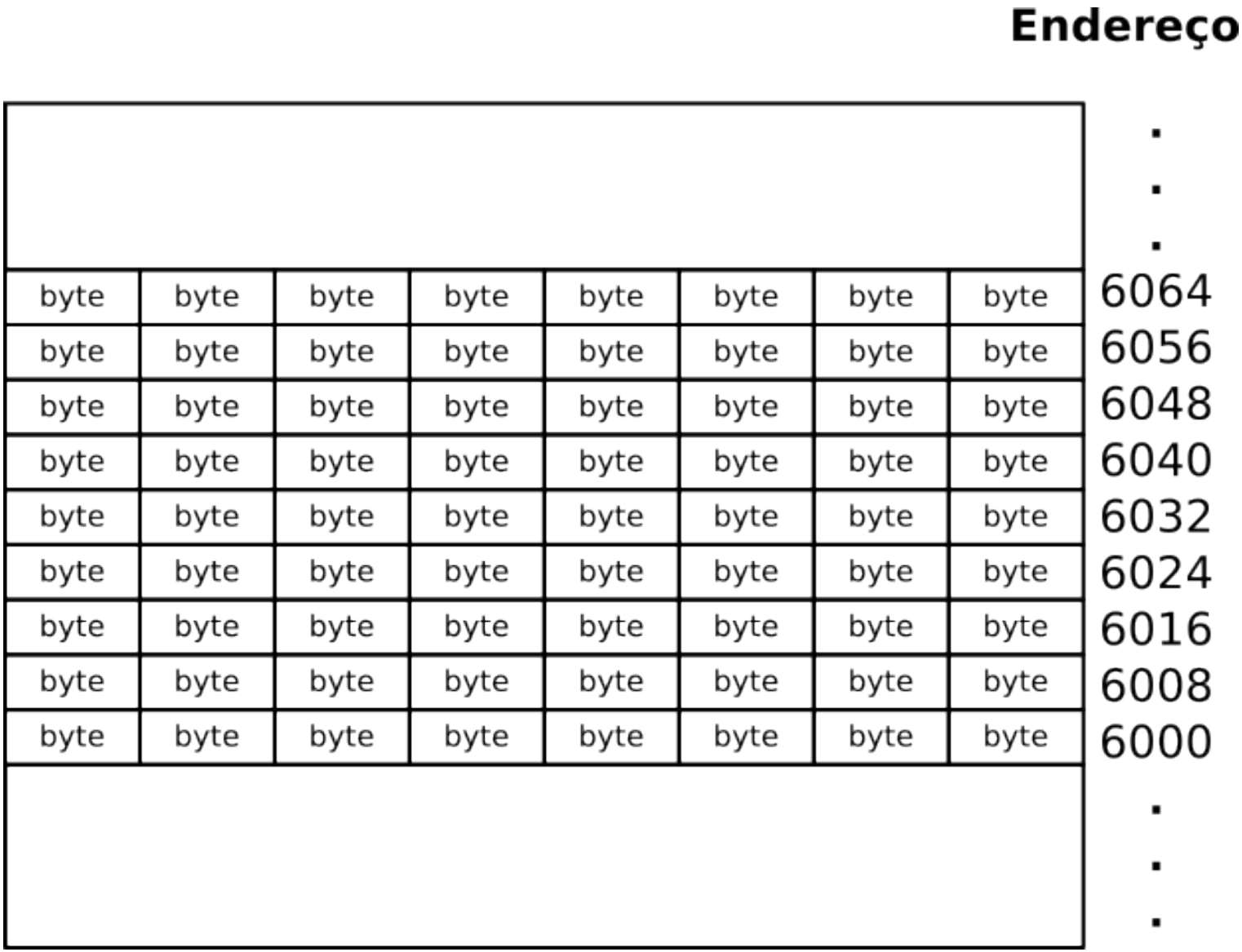
Memória física

- 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



Roteiro

Variáveis x Endereços

- Cada variável possui um endereço na memória
- Variáveis locais são alocadas na stack (pilha)
- Endereço = byte menos significativo (início da alocação)

```c
#include <stdio.h>

int main(){
    int x = 256; //2^8 = 00000000 00000000 00000001 0000000
    char y = 'a';
    int z = 0;

    printf("%d\n", x); //saída?
    printf("%ld\n", (long) &z); //140733520157276
    printf("%ld\n", (long) &x); //140733520157272
    printf("%ld\n", (long) &y); //140733520157271

    //desloca dois endereços de tamanhos de char
    printf("%ld\n", (long)(&y+2)); //140733520157273

    return 0;
}
```

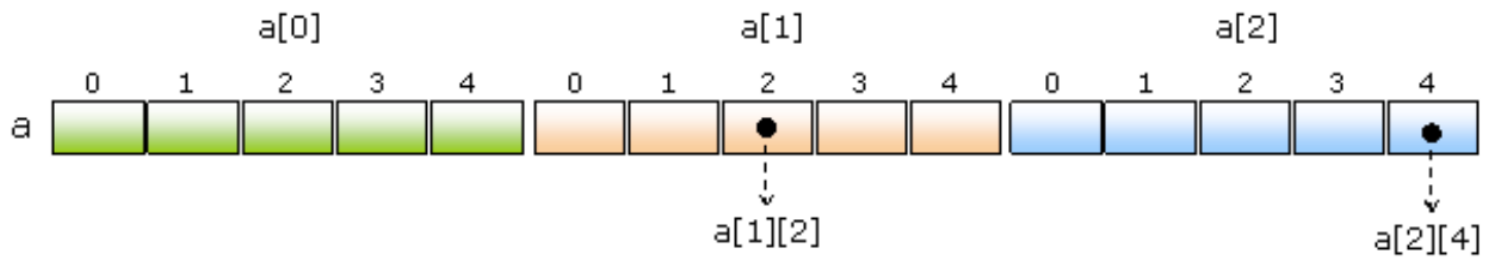
Vetor x Endereços

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

```c
#include <stdio.h>

int main(){
    int v[2] = {3, 7};

    printf("%d %d\n", v[0], v[1]); //3 7

    //140730478951952
    printf("%ld\n", (long)v);

    //140730478951952 140730478951956
    printf("%ld %ld\n", (long)&v[0], (long)&v[1]);

    return 0;
}
```

Matriz x Endereços

```c
#include <stdio.h>
int main(){
    int v[2][2] = {1, 2, 3, 4};
    for(int i=0; i<2; i++){
        for(int j=0; j<2; j++)
            printf("%d ", v[i][j]);
        printf("\n");
    }

    //alocação sequencial
    //140730053083440
    //140730053083440 140730053083444
    //140730053083448 140730053083452
    printf("%ld\n", (long)(v));
    printf("%ld %ld\n", (long)&v[0][0], (long)&v[0][1]);
    printf("%ld %ld\n", (long)&v[1][0], (long)&v[1][1]);

    return 0;
}
```



Roteiro

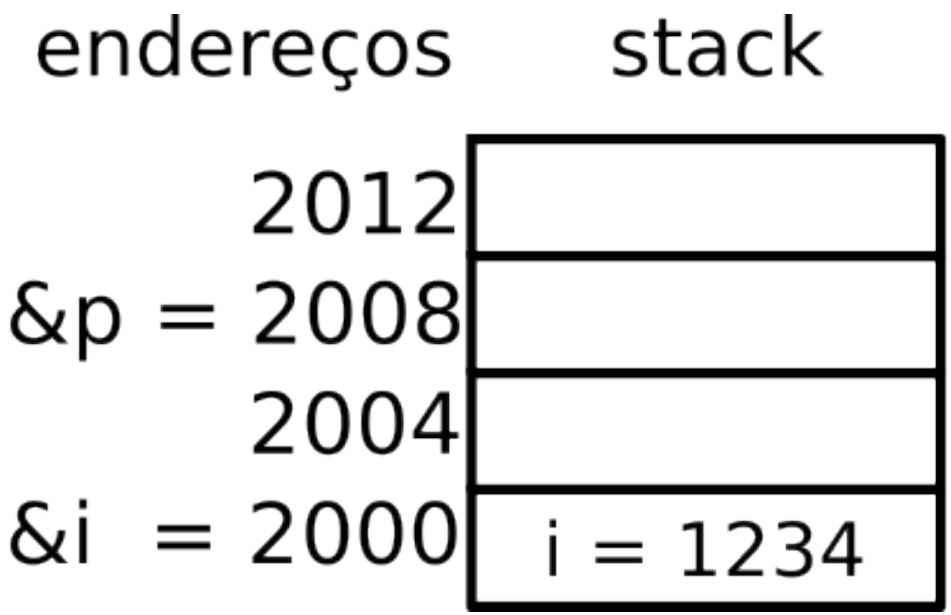
1. Memória
   - Variáveis x Endereços
   - Ponteiros - manipulação de endereços
2. Processo x Memória
   - Alocação estática de memória
   - Alocação automática de memória
   - Alocação dinâmica de memória

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

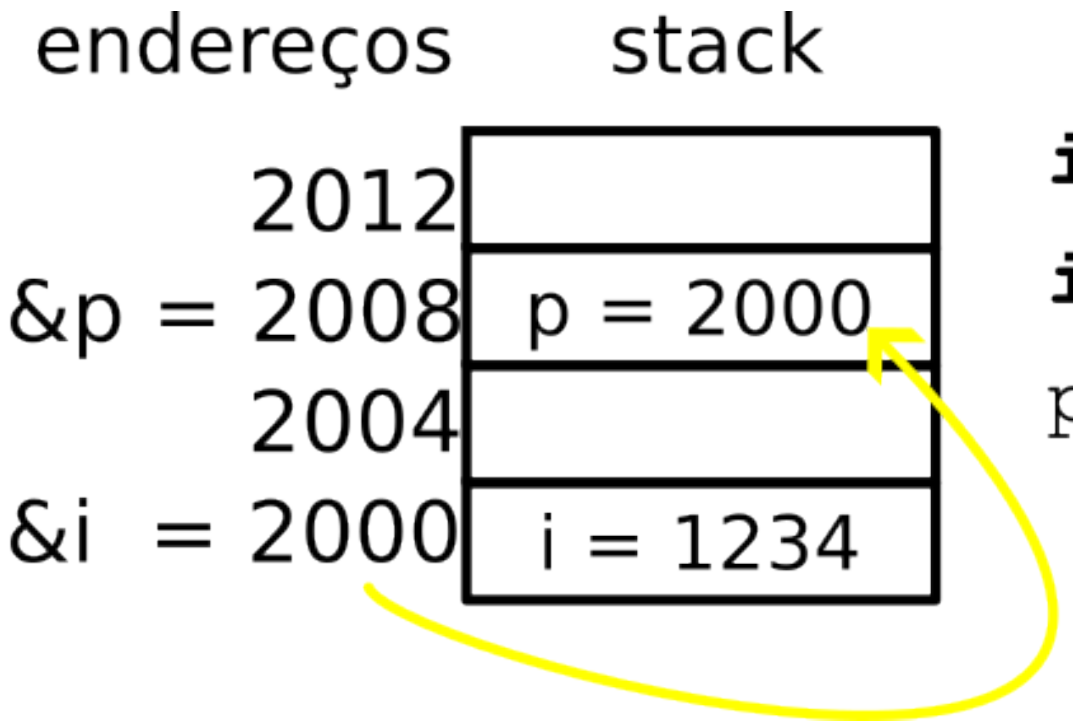
- Aloca i e p
- Conteúdo em i




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

Ponteiros

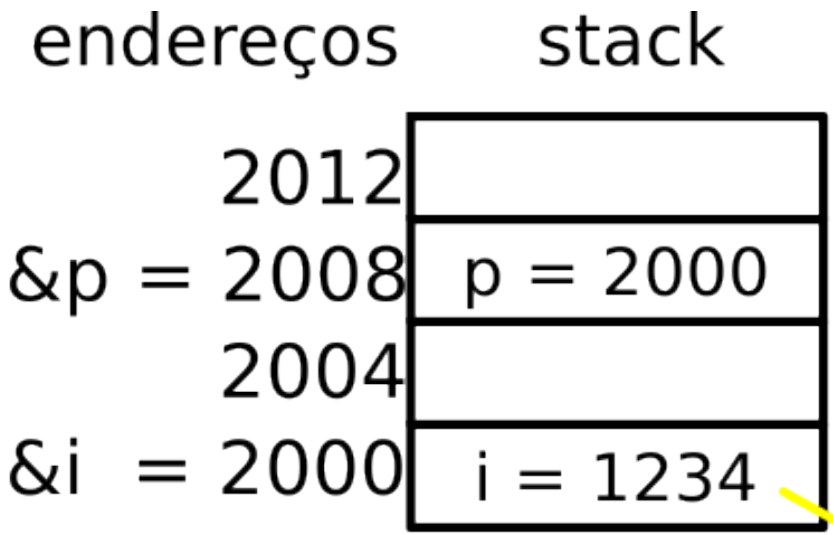
- Conteúdo de p = endereço de i




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

Ponteiros

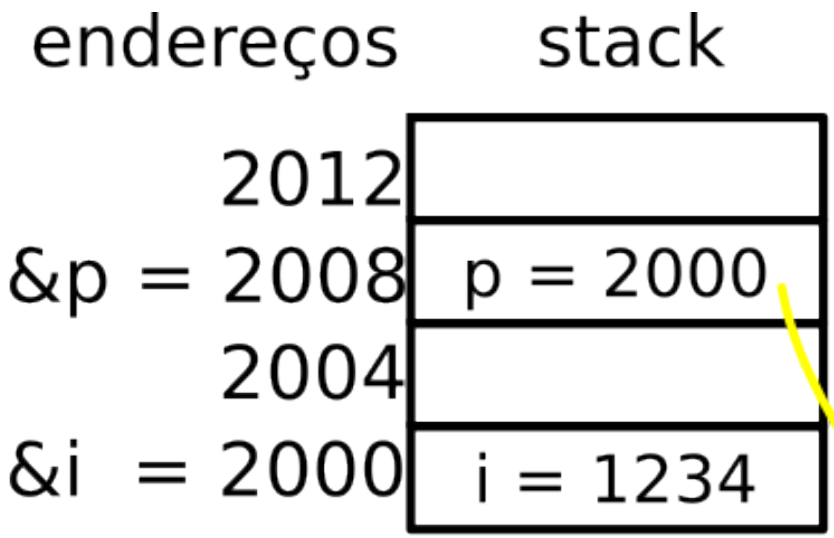
- Mostra o conteúdo de i




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

Ponteiros

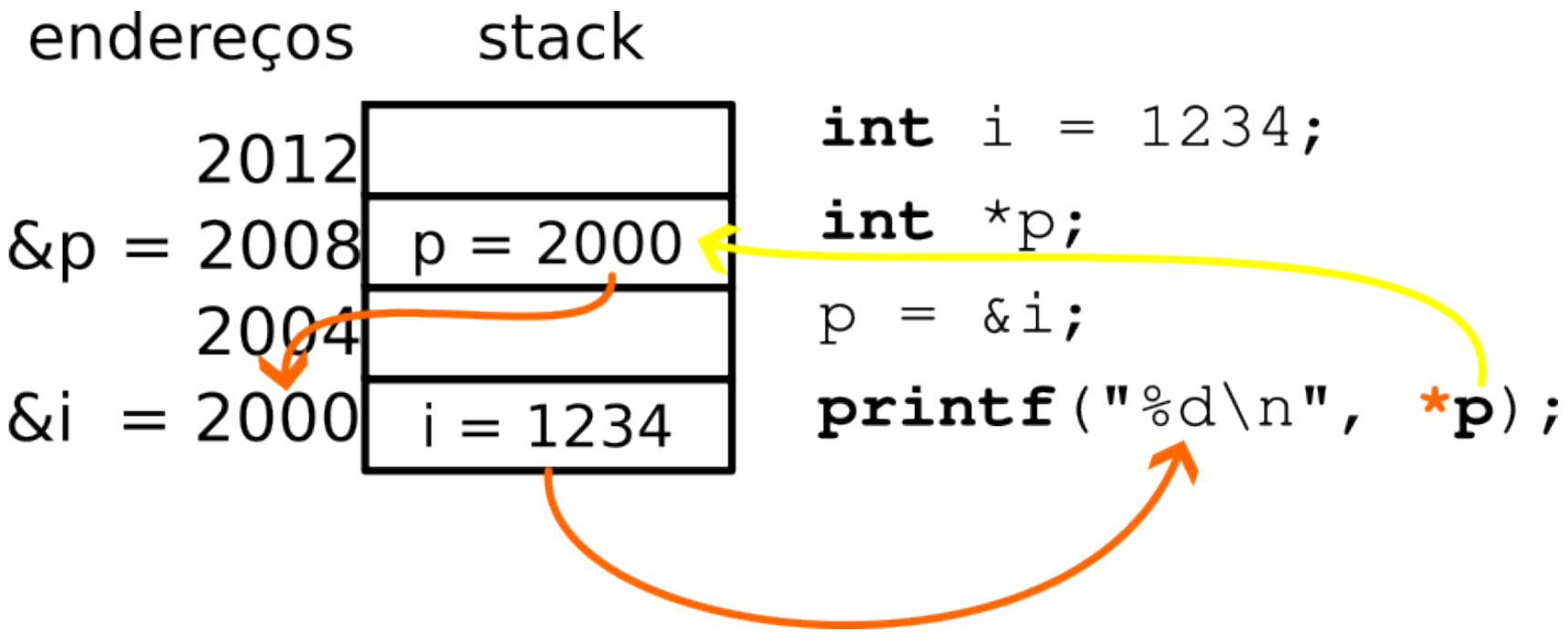
- Endereço conteúdo de p




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

Ponteiros

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

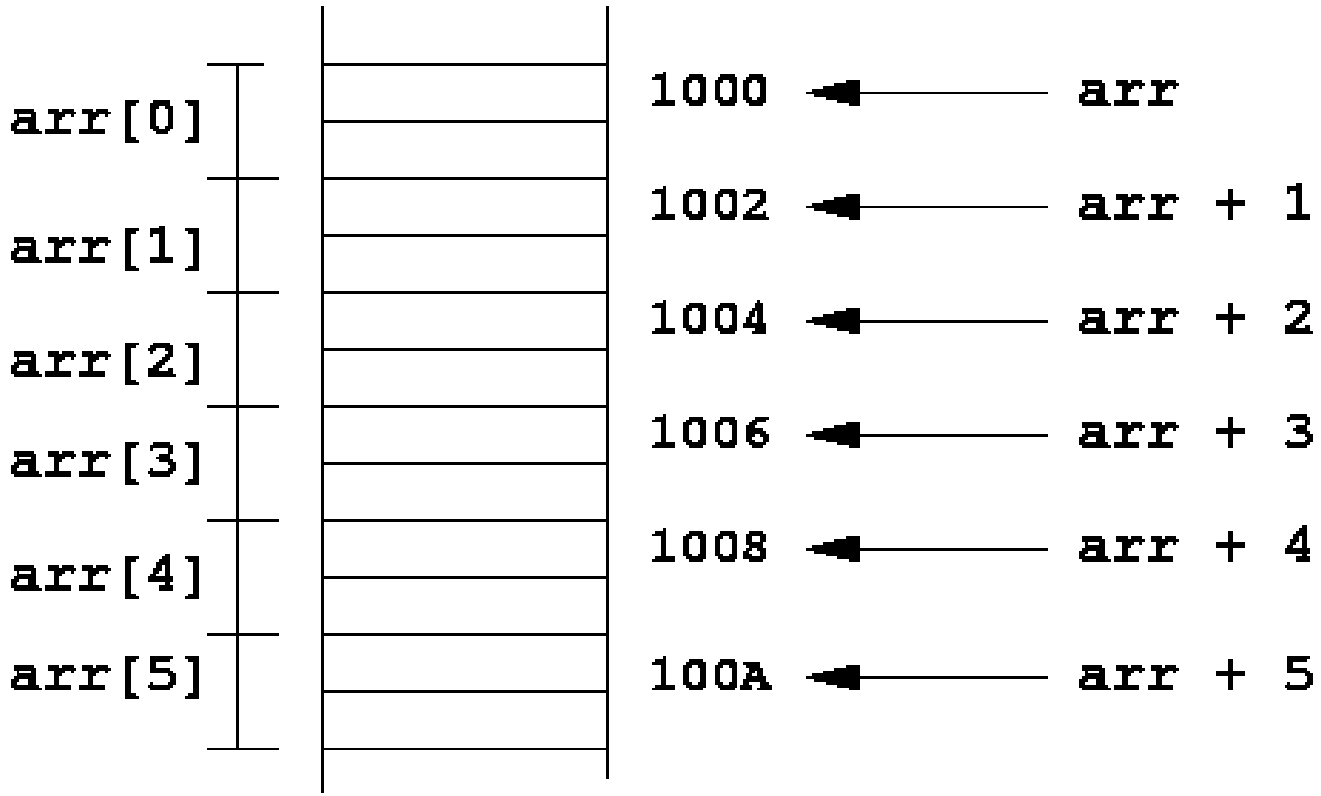


```c
#include <stdio.h>

int main(){
    int i;
    int *p;
    p = NULL;
    p = &i; //diz-se:
            //   p aponta para i
            //   p referencia a variavel i

    i = 5;

    //*p valor da variavel apontada por p, ou seja, valor de i
    printf("%d\n", *p); //*p eh igual a i
                        //saida: 5
    printf("%ld\n", sizeof(p));

    return 0;
}
```

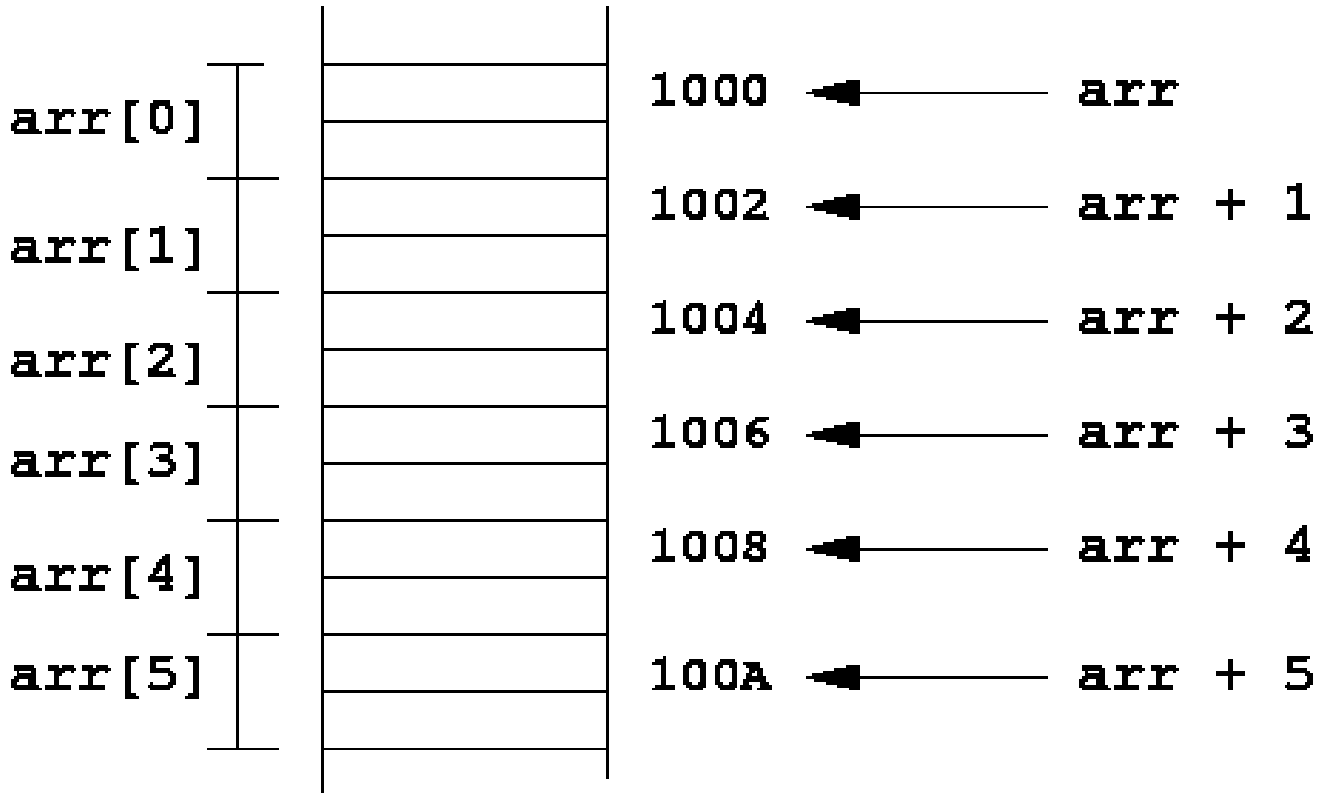
```c
#include <stdio.h>

int main(){
    int x = 256;  //2^8 = 00000000 00000000 00000001 0000000
    char y = 'a'; // 97 = 1100001
    int z = 0;    //  0 = 00000000 00000000 00000000 0000000

    printf("%d\n", x); //saída?
    printf("%ld\n", (long) &z); //140733520157276
    printf("%ld\n", (long) &x); //140733520157272
    printf("%ld\n", (long) &y); //140733520157271

    //memória
    //00000000 00000000 00000000 0000000 -> ..76
    //00000000 00000000 00000001 0000000 -> ..72
    //1100001 -> ..71

    //desloca dois endereços de tamanhos de char
    int *p = (int *)(&y+2);
    printf("%ld\n", (long)p); //140733520157273

    printf("%d\n", *p); //saída?

    return 0;
}
```

```c
#include <stdio.h>

int main(){
    int x = 256;  //2^8 = 00000000 00000000 00000001 0000000
    char y = 'a'; // 97 = 1100001
    int z = 0;    //  0 = 00000000 00000000 00000000 0000000

    printf("%d\n", x); //saída?
    printf("%ld\n", (long) &z); //140733520157276
    printf("%ld\n", (long) &x); //140733520157272
    printf("%ld\n", (long) &y); //140733520157271

    //memória
    //00000000 00000000 00000000 0000000 -> ..76
    //00000000 00000000 00000001 0000000 -> ..72
    //1100001 -> ..71

    //desloca dois endereços de tamanhos de char
    int *p = (int *)(&y+2);
    printf("%ld\n", (long)p); //140733520157273

    printf("%d\n", *p); //saída: 1

    return 0;
}
```

```c
#include <stdio.h>

int main(){

    int v[2] = {3, 7};

    printf("%d %d\n", v[0], v[1]);       //3 7
    printf("%d %d\n", *(v+0), *(v+1)); //? ?

    return 0;
}
```

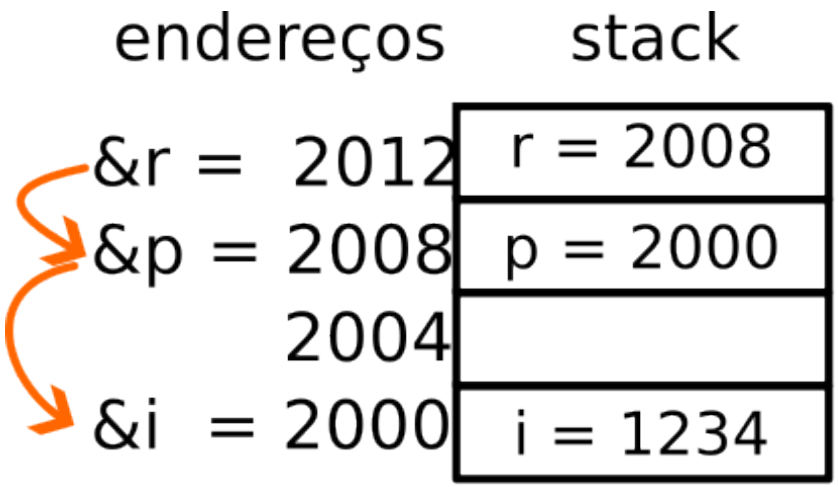


```c
#include <stdio.h>

int main(){

    int v[2] = {3, 7};

    printf("%d %d\n", v[0], v[1]);       //3 7
    printf("%d %d\n", *(v+0), *(v+1)); //3 7

    return 0;
}
```



Ponteiro para ponteiro

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

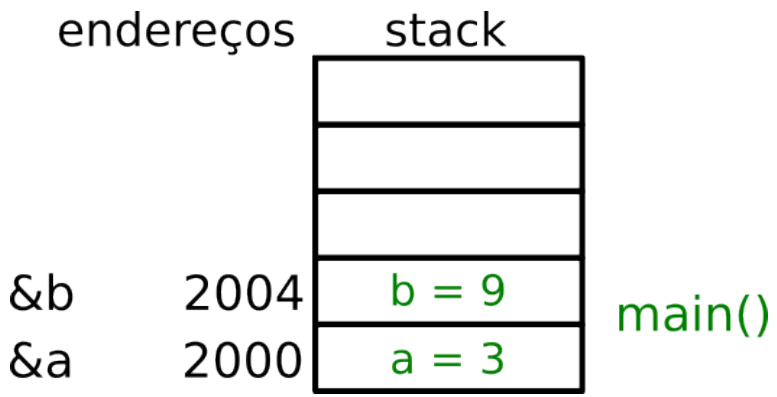



```
int i = 1234;
int *p;
int **r;
p = &i;
r = &p;
printf("%d\n", **r);
```

Ponteiros

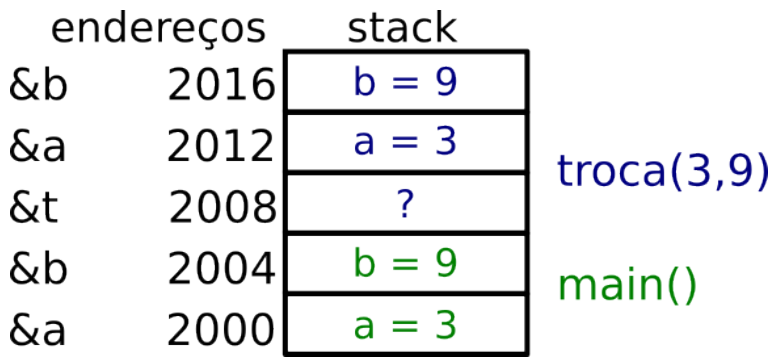
Parâmetros de funções - passagem por cópia/valor

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



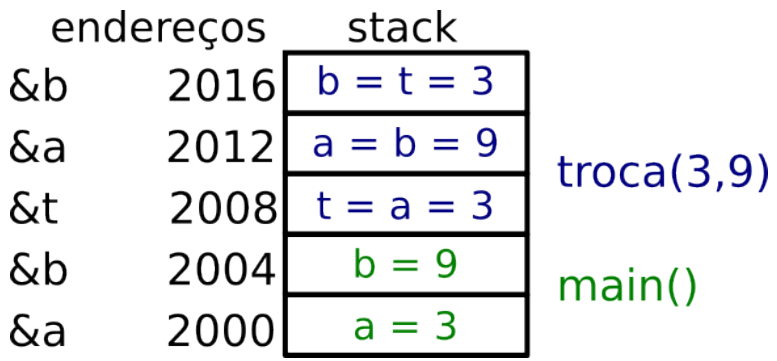
Ponteiros

```
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 }
```



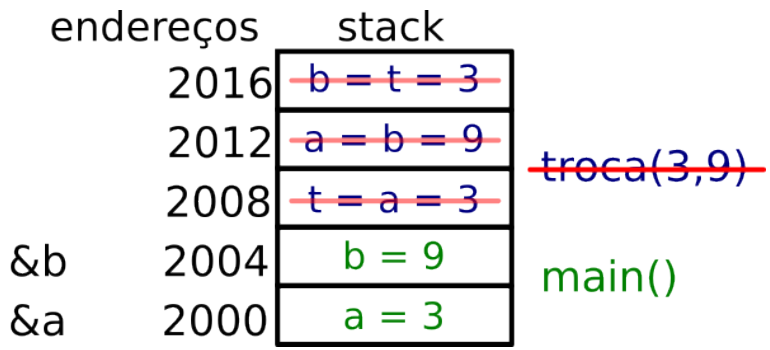
Ponteiros

```c
void troca (int a, int b) {
    int t;
    t = a;   //<<
    a = b;   //<<
    b = t;   //<<
}

int main() {
    int a = 3;
    int b = 9;

    troca(a, b);

    printf("%d %d\n", a, b);
}
```



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

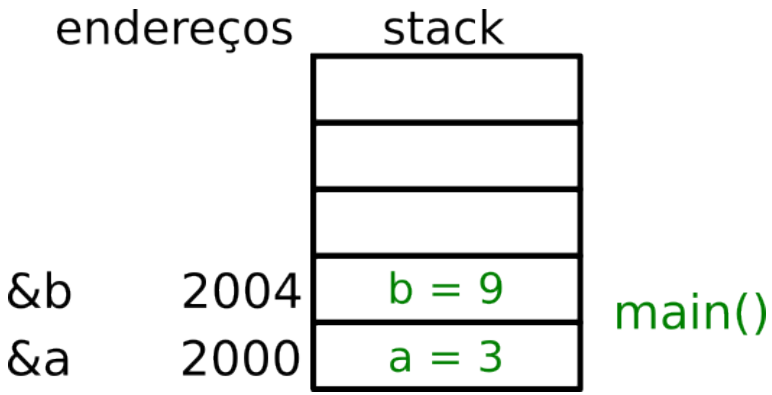
Parâmetros de funções - passagem por referência

- Passar o endereço de uma variável para salvar modificações

```c
void troca (int *p, int *q) { //ponteiros recebem endereços
    int t;
    t = *p;
    *p = *q;
    *q = t;
}

int main() {
    int a = 3, b = 9;

    troca(&a, &b); //passando os endereços

    printf("%d %d\n", a, b);
}
```

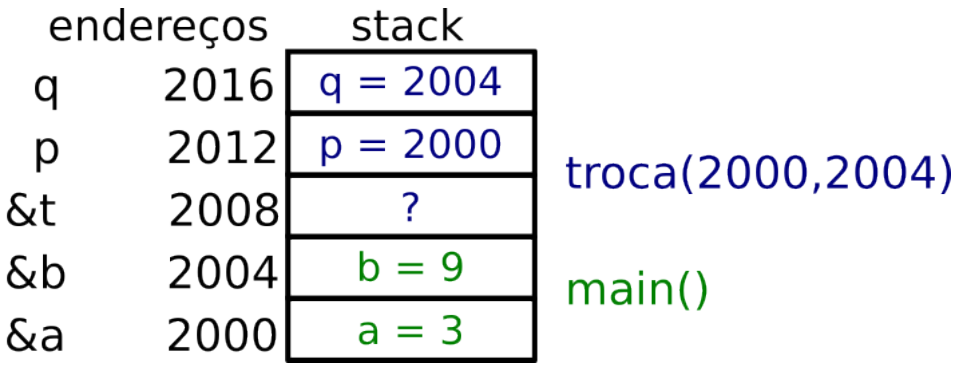
Ponteiros

```c
void troca (int *p, int *q) {
    int t;
    t = *p;
    *p = *q;
    *q = t;
}

int main() { //<<
    int a = 3, b = 9; //<<

    troca(&a, &b);

    printf("%d %d\n", a, b);
}
```



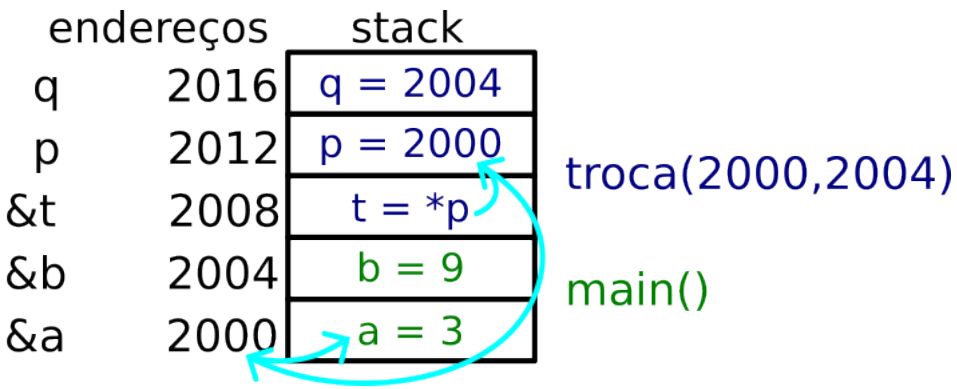
Ponteiros

```c
void troca (int *p, int *q) { //<<
    int t; //<<
    t = *p;
    *p = *q;
    *q = t;
}

int main() {
    int a = 3, b = 9;

    troca(&a, &b); //<<

    printf("%d %d\n", a, b);
}
```



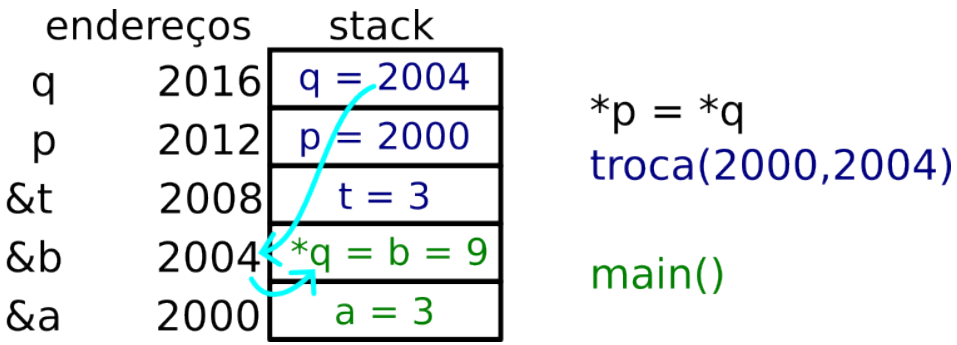
Ponteiros

```c
void troca (int *p, int *q) {
    int t;
    t = *p; //conteúdo de t = conteúdo do apontado por p
    *p = *q;
    *q = t;
}

int main() {
    int a = 3, b = 9;

    troca(&a, &b);

    printf("%d %d\n", a, b);
}
```



Ponteiros

```c
void troca (int *p, int *q) {
    int t;
    t = *p;
    *p = *q; //*q -> conteúdo do apontado por q
    *q = t;
}

int main() {
    int a = 3, b = 9;

    troca(&a, &b);

    printf("%d %d\n", a, b);
}
```

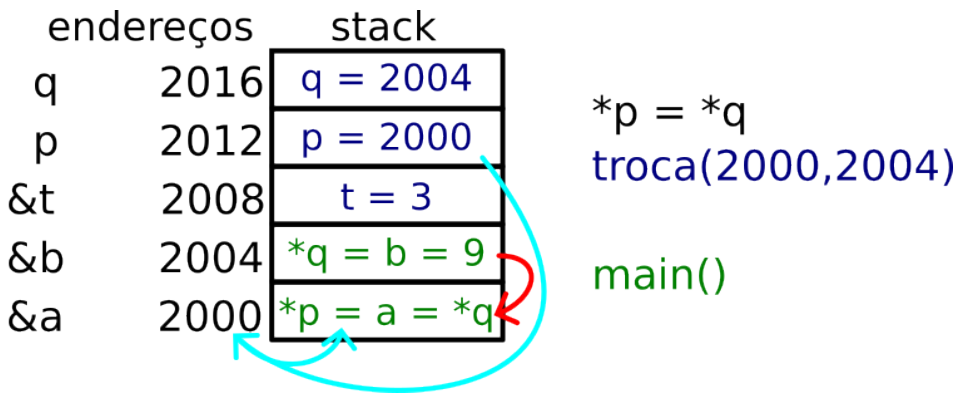


*p = *q
troca(2000,2004)

main()

Ponteiros

```c
void troca (int *p, int *q) {
    int t;
    t = *p;
    *p = *q; //*p -> conteúdo do apontado por p
             //cont. do apontado por p = cont. do apontado por q
    *q = t;
}

int main() {
    int a = 3, b = 9;

    troca(&a, &b);

    printf("%d %d\n", a, b);
}
```



Ponteiros

```c
void troca (int *p, int *q) {
    int t;
    t = *p;
    *p = *q; //cont. do apontado por p = cont. do apontado por q
    *q = t;
}

int main() {
    int a = 3, b = 9;

    troca(&a, &b);

    printf("%d %d\n", a, b);
}
```

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

*p = *q
troca(2000,2004)

main()

Ponteiros

```c
void troca (int *p, int *q) {
    int t;
    t = *p;
    *p = *q;
    *q = t; //conteúdo do apontado por q = conteúdo de t
}

int main() {
    int a = 3, b = 9;

    troca(&a, &b);

    printf("%d %d\n", a, b);
}
```

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

*q = t

troca(2000,2004)

main()

Ponteiros

```c
void troca (int *p, int *q) {
    int t;
    t = *p;
    *p = *q;
    *q = t; //conteúdo do apontado por q = conteúdo de t
}

int main() {
    int a = 3, b = 9;

    troca(&a, &b);

    printf("%d %d\n", a, b);
}
```

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

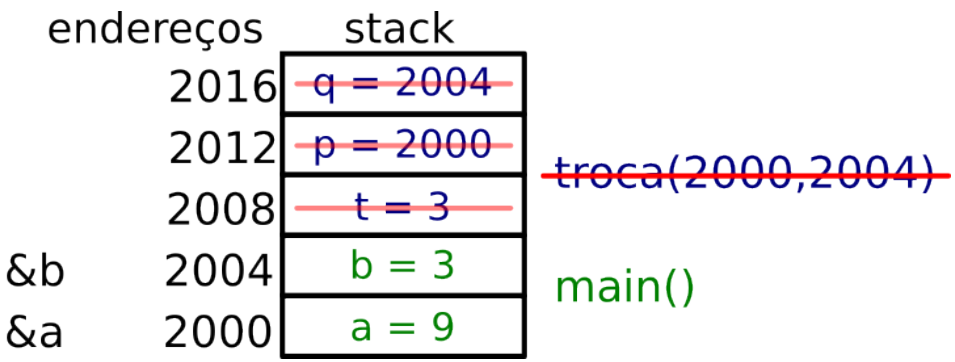
*q = t

troca(2000,2004)

main()

Ponteiros

```c
void troca (int *p, int *q) {
    int t;
    t = *p;
    *p = *q;
    *q = t;
} //<<

int main() {
    int a = 3, b = 9;

    troca(&a, &b); //<<

    printf("%d %d\n", a, b); //<< saída??
}
```



Ponteiro x Array

- Ponteiros podem referenciar arrays: apontar para o endereço inicial

```c
#include <stdio.h>

int main(){
    int v[2] = {1,2};
    int m[2][2] = {1,2,3,4};

    int *p;
    p = v;
    p[0] = 5;

    printf("%d %d\n", p[0], p[1]);

    p = m[0]; //(int *)m
    for(int i=0; i<4; i++)
        printf("%2d", p[i]);

    return 0;
}
```

Ponteiro x Struct

```c
#include <stdio.h>

typedef struct {
    int value;
}Point;

int main(){

    Point s;
    Point *ptr = &s;

    //variável simples
    s.value = 20; //campo acessado por '.'

    //conteúdo do apontado por ptr = variável simples
    (*ptr).value = 40; //campo acessado por '.'

    //acesso ao campo pelo ponteiro
    ptr->value = 30; //setinha ->

    printf("%d\n", s.value); //saída??
    return 0;
}
```

Ponteiro x Struct

```c
struct disciplina {
    int codigo;
    int periodo;
    struct disciplina *requisito;
};

int main() {
    struct disciplina d1, d2;
    d1.codigo = 123;
    d1.periodo = 1;

    d2.codigo = 345;
    d2.periodo = 2;
    d2.requisito = &d1;

    printf("%d %d\n", d2.codigo, d2.requisito->codigo);
}
```

Ponteiro x Função

```c
void f(int a) {
    printf("%d\n", a);
}

void f2(int a) {
    printf("%d\n", a+1);
}

int main(int argc, char **argv) {
    void (*fp)(int);

    if(argc>1 && strcmp(argv[1], "f")==0)
        fp = &f;
    else
        fp = &f2;

    (*fp)(10);

    void (*funcoes[2])(int) = { &f, &f2};
    for(int i=0; i<2; i++)
        (*funcoes[i])(i);

    return 0;
}
```

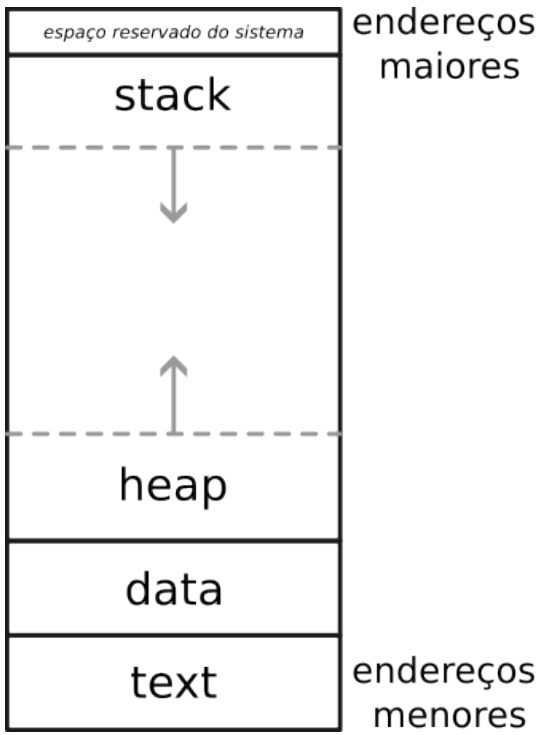
Roteiro

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:



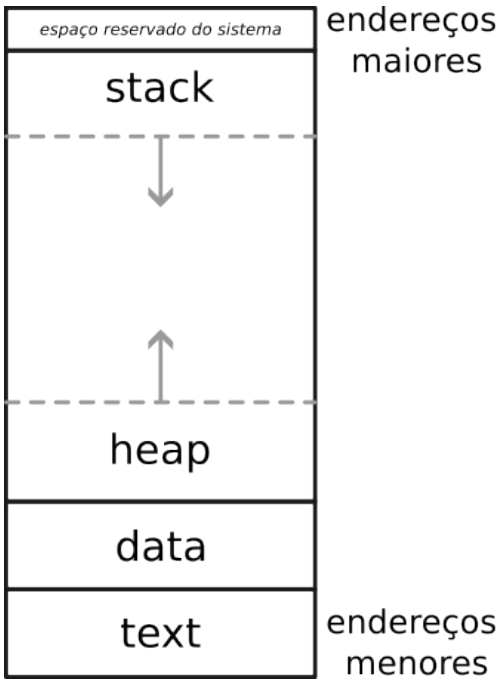
- stack: variáveis locais, parâmetros de funções e endereços de retorno (instrução que chamou uma determinada função)
- heap: blocos de memória alocadas dinamicamente, a pedido do processo (gerenciado pelo sistema operacional)
- data: variáveis globais e estáticas
- text: código que está sendo executado
- 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

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

```c
//variaveis.h
#ifndef _VARIAVEIS_H_
#define _VARIAVEIS_H_
#include <stdio.h>
#include <stdlib.h>
int retornaVar1();
int retornaVar2();
#endif
```

```c
//arquivo: variaveis.c
//compilar: gcc -c variaveis.c -o libvariaveis.o
#include "variaveis.h"

int var1 = 1;          //variável global, aloc. estática
static int var2 = 2; //variável estática global, aloc. estática
                       //acessível somente no arquivo

int retornaVar1(){
    return var1;
}

int retornaVar2(){
    return var2;
}
```

Alocação estática de memória - Data - exemplo

```c
//arquivo: teste.c
//compilar: gcc teste.c libvariaveis.o
//executar: ./a.out
#include "variaveis.h"

extern int var1; //global de variaveis.c
int var2;

int main() {
    printf("%d\n", var1); //1
    printf("%d\n", retornaVar1()); //1
    printf("%d\n", retornaVar2()); //2

    var1 = 4;
    printf("%d\n", var1); //4
    printf("%d\n", retornaVar1()); //4

    var2 = 5;
    printf("%d\n", var2); //5
    printf("%d\n", retornaVar2()); //2
    return 0;
}
```

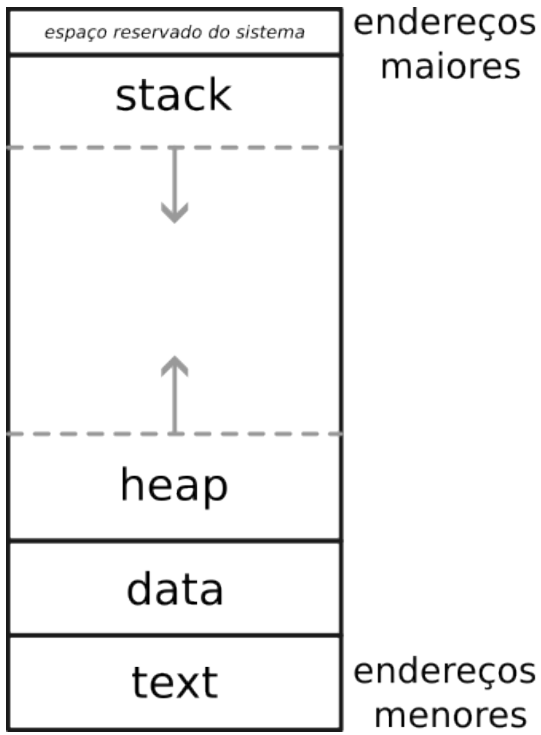
Alocação estática de memória - Data - exemplo

```c
int a = 0;

void incrementa(void) {
    static int c = 0 ; //variável local, aloc. estática
                       //alocada uma única vez e
                       //válida mesmo após o término da função

    printf ("a: %d, c: %d\n", a, c) ;
    a++;
    c++;
}
int main(void) {
    for (int i = 0; i < 5; i++)
        incrementa() ;
    return 0 ;
}
//A execução desse código gera a seguinte saída:
//  a: 0, c: 0
//  a: 1, c: 1
//  a: 2, c: 2
//  a: 3, c: 3
//  a: 4, c: 4
```

Roteiro

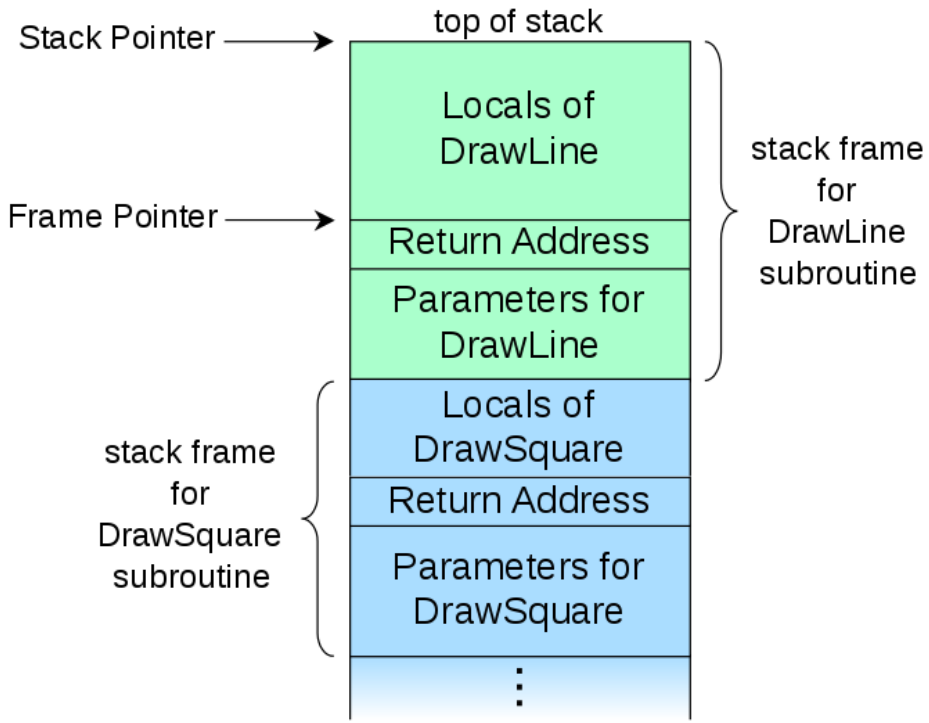
Alocação automática de memória - Stack

- 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	00000000 00000000	00000000 00000000	540 536	 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					...	

```
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);
```

Alocação estática x automática

```c
#include <stdio.h>

int *fa()
{
    int a = -9; //stack ou data?
    int *i = &a;
    return i;
}

int *fb()
{
    static int a = 1; //stack ou data?
    int *i = &a;
    return i;
}

int main() {
    int *b = fa();
    int *c = fb();
    printf("%d %d\n", *b, *c); //saída?

    return 0;
}
```

Alocação estática x automática

```c
#include <stdio.h>

int *fa()
{
    int a = -9; //stack ou data?
    int *i = &a;
    return i;
}

int *fb()
{
    static int a = 1; //stack ou data?
    int *i = &a;
    return i;
}

int main() {
    int *b = fa();
    int *c = fb();
    printf("%d %d\n", *b, *c); //-9 1

    return 0;
}
```

Alocação estática x automática

```c
#include <stdio.h>
int *fa() {
    int a = -9;
    int *i = &a;
    return i;
}

int *fb() {
    static int a = 1;
    int *i = &a;
    return i;
}

void fc() {
    int s[10] = {0}; //stack ou data?
}

int main() {
    int *b = fa(), *c = fb();
    fc();
    printf("%d %d\n", *b, *c); //saídas??
    return 0;
}
```

Alocação estática x automática

```c
#include <stdio.h>
int *fa() {
    int a = -9;
    int *i = &a;
    return i;
}

int *fb() {
    static int a = 1;
    int *i = &a;
    return i;
}

void fc() {
    int s[10] = {0}; //stack ou data?
}

int main() {
    int *b = fa(), *c = fb();
    fc();
    printf("%d %d\n", *b, *c); //0 1
    return 0;
}
```

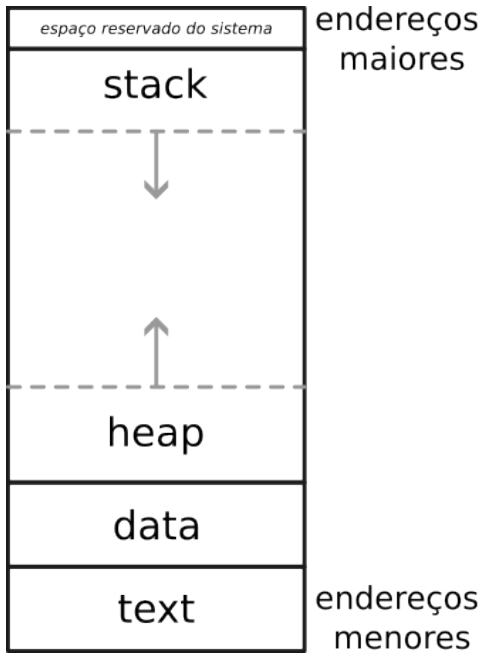
Roteiro

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

```c
#include <stdlib.h>

void *malloc(size_t size);
void free(void *ptr);
void *calloc(size_t nmemb, size_t size);
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  //typecast: 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

```c
#include <stdlib.h>

void free(void *ptr);
```

```c
int *p = malloc(sizeof(int));
free(p);

int b = 4;
int *a;
a = &b;
//free(a) ?
```

Alocação dinâmica de memória - Função calloc

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

```c
#include <stdlib.h>

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

```c
#include <stdlib.h>

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:

```c
int *p = malloc(sizeof(int));
p = realloc(p, 4*sizeof(int));
free(p);
```

```c
int *fa() {
    int *v = malloc(10*sizeof(int)); //stack ou heap
    for(int i=0; i<10; i++) v[i] = 1;
    return v;
}
int *fb() {
    int *v = malloc(10*sizeof(int)); //stack ou heap
    for(int i=0; i<10; i++) v[i] = 2;
    return v;
}
int main() {
    int *a, *b;
    a = fa();
    b = a;
    a = fb();

    for(int i=0; i<10; i++)
        printf("%d ", a[i]); //saída?
    printf("\n");

    for(int i=0; i<10; i++)
        printf("%d ", b[i]); //saída?
    printf("\n");

    return 0;
}
```

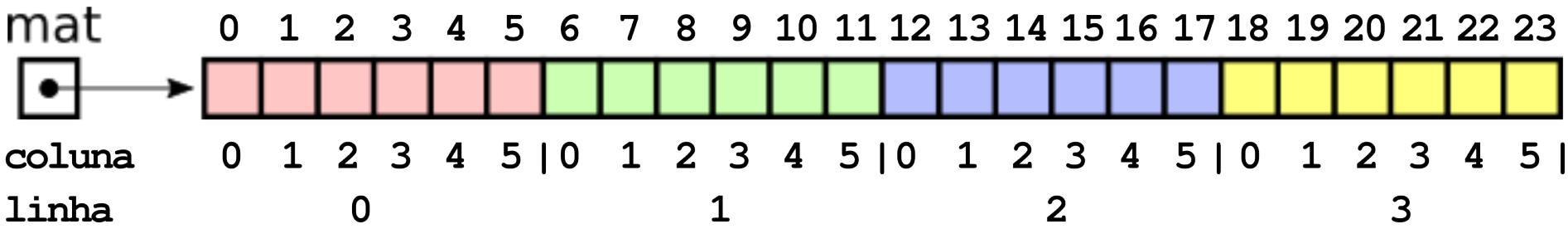
```c
int *fa() {
    int *v = malloc(10*sizeof(int)); //stack ou heap?
    for(int i=0; i<10; i++) v[i] = 1;
    return v;
}
int *fb() {
    int *v = malloc(10*sizeof(int)); //stack ou heap?
    for(int i=0; i<10; i++) v[i] = 2;
    return v;
}


int main() {
    int *a, *b;
    a = fa();
    b = a;
    free(a);
    a = fb();

    for(int i=0; i<10; i++)
        printf("%d ", a[i]); //saída?
    printf("\n");

    for(int i=0; i<10; i++)
        printf("%d ", b[i]); //saída?
    printf("\n");

```

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

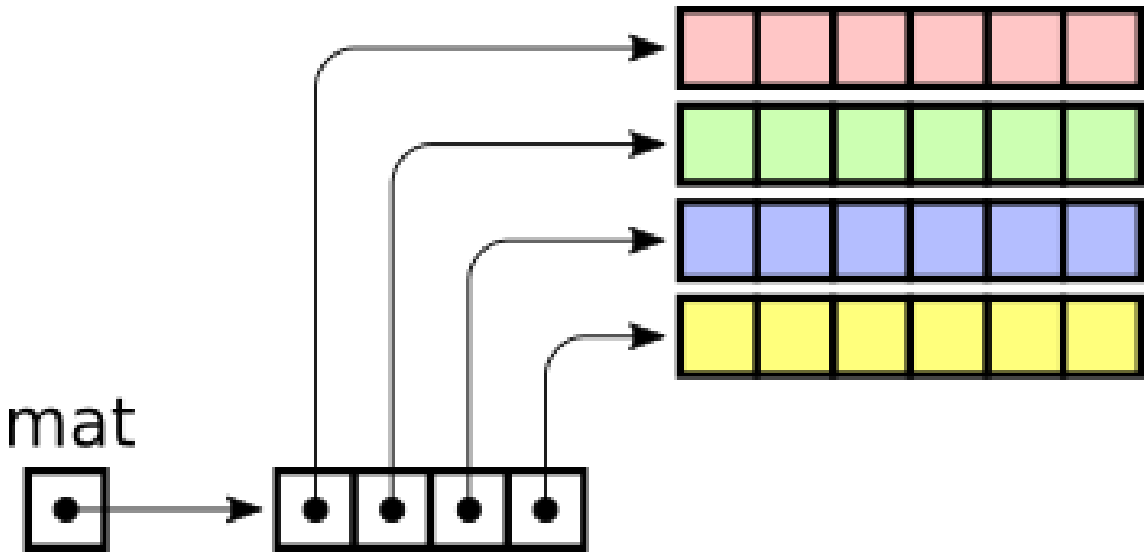
Exemplos: Alocação dinâmica de uma Matriz (linear)

```
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 }
```