Ponteiros e Memória

Profª. Rose Yuri Shimizu

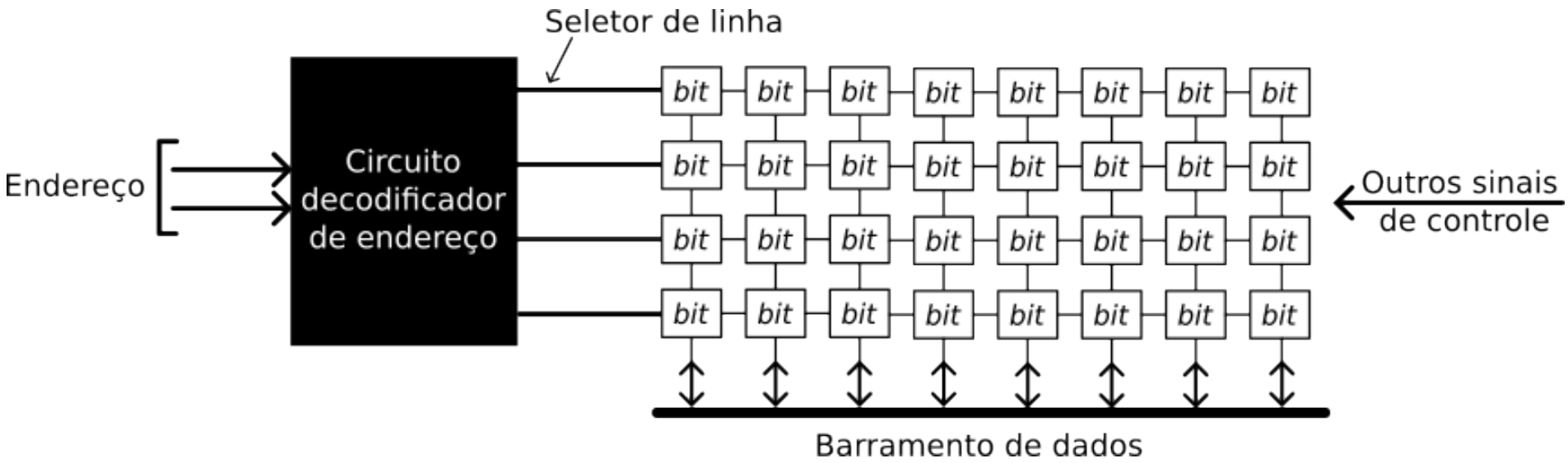
Roteiro

1. Memória
2. Processo x Memória
3. Alocação estática de memória
4. Ponteiros - manipulação de endereços
5. Alocação dinâmica de memória

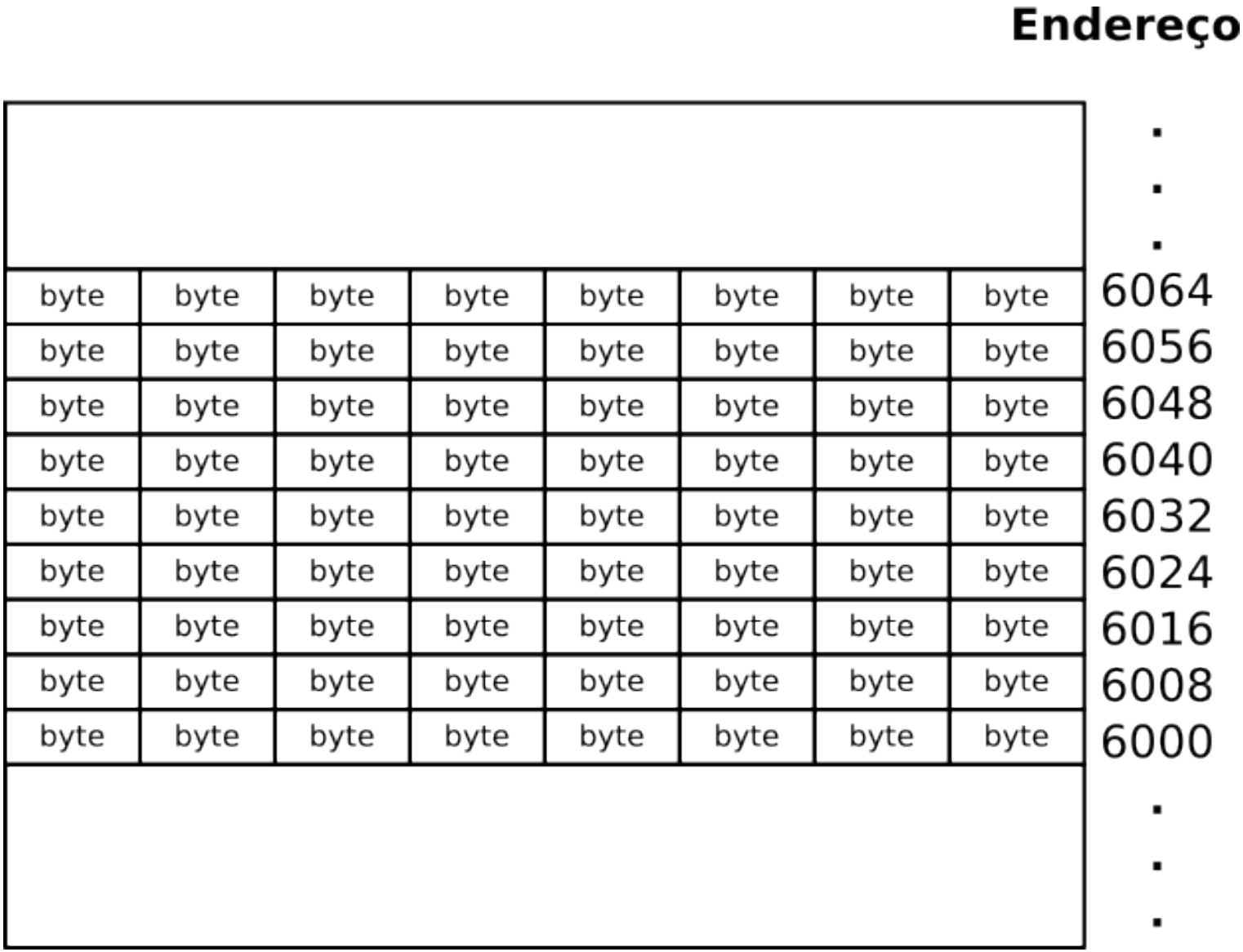
Memória física

- 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

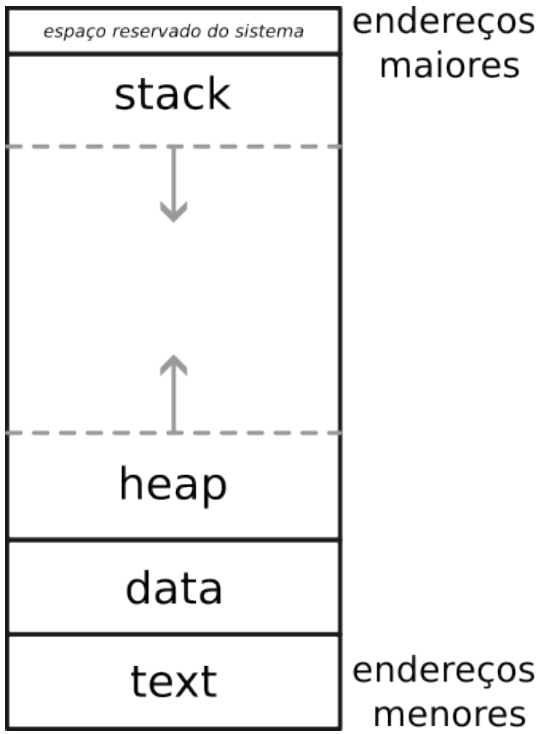


Roteiro

1. Memória
2. Processo x Memória
3. Alocação estática de memória
4. Ponteiros - manipulação de endereços
5. Alocação dinâmica de memória

Alocação de memória para os processos

- Programa em execução: processo
- Cada processo: aloca uma porção da memória
- Cada porção: organizada por segmentos
- Segmentos:



- stack: variáveis locais e endereços de retorno (instrução que chamou uma determinada função)
- heap: memória dinâmica (compartilhável com outras threads)
- data: variáveis globais inicializadas e não inicializadas (.bss)
- text: código que está sendo executado
- Comando: **size** *executavel*
  Lista os espaços ocupados por segmento de um executável

Alocação de memória para os processos

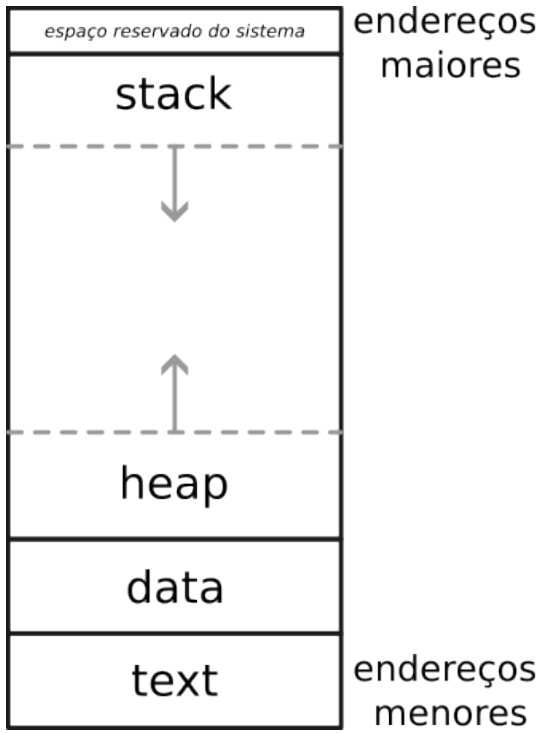
```
rysh@mundodalua:~/Documents/FGA$ gcc teste.c -o teste
rysh@mundodalua:~/Documents/FGA$ size teste
   text    data     bss     dec     hex filename
   1228     544       8    1780     6f4 teste
```

Roteiro

1. Memória
2. Processo x Memória
3. Alocação estática de memória
4. Ponteiros - manipulação de endereços
5. Alocação dinâmica de memória

Alocação na pilha de execução ou chamada (stack)

- Armazena
  - Variáveis locais
  - 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)



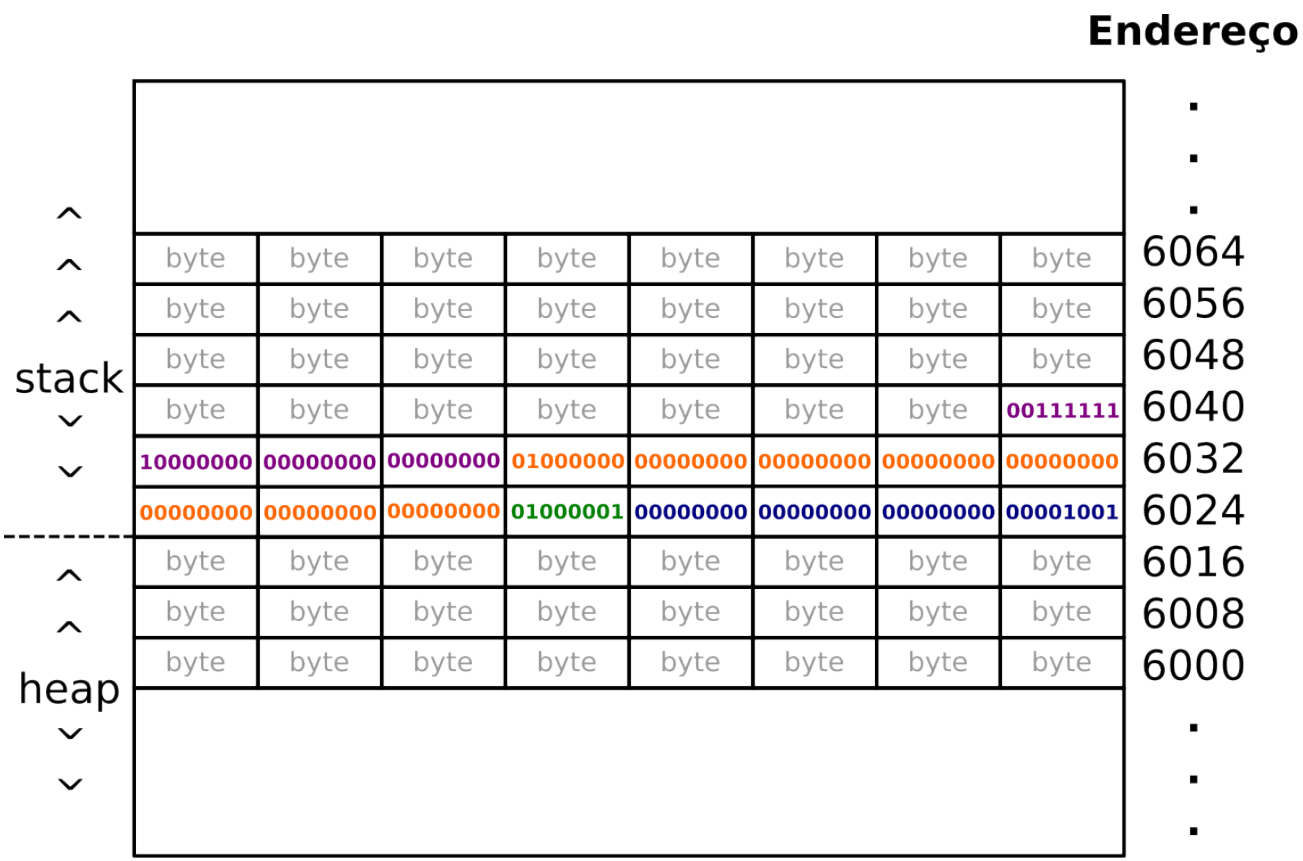
Alocação na pilha de execução ou chamada (stack)

Alocação de variáveis

- Cada tipo ocupa uma quantidade distinta
- Alocação estática (tamanho e quantidade definido antes da execução)
- Cada variável possui um **endereço na memória**
  - Byte menos significativo - início da alocação
  - Alocação contínua

Alocação na pilha de execução ou chamada (stack)

- Endereço = byte menos significativo (início da alocação)
- Alocação contínua



int x = 9;

int : ocupa 32 bits (4 bytes)
&x : 6024 (endereço hipotético)

---

char l = 'A';

char : ocupa 8 bits (1 byte)
&l : 6028 (endereço hipotético)

---

double z =2.0;

double : ocupa 64 bits (8 bytes)
&z : 6029 (endereço hipotético)

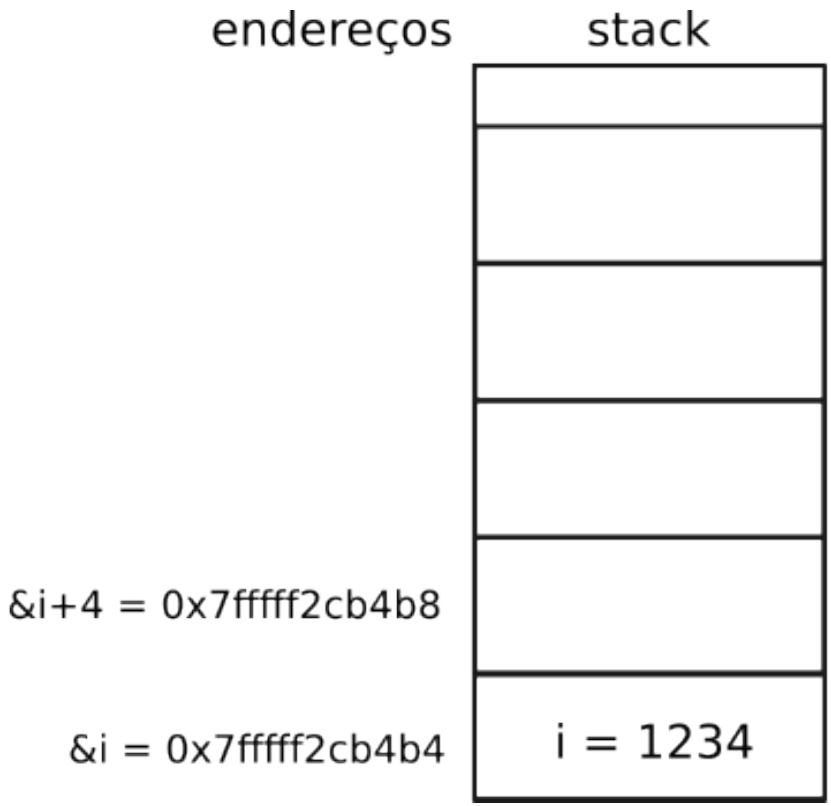
---

float y = 1.0;

float : ocupa 32 bits (4 bytes)
&y : 6037 (endereço hipotético)

Alocação na pilha de execução ou chamada (stack)

- Exemplo: variável na 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 = 140737474507956
 &i = 0x7fffff2cb4b4
```

Alocação na pilha de execução ou chamada (stack)

Array (vetor, matriz, string) na memória

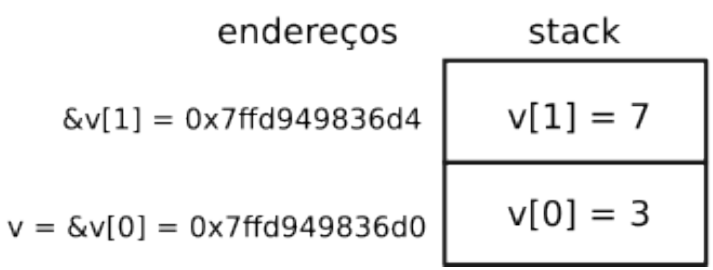
- Cada posição é uma variável local
- Portanto, cada posição tem um endereço
- Cada posição, é calculada a partir do endereço inicial
- Endereço inicial, é apontado pelo identificador (nome) do array

Alocação na pilha de execução ou chamada (stack)

- Exemplo: vetor na stack

```
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
```



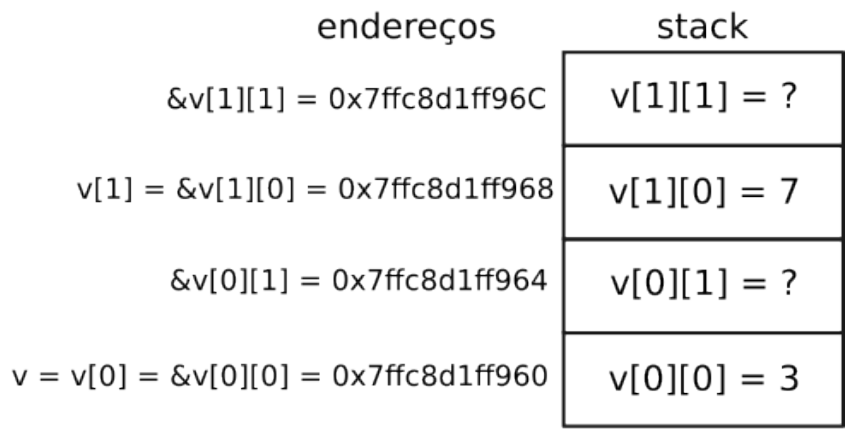
Alocação na pilha de execução ou chamada (stack)

- Exemplo: matriz na stack

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

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

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

//   endereco do primeiro elemento do primeiro vetor
printf ("&v[0][0] = %p\n", (void *) &v[0][0]);

//-----------------------------------------------------
//endereco da linha 1 (segundo vetor)
printf ("endereco de v[1] %p\n", (void *)v[1]);

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

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

Saída

```
endereco de v 0x7ffc8d1ff960

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

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



Roteiro

1. Memória
2. Processo x Memória
3. Alocação estática de memória
4. Ponteiros - manipulação de endereços
5. Alocação dinâmica de memória

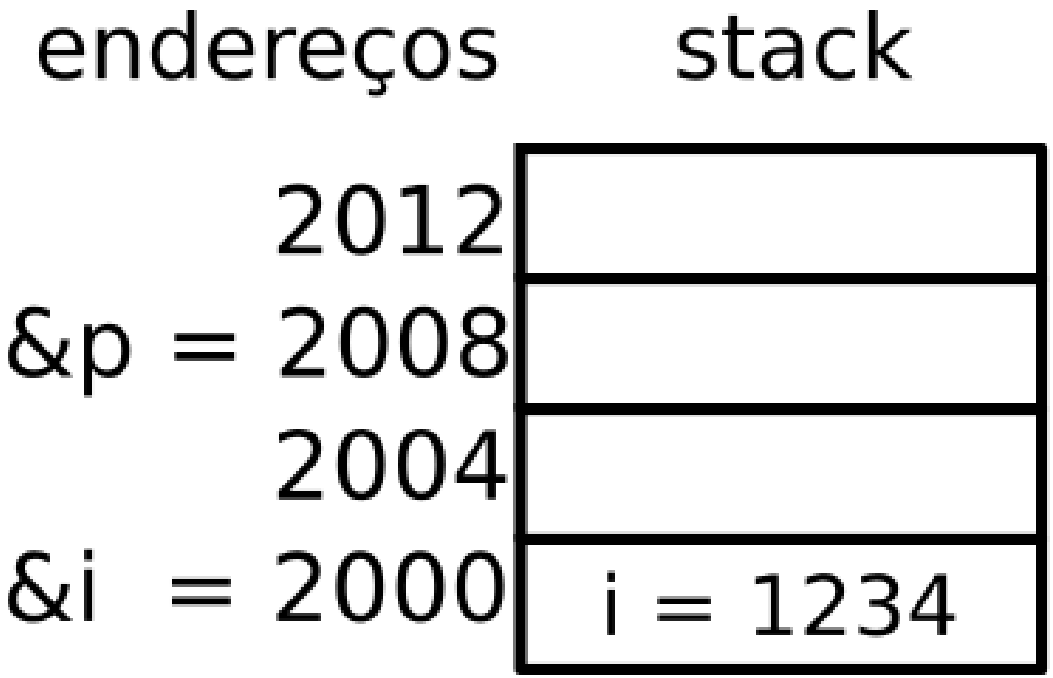
Como manipular os endereços?

- Através dos ponteiros
- Ponteiro
  - 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, ponteiro)
  - Pode ser NULL: indica endereço inválido; definida na interface stdlib.h; valor é 0 (zero)

```
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
```

Como manipular os endereços?

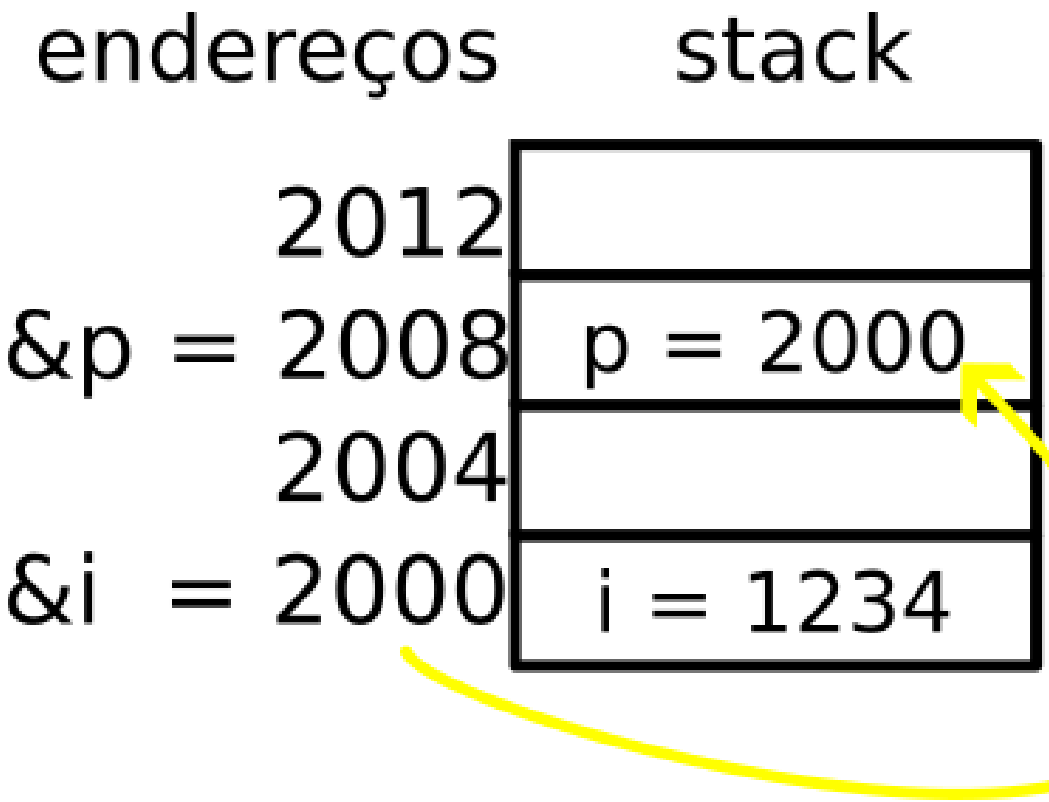
- Aloca i e p
- Conteúdo em i




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

Como manipular os endereços?

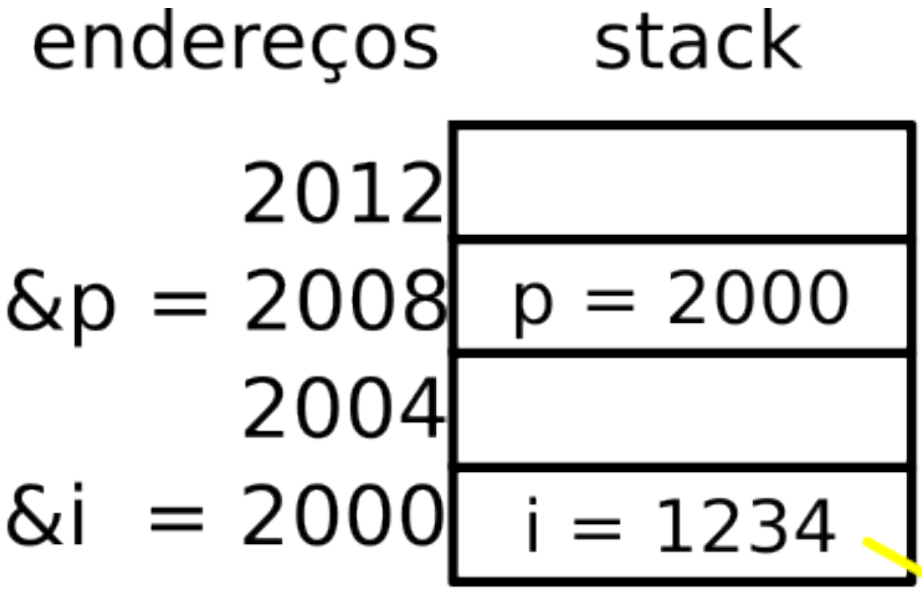
- Conteúdo de p = endereço de i




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

Como manipular os endereços?

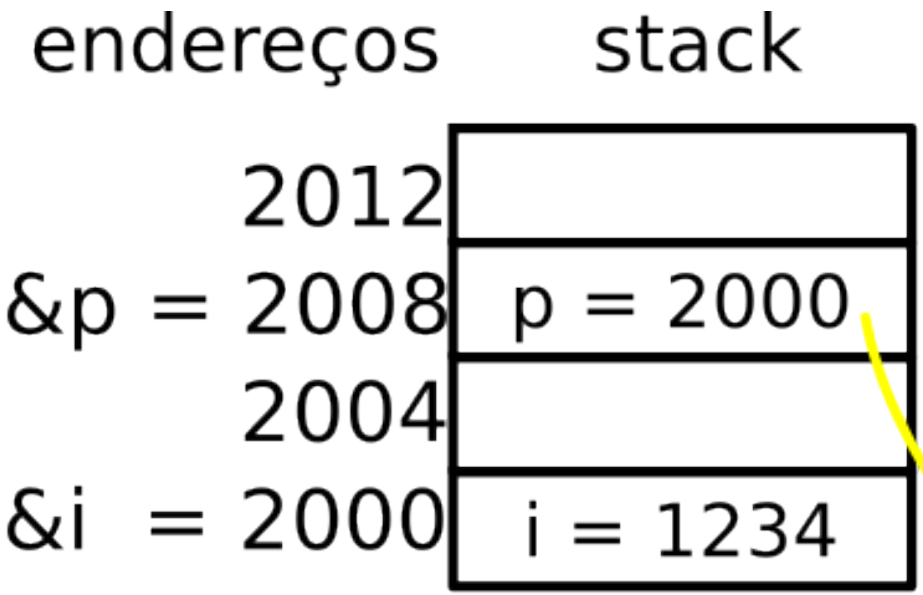
- Mostra o conteúdo de i




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

Como manipular os endereços?

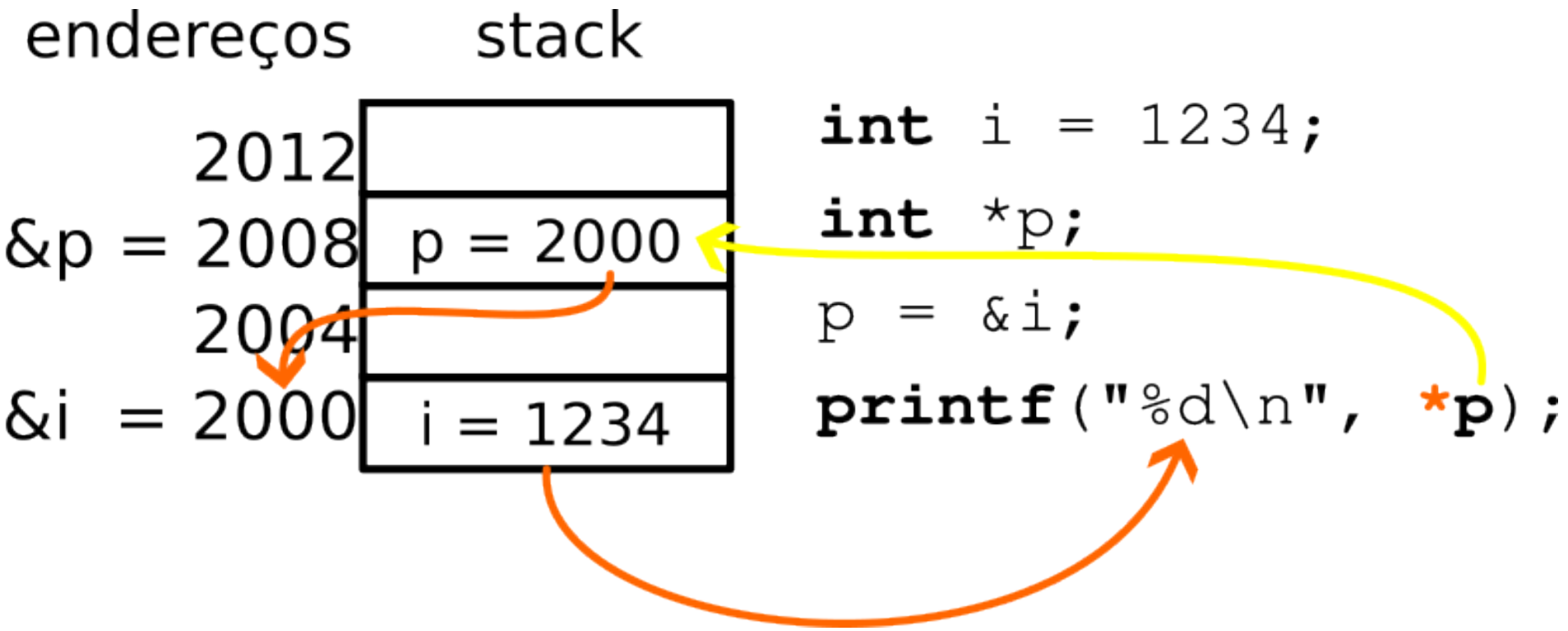
- Mostra o conteúdo de p




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

Como manipular os endereços?

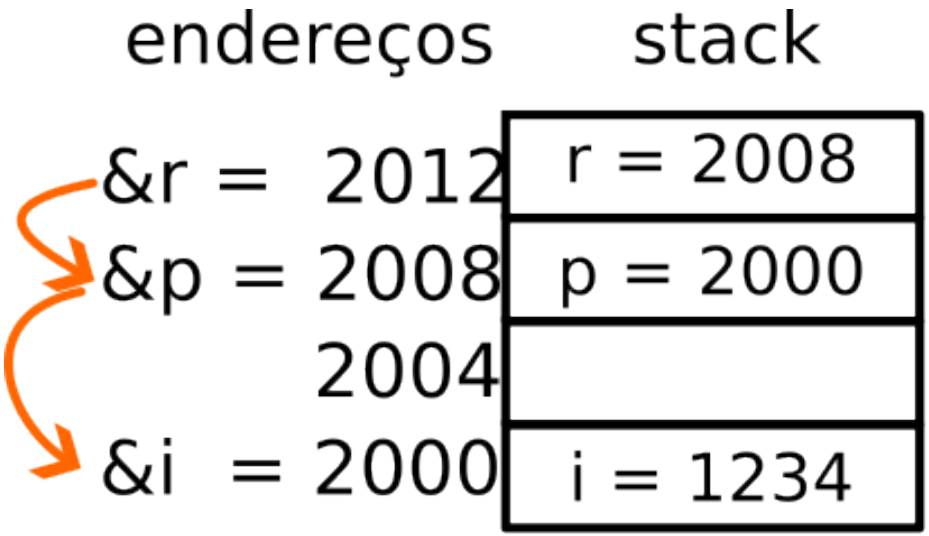
- Mostra o conteúdo da variável apontado por p



Como manipular os endereços?

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);
```

Como manipular os endereços?

Parâmetros de funções

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

Como manipular os endereços?

Parâmetros x Ponteiro

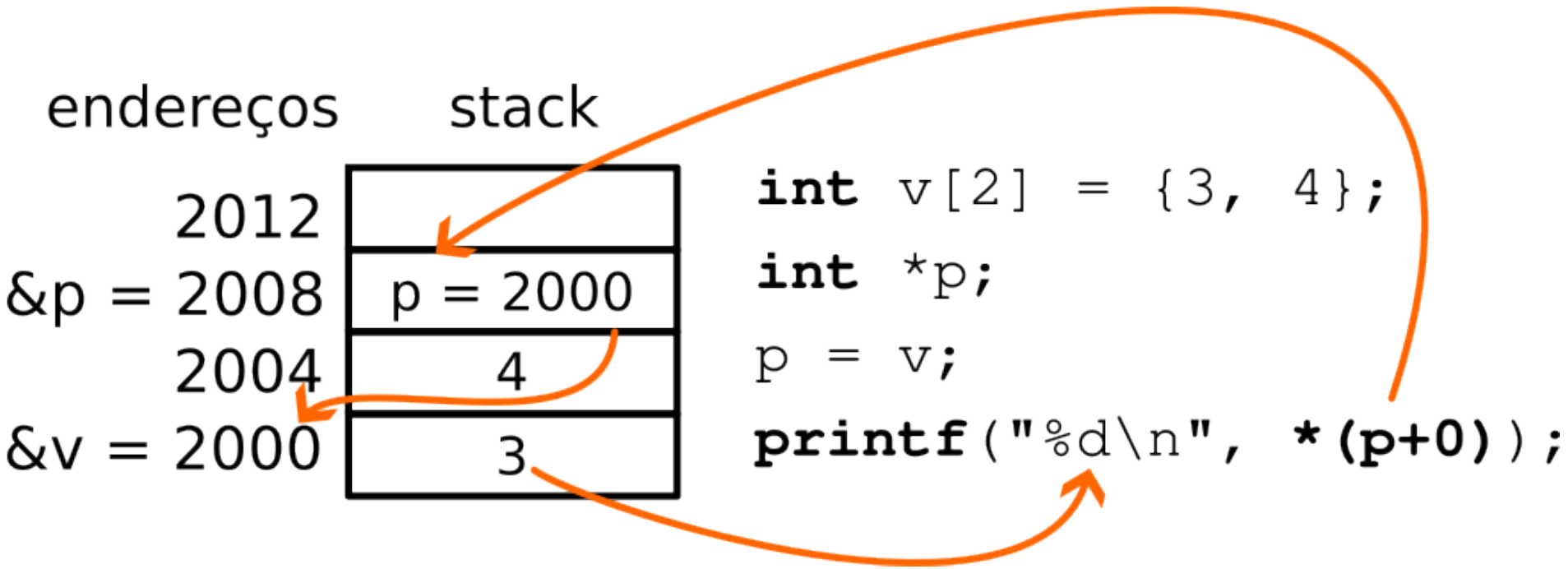
- Passar o endereço de uma variável para salvar modificações

```
1  void troca (int *p, int *q) {
2      int temp;
3      temp = *p; //conteudo de temp = conteudo apontado por p
4      *p = *q;  //conteudo de p = conteudo apontado por q
5      *q = temp; //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
13     int *p, *q;
14     p = &a;
15     q = &b;
16     troca (p, q);
17     printf("%d %d\n", a, b); //saida??
18 }
```

Como manipular os endereços?

Vetor x Ponteiro

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



Como manipular os endereços?

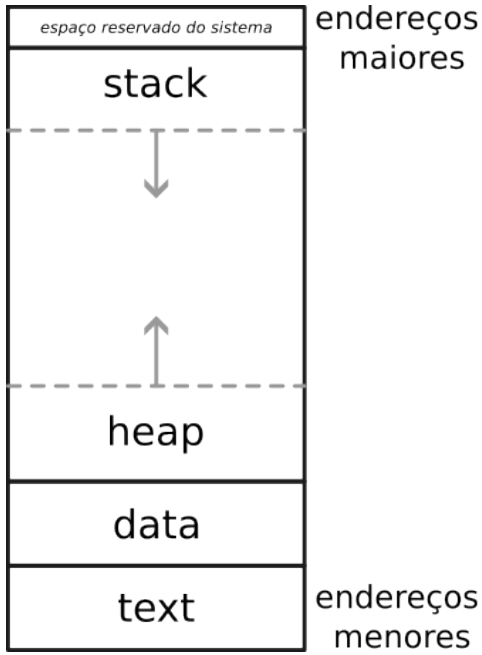
```
1     int i[2]; //i eh o endereco do vetor
2
3     int *p;
4     p = i;  //p armazena i,
5             //portanto, p armazena o endereco de vetor i
6
7     i[1] = 9; //alterando o valor da posicao 1 do endereco apontado
       por i
8
9     printf("%d\n", i[1]); //9
10    printf("%d\n", *(i+1));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

Roteiro

1. Memória
2. Processo x Memória
3. Alocação estática de memória
4. Ponteiros - manipulação de endereços
5. Alocação dinâmica de memória

Alocação dinâmica em C

- Alocar memória em **durante a execução** do programa
- Alocar **tamanhos maiores** que a stack permite
- Alocado no segmento **heap**
- Portanto, permite alocar espaços cujos tamanhos só são conhecidos durante a execução do processo ou tamanho maiores que a stack
- Permitem organizar os dados que vão aumentando/diminuindo com novas entradas
  - Utilizados na implementação das **estruturas de dados**



Alocação dinâmica em C

Funções malloc, realloc, calloc e free

- Biblioteca stdlib.h
- Utilizam uma ou mais heaps
- Estrutura utilizadas pelas funções
  - Arena: estrutura compartilhada por várias threads que contém a referência de uma ou mais heaps para saber quais heaps possuem chuncks (pedaços) livres
  - Heap: área contígua de de memória que são divididos por chuncks (pedaços) que podem ser alocados
  - Chunk: pequeno pedaço de memória que pode estar alocadas pelas aplicações, livres ou combinadas com chunks adjacentes para aumentar sua capacidade
  - Memória: espaço endereçado de armazenamento (RAM ou swap)
- Alocam o tamanho do argumento size_t (tamanho em bytes) - unsigned int
- Tamanho máximo do sizet_t: 18446744073709551615 bytes (macro SIZE_MAX - limits.h)

Alocação dinâmica em C

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 em C

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 em C

Funções malloc

- Aloca uma quantidade de bytes
- Retorna um ponteiro da memória alocada
- Retorna NULL em caso de erro
- Retorna NULL ou um ponteiro em caso da quantidade ser zero
- Não esqueça de verificar se a área foi reservada
- A memória não é inicializada

Alocação dinâmica em C

Exemplos - malloc

```c
int *p = malloc(sizeof(int)); //1 inteiro
char *nome = malloc(sizeof(char)*50); //string 50 posicoes
float *f = malloc(sizeof(float)*10); //vetor float - 10 posicoes

if(f){
    f[1] = 4;
    printf("%f\n",f[1]);
}

struct endereco {
    char rua[100];
    int numero;
};
struct endereco *end;
end = malloc(sizeof(struct endereco));

if(end){
    end->numero = 324;
}
```

Alocação dinâmica em C

Funções free

- Libera o espaço, previamente alocado dinamicamente, apontado por um ponteiro
- Não se esqueça de liberar a memória
- 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 em C

Funções calloc

- Aloca memória para um array de A elementos de tamanho N
  calloc(A, N);
- Retorna um ponteiro da memória alocada
- Retorna NULL em caso de erro
- Detecta overflow (A*N > capacidade)
- Retorna NULL ou um ponteiro em caso da quantidade ser zero
- A memória é inicializada com zero

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

Alocação dinâmica em C

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)
- Se o ponteiro não for NULL e a quantidade solicitada for zero, o espaço apontado é liberado (free)
- Retorna um ponteiro para a nova área alocada (pode ser a mesma ou diferete da original)
- Retorna NULL em caso de erro
  - Bloco original não é afetado, fica inalterado
- Retorna NULL ou um ponteiro em caso da quantidade ser zero

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

Alocação dinâmica em C

Alocação dinâmica - Observações

- Type casting das retornos das funções: versão antigas de C, ou para C++
- Alocação de vetores com variáveis como índices

```
1     int i = 5;
2     int v[i];
3
```

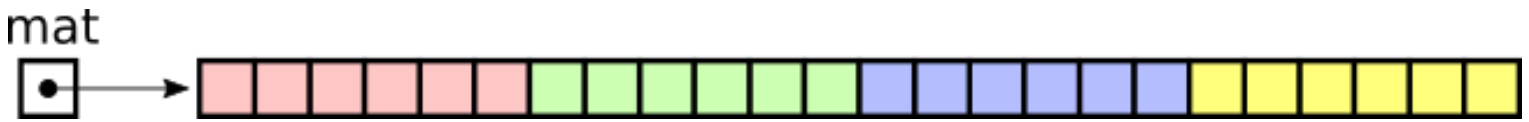
- Quando for grandes vetores, preferência por usar alocação dinâmica para evitar estouro da stack

```
1     int *v;
2     int n;
3     scanf ("%d", &n);
4     v = malloc (n * sizeof (int));
5     ...
6     free(v);
7
```

Alocação dinâmica em C

Exemplos: Alocação dinâmica de uma Matriz (linear)

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



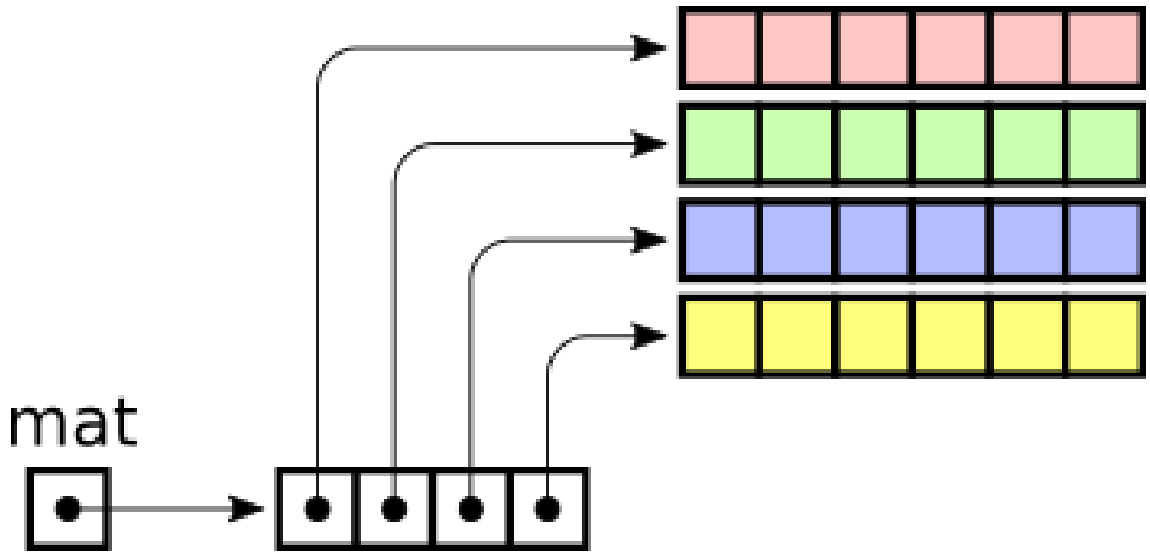
Alocação dinâmica em C

```c
//Elementos da matriz são alocados em um único vetor
#define LIN 4
#define COL 6
int *mat ;
int i, j ;

// aloca um vetor com todos os elementos da matriz
mat = malloc (LIN * COL * sizeof (int)) ;

if(mat){
    // percorre a matriz
    for (i=0; i < LIN; i++)
    for (j=0; j < COL; j++)
        mat[(i*COL) + j] = 0 ; // calcula a posição de cada elemento

    // libera a memória alocada para a matriz
    free (mat) ;
}
```

Alocação dinâmica em C

Exemplos: Alocação dinâmica de uma Matriz (vetores)

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



Alocação dinâmica em C

```c
#define LIN 4
#define COL 6
int **mat, i, j ;

// aloca um vetor de LIN ponteiros para linhas
mat = malloc (LIN * sizeof (int*)) ;

if(mat){
    // aloca cada uma das linhas (vetores de COL inteiros)
    for (i=0; i < LIN; i++)
        mat[i] = malloc (COL * sizeof (int)) ;

    // percorre a matriz
    for (i=0; i < LIN; i++)
       for (j=0; j < COL; j++)
          mat[i][j] = 0 ; // acesso com sintaxe mais simples

    // libera a memória da matriz
    for (i=0; i < LIN; i++) free (mat[i]) ;
    free (mat) ;
}
```