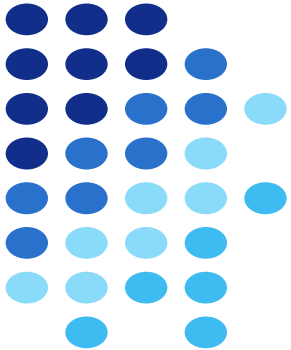
Universidade Federal de Sergipe
Departamento de Sistemas de Informação

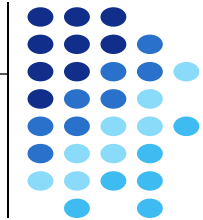
SINF0007 – Estrutura de Dados II

Revisão de C



2

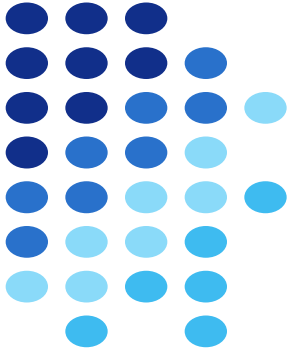
Prof. Dr. Raphael Pereira de Oliveira
raphael.oliveira@academico.ufs.br



Revisão de C

- Ponteiros
- Alocação Dinâmica de Memória
- Ponteiros e Tipos Estruturados
- Recursão

Ponteiros



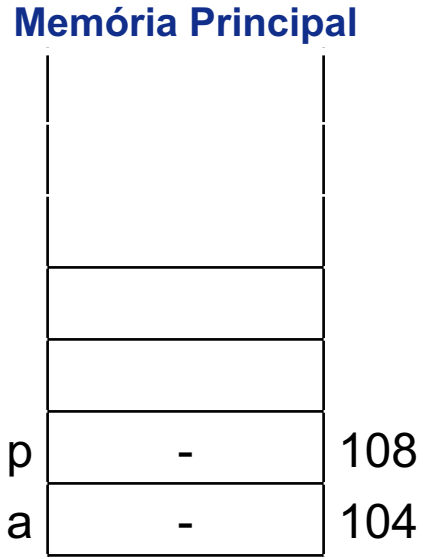
Ponteiros

- C permite o armazenamento e manipulação de valores de endereço de memória

- Para cada tipo de dados existente, há um tipo de ponteiro que pode armazenar endereços de memória onde existem valores daquele tipo armazenados

Exemplo

```
/* variável inteiro */
int a;


/* variável ponteiro para inteiro */
int *p;
```

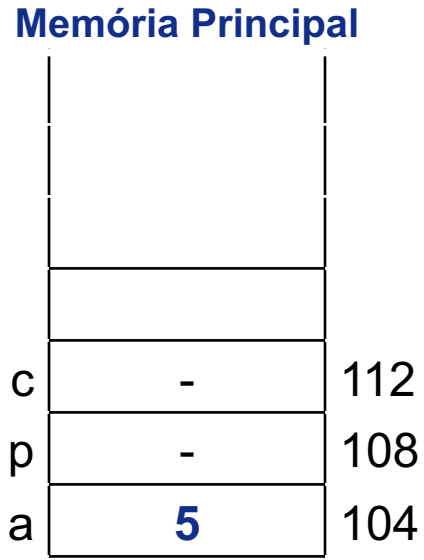


Operadores

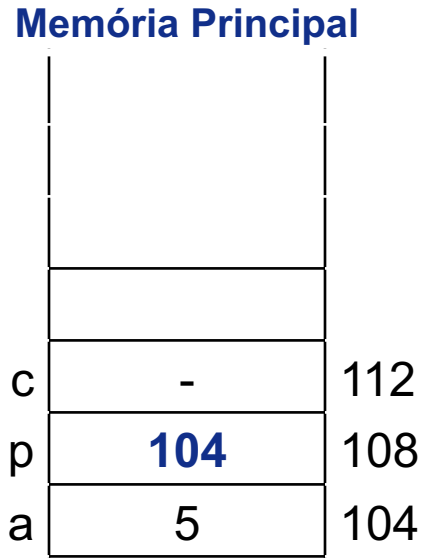
- Operador Unário **&**
  - **Endereço de**
  - Aplicado a uma variável x, resulta no **endereço da posição de memória** reservada para a variável x

- Operador Unário ***
  - **Conteúdo de**
  - Aplicado a variáveis do tipo ponteiro, acessa o **conteúdo do endereço de memória** armazenado pela variável ponteiro

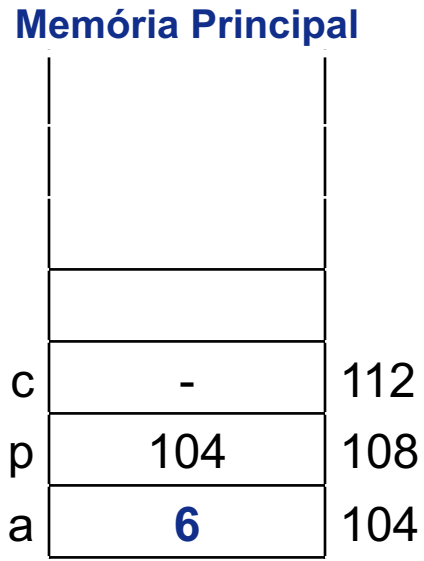
```
int a, *p, c;
/* a recebe o valor 5*/
a = 5;
```



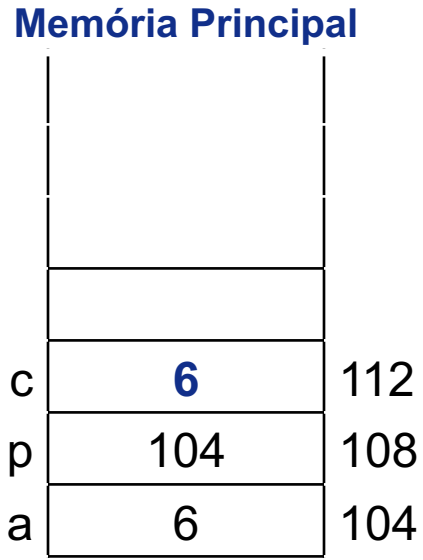
```
int a, *p, c;
/* a recebe o valor 5*/
a = 5;
/* p recebe o endereço de a, ou seja, p aponta para a */
p = &a;
```



```
int a, *p, c;
/* a recebe o valor 5*/
a = 5;
/* p recebe o endereço de a, ou seja, p aponta para a */
p = &a;
/* posição de memória apontada por p recebe 6 */
*p = 6;
```

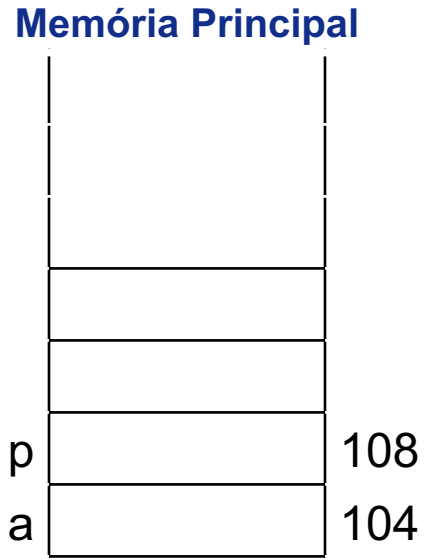


```
int a, *p, c;
/* a recebe o valor 5*/
a = 5;
/* p recebe o endereço de a, ou seja, p aponta para a */
p = &a;
/* posição de memória apontada por p recebe 6 */
*p = 6;
/* c recebe o valor armazenado na posição de memória apontada por p */
c = *p;
```



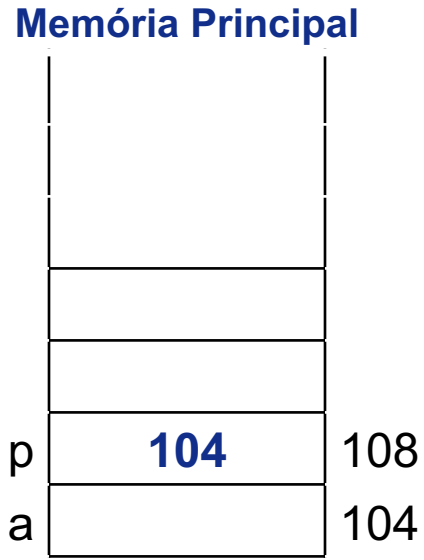
Exemplo

```
int main() {
        int a;
        int *p;
        p = &a;
        *p = 2;
        printf(" %d ",a);
        return 0;
}
```



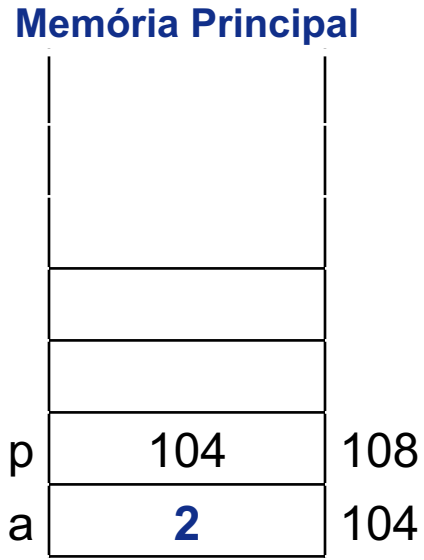
Exemplo

```
int main() {
        int a;
        int *p;
        p = &a;
        *p = 2;
        printf(" %d ",a);
        return 0;
}
```



Exemplo

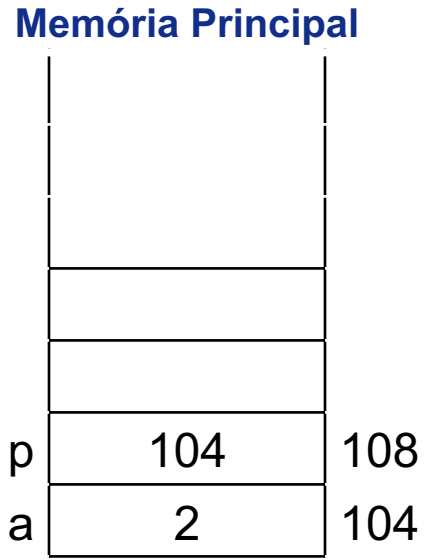
```
int main() {
        int a;
        int *p;
        p = &a;
        *p = 2;
        printf(" %d ",a);
        return 0;
}
```



Exemplo

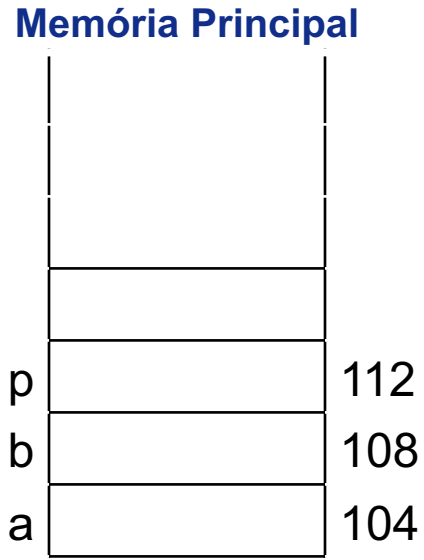
```
int main() {
        int a;
        int *p;
        p = &a;
        *p = 2;
        printf(" %d ",a);
        return 0;
}
```

Imprime o valor 2



Exemplo com ERRO

```
int main() {
        int a, b, *p;
        a = 2;
        *p = 3;
        b = a + (*p);

        printf(" %d ",b);
        return 0;
}
```

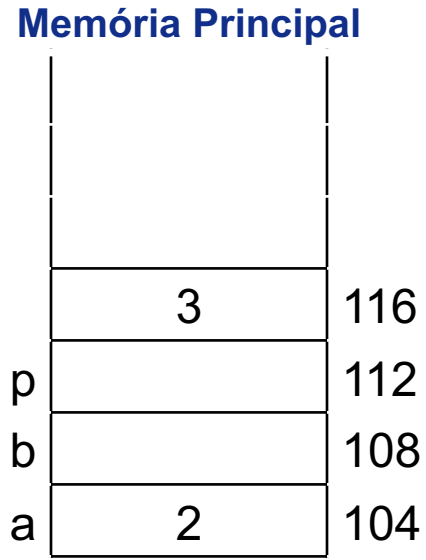


Exemplo com ERRO

```
int main() {
        int a, b, *p;
        a = 2;
        *p = 3;
        b = a + (*p);

        printf(" %d ",b);
        return 0;
}
```

Erro na atribuição *p = 3

- Utiliza a memória apontada por **p** para armazenar o valor **3**, sem que **p** tivesse sido inicializada
- Armazena **3** num espaço de memória desconhecido



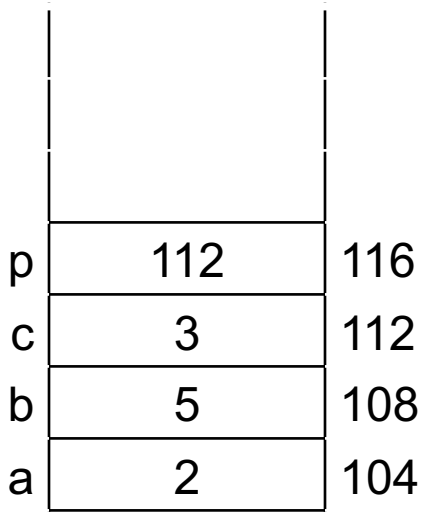
Exemplo CORRIGIDO

```
int main() {
        int a, b, c, *p;
        a = 2;
        p = &c;
        *p = 3;
        b = a + (*p);

        printf(" %d ",b);
        return 0;
}
```

Atribuição *p = 3

- `p` aponta para `c`
- Atribuição armazena `3` no espaço de memória da variável `c`

Memória Principal

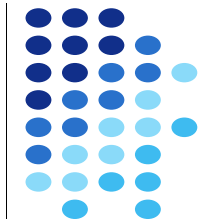


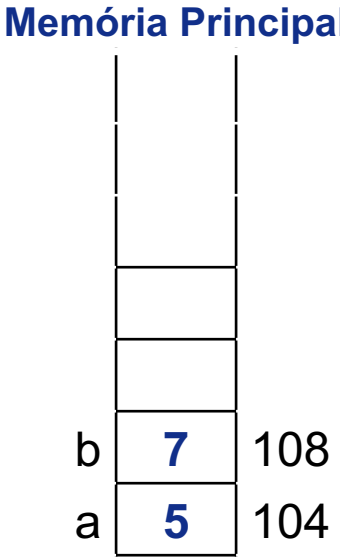
Passagem de Ponteiros para Funções

- Quando uma função **g** chama uma função **f**
  - **f** não pode alterar diretamente os valores das variáveis de **g**, porém,
  - Se **g** passar para **f** os valores dos endereços de memória onde as variáveis de **g** estão armazenadas, **f** pode alterar, indiretamente, os valores das variáveis de **g**

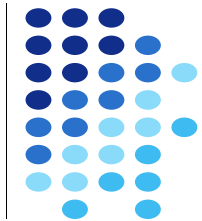
Passagem de Parâmetros por Referência

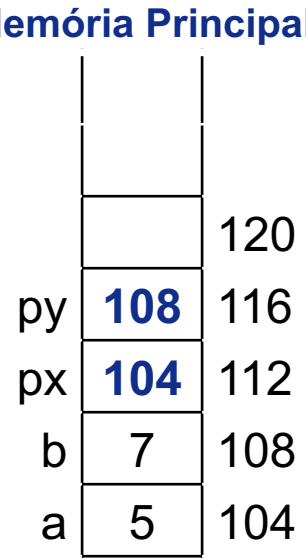
```c
#include <stdio.h>

void troca(int *px, int *py){
        int temp;
        temp = *px;
        *px = *py;
        *py = temp;
}
int main() {
        int a = 5, b = 7;
        troca(&a, &b);/* passamos os
           endereços das variáveis */

        printf("%d %d\n",a, b);
        return 0;
}
```



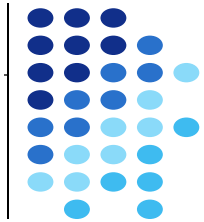


```c
#include <stdio.h>

void troca(int *px, int *py){
        int temp;
        temp = *px;
        *px = *py;
        *py = temp;
}
int main() {
        int a = 5, b = 7;
        troca(&a, &b);/* passamos os
           endereços das variáveis */

        printf("%d %d\n",a, b);
        return 0;
}
```





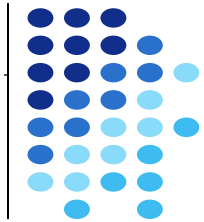
```c
#include <stdio.h>

void troca(int *px, int *py){
        int temp;
        temp = *px;
        *px = *py;
        *py = temp;
}
int main() {
        int a = 5, b = 7;
        troca(&a, &b);/* passamos os
           endereços das variáveis */

        printf("%d %d\n",a, b);
        return 0;
}
```



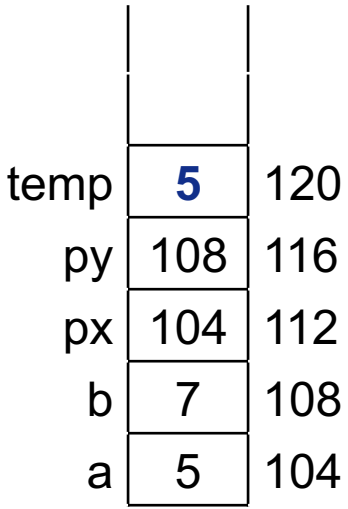
Memória Principal

temp	-	120
py	108	116
px	104	112
b	7	108
a	5	104

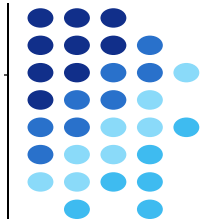
```c
#include <stdio.h>

void troca(int *px, int *py){
        int temp;
        temp = *px;
        *px = *py;
        *py = temp;
}
int main() {
        int a = 5, b = 7;
        troca(&a, &b);/* passamos os
           endereços das variáveis */

        printf("%d %d\n",a, b);
        return 0;
}
```



Memória Principal



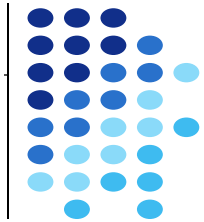
```c
#include <stdio.h>

void troca(int *px, int *py){
        int temp;
        temp = *px;
        *px = *py;
        *py = temp;
}
int main() {
        int a = 5, b = 7;
        troca(&a, &b);/* passamos os
           endereços das variáveis */

        printf("%d %d\n",a, b);
        return 0;
}
```



Memória Principal

temp	5	120
py	108	116
px	104	112
b	7	108
a	**7**	104

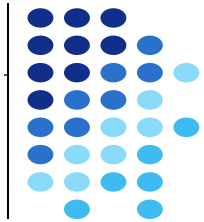
```c
#include <stdio.h>

void troca(int *px, int *py){
        int temp;
        temp = *px;
        *px = *py;
        *py = temp;
}
int main() {
        int a = 5, b = 7;
        troca(&a, &b);/* passamos os
           endereços das variáveis */

        printf("%d %d\n",a, b);
        return 0;
}
```

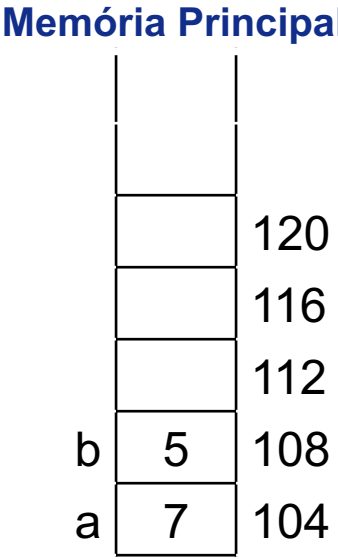


Memória Principal

temp	5	120
py	108	116
px	104	112
b	**5**	108
a	7	104

```c
#include <stdio.h>

void troca(int *px, int *py){
        int temp;
        temp = *px;
        *px = *py;
        *py = temp;
}
int main() {
        int a = 5, b = 7;
        troca(&a, &b);/* passamos os
           endereços das variáveis */

        printf("%d %d\n",a, b);
        return 0;
}
```



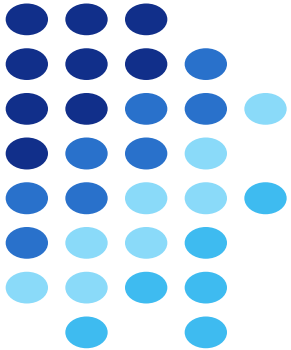


Imprime os valores
7 5

UNIVERSIDADE FEDERAL DE SERGIPE

dsi departamento de sistemas de informação

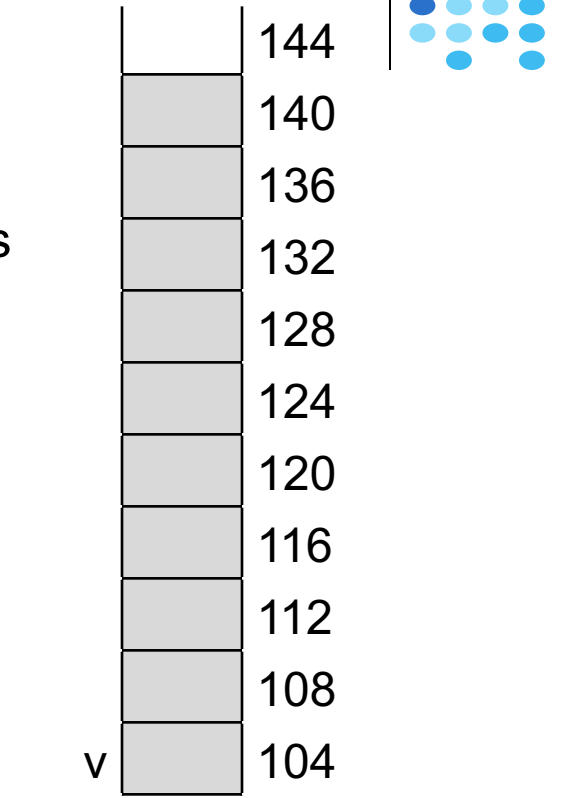
Alocação Dinâmica de Memória



Exemplo para Explicar o Conceito de Alocação Dinâmica: VETORES

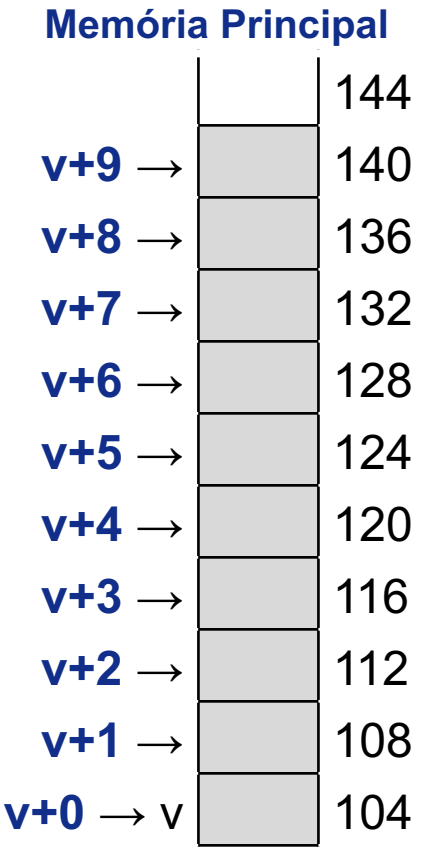
- Um vetor é alocado em posições contíguas de memória. Exemplo:
  - Seja **v** um vetor de **10** elementos inteiros
  - Espaço de memória de v = 10 x valores inteiros de 4 bytes = **40 bytes**
  - Alocação estática! (espaço de memória é reservado no momento da declaração do vetor)

```
int v[10];
```

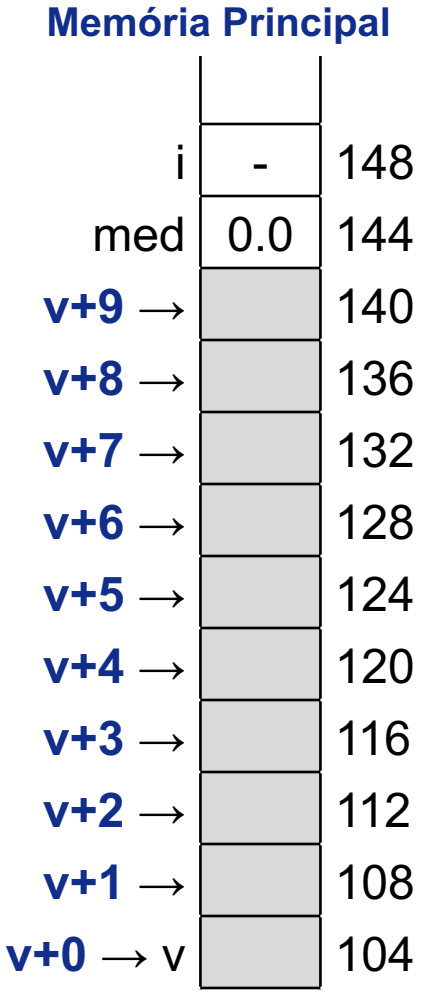


Vetores e Ponteiros

- Formas de Acesso:
  - `&v[i]` é equivalente à `v+i`
  - `*(v+i)` é equivalente à `v[i]`



```c
/* Média de 10 números reais */
#include <stdio.h>
int main () {
        float v[10];
        float med = 0.0;
        int i;
        // leitura dos valores (Endereços)
        for (i = 0; i < 10; i++)
           //scanf("%f", &v[i]); ou
           scanf("%f", v+i); // aritmética
        // cálculo da média (Valores)
        for (i = 0; i < 10; i++)
           //med = med + v[i]; ou
           med = med + *(v+i); // aritmética
        med = med / 10;
        // exibição do resultado
        printf ( "Media = %.2f \n", med );
        return 0;
}
```



Passagem de Parâmetro Vetor para Função

- Deve-se passar o **`endereço da primeira posição do vetor`**

- Função deve ter **`parâmetro do tipo ponteiro`** para armazenar o valor passado como parâmetro
  - Passar um vetor para uma função é equivalente a passar o endereço inicial do vetor
  - Elementos do vetor não são copiados para a função
  - Argumento copiado é apenas o endereço do primeiro elemento do vetor

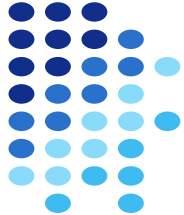
```c
#include <stdio.h>

float media(int n, float *v){
        int i;
        float s = 0.0;
        for (i = 0; i < n; i++)
                s += v[i];
        return s/n;
}
int main () {
        float v[10];
        float med;
        int i;
        for (i = 0; i < 10; i++)
            scanf("%f", &v[i]);
        med = media(10,v);
        printf ( "Media = %.2f \n", med );
        return 0;
}
```

Parâmetro do tipo ponteiro para **`float`**

Função pode Alterar Valores do Vetor

Como o que é passado é o endereço do vetor, e não uma cópia dos valores, a função pode alterar os valores do vetor e as alterações serão refletidas no programa principal

```c
/* Incrementa elementos de um vetor */
#include <stdio.h>
void incr_vetor ( int n, int *v ) {
        int i;
        for (i = 0; i < n; i++)
                v[i]++;
}

int main () {
        int a[ ] = {1, 3, 5};
        incr_vetor(3, a);
        printf("%d %d %d \n", a[0], a[1], a[2]);
        return 0;
}
```



Incrementa o vetor da **função main**

Imprime os valores 2 4 6

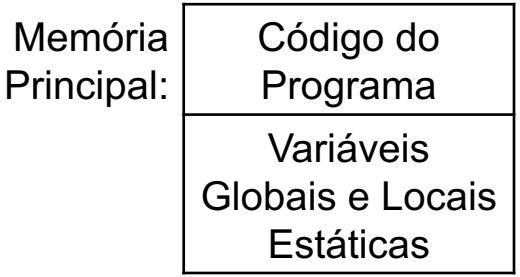
Alocação Dinâmica de Memória

- Alocação dinâmica é **usada quando não se sabe de antemão quanto espaço de memória será necessário**
  - Exemplo, quando não é possível determinar o tamanho do vetor de antemão

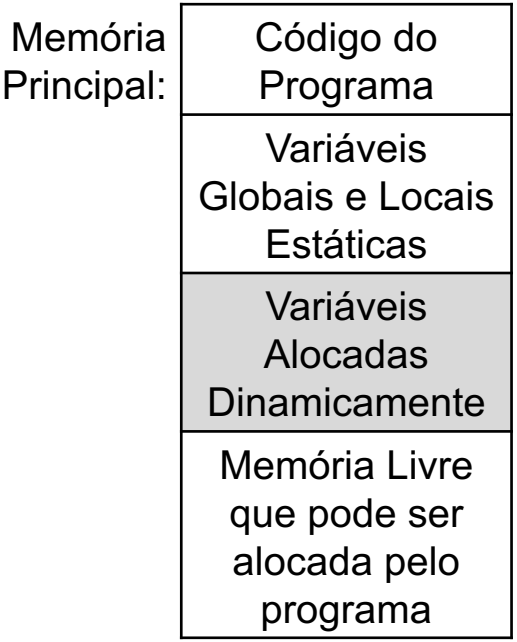
- Alocação dinâmica
  - Espaço de memória é requisitado em tempo de execução
  - Espaço permanece alocado até que seja explicitamente liberado
    - `Depois de liberado, espaço pode ser disponibilizado para outros usos`
    - `Espaço alocado e não liberado explicitamente é liberado ao final da execução do programa`

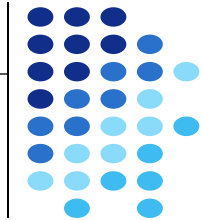
Alocação Estática X Alocação Dinâmica

Alocação Estática



Alocação Dinâmica





A Biblioteca `STDLIB.H`

- Essa biblioteca **`contém funções e constantes para a alocação dinâmica de memória`**

A Biblioteca `STDLIB.H`

- **`sizeof(tipo)`**
  - Retorna o número de bytes ocupado por um **`tipo`**

- **`malloc(tamanho)`**
  - Recebe como parâmetro o **`tamanho`** do espaço de memória (em bytes) que se deseja alocar
  - Retorna um ponteiro genérico para o endereço inicial da área de memória alocada, **se houver espaço livre**
    - Ponteiro genérico é representado por **`void*`**
    - Ponteiro é convertido automaticamente para o tipo apropriado
    - Ponteiro pode ser convertido explicitamente
  - Retorna um endereço nulo (**`NULL`**) se não houver espaço livre

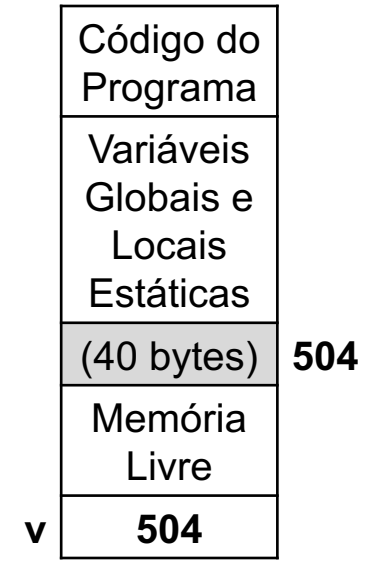
Exemplo

- Alocação dinâmica de um vetor de inteiros com 10 elementos
  - **malloc** retorna o endereço do espaço de memória alocado
  - Ponteiro de inteiro (v) recebe endereço inicial do espaço de memória alocado

```
int *v;
v = (int*) malloc(10 * sizeof(int));
```

  - no programa, **v** pode ser tratado como um vetor alocado estaticamente

Memória Principal



Tratamento de Erro

```
int *v;
v = (int*) malloc(10 * sizeof(int));
if(v == NULL){
      printf("Memoria insuficiente.\n");
      exit(1);/* aborta o programa e
                  retorna 1 para o sistema
                        operacional */
}
//...
```

Liberação de Memória

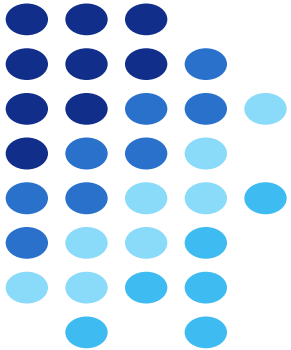
- **`free(ponteiro)`**
  - Libera a memória apontada por **`ponteiro`**
  - A posição de memória apontada por ponteiro **`deve ter sido alocada dinamicamente`**

```c
/* Cálculo da média de n números reais */
int main () {
  int i,n;
  float *v;
  float med;
  scanf("%d", &n); /* leitura do número n de valores */
  v = (float*) malloc(n * sizeof(float));/*alocação dinâmica*/
  if (v == NULL) {
    printf("Memoria insuficiente.\n");
    return 1;
  }
  for (i = 0; i < n; i++)
    scanf("%f", &v[i]);
  med = media(n,v);/*chama a função de cálculo de média */
  printf("Media = %.2f \n", med);
  free(v); //libera memória */
  return 0;
}
```

UNIVERSIDADE FEDERAL DE SERGIPE

dsi departamento de sistemas de informação

Ponteiros e Tipos Estruturados



Tipo Estrutura

- Tipo de dado com elementos (campos) **compostos por tipos mais simples**
- Campos são acessados através do **operador de acesso "ponto" (.)**

```
struct ponto2d{ /* declara ponto2d do tipo struct */
  float x;
  float y;
};
//...
struct ponto2d p; /* declara p como variável do tipo
                     struct ponto2d */
//...
p.x = 10.0; /* acessa os elementos de ponto2d */
p.y = 5.0;
```

Exemplo Completo

```c
/* Captura e imprime as coordenadas de um ponto qualquer */
#include <stdio.h>

struct ponto2d {
  float x;
  float y;
};

int main (void) {
  struct ponto2d p;
  printf("Digite as coordenadas do ponto(x y): ");
  scanf("%f %f", &p.x, &p.y);
  printf("O ponto fornecido foi: (%.2f,%.2f)\n", p.x, p.y);
  return 0;
}
```

Ponteiros para Estruturas

- Acesso ao valor de um campo **`x`** de uma variável estrutura **`p`**: **`p.x`**
- Acesso ao valor de um campo **`x`** de uma variável ponteiro **`pp`**: **`pp->x`** ou **`(*pp).x`**
- Acesso ao endereço do campo **`x`** de uma variável ponteiro **`pp`**: **`&pp->x`**

```
struct ponto2d *pp;
//...
(*pp).x = 10.0; /* formas equivalentes de acessar o
                        valor de um campo x */
pp->x = 10.0;
```

Alocação Dinâmica de Estruturas

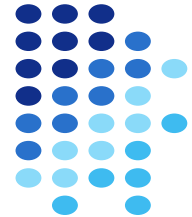
- Tamanho do espaço de memória alocado dinamicamente é dado pelo operador **`sizeof`** aplicado sobre o tipo da estrutura
- A função **`malloc`** retorna o endereço do espaço alocado, que é então convertido para o tipo ponteiro da estrutura

```
struct ponto2d *pp;
pp = (struct ponto2d*) malloc (sizeof(struct ponto2d));
//...
pp->x = 12.0;
pp->y = 5.5;
```

É mais Simples Definir um Nome para a Estrutura

- **typedef**
  - Permite criar nomes de tipos
  - Útil para abreviar nomes de tipos e para tratar tipos complexos

```
struct ponto2d {
  float x;
  float y;
};
typedef struct ponto2d TPonto2d; /* TPonto2d representa
                                    a estrutura ponto2d */
typedef struct ponto2d *TPPonto2d; /* TPPonto2d representa o tipo de
                                      ponteiro para a estrutura pondo2d */
```

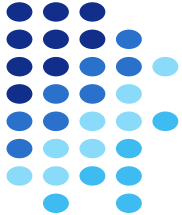


Comando Equivalentes

```
struct ponto2d {
  float x;
  float y;
};
typedef struct ponto2d TPonto2d;
```

```
typedef struct ponto2d {
  float x;
  float y;
}TPonto2d;
```

```c
/* Exemplo com estruturas aninhadas */
#include <stdio.h>
#include <math.h>
typedef struct ponto2d { /* ponto2d representa uma
                                    estrutura com 2 campos float */
  float x;
  float y;
} TPonto2d;
typedef struct circulo {
  TPonto2d p; /* centro do círculo */
  float r; /* raio do círculo */
} TCirculo;
/*  Função para a calcular distância entre 2 pontos:
    entrada: ponteiros para os pontos
    saída: distância correspondente
*/
float distancia (TPonto2d* p, TPonto2d* q) {
  float d = sqrt((q->x - p->x)*(q->x - p->x) + (q->y - p->y)*(q->y - p->y));
  return d;
}
/* ... */
```

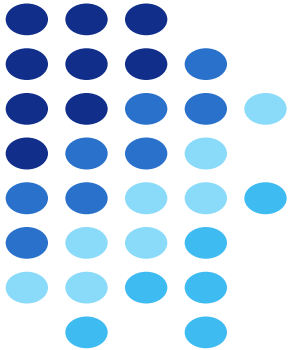


```c
/* Função para determinar se um ponto está ou não dentro de um
   círculo: entrada: ponteiros para um círculo e para um ponto
   saída: 1 = ponto dentro do círculo
   0 = ponto fora do círculo
*/
int interior (TCirculo* c, TPonto2d* p) {
  float d = distancia(&c->p, p);
  return (d < c->r);
}
int main (void) {
  TCirculo c;
  TPonto2d p;
  printf("Digite as coordenadas do centro e o raio do circulo:\n");
  scanf("%f %f %f", &c.p.x, &c.p.y, &c.r);
  printf("Digite as coordenadas do ponto:\n");
  scanf("%f %f", &p.x, &p.y);
  printf("Pertence ao interior = %d\n", interior(&c,&p));
  return 0;
}
```

&c->p: ponteiro para o centro de **c**

p: ponteiro para o ponto

Exemplo Prático: Lista Simplesmente Encadeada



Definição da Lista

- Vamos usar um exemplo onde a lista tem um campo inteiro **`info`**, e um ponteiro **`prox`** para o próximo nó da lista

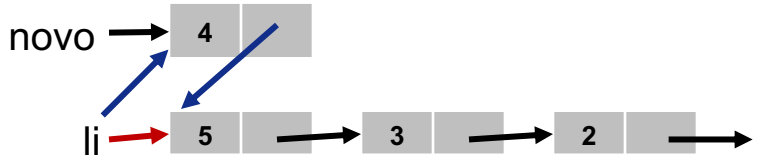
```
struct lista{
        int info;
        struct lista* prox;
};

typedef struct lista TLista;
```

Funções de Manipulação da Lista

- **`cria_lista`**
  - Cria uma lista vazia, representada pelo ponteiro **`NULL`**

```
TLista* cria_lista (void){
        return NULL;
}
```

Funções de Manipulação da Lista



- **insere_inicio**
  - Insere um elemento no início da lista
  - Retorna a lista atualizada

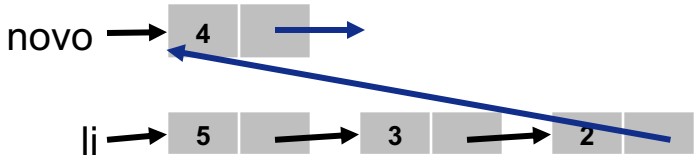
```
TLista* insere_inicio (TLista* li, int i) {
  TLista* novo = (TLista*) malloc(sizeof(TLista));
  novo->info = i;
  novo->prox = li;
  return novo;
}
```

Funções de Manipulação da Lista

- **imprime**
  - Imprime os elementos da lista

```
void imprime (TLista* li) {
      TLista* p;
      for (p = li; p != NULL; p = p->prox)
            printf("info = %d\n", p->info);
}
```

Funções de Manipulação da Lista

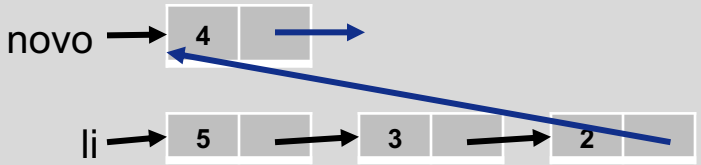


- **`insere_fim`**
  - Insere um elemento no fim da lista
  - Retorna ponteiro para a lista atualizada

Implemente a função insere_fim:

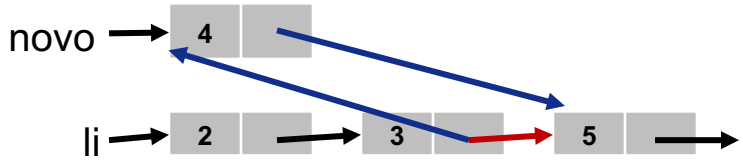
```
TLista* insere_fim (TLista* li, int i){
        //...
 }
```

```
TLista* insere_fim (TLista* li, int i) {
  TLista* novo = (TLista*) malloc(sizeof(TLista));
  novo->info = i;
  novo->prox = NULL;
  TLista* p = li;
  TLista* q = li;
  while (p != NULL) { /* encontra o ultimo elemento */
    q = p;
    p = p->prox;
  }
  if (q != NULL) /* se a lista original não estiver vazia */
    q->prox = novo;
  else
    li = novo;
  return li;
}
```



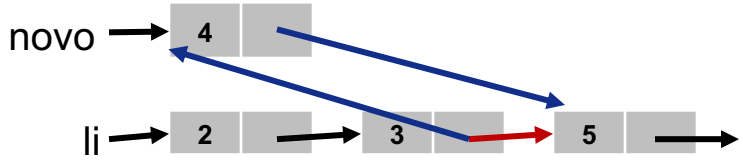
Funções de Manipulação da Lista

- **`insere_ordenado`**
  - Insere um elemento na lista de forma que ela esteja sempre ordenada
  - Retorna ponteiro para a lista atualizada



Funções de Manipulação da Lista

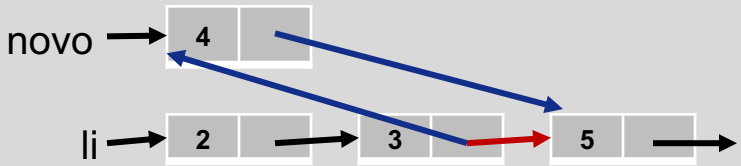
- **`insere_ordenado`**
  - Insere um elemento na lista de forma que ela esteja sempre ordenada
  - Retorna ponteiro para a lista atualizada



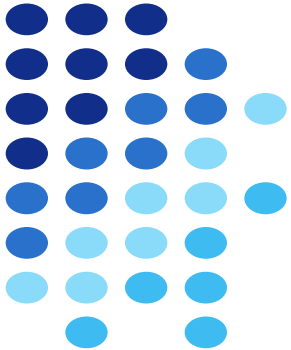
Implemente a função insere_ordenado:

```
TLista* insere_ordenado (TLista* li, int i) {
            //...
 }
```

```c
TLista* insere_ordenado (TLista* li, int i) {
  TLista* novo;
  TLista* ant = NULL; /* ponteiro para elemento anterior */
  TLista* p = li; /* ponteiro para percorrer a lista */
  /* procura posição de inserção */
  while (p != NULL && p->info < i) {
    ant = p;
    p = p->prox;
  }
  /* cria novo elemento */
  novo = (TLista*) malloc(sizeof(TLista));
  novo->info = i;
  /* encadeia elemento */
  if (ant == NULL) { /* insere elemento no início */
    novo->prox = li;
    li = novo;
  } else { /* insere elemento no meio da lista */
    novo->prox = ant->prox;
    ant->prox = novo;
  }
  return li;
}
```



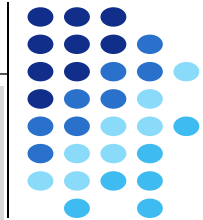
Funções Recursivas



Funções Recursivas

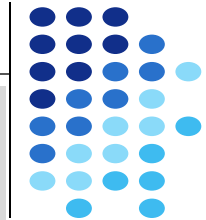
- Quando uma **`função chama a si mesma`**, dizemos que a função é recursiva

- **Pontos importantes**:
  - Forma fácil de dividir para conquistar – trata-se um item e chama-se recursivamente para os demais
  - É preciso ter um critério de parada para evitar loop infinito de chamadas



Implemente a função insere_fim_recursivo:

```
TLista* insere_fim_recursivo (TLista* li, int i) {
    //...
}
```



Implemente a função insere_fim_recursivo:

```
TLista* insere_fim_recursivo (TLista* li, int i) {
   //...
}
```

```
TLista* insere_fim_recursivo (TLista* li, int i) {
  if (li == NULL) { /* parada da recursão: quando encontra-se o
                       último elemento */
    TLista* novo = (TLista*) malloc(sizeof(TLista));
    novo->info = i;
    novo->prox = NULL;
    li = novo;
  }else{
    li->prox = insere_fim_recursivo(li->prox, i);
  }
  return li;
}
```

Referências

- Material baseado nos slides de **Vanessa Braganholo**, Disciplina de Estruturas de Dados e Seus Algoritmos. Instituto de Computação. Universidade Federal Fluminense (UFF), Niterói, Brasil.
- Waldemar Celes, Renato Cerqueira, José Lucas Rangel. **Introdução a Estruturas de Dados**. Editora Campus, 2004.

Desafio 2

Implementar as funções:

```
//excluir um item da lista
TLista* exclui(TLista* li, int i) {
    //TODO: completar aqui
}


//alterar um item da lista
TLista* altera(TLista* li, int vantigo, int vnovo) {
    //TODO: completar aqui
}


//inserir recursivamente de forma ordenada
TLista* insere_ordenado_recursivo(TLista *li, TLista* ant, int i) {
    //TODO: completar aqui
}
```