AULA 01

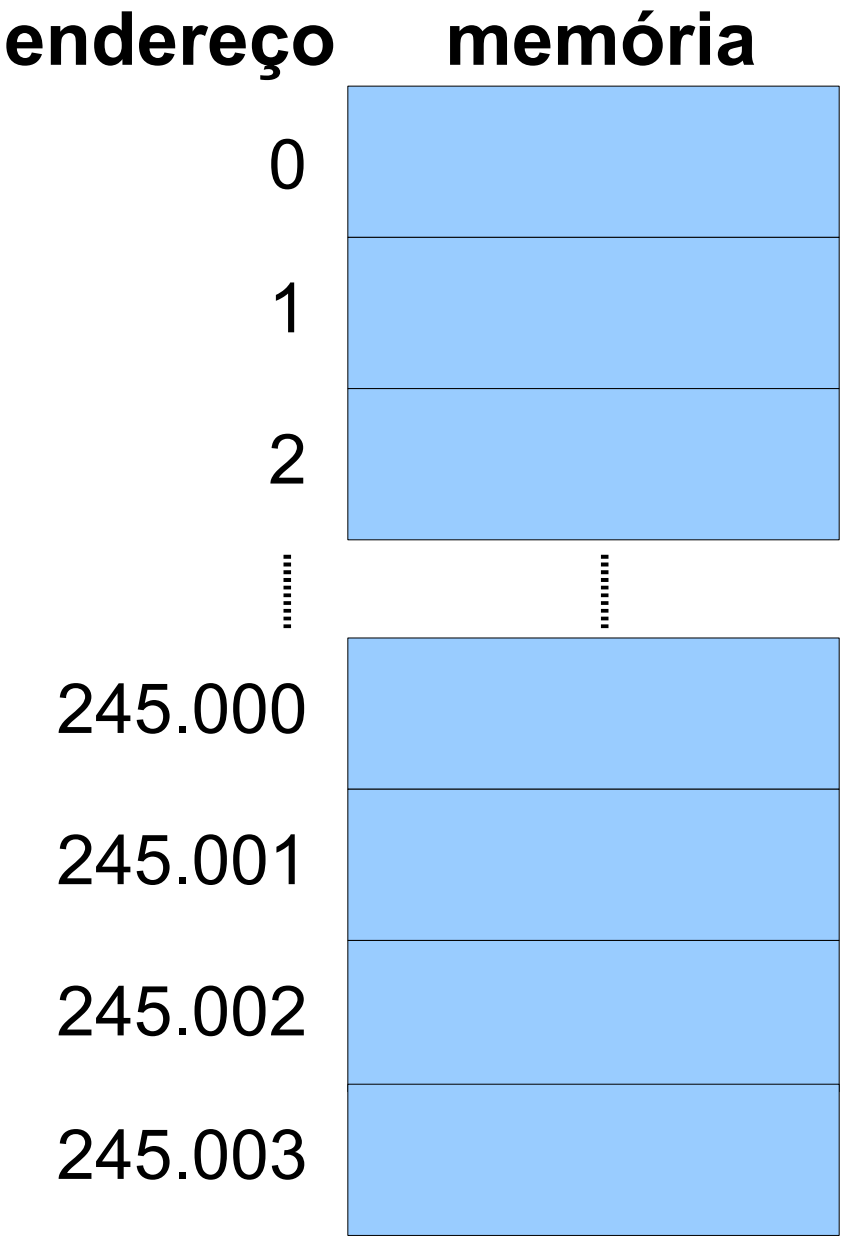
Revisão: Ponteiros

Prof. Tiago A. Almeida
talmeida@ufscar.br

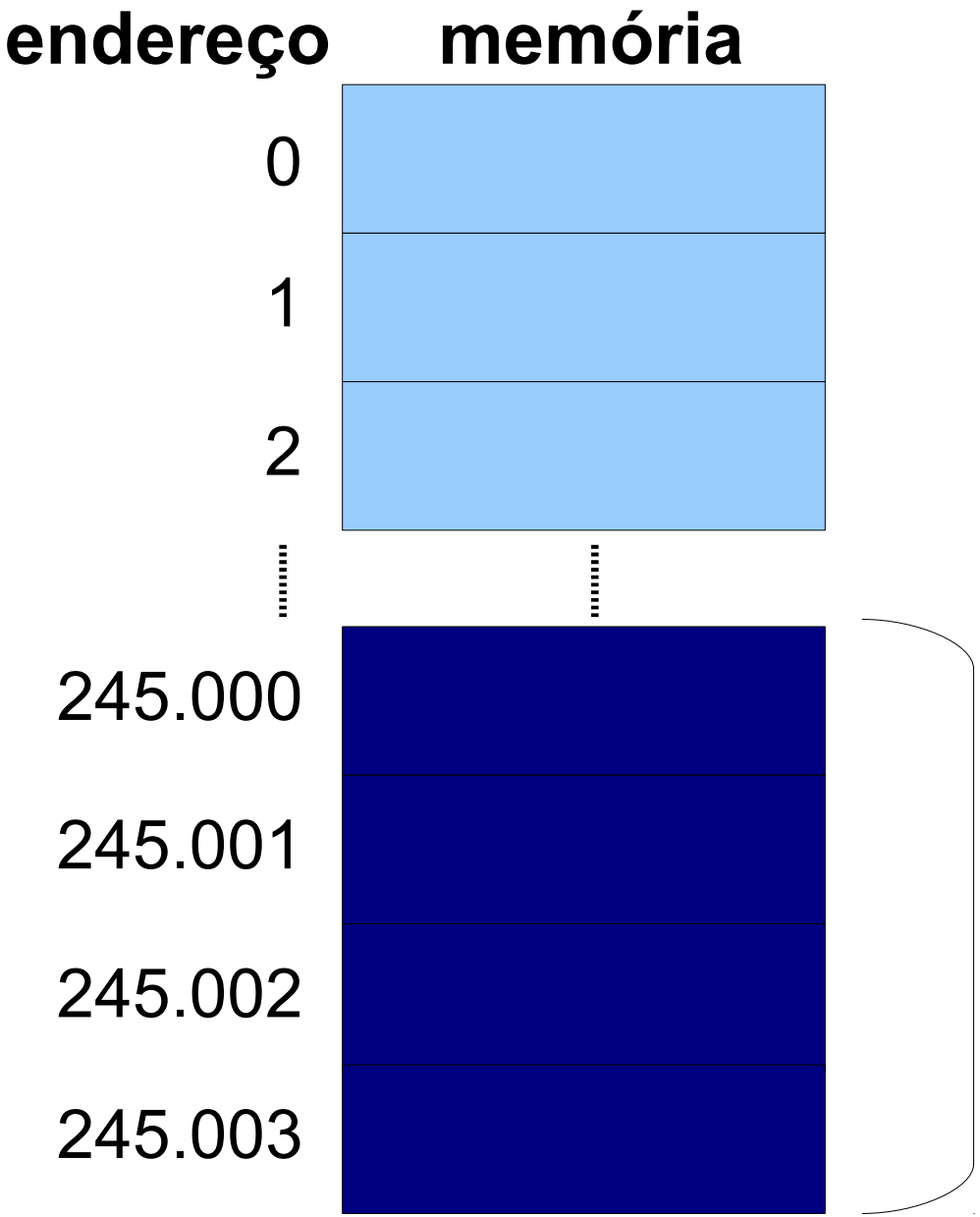
Endereços e Ponteiros

- A memória de qualquer computador é uma sequência de bytes
  - Cada byte armazena um de 256 possíveis valores. Os bytes são numerados sequencialmente
  - O número de um byte é o seu **endereço** (*address*)
- Cada objeto na memória do computador ocupa um certo número de bytes consecutivos. Ex:
  - um `char` ocupa 1 byte, um `int` ocupa 4 bytes e um `double` ocupa 8 bytes
- Cada objeto na memória do computador tem um endereço. Na maioria dos computadores, o endereço de um objeto é o endereço do seu primeiro byte

Endereços de memória



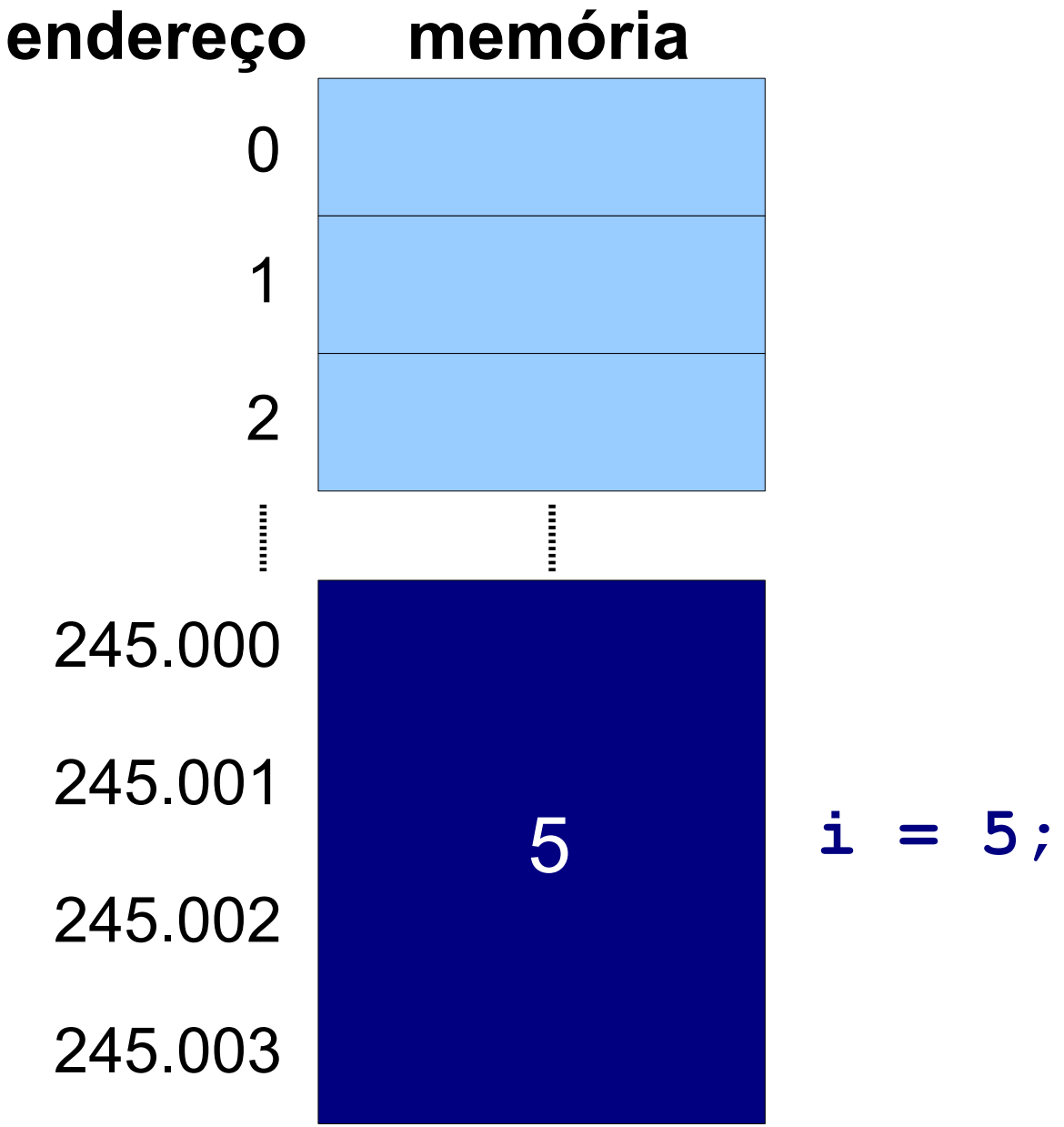
Endereços de memória




```
int i;
```

4 bytes reservados para `i` que inicia no endereço 245.000

Endereços de memória



Endereços e Ponteiros

- **Ponteiro é que uma variável que guarda um endereço de memória**
  - Com ela é possível acessar uma porção definida da memória
- Assim como variáveis comuns, ponteiros devem ser declarados. A única diferença é que ponteiros são identificados por um *****

```
int *p;
```

- `p` é o nome do ponteiro e "`int *`" informa o compilador que `p` guardará um endereço de memória onde será armazenado um inteiro

Endereços e Ponteiros

- Variáveis ponteiros podem ser declaradas juntas com outras variáveis:

```
int i, j, a[10], b[20], *p, *q;
```

- A linguagem C exige que cada variável ponteiro aponte apenas para objetos do mesmo tipo (tipo referenciado):

```
int *p; /*aponta apenas para inteiros*/

double *q; /*aponta apenas para double*/

char *r; /*aponta apenas para caracteres*/
```

Operadores: Endereço e Indireto

- A linguagem C oferece dois operadores designados especificamente para serem usados com ponteiros:
  - Operador endereço → **&**
    - Se `x` é variável, então `&x` é o endereço de memória de `x`
  - Operador indireto → *****
    - Se `p` é um ponteiro, então `*p` representa o objeto apontado por `p`

Operador de Endereço

- Declarar uma variável ponteiro reserva espaço na memória para o apontador, mas não faz referência a nenhum objeto:

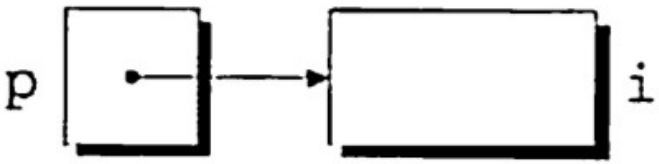
```
int *p; /* não aponta para nada */
```

- Para usar `p`, primeiro é preciso inicializá-la. Uma forma de fazer isso é associar o endereço de uma variável

```
int i, *p;

...

p = &i;
```



Operadores de Endereço

- É possível inicializar um ponteiro no momento da sua declaração:

```
int i;

int *p = &i;
```

- Podemos até mesmo combinar a declaração de `i` com a de `p`, com `i` declarada primeiro:

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

- Dica: é possível inicializar um ponteiro vazio fazendo

```
int *p;

p = NULL; /*NULL é definida em stdlib.h */
```

Operador Indireto

- Uma vez que o ponteiro aponta para um objeto, é possível usar o operador `*` para acessar o seu conteúdo

```
printf("%d\n", *p); /*mostra o valor de i*/
```

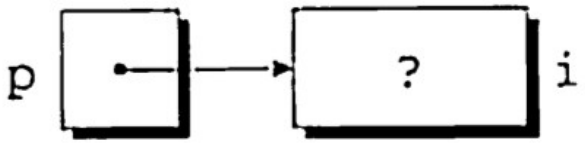
- O operador `*` é o inverso de `&`

```
j = *&i; /*é o mesmo que fazer j = i*/
```

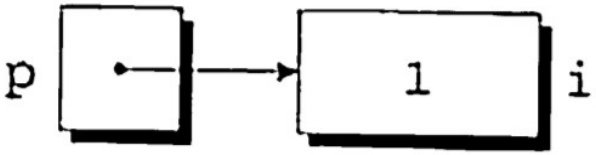
- Se `p` aponta para `i`, `*p` é dito ser um *alias* de `i`
  - `*p` tem o mesmo valor de `i`
  - Alterar `*p` também altera o valor de `i`

Operador Indireto

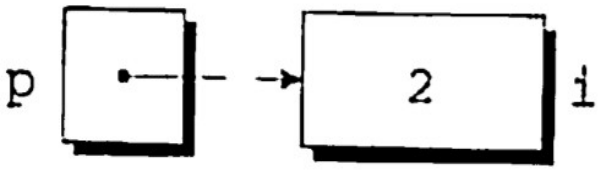
```
p = &i;
```




```
i = 1;
```




```
printf("%d\n",i); /* imprime 1 */
printf("%d\n",*p); /* imprime 1 */
*p = 2;
```




```
printf("%d\n",i); /* imprime 2 */
printf("%d\n",*p); /* imprime 2 */
```

Operador Indireto

- **Nunca** aplique um operador indireto em um apontador não inicializado!

```
int *p;

printf("%d\n",*p); /*** ERRADO ***/
```

  - Pode causar um comportamento indefinido

- Atribuir um valor para `*p` é perigoso! Se `p` apontar para um endereço de memória válido, isso irá modificar o dado armazenado no endereço

```
int *p;

*p = 1; /*** ERRADO ***/
```

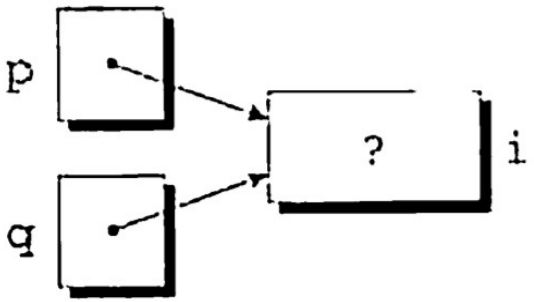
  - Se `p` apontar para um endereço de memória usado pelo SO, poderá travar o sistema

Atribuição de ponteiros

- C permite o operador de atribuição para copiar ponteiros

```
int i, j, *p, *q;


p = &i; /* endereço de i é copiado para p */
q = p; /* conteúdo de p é copiado para q */
```

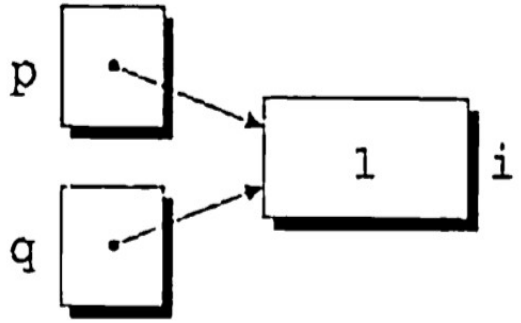


- Tanto `p` quanto `q` agora apontam para `i`

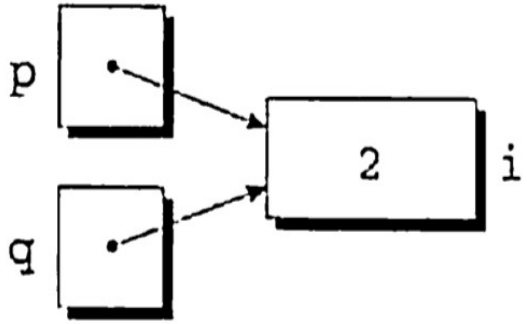
Atribuição de ponteiros

- Agora, podemos mudar o valor de i atribuindo novos valores para `*p` e `*q`

```
*p = 1;
```




```
*q = 2;
```



- Qualquer número de ponteiros pode apontar para um mesmo objeto

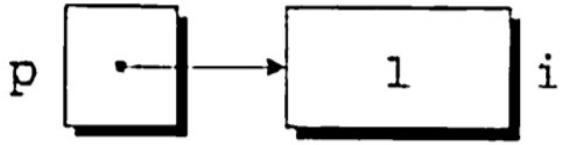
Atribuição de ponteiros

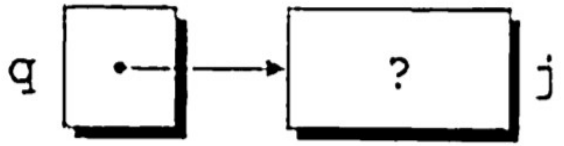
- **Cuidado!!! Não confunda**

  `q = p;` com `*q = *p;`

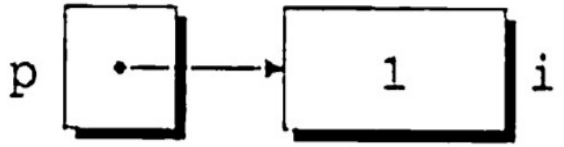
- O primeiro é uma atribuição de ponteiro; **o segundo não**

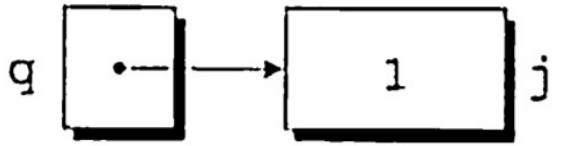
```
p = &i;
q = &j;
i = 1;
```






```
*q = *p;
```





- `*q = *p` copia o valor da variável apontada por `p` (valor de `i`) dentro do objeto apontado por `q` (variável `j`)

Funções: valores como argumentos

- Quando passamos argumentos a uma função, os valores fornecidos são copiados para os parâmetros formais da função. Este processo é idêntico a uma atribuição
  - Alterações nos parâmetros dentro da função não alteram os valores que foram passados
  - Isso limita muito a linguagem de programação!!!
  - Ex.:
    - função para trocar o valor entre duas variáveis;
    - função para decompor um número em parte inteira e fracionária
    - ...

```
void nao_troca(int x, int y){
   int aux = x;
   x = y;
   y = aux;
}
```

Funções: valores como argumentos

- Existe uma forma de alterarmos a variável passada como argumento, ao invés de usarmos apenas o seu valor
  - Passarmos como argumento o endereço da variável, e não o seu valor
- Para indicarmos que será passado o endereço do argumento, é preciso indicar na função

```
tipo nome (tipo *par1, tipo *par2, ..., tipo *parN)
{
        // comandos
}
```

Funções: valores como argumentos

- Um endereço de variável passado como parâmetro não é muito útil. Para acessarmos o valor de uma variável apontada por um endereço, usamos o operador *:
- Ao precedermos uma variável que contém um endereço com este operador, obtemos o equivalente a variável armazenada no endereço em questão:

```
void troca(int *end_x, int *end_y)
{
  int aux;
  aux = *end_x;
  *end_x = *end_y;
  *end_y = aux;
}
```

troca.c

Funções: exemplo 1

- Função para decompor um número em parte inteira e parte fracionária (ver `decomposicao.c`)

```
void decomposicao(double x, long *p_int, double *p_frac)
{
   *p_int = (long) x;
   *p_frac = x - *p_int;
}
```

- Os protótipos da função poderiam ser:

```
void decomposicao(double x, long *p_int, double *p_frac);
```

ou

```
void decomposicao(double, long *, double *);
```

Argumento por referência

- Não é novidade para nós: `scanf`

```
int i;
…
scanf("%d", &i); /* passamos o endereço de i */
```

- E se o argumento do `scanf` for um ponteiro?

```
int i, *p;
…
p = &i;
scanf("%d", p); /* p é igual ao endereço de i */
```

- Note que, nesse caso:

```
scanf("%d", &p); /*** ERRADO ***/
```

Funções: exemplo 2

- Escreva uma função para encontrar o valor máximo e o mínimo de um vetor com 10 posições (ver `encontrarMaxMin.c`)

- Para proteger ponteiros que não podem ser alterados por funções, podemos utilizar a palavra-chave `const`

```
void f(const int *p)
{
   *p = 0; /*** ERRADO ***/
}
```

- O compilador emitirá uma mensagem de erro!

Funções que retornam ponteiros

- É comum na linguagem C querermos retornar ponteiros

```
int *max (int *a, int *b)
{
   if (*a > *b)
      return a;
   else
      return b;
}
```

- Na chamada da função `max`, passamos ponteiros de duas variáveis inteiras e armazenamos o resultado em um ponteiro

```
int *p, i, j;
…
p = max(&i, &j);
```

- Durante a chamada de `max`, `*a` é um alias para `i` e `*b` é um *alias* para `j`, `max` retorna o endereço de `i` ou `j`. Após a chamada `p` aponta para a variável do endereço retornado

AULA 01

Revisão: Ponteiros - Vetores & Registros

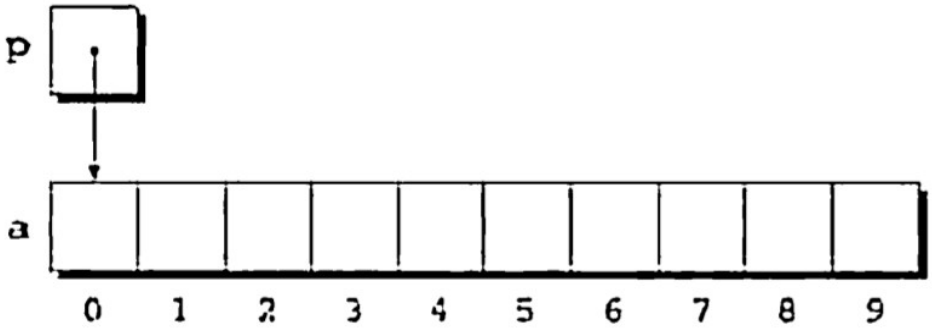
Prof. Tiago A. Almeida
talmeida@ufscar.br

Aritmética de ponteiros

- Um ponteiro pode apontar para um vetor de elementos:

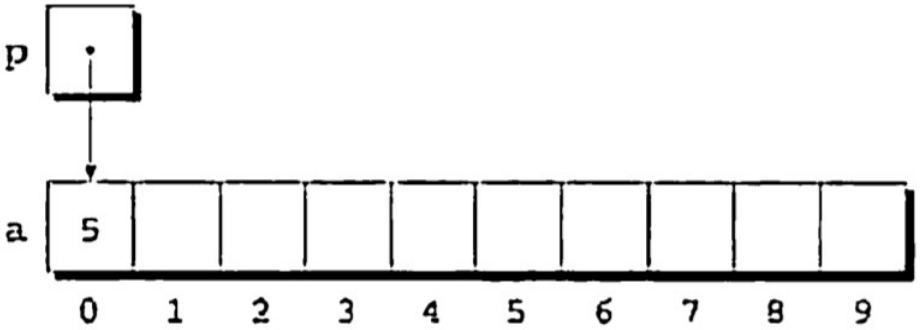
```
int a[10], *p;

p = &a[0];
```



- Assim, podemos acessar `a[0]` através de p; por exemplo, podemos armazenar o valor 5 em `a[0]` com:

```
*p = 5;
```



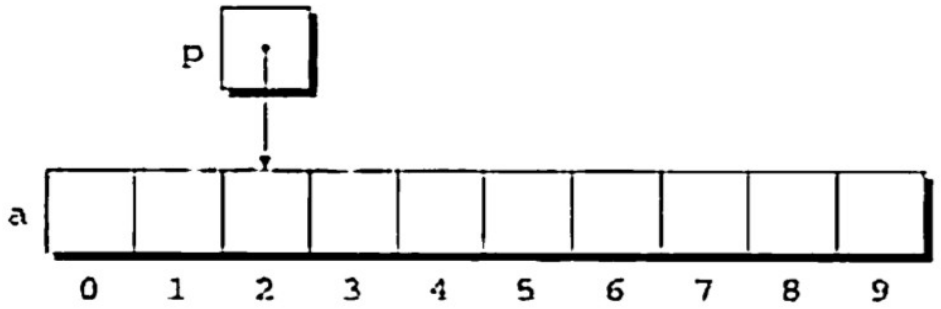
Aritmética de ponteiros

- Fazer um ponteiro apontar para um elemento de um vetor não é particularmente pancadão!
- Entretanto, através de aritmética de ponteiros é possível acessar outros elementos do vetor.
- A linguagem C suporta **três formas** de aritmética de ponteiros:
  - Adicionar um inteiro a um ponteiro;
  - Subtrair um inteiro de um ponteiro;
  - Subtrair um ponteiro de outro ponteiro.
- Exemplos:

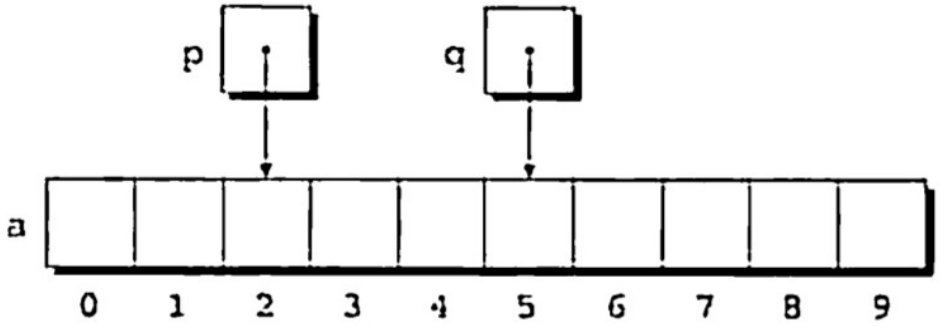
```
int a[10], *p, *q, i;
```

Adicionar inteiro em ponteiro

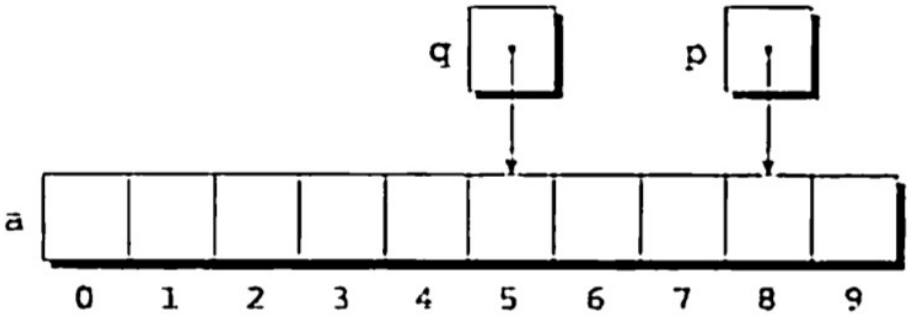
- `p = &a[2];`



- `q = p + 3;`

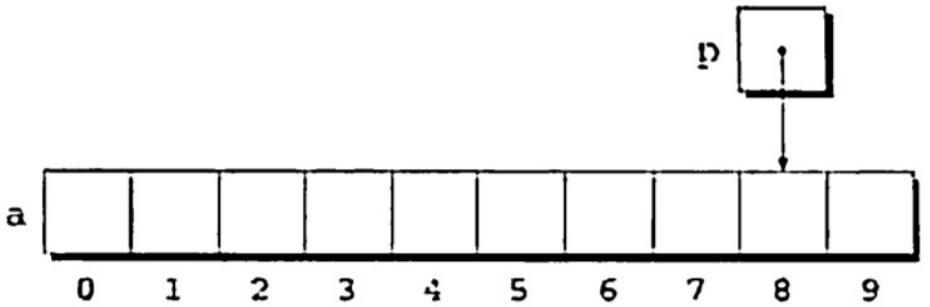


- `p = p + 6;`

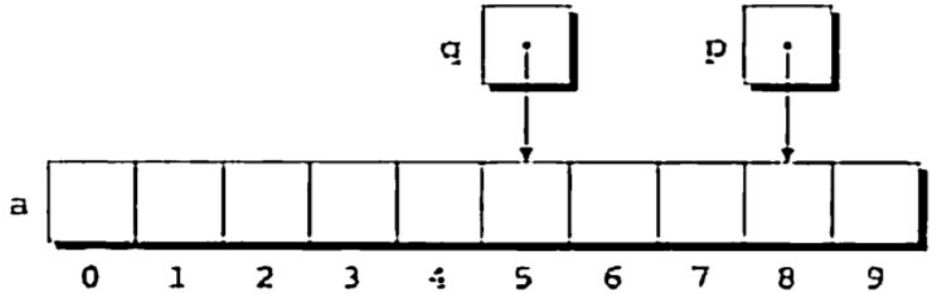


Subtrair inteiro de ponteiro

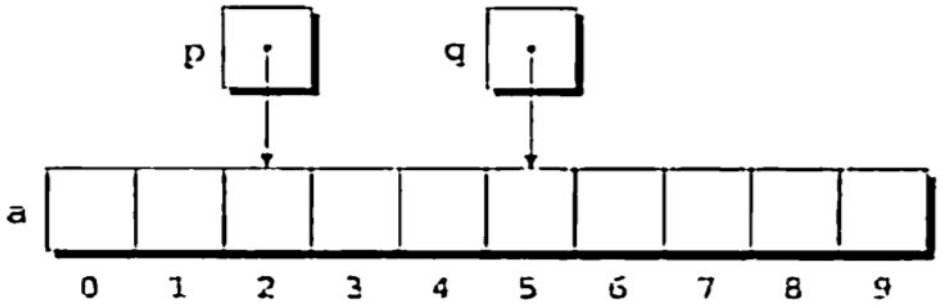
- ✔ `p = &a[8];`



- ✔ `q = p - 3;`

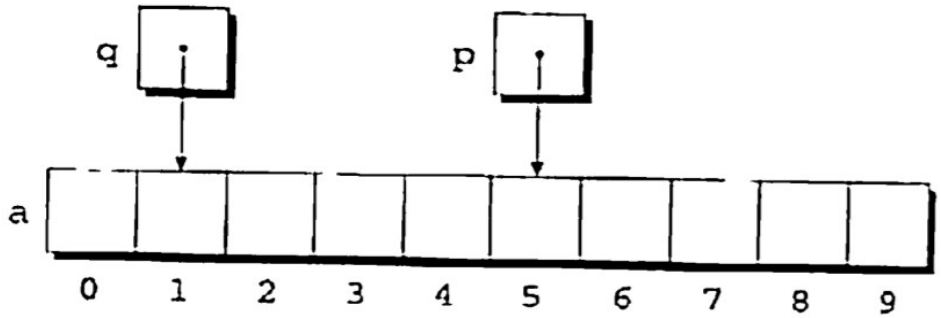


- ✔ `p = p - 6;`



Subtrair ponteiro de ponteiro

```
p = &a[5];
q = &a[1];
```




```
i = p - q; /* i é igual a 4 */
i = q - p; /* i é igual a -4 */
```

- Podemos comparar ponteiros usando operadores relacionais (`<`, `<=`, `>`, `>=`) e operadores de igualdade (`==` e `!=`)
  - `p = &a[5], q = &a[1];`
  - O valor de `p <= q` é 0 e o valor de `p >= q` é 1

Processamento de vetores

- Programa para somar elementos de um vetor:

```
#define N 10
…

int a[N], sum, *p;
…

sum = 0;

for (p = &a[0]; p < &a[N]; p++)
   sum = sum + *p;
```

- Obs: apesar do vetor `a[]` ir de `0` até `N-1`, é seguro utilizar `p<&a[N]`.
- **Vantagem:** aumenta velocidade de execução.

Combinando operadores * e ++

- Programadores de C frequentemente combinam operadores `*` (indireção) com `++`. Exemplos:

  `a[i++] = j` é o mesmo que fazer `*p++ = j` que é o mesmo que `*(p++) = j`

Expressão	Significado
*p++ ou *(p++)	Valor da expressão é *p antes do incremento; incrementa p depois
(*p)++	Valor da expressão é *p antes do incremento; incrementa *p depois
*++p ou *(++p)	Incrementa p primeiro; valor da expressão é *p após incremento
++*p ou ++(*p)	Incrementa *p primeiro; valor da expressão é *p após incremento

- Ver programa `AritmeticaPonteiros.c`

Combinando operadores * e ++

- Programa para somar elementos de um vetor:

```
#define N 10
…

int a[N], sum, *p;
…

p = &a[0];

while (p < &a[N])
   sum = sum + *p++;
```

Nome do vetor como ponteiro

- O nome de um vetor pode ser usado como um ponteiro para o primeiro elemento do vetor.

```
int a[10];

*a = 7;  /* armazena 7 em a[0] */

*(a+1) = 12;  /* armazena 12 em a[1] */
```

- Em geral, `a + i` é o mesmo que `&a[i]` e `*(a+i)` é equivalente a `a[i]`

Ponteiros para processar vetores

- Programa para somar elementos de um vetor:

```
#define N 10
…

int a[N], sum, *p;
…

sum = 0;

for (p = a; p < a + N; p++)
   sum = sum + *p;
```

- Ver programa `InverterSerie.c`

Vetores como argumentos

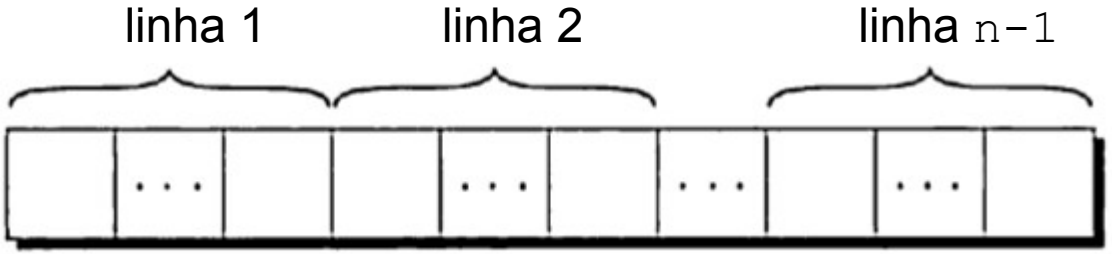
- Quando uma variável escalar qualquer é passada como parâmetro de uma função, ela é copiada e o seu conteúdo fica protegido.
- Porém, quando um vetor é passado para uma função, o nome do vetor é **sempre** tratado como um ponteiro.
- Consequentemente, os valores podem ser alterados pela função. Para proteger o conteúdo do vetor deve ser utilizado a palavra `const`. Exemplo:

```
InverterSerie(const int a[], int n)
```

- Ver programas `VetorParametro.c` e `VetorxVariavel.c`

Ponteiros e Matrizes

- As propriedades e operações de ponteiros **são as mesmas** de um vetor unidimensional, pois em linguagem C matrizes são armazenadas na memória como vetores.



- Exemplo: inicializar com zero os elementos de uma matriz bidimensional.

```
int a[N_LIN][N_COL], *p;
…
for (p=&a[0][0]; p<=&a[N_LIN-1][N_COL-1]; p++)
   *p = 0;
```

- `p` aponta para `a[0][0]`, depois para `a[0][1]` ...

Processando as linhas da matriz

- Para processar somente a linha `i` de uma matriz `a`:

  `p = &a[i][0]` é equivalente à `p = a[i]`, em ambos os casos `p` aponta para o primeiro elemento da linha `i`

- Para zerar os elementos da linha `i`, basta fazer

```
int a[N_LIN][N_COL], *p, i;
…
for (p = a[i]; p < a[i] + N_COL; p++)
   *p = 0;
```

Exercício

- Dado um vetor de notas e o número de notas que ele contém, faça uma função que retorne a média, a maior e a menor nota. Use os recursos vistos nessa aula.

Ponteiros e Registros

- Como um registro é uma variável qualquer que ocupa espaço em memória, obviamente podemos criar um ponteiro que aponta para ela.

```
struct ponto {
   double x; double y;
};

typedef struct ponto Ponto;
…
Ponto *ap_p;
```

- Nesse caso, utilizamos o operador `->` para acessar cada componente da estrutura.

```
ap_p->x = 4.0;
ap_p->y = 5.0;
```

- Também poderíamos fazer o acesso por `(*ap_p).x`

Ponteiros e Registros

- Operações de leitura e escrita:

```
scanf("%lf", &ap_p->x);
scanf("%lf", &ap_p->y);

…

printf("p.x = %.3lf\n", ap_p->x);
printf("p.y = %.3lf\n", ap_p->y);
```

- Ver `PonteirosRegistros.c`

Ponteiros e Registros

- Seguem a mesma regra de variáveis simples;

```
void troca(Ponto *p1, Ponto *p2) {
   Ponto aux;
   aux = *p1;
   *p1 = *p2;
   *p2 = aux;
}
```

- Ver `TrocaStruct.c`

AULA 01

Revisão: Alocação Dinâmica

Prof. Tiago A. Almeida
talmeida@ufscar.br

Alocação Dinâmica

- Até agora tínhamos que declarar todas as variáveis que íamos usar no programa, para que o computador pudesse alocar memória para elas.
- Mas existe um modo de definir uma variável enquanto o programa roda?

Alocação Dinâmica

- Até agora tínhamos que declarar todas as variáveis que íamos usar no programa, para que o computador pudesse alocar memória para elas.
- Mas existe um modo de definir uma variável enquanto o programa roda? **A resposta é: não exatamente.**

Alocação Dinâmica

- Até agora tínhamos que declarar todas as variáveis que íamos usar no programa, para que o computador pudesse alocar memória para elas.
- Mas existe um modo de definir uma variável enquanto o programa roda? **A resposta é: não exatamente.**
- Não há como declarar a variável enquanto o programa roda. O que dá para fazer é alocar memória para uma variável enquanto o programa roda.
- E qual a vantagem disso?

Alocação Dinâmica

- Até agora tínhamos que declarar todas as variáveis que íamos usar no programa, para que o computador pudesse alocar memória para elas.
- Mas existe um modo de definir uma variável enquanto o programa roda? **A resposta é: não exatamente.**
- Não há como declarar a variável enquanto o programa roda. O que dá para fazer é alocar memória para uma variável enquanto o programa roda.
- E qual a vantagem disso? **Poupa memória** (em casos específicos).
- Mas, isto tem um preço (como tudo na computação) - gasta mais processamento.

Alocação Dinâmica : Uso

- Apesar de poder ser usada com todos os tipos de dados, alocação dinâmica é mais frequentemente empregada para manipulação de *strings*, vetores e registros.
- Alocação dinâmica de registros é extremamente utilizada, uma vez que, é possível conectar os registros em forma de listas, árvores ou outra estrutura de dados.

Alocação Dinâmica : Funções

- Em linguagem C há três funções declaradas na `<stdlib.h>` que podem ser empregadas para alocar memória dinamicamente.

Alocação Dinâmica : Funções

- Em linguagem C há três funções declaradas na `<stdlib.h>` que podem ser empregadas para alocar memória dinamicamente. Elas são:

`malloc` – Aloca um bloco de memória mas não o inicializa (função mais empregada);

```
void *malloc(size_t size);
```

`calloc` - Aloca um bloco de memória e o inicializa (menos eficiente que `malloc`);

```
void *calloc(size_t nmemb, size_t size);
```

`realloc` – Redimensiona um bloco de memória previamente alocado.

```
void *realloc(void *ptr, size_t size);
```

Alocação Dinâmica : malloc

- A função `malloc` aloca um determinado número de bytes na memória, retornando um ponteiro para o primeiro byte alocado, ou `NULL` caso não tenha conseguido alocar.
- A função `free`, por outro lado, libera o espaço alocado.

```
p = malloc(10000);
if (p == NULL) {
   /* tratamento para falha de alocação */
}
free(p);
```

Alocação Dinâmica : exemplo 1

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char *c; /* ponteiro para o espaço alocado */

   c = (char *)malloc(1); /* aloco um único byte
                             na memória */
   if (c == NULL) { /* testa se conseguiu alocar.
                       Equivalente a "if (!c)" */
      printf("Não conseguiu alocar a memória\n");
      exit(1);
   }

   *c = 'd'; /* carrego um valor na região
                de memória alocada */
   printf("%c\n", *c); /* escrevo este valor */
   free(c); /* libero a memória alocada */
   return 0;
}
```

Alocação Dinâmica : malloc

- A função `malloc` aloca um determinado número de bytes na memória, retornando um ponteiro para o primeiro byte alocado, ou `NULL` caso não tenha conseguido alocar.
- A função `free`, por outro lado, libera o espaço alocado.

```
p = malloc(10000);
if (p == NULL) {
   /* tratamento para falha de alocação */
}
free(p);
```

- É possível combinar a alocação e o teste:

```
if ((p = malloc(10000)) == NULL) {
   /* tratamento para falha de alocação */
}
```

Alocação Dinâmica : exemplo 1

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char *c; /* ponteiro para o espaço alocado */

   c = (char *)malloc(1); /* aloco um único byte
                             na memória */
   if (c == NULL) { /* testa se conseguiu alocar.
                       Equivalente a "if (!c)" */
      printf("Não conseguiu alocar a memória\n");
      exit(1);
   }

   *c = 'd'; /* carrego um valor na região
                de memória alocada */
   printf("%c\n", *c); /* escrevo este valor */
   free(c); /* libero a memória alocada */
   return 0;
}
```

Alocação Dinâmica : exemplo 1

- Declaramos um ponteiro para caractere e usamos `malloc` para alocar um byte na memória (tamanho de um caractere).
- Quando `malloc` aloca a memória, ela não faz ideia do que será posto lá: se é `int`, `char`, `float` ou o que for. Então, ela retorna um ponteiro genérico (`void *`). Isso é possível porque ponteiros são endereços de memória e, como tal, possuem sempre o mesmo tamanho, não importando o tipo para o qual eles apontam.
- Contudo, C precisa que o ponteiro tenha um tipo, para poder executar operações de aritmética de ponteiros. Por isso **temos que fazer um `cast` no retorno de `malloc`** para o tipo de ponteiro. No nosso caso, como queríamos um ponteiro para caractere, forçamos a saída de `malloc` a ser `char *`.

Alocação Dinâmica : exemplo 1

- No caso de `c` ser `NULL`, é impressa uma mensagem de erro e encerramos o programa. Se `c` não for `NULL`, então ele contém o endereço na memória onde cabe um caractere. Agora é só agir como faríamos com um ponteiro que apontou para uma variável `char`, guardando um valor na região de memória apontada.
- Antes do encerramento do programa, é necessário liberar a memória alocada, usando a função `free`.

Alocação Dinâmica : exemplo 1

- **Observação**
  - Neste exemplo, o programa fica mais lento e ocupa mais memória que se fizéssemos ele sem alocação.

- **Utilidade (exemplo 2)**
  - Imagine uma situação onde você tem um mínimo de memória e precisa usar uma variável (um `double`, por exemplo) somente no início do programa. Se o seu programa for grande o suficiente, você vai declarar este `double` no início, usar, e segurar a memória alocada até o fim. Um jeito de minimizar este problema seria declarar um ponteiro para `double`, alocá-lo, e depois de usar, liberá-lo.

Alocação Dinâmica : exemplo 2

```
#include <stdio.h>
#include <stdlib.h>

int main(void){
   double *n; /* ponteiro para o espaço a ser alocado */

   n = (double *)malloc(sizeof(double));

   if (!n){ /* testa a alocação */
      printf("Não conseguiu alocar a memória\n");
      exit(1);
   }

   /* usa o double */
   ...
   /* libera a memória alocada */
   free(n);
   /* o programa continua */
   ...
   return 0;
}
```

Alocação Dinâmica de Vetores

- Suponha que queremos tirar a média de *n* notas.
- Pedimos o valor de *n* e então as *n* notas, certo?
  - E como guardaríamos?
    - Até agora, tínhamos que declarar um mega vetor e torcer para que *n* não fosse maior que nosso vetor.
    - Mas com alocação dinâmica...

Alocação Dinâmica de Vetores

- Primeiro, é preciso declarar um ponteiro para `float`, **`float *v`**;
  - Lembra de `float v[10];` ?
  - O compilador aloca espaço na memória suficiente para guardar 10 `floats`, guardando o endereço do primeiro elemento do vetor `v`. Isso significa que, em `float v[10]`, `v` nada mais é que "`float *`".
- Sendo assim, se declararmos um ponteiro para `float`, tudo que temos que fazer para transformá-lo em um vetor é apontá-lo para um grupo sequencial de `floats` na memória.

Alocação Dinâmica de Vetores

- Bom, se fizermos:

```
v = (float *)malloc(n * sizeof(float));
```

  é exatamente isso que estamos fazendo. Note que pegamos o tamanho de um `float` e multiplicamos pelo número de `floats` (*n*) que o vetor conterá, ou seja, calculamos o tamanho em bytes de `n floats`.

- Quando vimos ponteiros, também vimos que ao fazermos "`v[i]`" estamos fazendo, na verdade, "`*(v+i)`".
  - Portanto, podemos tratar nosso ponteiro como um vetor comum.

Alocação Dinâmica de Vetores

- Por fim, porém não menos importante, para desalocar nosso vetor, basta fazer `free(v);`

- Ok... vamos ao código (ver `malloc.c`)!

Alocação Dinâmica de Vetores

```
int main(void) {

    float *v; /* vetor de notas */
    int i, n; /* contador e número de elementos do vetor */

    printf("Qual o número de notas? ");
    scanf("%d",&n);

    /* aloco espaço suficiente para o vetor de n notas */
    v = (float *)malloc(n * sizeof(float));

    if (v == NULL){
        printf("Não foi possível alocar o vetor\n");
        exit(0);
    }

    for (i=0; i<n; i++) /* carrego o vetor de notas */
        v[i] = nota;
    for (i=0; i<n; i++) /* imprimo o vetor */
        printf("Nota: %f\n", v[i]);

    free(v); /* desaloco o vetor */
    return 0;
}
```

Alocação Dinâmica de Registros

- **O procedimento é igual!**
  - Basta trocar o tipo da variável pelo registro.

- **Exemplo**
  - Programa para armazenar e exibir um círculo de centro (x,y) e raio r.

Alocação Dinâmica de Registros

```
#include <stdio.h>
#include <stdlib.h>

struct s_pos { int x; int y; };
struct s_circulo { struct s_pos c; /* centro do círculo */
                   float r; /* seu raio */};

int main(void){
    struct s_circulo *p; /* o ponteiro para o espaço alocado */

    /* aloco espaço para um struct s_circulo */
    p = (struct s_circulo *)malloc(sizeof(struct s_circulo));

    if (p == NULL){
        printf("Não conseguiu alocar a memória\n");
        exit(1);
    }

    p->c.x = 2; // ou ainda (*p).c.x = 2;
    p->c.y = 4; // ou ainda (*p).c.y = 4;
    p->r = 3.2; // ou ainda (*p).r = 3.2;
    printf("x = %d, y = %d\n",p->c.x, p->c.y);
    printf("r = %f\n",p->r);
    free(p);
    return 0;
}
```

Alocação Dinâmica: realloc

- A função `realloc` faz um bloco já alocado crescer ou diminuir, preservando o conteúdo já existente:

```
int *x, i;

x = (int *) malloc(4000*sizeof(int));

if (x == NULL){
   printf("Não foi possível alocar o vetor\n");
   exit(0);
}

for(i=0; i<4000; i++)
   x[i] = rand()%100;

x = (int *) realloc(x, 8000*sizeof(int));

x = (int *) realloc(x, 2000*sizeof(int));

free(x);
```

Alocação Dinâmica: realloc

- Muitos erram quando utilizam a `realloc`.
- Isso acontece por que na maioria das vezes o programador esquece de “pegar” o retorno da função.
- O `realloc` tenta realocar a quantidade de memória pedida na sequência da já alocada, se não consegue, ele aloca uma nova área e retorna o ponteiro para essa área, liberando a área previamente alocada, e é aí que ocorre o erro.

```
char *pointer;

pointer = (char *) malloc(10 * sizeof(char));

realloc(pointer, 20 * sizeof(char)); /* ERRADO */

pointer = (char *)realloc(pointer, 20 * sizeof(char)); /* CERTO */
```

Alocação Dinâmica: calloc

- A função `calloc` é parecida com a função `malloc`. Exceto pelo fato de que ela inicializa os elementos alocados com zeros.

```
void *calloc(size_t nmemb, size_t size);
```

- `nmemb` – quantidade de elementos a ser alocada
- `size` – tamanho de cada elemento

```
int *vetor, i;

scanf ("%d",&i); /* tamanho do vetor /*

vetor = (int*) calloc(i,sizeof(int)); /*aloca e inicializa/*
```

- Ver `calloc.c`

Parâmetros de entrada

- `argc` (*Argument count*) e `argv` (*Argument vector*)

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
   int a, b;

   if (argc != 3) {   /* testa qtde de parâmetros */
      printf("Qtde de parâmetros inválida!\n");
      return -1;
   }

   a = atoi(argv[1]);   /* converte arg. 2 para int */
   b = atoi(argv[2]);   /* converte arg. 3 para int */

   printf("media = %d\n", (a+b)/2);

   return 0;
}
```

AULA 01

Revisão: Parâmetros de entrada de programa

Prof. Tiago A. Almeida
talmeida@ufscar.br

Parâmetros de entrada

- Podemos passar **entradas de dados como parâmetros da função `main`**

```
#include <stdio.h>

int main (void)
{
   int a, b;

   scanf("%d %d", &a, &b);  /* quero tirar scanf */

   printf("media = %d\n", (a+b)/2);

   return 0;
}
```

- Como passar `a` e `b` na chamada do programa?
- Ex: `./media 5 7`

Parâmetros de entrada

- `argc` (*Argument count*) e `argv` (*Argument vector*)

```
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char *argv[])
{
   int a, b;

   if (argc != 3) {   /* testa qtde de parâmetros */
      printf("Qtde de parâmetros inválida!\n");
      return -1;
   }

   a = atoi(argv[1]);   /* converte arg. 2 para int */
   b = atoi(argv[2]);   /* converte arg. 3 para int */

   printf("media = %d\n", (a+b)/2);

   return 0;
}
```

AULA 01

Revisão: Exercícios

Prof. Tiago A. Almeida
talmeida@ufscar.br