



Disciplina:

Programação Computacional

Prof. Fernando Rodrigues
e-mail: fernandorodrigues@sobral.ufc.br



Aula 12: Programação em C

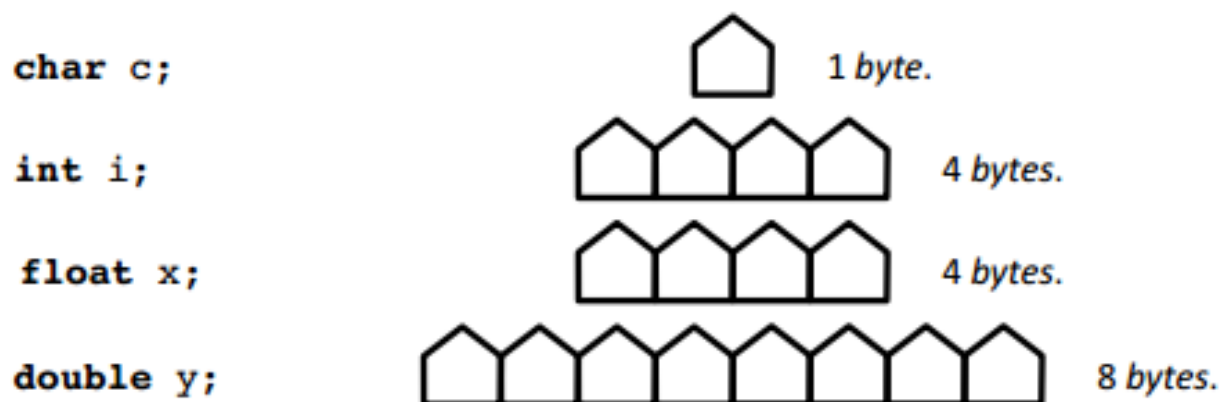
- ❖ Ponteiros

- ❖ Alocação Dinâmica de Memória

Identificadores e endereços

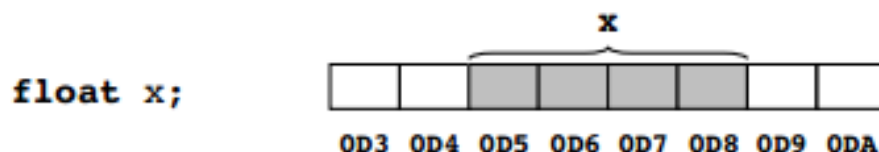
Na linguagem C, uma **declaração** faz a **associação** entre **identificadores** e **endereços de memória**.

O número de endereços (*bytes*) de memória depende do tipo utilizado na declaração.



Identificadores e endereços

Considere a declaração de uma variável do tipo **float**, identificada por **x**:



Após a declaração, os 4 *bytes* que se iniciam no endereço **0D5** estão reservados e são acessíveis pelo programa através do identificador **x**.

```
printf("Endereço de x: %X", &x); // irá exibir 0D5
```

Evolução

Linguagens de alto nível associam identificadores de variáveis a endereços de memória.

Ponteiros

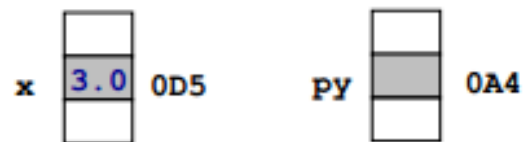
Outra forma de acesso à memória é através dos endereços dos *bytes*. Para isso, é preciso utilizar uma entidade denominada **ponteiro**.

Definição

Ponteiros são variáveis que armazenam **apenas** endereços de memória.

Para declarar uma variável do tipo ponteiro, utiliza-se o símbolo ***** entre o tipo e o identificador do mesmo:

```
float x = 3.0;    // x é uma variável do tipo float
float *py;        // py é ponteiro para variáveis do tipo float
```



Os operadores “*” e “&”

- Ao se trabalhar com ponteiros, duas tarefas básicas serão sempre executadas:
 - Acessar o endereço de memória de uma variável.
 - Acessar o conteúdo de um endereço de memória.
- Para realizar essas tarefas, vamos sempre utilizar apenas dois operadores: o operador “*” e o operador “&”.

Exemplo: operador “*” versus operador “&”

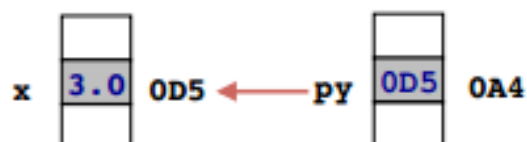
“*”	Declara um ponteiro: <code>int *x;</code>
	Conteúdo para onde o ponteiro aponta: <code>int y = *x;</code>
“&”	Endereço onde uma variável está guardada na memória: <code>&y</code>

Ponteiros

Para armazenar endereços em um ponteiro basta atribuir-lhe o endereço de alguma variável do mesmo **tipo base**:

```
py = &x;           // py aponta para x
```

Esquemáticamente:



```
char  c, *pc;
int    i, *pi;
double x, *px;

pc = &c;
printf("Tamanho de c: %dB\tEndereco de c: %X\t Próximo endereco: %X\n", sizeof(c), pc, pc+1);
pi = &i;
printf("Tamanho de i: %dB\tEndereco de i: %X\t Próximo endereco: %X\n", sizeof(i), pi, pi+1);
px = &x;
printf("Tamanho de x: %dB\tEndereco de x: %X\t Próximo endereco: %X\n", sizeof(x), px, px+1);
```

Tipos de referência

- Referência direta: acesso à memória através do **identificador** associado ao endereço.
- Referência indireta: acesso à memória através do **ponteiro** associado ao endereço.

```
int a, *pa;  
pa = &a;
```

```
// Referência direta  
a = 500;  
printf("Valor de a: %d\n", a);  
printf("Endereco de a: %X\n", &a);  
a++;  
printf("%d" ,a);
```

```
// Referência indireta  
*pa = 500;  
printf("Valor de a: %d\n", *pa);  
printf("Endereco de a: %X\n", pa);  
(*pa)++;  
printf("%d" ,*pa);
```

ATENÇÃO: $pa \neq *pa$

pa significa um endereço; ***pa** significa "o conteúdo armazenado no endereço **pa**" ou "o valor apontado por **pa**."

Aritmética de ponteiros

Se **pa** é um ponteiro para inteiros, que armazena o valor **DA3**, o comando:

```
pa = pa + 1;    // ou pa++;
```

causa uma mudança no valor de **pa**, que passa a apontar para o próximo endereço do tipo inteiro, ou seja, **pa** passa a armazenar o endereço **DA7**.

Apenas os operadores aritméticos **+** e **-** podem ser utilizados sobre ponteiros. Incrementos ou decrementos no valor de um ponteiro fazem com que o mesmo aponte para posições de memória imediatamente posteriores ou anteriores à atual, respectivamente.

Com isso, pode-se utilizar ponteiros para percorrer toda a memória do computador em incrementos inteiros definidos pelo usuário.

Ponteiros e vetores

Trabalhar com ponteiros em estruturas seqüenciais (*strings*, vetores e matrizes) pode melhorar o desempenho de programas.

O endereço inicial de um vetor corresponde ao nome do mesmo.

```
char s[80], *ps, c;  
ps = s;           // equivalente à ps = &s[0];  
  
// Atribuir o elemento da posição 4 da string s à variável c  
  
c = s[4];         // indexação de vetores, ou  
c = *(ps + 4);    // aritmética de ponteiros
```

Alocação de memória

A linguagem C exige que todas as constantes e variáveis de um programa sejam declaradas **antes** de serem utilizadas.

Isso determina, por exemplo, que estruturas seqüenciais (vetores, matrizes e *strings*) sejam declarados com um tamanho fixo pré-determinado (**alocação estática de memória**), comprometendo a utilização do programa na execução de dados de diferentes dimensões.

Ponteiros permitem definir o tamanho de estruturas seqüenciais em tempo de execução do programa.

Este recurso denomina-se **alocação dinâmica de memória**.

Alocação Dinâmica de Memória

- A linguagem C ANSI usa apenas quatro funções para a alocação dinâmica, disponíveis na biblioteca `stdlib.h`:
 - `calloc` : `void *calloc (unsigned int num, unsigned int size)`
 - Ex: `p = (int *) calloc(5,sizeof(int));`
 - `malloc` : `void *malloc (unsigned int num)`
 - Ex: `p = (int *) malloc(50*sizeof(int));`
 - `realloc` : `void *realloc (void *ptr, unsigned int num)`
 - Ex: `p = realloc(p,10*sizeof(int));`
 - `free` : `void free (void *p)`
 - Ex: `free(p);`



Alocação dinâmica de memória: vetores

Um vetor de inteiros com tamanho definido pelo usuário em tempo de execução será declarado como um ponteiro de inteiros.

```
int *x;    // um vetor de inteiros
int n;     // a dimensão do vetor x
int i;

printf("Tamanho do vetor: ");
scanf("%d", &n);
x = (int *) calloc(n, sizeof(int)); // aloca n posições de tamanho int
puts("Informe os elementos do vetor x:");
for (i = 0; i < n; i++)
    scanf("%d", &x[i]);
...
free(x);
```

Importante

A memória alocada dinamicamente **deve ser liberada** pela função `free()`.

Alocação dinâmica de memória: matrizes

Para matrizes, a alocação (e a desalocação) deve ser feita em duas etapas.

```
float **a;      // uma matriz de reais
int m, n;       // as dimensões da matriz a
int i, j;       // para os índices da matriz
```

- Alocar memória para as linhas:

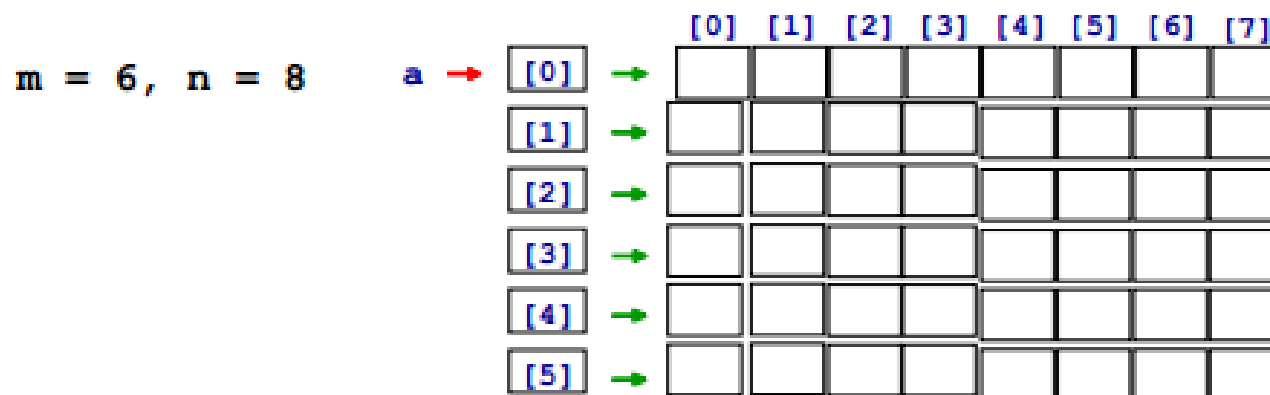
```
printf("Numero de linhas da matriz: ");
scanf("%d", &m);
a = (float **) calloc(m, sizeof(float *)); // aloca m linhas
```

- Para cada linha, alocar memória para as colunas:

```
printf("Numero de colunas da matriz: ");
scanf("%d", &n);
for (i = 0; i < m; i++)
    a[i] = (float *) calloc(n, sizeof(float)); // aloca n colunas
```

Alocação dinâmica de memória: matrizes

Esquemáticamente:



- Para desalocar a memória:

```
for (i = 0; i < m; i++)  
    free(a[i]);  
free(a);
```

Função malloc()

- Serve para alocar memória durante a execução do programa. É ela que faz o pedido de memória ao computador e retorna um ponteiro com o endereço do início do espaço de memória alocado. A função malloc() possui o seguinte protótipo:

`void *malloc (unsigned int num);`

- A função malloc() recebe um parâmetro de entrada
 - num: o tamanho do espaço de memória a ser alocado.
- e retorna
 - NULL: no caso de erro;
 - retorna o ponteiro para a primeira posição do array alocado: Caso contrário.

malloc() x calloc()

Exemplo: malloc() versus calloc()

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      //alocação com malloc
05      int *p;
06      p = (int *) malloc(50*sizeof(int));
07      if(p == NULL){
08          printf("Erro: Memoria Insuficiente!\n");
09      }
10      //alocação com calloc
11      int *p1;
12      p1 = (int *) calloc(50,sizeof(int));
13      if(p1 == NULL){
14          printf("Erro: Memoria Insuficiente!\n");
15      }
16      system("pause");
17      return 0;
18  }
```


malloc() x calloc()



Existe outra diferença entre a função **calloc()** e a função **malloc()**: ambas servem para alocar memória, mas a função **calloc()** inicializa todos os **BITS** do espaço alocado com 0s.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03
04  int main(){
05      int i;
06      int *p, *p1;
07      p = (int *) malloc(5*sizeof(int));
08      p1 = (int *) calloc(5,sizeof(int));
09      printf("calloc \t\t malloc\n");
10      for (i=0; i<5; i++)
11          printf("p1[%d]=%d \t p[%d] = %d\n",i,p1[i],i,p[i]);
12      system("pause");
13      return 0;
14  }
```

Função realloc()

- Serve para alocar memória ou realocar blocos de memória previamente alocados pelas funções malloc(), calloc() ou realloc(). Essa função tem o seguinte protótipo:

`void *realloc (void *ptr, unsigned int num);`

- A função realloc() recebe dois parâmetros de entrada:
 - Um ponteiro para um bloco de memória previamente alocado.
 - **num** : o **tamanho em bytes** do espaço de memória a ser alocado.
- e retorna
 - NULL: no caso de erro;
 - O ponteiro para a primeira posição do array alocado/realocado: caso contrário.

Função realloc() (Cont.)

- Basicamente, a função realloc() modifica o tamanho da memória previamente alocada e apontada pelo ponteiro ptr para um novo valor especificado por num, sendo **num** o tamanho em bytes do bloco de memória solicitado (igual à função malloc()).
- A função realloc() retorna um ponteiro (void *) para o novo bloco alocado. Isso é necessário porque a função realloc() pode precisar mover o bloco antigo para aumentar seu tamanho. Se isso ocorrer, o conteúdo do bloco antigo é copiado para o novo bloco e nenhuma informação é perdida.

Constante NULL

- Está definida na biblioteca `stdlib.h`
- Quando um ponteiro é declarado, ele não possui um endereço associado.
- Qualquer tentativa de uso desse ponteiro causa um comportamento indefinido no programa.
- Um ponteiro pode ter um valor especial **NULL**, que é o endereço de “nenhum lugar”.
- Trata-se de um valor reservado que indica que aquele ponteiro aponta para uma posição de memória inexistente.



Não confunda um ponteiro apontando para **NULL** com um ponteiro não inicializado. O primeiro possui valor fixo, enquanto um ponteiro não inicializado pode possuir qualquer valor.

Verificando a alocação de memória

Seja *p* um ponteiro para um tipo qualquer. O seguinte teste:

```
if (p == NULL)
{
    puts("Memoria insuficiente para alocao de p");
    exit(0);
}
```

poderá ser utilizado para verificar se a alocação de memória para o ponteiro *p* foi bem sucedida. Caso contrário, o programa poderá retornar uma mensagem de erro e terminar a execução.

Operações relacionais com ponteiros

- A linguagem C permite comparar os endereços de memória armazenados por dois ponteiros utilizando uma expressão relacional.
- Por exemplo, os operadores `==` e `!=` são usados para saber se dois ponteiros são iguais ou diferentes, ou seja, se apontam para um mesmo endereço de memória ou não.
- Já os operadores `>`, `<`, `>=` e `<=` são usados para saber se um ponteiro aponta para uma posição mais adiante na memória do que outro.
- Ex: Dado que `p` e `p1` são ponteiros.

```
if(p > p1)
    printf("O ponteiro p aponta para uma posição a frente de p1\n");
else
    printf("O ponteiro p NÃO aponta para uma posição a frente de p1\n");
```

Ponteiros Genéricos (void)

- Normalmente, um ponteiro aponta para um tipo específico de dado.
- Porém, pode-se criar um ponteiro *genérico*.
- Esse tipo de ponteiro pode apontar para todos os tipos de dados existentes ou que ainda serão criados.
- Em linguagem C, a declaração de um ponteiro genérico segue esta forma geral:

void *nome_do_ponteiro;

- o ponteiro genérico permite guardar o endereço de qualquer tipo de dado. Essa vantagem também carrega uma desvantagem: sempre que tivermos de acessar o conteúdo de um ponteiro genérico, será necessário utilizar o operador de *typecast* (*conversão explícita*) sobre ele antes de acessar o seu conteúdo.

Ponteiros Genéricos (Ex. *typedef*)



Sempre que se trabalhar com um ponteiro genérico é preciso convertê-lo para o tipo de ponteiro com o qual se deseja trabalhar antes de acessar o seu conteúdo.

```
01  #include <stdio.h>
02  #include <stdlib.h>
03  int main(){
04      void *pp;
05      int p2 = 10;
06      // ponteiro genérico recebe o endereço de um
//inteiro
07      pp = &p2;
08      //tenta acessar o conteúdo do ponteiro genérico
09      printf("Conteudo: %d\n",*pp); //ERRO
10      //converte o ponteiro genérico pp para (int *)
//antes de acessar seu conteúdo.
11      printf("Conteudo: %d\n",*(int*)pp); //CORRETO
12      system("pause");
13      return 0;
14  }
```


Ponteiros Genéricos (Cont.)

- Como o compilador não sabe qual o tipo do ponteiro genérico, acessar o seu conteúdo gera um tipo de erro. Somente é possível acessar o seu conteúdo depois de uma operação de *typecast*.

- Ex:

```
printf("Conteudo: %d\n", *(int*)pp); //CORRETO
```

- Outro cuidado que devemos ter com ponteiros genéricos: como ele não possui tipo definido, deve-se tomar cuidado ao realizar operações aritméticas.
- Como o compilador não sabe qual o tipo do ponteiro genérico, nas operações de adição e subtração é adicionado/subtraído um total de 1 byte por incremento/decremento, pois esse é o tamanho de uma unidade de memória.

Funções para Alocação Dinâmica de Memória

- A linguagem C ANSI usa apenas quatro funções para o sistema de alocação dinâmica, disponíveis na biblioteca `stdlib.h`. São elas:
 - `free`
 - `calloc`
 - `malloc`
 - `realloc`



Exercícios I

- 1) Escreva um programa que contenha duas variáveis inteiras. Leia essas variáveis do teclado. Em seguida, compare seus endereços e exiba o conteúdo do maior endereço.
- 2) Crie um programa que contenha um array de float contendo 10 elementos. Imprima o endereço de cada posição desse array.
- 3) Crie um programa que contenha um array de inteiros contendo cinco elementos. Utilizando apenas aritmética de ponteiros, leia esse array do teclado e imprima o dobro de cada valor lido.

Exercícios II

- 1) Escreva um programa que mostre o tamanho em bytes que cada tipo de dados ocupa na memória: char, int, float, double.
- 2) Crie uma estrutura representando um aluno de uma disciplina. Essa estrutura deve conter o número de matrícula do aluno, seu nome e as notas de três provas. Escreva um programa que mostre o tamanho em bytes dessa estrutura.
- 3) Crie uma estrutura chamada Cadastro. Essa estrutura deve conter o nome, a idade e o endereço de uma pessoa. Agora, receba um inteiro positivo N e crie um ponteiro para um vetor de tamanho N, alocado dinamicamente, para essa estrutura. Solicite também que o usuário digite os dados desse vetor e depois imprima os dados digitados.

Fim