



LP1

Prof. Luciano Bernardes de Paula



Tópico 1 - Argumentos da função main



Argumentos da função main (argumentos de linha de comando)

São os argumentos que podem ser passados para a função main.

Esses parâmetros são passados ao programa na sua execução.

A função main contabiliza dois parâmetros:

`int argc` e `char *argv[]`

`int main(int argc, char *argv[])`



`argc` é um inteiro que conta o número de parâmetros inseridos (considerando inclusive o próprio nome do programa).

`argv` é um vetor de strings que armazena os parâmetros passados, sendo o índice 0 o próprio nome do programa.



```
int main(int argc, char *argv[]){  
    int i = 0;  
    printf("\n\nnumero de parametros: %d\n", argc);  
    for(i; i < argc; i++) printf("%s ", argv[i]);  
    return 0;  
}
```

Execução: teste.exe 1 2 3

Saída:

Numero de parametros: 4

teste.exe

1 2 3



Para testar há duas formas:

- Execute o programa direto do prompt de comandos do Windows (usando o cmd);
- Configure o Dev-C++ para passar parâmetros para o programa (menu *Executar* → *Parâmetros*).



Como visto, a lista de parâmetros assume que esses são strings.

Caso seja necessário utilizar o parâmetro passado como valor numérico, use as funções `atoi()` e `atof()` (precisam da biblioteca `stdlib.h`).

Converte uma string em inteiro (`atoi(string)`) ou uma string em float (`atof(string)`).



Exemplo – suponha que seu programa receba na função main dois valores, o primeiro inteiro e o segundo como float:

```
int main(int argc, char *argv[]){  
    int i;  
    float f;  
  
    i = atoi(argv[1]);  
    f = atof(argv[2]);  
    ...  
}
```




Tópico 2 – Recursividade (funções recursivas)



Funções recursivas

É uma função que executa a si mesma no seu corpo.

Exemplo - o fatorial de um número qualquer

$$n! = n * (n - 1)!$$



Três pontos devem ser lembrados quando queremos escrever uma função recursiva:

1 – defina o problema em termos recursivos

Por exemplo, no fatorial, é possível definir o cálculo em termos que utilizam a própria função fatorial.



2 – encontre uma condição básica

Por exemplo, na função fatorial, por definição,

$$0! = 1$$

Essa é a condição de “parada” da recursão.



3 – Certificar-se de que a cada chamada, a condição de parada se aproxima.



Exemplo: Fatorial de um número n.

```
double fatorial(double n)
{
    if(n == 0) return 1;

    return n * fatorial(n - 1);
}
```



Há várias situações em que é possível usar funções recursivas.

Há vantagens e desvantagens de se usar recursividade.

Vantagens:

- Torna a escrita do código mais simples e elegante, tornando-o fácil de entender e de manter.

Desvantagens:

- Quando o loop recursivo é muito grande é consumida muita memória nas chamadas a diversos níveis de recursão, pois cada chamada recursiva aloca memória para os parâmetros, variáveis locais e de controle.
- Em muitos casos uma solução iterativa gasta menos memória, e torna-se mais eficiente em termos de performance do que usar recursão.



Tópico 3 - Ponteiros



Ponteiros

- O nome de uma variável indica o **conteúdo** que está armazenado nela.
- O **endereço** de uma variável pode ser armazenado em uma variável do tipo ponteiro.
- O conteúdo de um ponteiro é um endereço de memória que aponta para alguma variável.



Por que utilizar ponteiros?

- Fornecem maneiras com as quais as funções podem realmente modificar argumentos recebidos;
- Passar vetores ou *strings* de uma função à outra de forma mais conveniente;
- Manipular elementos de vetores e matrizes mais facilmente;
- Criar estruturas de dados complexas;
- Alocar e desalocar memória dinamicamente;
- Etc...



Ponteiro variável

Em C, é possível declarar ponteiros como variáveis.

Essas variáveis armazenarão endereços de variáveis.

Ou seja, um ponteiro “apontará” para uma variável (para o trecho de memória alocado para aquela variável).



O operador & retorna o endereço da variável.

Com o printf, a opção %p imprime o endereço de uma variável qualquer.

```
printf("%p", &x);
```



Declarando ponteiros

Para declarar um ponteiro, basta fazer da seguinte forma:

```
int *p;
```

Essa declaração cria a variável `p` como um ponteiro para um inteiro.

É possível criar ponteiros para todos os tipos de variáveis do C.

O tipo define para quantos bits o ponteiro aponta.



Quando um ponteiro é criado, ele possui como conteúdo NULL.

Para “apontá-lo” para alguma variável, basta:

```
p = &var;
```

Dessa forma, p passa a conter o endereço da variável var.



Alterando o conteúdo da variável apontada

É possível manipular o conteúdo da variável apontada usando um ponteiro.

```
p = &var;
```

```
*p = 10; //equivale a var = 10;
```



Lembre-se

Ao utilizar um ponteiro, se for colocado o asterisco, a operação estará manipulando o conteúdo que existe no endereço apontado.



Operações com ponteiros

Atribuição

```
pvar1 = &var1;  
pvar2 = &var2;
```

Atribuição ao conteúdo do endereço apontado

```
*pvar1 = 10; //equivale a var1 = 10;
```



É possível incrementar ou decrementar um ponteiro.

A cada incremento ou decremento, o ponteiro “anda” um certo tanto de bits na memória igual ao seu tipo.

Exemplo:

```
int *pvar;  
pvar = &var;  
pvar++;
```

Faz com que o ponteiro pvar aponte para os próximos 32 bits após o endereço de var.

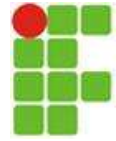


É possível calcular a diferença entre dois ponteiros do mesmo tipo:

```
diferenca = pvar1 - pvar2;
```

Isso resultará na diferença entre os dois endereços apontados em número de variáveis do tipo dos ponteiros.

Operadores relacionais (>, <, ==, etc) são permitidos apenas entre ponteiros do mesmo tipo.



O nome de um vetor ou matriz representa um ponteiro para sua posição inicial.

É possível ler ou alterar seus valores utilizando a notação de ponteiro.

```
int M[] = {1, 2, 3, 4, 5};
```

```
for(i=0; i < 5; i++){  
    printf("%d", M[i]);    //notação de vetor  
}
```

```
for(i=0; i < 5; i++){  
    printf("%d", *(M + i)); //notação de ponteiro  
}
```



Nesse exemplo:

$M + i$ é equivalente a $\&M[i]$, portanto

$*(M+i)$ é equivalente ao conteúdo de $(M + i)$, ou seja, $M[i]$.



Ponteiros como vetores

```
#include <stdio.h>
main ()
{
    int vetor[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int *p;
    p = vetor;
    printf ("O terceiro elemento do vetor é: %d",p[2]);
}
```

Podemos ver que `p[2]` equivale a `*(p+2)`



Apesar de o nome de um vetor ser um ponteiro,
não é possível alterá-lo.

```
int vetor[10] = {0, 1, 2, 3};
```

```
vetor++; // ERRO!
```

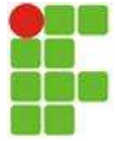
Isso pq se trata de um ponteiro constante, ou seja,
ele nunca muda do endereço original.



Passando argumentos para funções por referência

É possível declarar funções que recebam endereços de variáveis.

```
void soma(int a, int b, int *c){  
    *c = a + b;  
}
```

Chamada no main():

```
soma(a, b, &c);
```

Ou

```
pc = &c;
```

```
soma(a, b, pc);
```

Sendo pc um ponteiro para a variável que receberá o conteúdo.

Exemplo



```
void troca(int a, int b){  
    int aux;  
    aux = a;  
    a = b;  
    b = aux;  
}
```

No programa principal, a troca será efetuada?

Não! Lembre-se de passagem de parâmetro por valor e por referência.



```
void troca(int *a, int *b){  
    int aux;  
  
    aux = *a;  
    *a = *b;  
    *b = aux;  
}
```

Agora sim!



Ponteiros void

É um ponteiro que pode apontar para qualquer tipo de variável.

```
void *p;
```

O conteúdo de um ponteiro void não pode ser acessado diretamente.



Para acessar o valor que ele possui, é preciso feito um *cast* de tipo:

```
void *p;
```

```
int *c;
```

```
int i = 5;
```

```
int b = 4;
```

```
c = &i;
```

```
p = (int *)b;
```

```
printf("\np = %i c = %i\n", (int *)p, *c);
```



É possível ter um ponteiro apontando para uma variável que é uma estrutura.

```
typedef struct{  
    int a;  
    float b;  
}teste;
```

```
teste s;  
teste *ps;
```

```
ps = &s;
```



Para acessar os membros de uma estrutura a partir de um ponteiro:

```
ps.a = 1; // ERRADO!
```

```
(*ps).a = 1; // Correto
```

```
ps->a = 1; // Correto e o mais usado
```



Ponteiros para ponteiros

É possível criar um ponteiro para um ponteiro.

Esse ponteiro apontará para outro ponteiro.

```
int **p;
```

Útil para usar com matrizes com mais de uma dimensão.



Tópico 4 – Alocação dinâmica de memória



Alocação dinâmica de memória

Imagine um programa que armazenará o cadastro de usuários.

Ao implementá-lo, é possível criar, por exemplo, um vetor de estruturas com 50 posições.

O seu programa poderá, portanto, armazenar 50 pessoas ao mesmo tempo.



Há dois problemas:

- Se houver menos que 50 pessoas, haverá memória alocada sem utilização;
- Se o total de pessoas for maior que 50, não será possível inserir os dados das pessoas excedentes sem extravasar o vetor.



Todo programa para ser executado deve ser carregado na memória.

O mesmo deve acontecer com as funções (instruções) e variáveis que o programa utilize.

Toda memória disponível que não está sendo utilizada é chamada de *heap*.



É possível alocar dinamicamente memória, ou seja, durante a execução do programa.

Em C, existem as funções `malloc()`, `calloc()` e `free()`, que são utilizadas para esse fim.



`malloc()`

Essa função recebe um número inteiro positivo que representa a quantidade de bytes de memória desejada.

Solicita memória ao sistema operacional e retorna um ponteiro void para o primeiro byte do novo bloco alocado.

Se não há memória suficiente, o `malloc()` retorna um ponteiro NULL.



Exemplo:

```
typedef {  
    int dia;  
    int mes;  
    int ano;  
} Data;
```

```
Data *ptr;  
ptr = (Data *) (malloc(sizeof(Data)));
```

ptr apontará para uma struct do tipo Data.



calloc()

A função `calloc()` aloca memória para um vetor de `x` posições, inicializados com 0.

Ela recebe dois argumentos: o primeiro é o número de itens desejados e o segundo é o tamanho do item.

```
int *mem;  
mem = (int *) calloc(100, sizeof(int));
```

`mem` pode ser usado como um vetor comum.



```
float *notas;
```

```
notas = (float *) calloc(10, sizeof(float));
```

```
notas[0] = 5.0;
```

```
...
```



free()

A função `free()` é usada para liberar um espaço de memória previamente alocado com `malloc()` ou `calloc()`.

`free()` recebe um ponteiro para a memória alocada.

Após utilizar um espaço de memória que foi alocado (e não será mais utilizado), uma boa prática é liberar a memória, que passará a estar disponível, usando `free()`.



Exemplo de alocação de uma variável de tipo básico.

```
int *p;
```

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

```
...
```

```
free(p);
```



```
#define TAM 10
```

```
...
```

```
float *notas;
```

```
notas = (float *) calloc(TAM, sizeof(float));
```

```
...
```

```
free(notas);
```



É recomendado sempre checar se o malloc() ou calloc() realmente alocaram a memória pedida, já que se não houver memória disponível, é retornado um ponteiro para NULL.

Sempre teste se o ponteiro é diferente de NULL.

```
notas = (float *)calloc(tamanho, sizeof(float));
```

```
if(notas == NULL) {  
    printf("Não foi possível alocar memória...");  
    return 1;  
}
```



CUIDADO!

Como os ponteiros retornados por `malloc()` ou `calloc()` não são ponteiros estáticos, é possível incrementá-los, passando para o trecho de memória vizinho.

Se não for tomado cuidado, é possível “perdê-los” na memória...



Exercícios.