

Rodrigo Henrich

rodrigohenrich@faccat.br



Ponteiros em C

- Ponteiros ou apontadores, são variáveis que armazenam o endereço de memória de outras variáveis.
- Dizemos que um ponteiro “aponta” para uma variável quando contém o endereço da mesma.
- Os ponteiros podem apontar para qualquer tipo de variável.
- Assim eles podem ser de qualquer tipo de dado presente na linguagem

Ponteiros em C

- Ponteiros são úteis em várias situações, por exemplo
 - Alocação dinâmica de memória
 - Manipulação de arrays
 - Retorno múltiplo em funções
 - Referência para listas, pilhas, árvores e grafos

Declarando um ponteiro em C

- Para declarar um ponteiro em C usamos a sintaxe

tipo *nome_ponteiro;

- De forma simples a única diferença entre uma variável e um ponteiro é o *
- Exemplo

int *ponteiro

Declarando um ponteiro em C

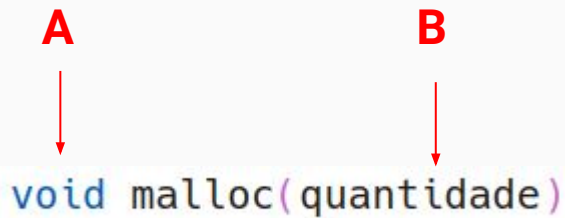
```
1 | #include<stdio.h>
2 | #include<locale.h>
3 |
4 | int main(){
5 |     setlocale(LC_ALL, "");
6 |     //Declarando uma variável do tipo inteiro
7 |     int valor = 100;
8 |     //Declarando um ponteiro
9 |     int *ponteiro;
10 |    //Atribuindo o endereço de valor ao ponteiro
11 |    ponteiro = &valor;
12 |    //Saídas
13 |    printf("\nValor da variável 'valor' %d", valor);
14 |    printf("\nEndereço da variável 'valor' %x", &valor);
15 |    printf("\nConteúdo da variável 'ponteiro' %x", ponteiro);
16 |    printf("\nValor que o ponteiro 'ponteiro' aponta %d", *ponteiro);
17 | }
```

Alocação dinâmica de memória

- A alocação dinâmica de memória permite alocar memória em tempo de execução
- Desta forma não ficamos limitados as variáveis definidas ou ao tamanho de um vetor
- O processo é realizado por meio de um ponteiro e algumas funções
 - malloc()
 - calloc()
 - realloc()
 - free()

Alocação dinâmica de memória

- A função **malloc()** permite alocar espaço de memória



A ↓
`void malloc(quantidade)`
B ↓

A - Retorna um ponteiro do tipo **void** para o primeiro byte alocado

Esse ponteiro poderá ser convertido para qualquer tipo

B - Quantidade de bytes que serão alocados

Alocação dinâmica de memória

- dificilmente será possível saber o tamanho de cada tipo de dado
- desta forma normalmente a função **malloc** será usada em conjunto com a função **sizeof()**
- Essa função retorna a quantidade de bytes de determinado tipo de dado

A **B**

↓ ↓

```
int sizeof(objeto)
```

A - A função retorna um inteiro com a quantidade de bytes do objeto

B - Objeto ou tipo de que se pretende descobrir o tipo

Alocação dinâmica de memória

- Vamos usar a função `malloc` em conjunto com a função `sizeof`



The diagram shows two lines of C code. Above the first line, 'A' has a red arrow pointing to the asterisk in `*p`. Above the second line, 'B' has a red arrow pointing to the opening parenthesis of the cast `(char*)`, and 'C' has a red arrow pointing to the `sizeof(char)` expression.

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

A - Alocação de um ponteiro do tipo **char**

B - Aqui estamos fazendo um casting do tipo `void` para um ponteiro **char**

C - Usando a função **sizeof** para saber quantos bytes alocar

Alocação dinâmica de memória

- Podemos usar esse ponteiro como se fosse uma variável



```
char *p;  
p = (char*) malloc(sizeof(char));  
  
printf("Digite uma letra: ");  
scanf("%c", p);  
  
printf("O valor de c é %c", *p);
```

A - Alocação de um ponteiro do tipo **char**

B - Aqui não vai o &, p já é um ponteiro

C - Aqui precisa do * para acessar o endereço do ponteiro

Alocação dinâmica de memória

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

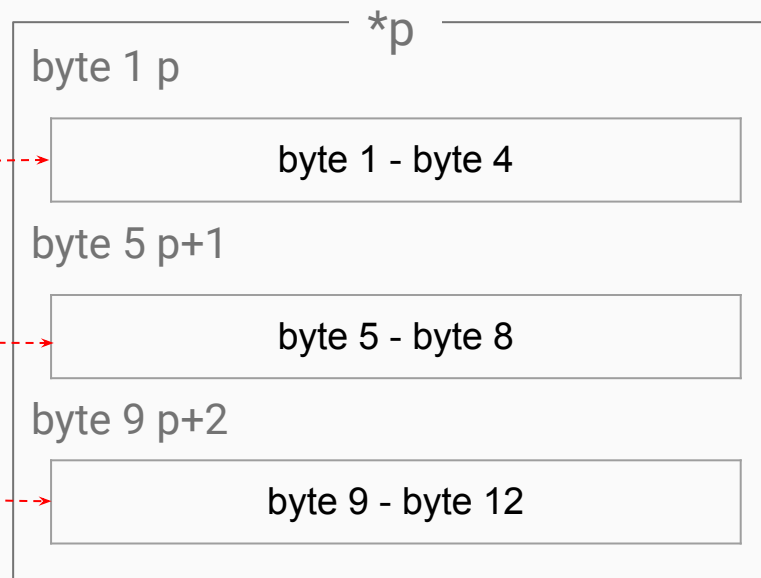
1 byte



0	1	0	1	0	0	1	0
---	---	---	---	---	---	---	---

Alocação dinâmica de memória

```
int *p;  
p = (int*) malloc(3*sizeof(int));  
  
printf("Digite um número: ");  
scanf("%d",p);  
printf("Digite um número: ");  
scanf("%d",p+1);  
printf("Digite um número: ");  
scanf("%d",p+2);  
  
printf("O valor é %d\n",*p);  
printf("O valor é %d\n",*(p+1));  
printf("O valor é %d\n",*(p+2));
```



Alocação dinâmica de memória

- A função **calloc()** permite alocar espaço de memória

A **B** **C**

↓ ↓ ↓

```
void calloc(quantidade,tamanho);
```

A principal vantagem da função **calloc()** é que ela inicializa os espaços com zeros

A - Retorna um ponteiro do tipo **void** para o primeiro byte alocado

B - Quantidade de bytes que serão alocados

C - Tamanho do byte a ser alocado



Mas qual a utilidade da
alocação dinâmica, até
agora poderia ter usado
apenas um vetor e ter o
mesmo resultado

Alocação dinâmica de memória

- A função **realloc()** permite redimensionar o espaço alocado pelas funções **malloc()** e **calloc()**

A



B



C



```
void realloc(enderecoAlocado, novoTamanho)
```

A - Retorna um ponteiro do tipo **void** para o primeiro byte alocado

B - Ponteiro que será realocado

C - Nova quantidade de bytes do ponteiro

```
int *p;  
p = (int*) calloc(3, sizeof(int));
```

Declaração e inicialização do ponteiro

```
printf("Digite um número: ");  
scanf("%d", p);  
printf("Digite um número: ");  
scanf("%d", p+1);  
printf("Digite um número: ");  
scanf("%d", p+2);
```

Lendo dados para ocupar os espaços do ponteiro

```
p = realloc(p, 6*sizeof(int));
```

Realocando o ponteiro, aumentando sua capacidade para
6 valores do tipo inteiro

```
printf("Digite um número: ");  
scanf("%d", p+3);  
printf("Digite um número: ");  
scanf("%d", p+4);  
printf("Digite um número: ");  
scanf("%d", p+5);
```

Lendo valores para as novas posições do ponteiro

```
printf("O valor é %d\n", *p);  
printf("O valor é %d\n", *(p+1));  
printf("O valor é %d\n", *(p+2));  
printf("O valor é %d\n", *(p+3));  
printf("O valor é %d\n", *(p+4));  
printf("O valor é %d\n", *(p+5));
```

Os valores que já estavam armazenados, continuam salvos

junto com os valores novos

Alocação dinâmica de memória

- A função **free()** permite liberar a memória

A
↓
`free(p);`

A - Ponteiro que será liberado da memória

Alocação dinâmica de memória

<https://www.ime.usp.br/~pf/algoritmos/aulas/aloca.html#malloc>

https://sae.unb.br/cae/conteudo/unbfga/cb/new_alocacaodinamica.html

Passagem de parâmetro por referência

- Caso alguma função precise retornar mais de um valor podemos contar com um artifício
- Passamos um ponteiro como parâmetro e ao modificar ele dentro da função ele será modificado na função principal também
- Um exemplo criamos uma função que soma 100 a um número

Passagem de parâmetro por referência

```
1  #include<stdio.h>
2
3  void incrementaNumero(int *valor){
4      *valor+=100;
5  }
6  int main(){
7      int n;
8      n=10;
9      printf("\nn=%d",n);
10     incrementaNumero(&n);
11     printf("\nn=%d",n);
12
13 }
```

Arquivos em C

- Usando arquivos em C é possível realizar o armazenamento de dados mesmo quando o programa é fechado;
- Em C temos dois tipos de arquivos, de texto ou binários
- O primeiro armazena apenas texto
- O segundo pode armazenar uma série de dados, que estarão em formato legível apenas pelo nosso programa

Ponteiro para um arquivo

- O arquivo é manipulado através de um ponteiro especial para o arquivo que aponta para a localização de um registro.

FILE < *ponteiro >

- Exemplo

FILE *arquivo;

Abrindo um arquivo

- Para abrir um arquivo usamos o seguinte comando

<ponteiro> = fopen("nome do arquivo", "tipo de abertura");

- **"nome do arquivo"** é o nome do arquivo que desejo abrir
- **"tipo de arquivo"** é o tipo de abertura ou acesso para o arquivo

Abrindo um arquivo tipos

- **r**: Permissão de abertura somente para leitura. O arquivo precisa existir
- **w**: Permissão de abertura para escrita (gravação). Ele cria o arquivo ou sobrescreve se já existir (**Cuidado para não perder informações**)
- **a**: Permissão para abrir um arquivo texto para escrita(gravação), permite acrescentar novos dados ao final do arquivo. Caso não exista, ele será criado.

Fechando um arquivo

- Para fechar um arquivo usamos a função **`fclose(<ponteiro>);`**

Exemplo

```
1  #include<stdio.h>
2
3  int main(){
4      FILE *arquivo;
5      arquivo = fopen("arquivo.txt","a");
6      fclose(arquivo);
7      printf("Arquivo criado.");
8  }
```

Problemas que podem ocorrer...

- Tentar abrir um arquivo que não existe para leitura;
- Não ter permissão de escrita no arquivo;
- O arquivo está bloqueado por estar sendo usado por outro programa.
- Se ocorrer algum erro ao abrir o arquivo a função fopen retorna NULL
- Desta forma podemos testar se ela foi bem sucedida.

Problemas que podem ocorrer...

```
1  #include<stdio.h>
2
3  int main(){
4      FILE *arquivo;
5      arquivo = fopen("arquivo.txt","r");
6      if(arquivo!=NULL){
7          printf("Arquivo carregado com sucesso!");
8          fclose(arquivo);
9      }
10     else
11         printf("Erro ao carregar o arquivo.");
12 }
```

Gravar informações no arquivo

- Para gravar informações em um arquivo precisamos usar uma função especial da biblioteca `stdio.h`
- Ela é a função `fprintf()`

`fprintf(nome_do_ponteiro_para_o_arquivo, "%s", variavel_string)`

- Exemplo:

Gravar informações no arquivo

```
1  #include<stdio.h>
2
3  int main(){
4      FILE *arquivo;
5      arquivo = fopen("arquivo.txt","a");
6      if(arquivo!=NULL){
7          for(int i=0;i<10;i++)
8              fprintf(arquivo,"Linha %d do arquivo\n",i+1);
9          fclose(arquivo);
10     }
11     else
12         printf("Erro ao carregar o arquivo.");
13 }
```

Lendo informações no arquivo

- Para ler os dados do arquivo podemos usar a função `getc()`;
- `getc(ponteiro_do_arquivo)`;
- Essa função retorna cada caracter a cada chamada dela até chegar ao final do arquivo
- Quando chegar ao final do arquivo a função retorna **EOF** (End Of File)
- Neste caso o arquivo precisa ser carregado em **modo leitura "r"**
- Exemplo

Lendo informações no arquivo

```
1  #include<stdio.h>
2
3  int main(){
4      FILE *arquivo;
5      arquivo = fopen("arquivo.txt","r");
6      if(arquivo!=NULL){
7          char letra;
8          do{
9              letra = getc(arquivo);
10             printf("%c",letra);
11         }
12         while(letra!=EOF);
13     }
14     else
15         printf("Erro ao carregar o arquivo.");
16 }
```


Escrevendo structs em arquivo

- Para escrever structs ou outros tipos de dados em arquivos usamos a função `fwrite()`
- A sintaxe desta função é um pouco mais complexa

```
int fwrite (void *dados, int tamanho_do_elemento, int num_elementos,  
FILE *fluxo);
```

Escrevendo structs em arquivo

```
int fwrite (void *dados, int tamanho_do_elemento, int num_elementos,  
FILE *fluxo);
```

- ***dados**, deve ser um ponteiro para os dados que vamos escrever
- **tamanho do elemento**, para descobrir isso vamos precisar de uma outra função **sizeof(elemento)**
- **num_elementos** indica a quantidade de elementos que serão escritos
- ***fluxo** indica o arquivo onde serão escritos os dados

Exemplo

```
1  #include<stdio.h>
2  typedef struct{
3      char nome[50];
4      int idade;
5  } Pessoa;
6
7  int main(){
8      FILE *arquivo;
9      Pessoa p1 = {"Ana", 13};
10     arquivo = fopen("arquivoDados", "w");
11     if(arquivo!=NULL){
12         fwrite(&p1, sizeof(Pessoa), 1, arquivo);
13     }
14     else
15         printf("Erro ao carregar o arquivo.");
16 }
```

Lendo structs em arquivo

```
int fread (void *dados, int tamanho_do_elemento, int num_elementos,  
FILE *fluxo);
```

- ***dados**, ponteiro para onde serão armazenados os dados lidos
- **tamanho do elemento**, para descobrir isso vamos precisar de uma outra função **sizeof(elemento)**
- **num_elementos** indica a quantidade de elementos que serão lidos
- ***fluxo** indica o arquivo de onde serão lidos os elementos

Exemplo

```
1  #include<stdio.h>
2  typedef struct{
3      char nome[50];
4      int idade;
5  } Pessoa;
6
7  int main(){
8      FILE *arquivo;
9      arquivo = fopen("arquivoDados","r");
10     Pessoa p1;
11     if(arquivo!=NULL){
12         fread(&p1,sizeof(Pessoa),1, arquivo);
13         printf("Nome: %s\n",p1.nome);
14         printf("Idade: %d",p1.idade);
15     }
16     else
17         printf("Erro ao carregar o arquivo.");
18 }
```

Exercício

- Crie um programa em que se possa cadastrar uma turma com até 10 alunos, cada aluno possui um
 - nome,
 - matrícula,
 - três notas,
 - média das notas
- o programa deve ter um menu para adicionar novos alunos e deve permitir listar os alunos.
- Além disso as informações devem ser salvas em arquivos.