
# Ciência da Computação
# Programação

Prof. Sérgio Yunes
syunes@gmail.com



# Ponteiros

## Como funcionam os ponteiros

- ints guardam inteiros.
- floats guardam números de ponto flutuante.
- chars guardam caracteres.
- Ponteiros guardam endereços de memória.

- Quando você anota o endereço de um colega você está criando um ponteiro.
- O ponteiro é este seu pedaço de papel. Ele tem anotado um endereço.
- Qual é o sentido disto? Simples. Quando você anota o endereço de um colega, depois você vai usar este endereço para achá-lo.



## Como funcionam os ponteiros

- O C funciona assim. Voce anota o endereço de algo numa variável ponteiro para depois usar.

- Da mesma maneira, uma agenda, onde são guardados endereços de vários amigos, poderia ser vista como sendo uma matriz de ponteiros no C.

- Um ponteiro também tem tipo. Veja: quando você anota um endereço de um amigo você o trata diferente de quando você anota o endereço de uma firma. Apesar de o endereço dos dois locais ter o mesmo formato (rua, número, bairro, cidade, etc.) eles indicam locais cujos conteúdos são diferentes. Então os dois endereços são ponteiros de *tipos* diferentes.

## Como funcionam os ponteiros

- No C quando declaramos ponteiros nós informamos ao compilador para que tipo de variável vamos apontá-lo.
- Um ponteiro int aponta para um inteiro, isto é, guarda o endereço de um inteiro.



## Declarando e utilizando ponteiros

- Para declarar um ponteiro temos a seguinte forma geral:

*tipo_do_ponteiro *nome_da_variável;*

- É o asterisco (*****) que faz o compilador saber que aquela variável não vai guardar um valor mas sim um endereço para aquele tipo especificado. Exemplos de declarações:

```
int *pt;        //  declara um ponteiro para um inteiro
char *temp,*pt2; // dois ponteiros para caracteres
```

- Eles ainda não foram inicializados (como toda variável do C que é apenas declarada). Isto significa que eles apontam para um lugar indefinido. Este lugar pode estar, por exemplo, na porção da memória reservada ao sistema operacional do computador.

## Declarando e utilizando ponteiros

- Usar o ponteiro nestas circunstânicias pode levar a um travamento do micro, ou a algo pior. *O ponteiro deve ser inicializado (apontado para algum lugar conhecido) antes de ser usado!* Isto é de suma importância!



## Declarando e utilizando ponteiros

- Para atribuir um valor a um ponteiro recém-criado poderíamos igualá-lo a um valor de memória. Mas, como saber a posição na memória de uma variável do nosso programa?
- Seria muito difícil saber o endereço de cada variável que usamos, mesmo porque estes endereços são determinados pelo compilador na hora da compilação e realocados na execução.
- Podemos então deixar que o compilador faça este trabalho por nós. Para saber o endereço de uma variável basta usar o operador **&**.

## Declarando e utilizando ponteiros

- Veja o exemplo:

```
int count=10;
int *pt;     //  cria um apontador para um inteiro pt
pt=&count;
```

- A expressão **&count** nos dá o endereço de count, o qual armazenamos em **pt**. Simples, não é? Repare que *não* alteramos o valor de **count**, que continua valendo 10.

- Como nós colocamos um endereço em **pt**, ele está agora "liberado" para ser usado.



## Declarando e utilizando ponteiros

- Podemos, por exemplo, alterar o valor de **count** usando **pt**.

- Para tanto vamos usar o operador "inverso" do operador **&**.

- É o operador *****.

- No exemplo acima, uma vez que fizemos **pt=&count** a expressão ***pt** é equivalente ao próprio **count**.

- Isto significa que, se quisermos mudar o valor de count para 12, basta fazer ***pt=12**.

## Declarando e utilizando ponteiros

- Vamos fazer uma pausa e voltar à nossa analogia para ver o que está acontecendo.
- Digamos que exista uma firma. Ela é como uma variável que já foi declarada. Você tem um papel em branco onde vai anotar o endereço da firma. O papel é um ponteiro do tipo firma. Você então liga para a firma e pede o seu endereço, o qual você vai anotar no papel. Isto é equivalente, no C, a associar o papel à firma com o operador **&**. Ou seja, o operador **&** aplicado à firma é equivalente a você ligar para a mesma e pedir o endereço. Uma vez de posse do endereço no papel você poderia, por exemplo, fazer uma visita à firma. No C você faz uma visita à firma aplicando o operador ***** ao papel. Uma vez dentro da firma você pode copiar seu conteúdo ou modificá-lo.
-



## Declarando e utilizando ponteiros

- O operador ***** (multiplicação) não é o mesmo operador que o ***** (referência de ponteiros). Para começar o primeiro é binário, e o segundo é unário pré-fixado.
- Aqui vão dois exemplos de usos simples de ponteiros:

```
#include <stdio.h>
int main () {
      int num,valor;
      int *p;
      num=55;
      p=&num;     /* Pega o endereco de num */
      valor=*p;       /* Valor e igualado a num de uma maneira indireta */
      printf ("\n\n%d\n",valor);
      printf ("Endereco para onde o ponteiro aponta: %p\n",p); //%p  indica à função que
         ela deve imprimir um endereço.
      printf ("Valor da variavel apontada: %d\n",*p);
      return(0);
}
```

## Declarando e utilizando ponteiros

```
#include <stdio.h>
int main () {
int num,*p;
num=55;
p=&num;     /* Pega o endereco de num */
printf ("\nValor inicial: %d\n",num);
*p=100; /* Muda o valor de num de uma maneira indireta */
printf ("\nValor final: %d\n",num);
return(0);
}
```



## Declarando e utilizando ponteiros

- Podemos fazer algumas operações aritméticas com ponteiros.
- A primeira, e mais simples, é igualar dois ponteiros.
- Se temos dois ponteiros **p1** e **p2** podemos igualá-los fazendo **p1**=**p2**.
- Repare que estamos fazendo com que **p1** aponte para o mesmo lugar que **p2**.
-

## Declarando e utilizando ponteiros

- Se quisermos que a variável apontada por **p1** tenha o mesmo conteúdo da variável apontada por **p2** devemos fazer

```
*p1=*p2
```

- Basicamente, depois que se aprende a usar os dois operadores (**&** e *****) fica fácil entender operações com ponteiros.
-



## Declarando e utilizando ponteiros

- As próximas operações, também muito usadas, são o incremento e o decremento.
- Quando incrementamos um ponteiro ele passa a apontar para o próximo valor do mesmo tipo para o qual o ponteiro aponta.
- Isto é, se temos um ponteiro para um inteiro e o incrementamos ele passa a apontar para o próximo inteiro.

## Declarando e utilizando ponteiros

- Esta é mais uma razão pela qual o compilador precisa saber o tipo de um ponteiro: se você incrementa um ponteiro **char*** ele anda 1 byte na memória e se você incrementa um ponteiro **double*** ele anda 8 bytes na memória.

- O decremento funciona semelhantemente. Supondo que **p** é um ponteiro, as operações são escritas como:

```
p++;
p--;
```



## Declarando e utilizando ponteiros

- Mais uma vez insisto. Estamos falando de operações com *ponteiros* e não de operações com o conteúdo das variáveis para as quais eles apontam. Por exemplo, para incrementar o conteúdo da variável apontada pelo ponteiro **p**, faz-se:

```
(*p)++;
```

- Outras operações aritméticas úteis são a soma e subtração de inteiros com ponteiros. Vamos supor que você queira incrementar um ponteiro de 15. Basta fazer:

```
p=p+15;  ou  p+=15;
```

## Declarando e utilizando ponteiros

- E se você quiser usar o conteúdo do ponteiro 15 posições adiante:

```
*(p+15);  // A subtração funciona da mesma maneira.
```

- Uma outra operação, às vezes útil, é a comparação entre dois ponteiros. Mas que informação recebemos quando comparamos dois ponteiros?

- Bem, em primeiro lugar, podemos saber se dois ponteiros são iguais ou diferentes (== e !=). No caso de operações do tipo >, <, >= e <= estamos comparando qual ponteiro aponta para uma posição mais alta *na memória*.



## Declarando e utilizando ponteiros

- Então uma comparação entre ponteiros pode nos dizer qual dos dois está "mais adiante" na memória. A comparação entre dois ponteiros se escreve como a comparação entre outras duas variáveis quaisquer:

```
p1>p2
```

- Há entretanto operações que você *não* pode efetuar num ponteiro.

- Você não pode dividir ou multiplicar ponteiros, adicionar dois ponteiros, adicionar ou subtrair floats ou doubles de ponteiros.

# Ponteiros

AUTO AVALIAÇÃO

Veja como você está:

a) Explique a diferença entre p++; (*p)++; *(p++);
- O que quer dizer *(p+10);?
- Explique o que você entendeu da comparação entre ponteiros

b) Qual o valor de y no final do programa? Tente primeiro descobrir e depois verifique no computador o resultado. A seguir, escreva um /* comentário */ em cada comando de atribuição explicando o que ele faz e o valor da variável à esquerda do '=' após sua execução.

```
int main()
{
   int y, *p, x;
   y = 0;
   p = &y;
   x = *p;
   x = 4;
   (*p)++;
   x--;
   (*p) += x;
   printf ("y = %d\n", y);
   return(0);
}
```



# Ponteiros e Vetores

**Vetores como ponteiros**

- Quando você declara uma matriz da seguinte forma:

*tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN];*

- o compilador C calcula o tamanho, em bytes, necessário para armazenar esta matriz. Este tamanho é:
- *tam1 x tam2 x tam3 x ... x tamN x tamanho_do_tipo*
- O compilador então aloca este número de bytes em um espaço livre de memória.
- O *nome da variável* que você declarou é na verdade *um ponteiro para o tipo da variável da matriz*.
- Este conceito é fundamental. Eis porque: Tendo alocado na memória o espaço para a matriz, ele toma o nome da variável (que é um ponteiro) e aponta para o *primeiro* elemento da matriz.

## Ponteiros e Vetores

**Vetores como ponteiros**

- Mas aí surge a pergunta: então como é que podemos usar a seguinte notação?

*nome_da_variável[índice]*

- Isto pode ser facilmente explicado desde que você entenda que a notação acima é *absolutamente equivalente* a se fazer:

**(nome_da_variável+índice)*

- Agora podemos entender como é que funciona um vetor! Vamos ver o que podemos tirar de informação deste fato.



## Ponteiros e Vetores

**Vetores como ponteiros**

- Fica claro, por exemplo, porque é que, no C, a indexação começa com zero. É porque, ao pegarmos o valor do primeiro elemento de um vetor, queremos, de fato, ***nome_da_variável** e então devemos ter um índice igual a zero. Então sabemos que:

**nome_da_variável* é equivalente a *nome_da_variável[0]*

- Outra coisa: apesar de, na maioria dos casos, não fazer muito sentido, poderíamos ter índices negativos. Estaríamos pegando posições de memória antes do vetor. Isto explica também porque o C não verifica a validade dos índices.
- Ele *não* sabe o tamanho do vetor. Ele apenas aloca a memória, ajusta o ponteiro do nome do vetor para o início do mesmo e, quando você usa os índices, encontra os elementos requisitados.

# Ponteiros e Vetores

**Vetores como ponteiros**

- Vamos ver agora um dos usos mais importantes dos ponteiros: a varredura sequencial de uma matriz. Quando temos que varrer todos os elementos de uma matriz de uma forma sequencial, podemos usar um ponteiro, o qual vamos incrementando. Qual a vantagem? Considere o seguinte programa para zerar uma matriz:

```
int main () {
float matrx [50][50];
int i,j;
for (i=0;i<50;i++)
    for (j=0;j<50;j++)
          matrx[i][j]=0.0;
return(0);
}
```

```
int main () {
float matrx [50][50];
float *p; int count;
p=matrx[0];
for(count=0;count<2500;count++){
  *p=0.0;
   p++;
}
return(0);
}
```



# Ponteiros e Vetores

**Vetores como ponteiros**

- No primeiro programa, *cada* vez que se faz **matrx[i][j]** o programa tem que calcular o deslocamento para dar ao ponteiro. Ou seja, o programa tem que calcular 2500 deslocamentos.
- No segundo programa o único cálculo que deve ser feito é o de um incremento de ponteiro. Fazer 2500 incrementos em um ponteiro é muito mais rápido que calcular 2500 deslocamentos completos.
- Há uma diferença entre o nome de um vetor e um ponteiro que deve ser frisada: um ponteiro é uma variável, mas o nome de um vetor não é uma variável. Isto significa, que não se consegue alterar o endereço que é apontado pelo "nome do vetor". Seja:

```
int vetor[10];
int *ponteiro, i;
ponteiro = &i; /* as operacoes a seguir sao invalidas */
vetor = vetor + 2; /* ERRADO: vetor nao e' variavel*/
vetor++;           /* ERRADO: vetor nao e' variavel */
vetor = ponteiro;  /* ERRADO: vetor nao e' variavel */
```

## Ponteiros e Vetores

**Vetores como ponteiros**

- Teste as operações acima no seu compilador. Ele dará uma mensagem de erro. Alguns compiladores dirão que vetor não é um Lvalue. Lvalue, significa "Left value", um símbolo que pode ser colocado do lado esquerdo de uma expressão de atribuição, isto é, uma variável. Outros compiladores dirão que tem-se "incompatible types in assignment", tipos incompatíveis em uma atribuição.
- /* as operacoes abaixo sao validas */

```
ponteiro = vetor;  /* CERTO: ponteiro e' variavel */
ponteiro = vetor+2;  /* CERTO: ponteiro e' variavel */
```

- O que você aprendeu nesta seção é de suma importância. Não siga adiante antes de entendê- la bem.



## Ponteiros e Vetores

**Ponteiros como vetores**

- Sabemos agora que, na verdade, o nome de um vetor é um ponteiro constante. Sabemos também que podemos indexar o nome de um vetor.
- Como consequência podemos também indexar um ponteiro qualquer. O programa mostrado a seguir funciona perfeitamente:

```
#include <stdio.h>
int main () {
int matrx [10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
int *p;
p=matrx;
printf ("O terceiro elemento do vetor e: %d",p[2]);
return(0); }
```

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

# Ponteiros e Vetores

**Strings**

- Seguindo o raciocínio acima, nomes de strings, são do tipo **char***. Isto nos permite escrever a nossa função **StrCpy()**, que funcionará de forma semelhante à função strcpy() da biblioteca:

```
#include <stdio.h>
void StrCpy (char *destino,char *origem) {
while(*origem) {
    *destino=*origem;
    origem++;
    destino++;  }
*destino='\0';
}
```

```
int main () {
char str1[100],str2[100],str3[100];
printf ("Entre com uma string: ");
gets (str1);
StrCpy (str2,str1);
StrCpy (str3,"Voce digitou a string ");
printf ("\n\n%s%s",str3,str2);
return(0); }
```

- Observe que podemos passar ponteiros como argumentos de funções. Na verdade é assim que funções como **gets()** e strcpy() funcionam. Passando o ponteiro você possibilita à função *alterar* o conteúdo das strings. Você já estava passando os ponteiros e não sabia.
- No comando **while (*origem)** estamos usando o fato de que a string termina com '\0' como critério de parada.
- Quando fazemos **origem++** e **destino++** o leitor poderia argumentar que estamos alterando o valor do ponteiro-base da string, contradizendo o que recomendei que se deveria fazer, no final de uma seção anterior. O que o leitor talvez não saiba ainda (e que será estudado em detalhe mais adiante) é que, no C, são passados para as funções *cópias* dos argumentos. Desta maneira, quando alteramos o ponteiro **origem** na função **StrCpy()** o ponteiro **str2** permanece inalterado na função **main()**.



# Ponteiros e Vetores

Endereços de elementos de vetores

- Nesta seção vamos apenas ressaltar que a notação

*&nome_da_variável[índice]*

- é válida e retorna o endereço do ponto do vetor indexado por índice. Isto seria equivalente a nome_da_variável + indice.
- É interessante notar que, como consequência, o ponteiro nome_da_variável tem o endereço &nome_da_variável[0], que indica onde na memória está guardado o valor do primeiro elemento do vetor.

## Ponteiros e Vetores

Vetores de ponteiros

- Podemos construir vetores de ponteiros como declaramos vetores de qualquer outro tipo. Uma declaração de um vetor de ponteiros inteiros poderia ser:

```
int *pmatrx [10];
```

- No caso acima, pmatrx é um vetor que armazena 10 ponteiros para inteiros.



## Funções

## Função

- São as estruturas que permitem ao usuário separar seus programas em blocos
- Para fazermos programas grandes e complexos temos de construí-los bloco a bloco.
- Forma geral:

*tipo_de_retorno nome_da_função (declaração_de_parâmetros)*
*{*
*corpo_da_função*
*}*

- O tipo-de-retorno é o tipo de variável que a função vai retornar. O default é o tipo **int**.



## Função

- Forma geral:

*tipo_de_retorno nome_da_função (declaração_de_parâmetros)*
*{*
*corpo_da_função*
*}*

- A declaração de parâmetros é uma lista com a seguinte forma geral:

*tipo nome1, tipo nome2, ... , tipo nomeN*

- O tipo deve ser especificado para cada uma das N variáveis de entrada.
- É na declaração de parâmetros que informamos ao compilador quais serão as entradas da função
- O corpo da função é a sua alma. É nele que as entradas são processadas, saídas são geradas ou outras coisas são feitas

## O Comando return

- Forma geral:

*return valor_de_retorno;* ou *return;*

- Digamos que uma função está sendo executada. Quando se chega a uma declaração **return** a função é encerrada imediatamente e, se o valor de retorno é informado, a função retorna este valor.
- É importante lembrar que o valor de retorno fornecido tem que ser compatível com o tipo de retorno declarado para a função.
- Uma função pode ter mais de uma declaração **return**. Isto se torna claro quando pensamos que a função é terminada quando o programa chega à primeira declaração **return**.



## O Comando return

- Exemplo de uso do **return**:

```
#include <stdio.h>
int EPar (int a) {
    if (a%2!=0)          /* Verifica se a e divisivel por dois */
          return 0;       /* Retorna 0 se nao for divisivel */
    else
          return 1;       /* Retorna 1 se for divisivel */
}
int main () {
    int num;
    printf ("Entre com numero: ");
    scanf ("%d",&num);
    if (EPar(num)==1)
          printf ("\n\nO numero e par.\n");
    else
          printf ("\n\nO numero e impar.\n");
    return 0;
}
```

## O Comando return

- É importante notar que, como as funções retornam valores, podemos aproveitá-los para fazer atribuições, ou mesmo para que estes valores participem de expressões. Mas *não* podemos fazer:

```
func(a,b)=x;   /* Errado! */
```

- Fato importante: se uma função retorna um valor você *e* se você não fizer nada com o valor ele será descartado.

- Por exemplo, a função **printf()** retorna um inteiro que nós nunca usamos para nada. Ele é descartado.



## Funções

AUTO AVALIAÇÃO

Veja como você está:

Escreva a função 'EDivisivel(int a, int b)' (tome como base EPar(int a)). A função deverá retornar 1 se o resto da divisão de a por b for zero. Caso contrário, a função deverá retornar zero.

## Protótipo de funções

- Até agora, nos exemplos apresentados, escrevemos as funções antes de escrevermos a função **main()**. Isto é, as funções estão fisicamente antes da função **main()**.
- Isto foi feito por uma razão. Imagine-se na pele do compilador. Se você fosse compilar a função **main()**, onde são chamadas as funções, você teria que saber com antecedência quais são os tipos de retorno e quais são os parâmetros das funções para que você pudesse gerar o código corretamente.
- Foi por isto as funções foram colocadas antes da função **main()**: quando o compilador chegasse à função **main()** ele já teria compilado as funções e já saberia seus formatos.



## Protótipo de funções

- Mas, muitas vezes, não poderemos nos dar ao luxo de escrever nesta ordem. Muitas vezes teremos o nosso programa espalhado por vários arquivos. Ou seja, estaremos chamando funções em um arquivo que serão compiladas em outro arquivo. Como manter a coerência?
- A solução são os protótipos de funções. Protótipos são nada mais, nada menos, que declarações de funções. Isto é, você declara uma função que irá usar. O compilador toma então conhecimento do formato daquela função antes de compilá-la. O código correto será então gerado.

## Protótipo de funções

- Um protótipo tem o seguinte formato:

*tipo_de_retorno nome_da_função (declaração_de_parâmetros);*

- onde o tipo-de-retorno, o nome-da-função e a declaração-de-parâmetros são os mesmos que você pretende usar quando realmente escrever a função.

- Vamos implementar agora um dos exemplos da seção anterior com algumas alterações e com protótipos:



## Protótipo de funções

```
#include <stdio.h>
float Square (float a);
int main () {
      float num;
      printf ("Entre com um numero: ");
      scanf ("%f",&num);
      num=Square(num);
      printf ("\n\nO seu quadrado vale: %f\n",num);
      return 0;
}

float Square (float a) {
    return (a*a);
}
```

- Observe que a função **Square()** está colocada depois de **main()**, mas o seu protótipo está antes. Sem isto este programa não funcionaria corretamente.

- Os protótipos não só ajudam o compilador. Eles ajudam a você também: usando protótipos, o compilador evita erros, não deixando que o programador use funções com os parâmetros errados e com o tipo de retorno errado, o que é uma grande ajuda!

## O tipo void

- Em inglês, **void** quer dizer vazio e é isto mesmo que o **void** é.
- Void nos permite fazer funções que não retornam nada e funções que não têm parâmetros!
- Podemos agora escrever o protótipo de uma função que não retorna nada:

```
void nome_da_função (declaração_de_parâmetros);
```

- Numa função, como a acima, não temos valor de retorno na declaração return. Aliás, neste caso, o comando return não é necessário na função.
- Podemos, também, fazer funções que não têm parâmetros:

```
tipo_de_retorno nome_da_função (void);
```

- ou, ainda, que não tem parâmetros e não retornam nada:

```
void nome_da_função (void);
```



## O tipo void

- Um exemplo de funções que usam o tipo **void**:

```
#include <stdio.h>
void Mensagem (void);
int main () {
Mensagem();
printf ("\tDiga de novo:\n");
Mensagem();
return 0;
}

void Mensagem (void) {
    printf ("Ola! Eu estou vivo.\n");
}
```

- Se quisermos que a função retorne algo, devemos usar a declaração return. Se não quisermos, basta declarar a função como tendo tipo-de-retorno **void**.

## O tipo void

- *O compilador acha que a função **main()** deve retornar um inteiro. Isto pode ser interessante se quisermos que o sistema operacional receba um valor de retorno da função **main()**.*
- *Convenção para retorno de programas:*
- *retornar zero, significa que ele terminou normalmente*
- *retornar um valor diferente de zero, significa que o programa teve um término anormal.*

- *Se não estivermos interessados neste tipo de coisa, basta declarar a função main como retornando **void**.*
- *As duas funções **main()** abaixo são válidas:*

```
main (void) { .... return 0; }
void main (void) { .... }
```



## Arquivos cabeçalho

- *São aqueles que temos mandado o compilador incluir no início de nossos exemplos e que sempre terminam em **.h**.*
- *A extensão **.h** vem de **header** (cabeçalho em inglês). Já vimos exemplos como **stdio.h**, **conio.h**, **string.h**. Estes arquivos, na verdade, não possuem os códigos completos das funções.*
- *Eles só contêm protótipos de funções. É o que basta. O compilador lê estes protótipos e, baseado nas informações lá contidas, gera o código correto.*
- *O corpo das funções cujos protótipos estão no arquivo-cabeçalho, no caso das funções do próprio C, já estão compiladas e normalmente são incluídas no programa no instante da "linkagem".*
- *Este é o instante em que todas as referências a funções cujos códigos não estão nos nossos arquivos fontes são resolvidas, buscando este código nos arquivos de bibliotecas. .*

## Arquivos cabeçalho

- Se você criar algumas funções que queira aproveitar em vários programas futuros, ou módulos de programas, você pode escrever arquivos-cabeçalhos e incluí-los também

- Suponha que a função 'int EPar(int a)' seja importante em vários programas, e desejemos declará-la num módulo separado. No arquivo de cabeçalho chamado por exemplo de 'funcao.h' teremos a seguinte declaração:

```
int EPar(int a);
```

- O código da função será escrito num arquivo a parte. Vamos chamá-lo de 'funcao.c'. Neste arquivo teremos a definição da função:

```
int EPar (int a) {
    if (a%2)            /* Verifica se a e divisivel por dois */
        return 0;
    else
        return 1;
}
```



## Arquivos cabeçalho

- Por fim, no arquivo do programa principal teremos o programa principal. Vamos chamar este arquivo aqui de 'princip.c'.

```
#include <stdio.h>
#include "funcao.h"
void main () {
    int num;
    printf ("Entre com numero: ");
    scanf ("%d",&num);
    if (EPar(num))  printf ("\n\nO numero e par.\n");
    else   printf ("\n\nO numero e impar.\n");
}
```

## Arquivos cabeçalho

- Este programa poderia ser compilado da seguinte forma:
  - Criar Novo Projeto (file, new, project)
  - Criar arquivo princip.c
  - Criar arquivo funcao.h (file, new, source file. Mudar extensão na hora de salvar para .h)
  - Criar funcao.c (file, new, source file. Mudar extensão na hora de salvar para .c)
  - F9 para compilar e rodar



## Arquivos cabeçalho

AUTO AVALIAÇÃO

Veja como você está:

Escreva um programa que faça uso da função EDivisivel(int a, int b), criada no slide anterior. Organize o seu programa em três arquivos: o arquivo prog.c , conterá o programa principal; o arquivo func.c conterá a função; o arquivo func.h conterá o protótipo da função. Compile os arquivos e gere o executável a partir deles.

# Tipos de Dados Definidos Pelo Usuário



## Estruturas

- Uma estrutura agrupa várias variáveis numa só.
- Funciona como uma ficha pessoal que tenha nome, telefone e endereço.
- A ficha seria uma estrutura. A estrutura, então, serve para agrupar um conjunto de dados não similares, formando um novo tipo de dados.

## Criando estruturas

- Para se criar uma estrutura usa-se o comando struct.
- Sua forma geral é:

```
struct nome_do_tipo_da_estrutura
    {
    tipo_1 nome_1;
    tipo_2 nome_2;
    ...
    tipo_n nome_n;
    } variáveis_estrutura;
```

- O nome_do_tipo_da_estrutura é o nome para a estrutura.
- As variáveis_estrutura são opcionais e seriam nomes de variáveis que o usuário já estaria declarando e que seriam do tipo nome_do_tipo_da_estrutura. Um primeiro exemplo:
- 
```
struct est{
    int i;
    float f;
} a, b;
```



## Criando estruturas

- Um primeiro exemplo:

```
struct est{
    int i;
    float f;
} a, b;
```

- *Neste caso, est é uma estrutura com dois campos, i e f.*
- *Foram também declaradas duas variáveis, a e b que são do tipo da estrutura, isto é, a possui os campos i e f, o mesmo acontecendo com b.*

# Criando estruturas

- *Vamos criar uma estrutura de endereço:*

```
struct tipo_endereco {
    char rua [50];
    int numero;
    char bairro [20];
    char cidade [30];
    char sigla_estado [3];
    long int CEP; };
```

- Vamos agora criar uma estrutura chamada ficha_pessoal com os dados pessoais de uma pessoa:

```
struct ficha_pessoal {
    char nome [50];
    long int telefone;
    struct tipo_endereco endereco;
};
```

- Vemos, pelos exemplos acima, que uma estrutura pode fazer parte de outra (a struct tipo_endereco é usada pela struct ficha_pessoal).



# Usando estruturas

- Vamos agora utilizar as estruturas declaradas na seção anterior para escrever um programa que preencha uma ficha.

```
#include <stdio.h>
#include <string.h>
struct tipo_endereco {
    char rua [50];
    int numero;
    char bairro [20];
    char cidade [30];
    char sigla_estado [3];
    long int CEP;
};

struct ficha_pessoal {
    char nome [50];
    long int telefone;
    struct tipo_endereco endereco;
};

main (void) {
    struct ficha_pessoal ficha;
    strcpy (ficha.nome,"Luiz Osvaldo Silva");
    ficha.telefone=4921234;
    strcpy (ficha.endereco.rua,"Rua das Flores");
    ficha.endereco.numero=10;
    strcpy (ficha.endereco.bairro,"Cidade Velha");
    strcpy (ficha.endereco.cidade,"Belo Horizonte");
    strcpy (ficha.endereco.sigla_estado,"MG");
    ficha.endereco.CEP=31340230;
    return 0;
}
```

- O programa declara uma variável ficha do tipo **ficha_pessoal** e preenche os seus dados
- Para acessar um elemento de uma estrutura: basta usar o ponto (.).
- Assim, para acessar o campo telefone de ficha, escrevemos:

```
ficha.telefone = 4921234;
```

- Como a struct ficha pessoal possui um campo, endereco, que também é uma struct, podemos fazer acesso aos campos desta struct interna da seguinte maneira:

```
ficha.endereco.numero = 10;
ficha.endereco.CEP=31340230;
```

- Desta forma, estamos acessando, primeiramente, o campo endereco da struct ficha e, dentro deste campo, estamos acessando o campo numero e o campo CEP.

## Matrizes de estruturas

- Um estrutura é como qualquer outro tipo de dado no C.
- Podemos, portanto, criar matrizes de estruturas. Vamos ver como ficaria a declaração de um vetor de 100 fichas pessoais:

*struct ficha_pessoal fichas [100];*

- Poderíamos então acessar a segunda letra da sigla de estado da décima terceira ficha fazendo:

*fichas[12].endereco.sigla_estado[1];*

- Analise atentamente como isto está sendo feito ...



## Estruturas

AUTO AVALIAÇÃO

Veja como você está:

Escreva um programa fazendo o uso de struct's. Você deverá criar uma struct chamada Ponto, contendo apenas a posição x e y (inteiros) do ponto. Declare 2 pontos, leia a posição (coordenadas x e y) de cada um e calcule a distância entre eles. Apresente no final a distância entre os dois pontos.

## Atribuindo estruturas

- Podemos atribuir duas estruturas que sejam do *mesmo* tipo. O C irá, neste caso, copiar uma estrutura, campo por campo, na outra. Veja o programa abaixo:

```
struct est1 {
        int i;
        float f;
};
void main() {
     struct est1 primeira, segunda;  /* Declara primeira e segunda como
    structs do tipo est1 */
    primeira.i = 10;
    primeira.f = 3.1415;
    segunda = primeira; /* A segunda struct e' agora igual a primeira */
    printf(" Os valores armazenasdos na segunda struct sao :  %d  e  %f ",
    segunda.i , segunda.f);
    }
```



## Atribuindo estruturas

- São declaradas duas estruturas do tipo *est1,* uma chamada *primeira* e outra chamada *segunda*.
- Atribuem-se valores aos dois campos da struct primeira.
- Os valores de *primeira* são copiados em *segunda* apenas com a expressão de atribuição:

```
segunda = primeira;
```

# Atribuindo estruturas

- Porém, devemos tomar cuidado na atribuição de structs que contenham campos ponteiros. Veja abaixo:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct tipo_end {
char *rua; /* A struct possui um campo que é um
     ponteiro */
int numero; };
void main() {
struct tipo_end end1, end2; char buffer[50];
printf("\nEntre o nome da rua:");
gets(buffer); /* Le o nome da rua em uma string
     de buffer */
end1.rua = (char *) malloc((strlen(buffer)+1)*sizeof(char));
     /* Aloca a quantidade de memoria suficiente para
     armazenar a string */
 strcpy(end1.rua, buffer); /* Copia a string */
printf("\nEntre o numero:");
scanf("%d", &end1.numero);
end2 = end1; /* ERRADO end2.rua e end1.rua estao
     apontando para a mesma regiao de memoria */
printf("Depois da atribuicao:\n Endereco em end1 %s %d
     \n Endereco em end2 %s %d",
     end1.rua,end1.numero,end2.rua, end2.numero);
strcpy(end2.rua, "Rua Mesquita"); /* Uma modificacao na
     memoria apontada por end2.rua causara' a modificacao
     do que e' apontado por end1.rua, o que, esta' errado !!!
     */
end2.numero = 1100; /* Nesta atribuicao nao ha problemas
     */
printf(" \n\nApos modificar o endereco em end2:\n
     Endereco em end1 %s %d \n Endereco em end2 %s
     %d", end1.rua, end1.numero, end2.rua, end2.numero); }
```



# Atribuindo estruturas

- Neste programa há um erro grave, pois ao se fazer a atribuição end2 = end1, o campo rua de end2 estará apontando para a mesma posição de memória que o campo rua de end1. Assim, ao se modificar o conteúdo apontado por end2.rua estaremos também modificando o conteúdo apontado por end1.rua !!!

## Passando estruturas para funções

- Podemos também passar para uma função uma estrutura inteira. Veja a seguinte função:

*void PreencheFicha (struct ficha_pessoal ficha) { ... }*

- Devemos observar que, como em qualquer outra função no C, a passagem da estrutura é feita por valor.

- A estrutura que está sendo passada, vai ser copiada, campo por campo, em uma variável local da função PreencheFicha.
- Isto significa que alterações na estrutura dentro da função não terão efeito na variável fora da função. Podemos contornar este pormenor usando ponteiros e passando para a função um ponteiro para a estrutura.



## Ponteiro de estruturas

- Vamos ver como poderia ser declarado um ponteiro para as estruturas de ficha que estamos usando:

*struct ficha_pessoal *p;*

- Os ponteiros para uma estrutura funcionam como os ponteiros para qualquer outro tipo de dados no C.
- Para usá-lo, há duas possibilidades. A primeira é apontá-lo para uma variável struct já existente, da seguinte maneira:

```
struct ficha_pessoal ficha;
struct ficha_pessoal *p;
p  = &ficha;
```

## Ponteiro de estruturas

- A segunda é alocando memória para ficha_pessoal usando, por exemplo, malloc():

```
#include <stdlib.h>
main() {
   struct ficha_pessoal *p;
   int a = 10; /* Faremos a alocacao dinamica de 10 fichas pessoais */
   p = (struct ficha_pessoal *) malloc (a * sizeof(struct
   ficha_pessoal));
   p[0].telefone = 3443768; /* Exemplo de acesso ao campo telefone da
   primeira ficha apontada por p */
   free(p);
   }
```



## Ponteiro de estruturas

- Se apontarmos o ponteiro **p** para uma estrutura qualquer (como fizemos em p = &ficha; ) e quisermos acessar um elemento da estrutura poderíamos fazer:

  *(*p).nome*

- Os parênteses são necessários, porque o operador . tem precedência maior que o operador * .

- Porém, este formato não é muito usado. O que é comum de se fazer é acessar o elemento **nome** através do operador seta, que é formado por um sinal de "menos" (-) seguido por um sinal de "maior que" (>), isto é: **->** .

  *p->nome*

- A declaração acima é muito mais fácil e concisa. Para acessarmos o elemento **CEP** dentro de **endereco** faríamos:

  *p->endereco.CEP*

# Estruturas

AUTO AVALIAÇÃO
Veja como você está:

Seja a seguinte struct que é utilizada para descrever os produtos que estão no estoque de uma loja :

```
struct Produto {
        char nome[30];    /* Nome do produto */
        int codigo;       /* Codigo do produto */
        double  preco;    /* Preco do produto */
};
```

a) Escreva uma instrução que declare uma matriz de Produto com 10 itens de produtos;

b) Atribua os valores "Pe de Moleque", 13205 e R$0,20 aos membros da posição 0 e os valores "Cocada Baiana", 15202 e R$0,50 aos membros da posição 1 da matriz anterior;

c) Faça as mudanças que forem necessárias para usar um ponteiro para Produto ao invés de uma matriz de Produtos. Faça a alocação de memória de forma que se possa armazenar 10 produtos na área de memória apontada por este ponteiro e refaça as atribuições da letra b;

d) Escreva as instruções para imprimir os campos que foram atribuídos na letra c.



# Manipulando Arquivos

## Abrindo e Fechando um Arquivo

- Sistema de entrada e saída do C é composto de funções que trabalham com o conceito de "ponteiro de arquivo"
- Os protótipos desta funções estão definidos em *stdio.h*
- Podemos declarar um ponteiro de arquivo da seguinte forma:

```
FILE *p;
```

- É usando este tipo que podemos manipular arquivos no C



## Strings

```
#include <stdio.h>
int main ()
{
      char string[100];
      printf ("Digite uma string: ");
      gets (string);
      printf ("\n\nVoce digitou %s",string);
      return(0);
}
```

- No programa acima, tamanho máximo da string que você pode entrar é uma string de 99 caracteres.
- Se você entrar com uma string de comprimento maior, o programa irá aceitar, mas os resultados podem ser desastrosos. Veremos porque posteriormente.

## Abrindo e Fechando um Arquivo

- fopen, Função utilizada para abertura de arquivo
- Protótipo:

```
FILE  *fopen (char *nome_do_arquivo, char *modo);
```

- nome_do_arquivo, Determina qual arquivo deverá ser aberto
- Este nome deve ser válido no sistema operacional que estiver sendo utilizado



## Abrindo e Fechando um Arquivo

- *modo, modo de abertura diz à função **fopen()** que tipo de uso você vai fazer do arquivo
- A tabela seguinte mostra os valores de modo válidos:

| Modo | Significado |
|---|---|
| "r" | Abre um arquivo texto para leitura. O arquivo deve existir antes de ser aberto. |
| "w" | Abrir um arquivo texto para gravação. Se o arquivo não existir, ele será criado. Se já existir, o conteúdo anterior será destruído. |
| "a" | Abrir um arquivo texto para gravação. Os dados serão adicionados no fim do arquivo ("append"), se ele já existir, ou um novo arquivo será criado, no caso de arquivo não existente anteriormente. |
| "r+" | Abre um arquivo texto para leitura e gravação. O arquivo deve existir e pode ser modificado. |
| "w+" | Cria um arquivo texto para leitura e gravação. Se o arquivo existir, o conteúdo anterior será destruído. Se não existir, será criado. |
| "a+" | Abre um arquivo texto para gravação e leitura. Os dados serão adicionados no fim do arquivo se ele já existir, ou um novo arquivo será criado, no caso de arquivo não existente anteriormente. |
| "rb" | Abre um arquivo binário para leitura. Igual ao modo "r" anterior, só que o arquivo é binário. |
| "wb" | Cria um arquivo binário para escrita, como no modo "w" anterior, só que o arquivo é binário. |
| "ab" | Acrescenta dados binários no fim do arquivo, como no modo "a" anterior, só que o arquivo é binário. |
| "r+b" | Abre um arquivo binário para leitura e escrita. O mesmo que "r+" acima, só que o arquivo é binário. |
| "w+b" | Cria um arquivo binário para leitura e escrita. O mesmo que "w+" acima, só que o arquivo é binário. |
| "a+b" | Acrescenta dados ou cria uma arquivo binário para leitura e escrita. O mesmo que "a+" acima, só que o arquivo é binário |



# Abrindo e Fechando um Arquivo

- Poderíamos então, para abrir um arquivo binário para escrita, escrever:

```
FILE *fp;    /* Declaração da estrutura */
/* exemplo.bin deve estar localizado no diretório corrente */
fp = fopen ("exemplo.bin","wb");
if (fp == NULL)   /*  testa se o arquivo foi aberto com sucesso */
   printf ("Erro na abertura do arquivo.");
```

- No caso de um erro a função **fopen()** retorna um ponteiro nulo (**NULL**)

## A função exit()

- O protótipo desta função está definido em *stdio.h*

```
void exit (int codigo_de_retorno);
```

- A função exit() faz com que o programa termine e retorne para o sistema operacional o ***codigo_de_retorno***
- A convenção mais usada é que um programa retorne:
  - ***zero*** no caso de um término normal
  - ***um número não nulo*** no caso de ter ocorrido um problema
- Pode ser usada em casos como alocação dinâmica e abertura de arquivos
- Se o programa não conseguir a memória necessária ou abrir o arquivo, a melhor saída pode ser terminar a execução do programa



## A função exit()

- Poderíamos reescrever o exemplo da seção anterior usando agora o exit() para garantir que o programa não deixará de abrir o arquivo:

```
#include <stdio.h>
#include <stdlib.h> /* Para a função exit() */
int main (void) {
  FILE *fp;
  ...
  fp=fopen ("exemplo.bin","wb");
  if (fp ==  NULL) {
     printf ("Erro na abertura do arquivo. Fim de programa.");
     exit (1);
  }
  ...
  return 0;
}
```

## Abrindo e Fechando um Arquivo

- Fclose: Função utilizada para fechar um arquivo:
- Protótipo: int fclose (FILE *fp);
- *fp, determina o arquivo a ser fechado. A função retorna zero no caso de sucesso
- Fechar um arquivo faz com que qualquer caracter que tenha permanecido no "buffer" associado ao fluxo de saída seja gravado. *Buffer???*
- A função *exit()* fecha todos os arquivos que um programa tiver aberto



## Lendo e escrevendo caracteres em arquivos

- fputc, Função que escreve um caractere no arquivo
- Protótipo: int fputc (int ch, FILE *fp);
- ch, determina o caracter que será gravado no arquivo
- fp, ponteiro para FILE que está associado ao arquivo aonde será feita a gravação
- A função retorna o caracter ch se a gravação for realizada com sucesso ou EOF caso ocorra um erro no momento da gravação.
- EOF ("End of file") indica o fim de um arquivo

**Slide 77**

**A1** Quando você envia caracteres para serem gravados em um arquivo, estes caracteres são armazenados temporariamente em uma área de memória (o "buffer") em vez de serem escritos em disco imediatamente. Quando o "buffer" estiver cheio, seu conteúdo é escrito no disco de uma vez. A razão para se fazer isto tem a ver com a eficiência nas leituras e gravações de arquivos. Se, para cada caracter que fossemos gravar, tivéssemos que posicionar a cabeça de gravação em um ponto específico do disco, apenas para gravar aquele caracter, as gravações seriam muito lentas. Assim estas gravações só serão efetuadas quando houver um volume razoável de informações a serem gravadas ou quando o arquivo for fechado.
Administrator; 18/2/2005

## Lendo e escrevendo caracteres em arquivos

- O programa a seguir lê uma string do teclado e escreve-a, caractere por caractere em um arquivo em disco:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  FILE *fp;
  char string[100];  int i;
  fp = fopen("arquivo.txt","w"); /* Arquivo ASCII, para escrita */
  if( fp == NULL) {
    printf( "Erro na abertura do arquivo");
    exit(1);
  }
  printf("Entre com a string a ser gravada no arquivo:");
  gets(string);
  for(i=0; string[i]; i++) fputc(string[i], fp);/* Grava a string, caractere a
   caractere */
  fclose(fp);
  return 0;
}
```



## Lendo e escrevendo caracteres em arquivos

- getc, Função que lê e retorna o próximo caractere do arquivo como um inteiro
- Protótipo: int getc (FILE *fp);
- fp, ponteiro para FILE que está associado ao arquivo donde o caracter será lido
- Caso ocorra um erro, getc retorna EOF. A função também retorna EOF quando o final do arquivo é alcançado

## Lendo e escrevendo caracteres em arquivos

- feof, verifica se um arquivo chegou ao fim.
- Protótipo: int feof (FILE *fp);
- fp, determina o arquivo a ser verificado
- A função retorna não-zero se o arquivo chegou a EOF, caso contrário retorna zero
- Outra forma de se verificar se o final do arquivo foi atingido é comparar o caractere lido por getc com *EOF*



## Lendo e escrevendo caracteres em arquivos

- O programa a seguir abre um arquivo já existente e o lê, caracter por caracter, até que o final do arquivo seja atingido. Os caracteres lidos são apresentados na tela:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   FILE *fp;
   char c;
   fp = fopen("arquivo.txt","r");   /* Arquivo ASCII, para leitura */
   if(!fp) {
      printf( "Erro na abertura do arquivo");
      exit(1);
   }
   while((c = fgetc(fp) ) != EOF)     /* Enquanto não chegar ao final do arquivo */
      printf("%c", c);    /* imprime o caracter lido */
   fclose(fp);
   return 0;
}
```

## Lendo e escrevendo caracteres em arquivos

AUTO AVALIAÇÃO

Veja como você está: escreva um programa que abra um arquivo texto e conte o número de caracteres presentes nele. Imprima o número de caracteres na tela.



## Outros Comandos de Acesso a Arquivos

- fgets, Função lê uma string do arquivo
- Protótipo:

```
char *fgets (char *str, int tamanho,FILE *fp);
```

- str, string a ser lida
- tamanho, o limite máximo de caracteres a serem lidos
- fp, ponteiro para FILE que está associado ao arquivo de onde a string será lida.
- A função lê a string até que um caracter de nova linha seja lido, *tamanho-1* caracteres sejam lidos ou o final do arquivo seja alcançado

## Outros Comandos de Acesso a Arquivos

- Protótipo:

  char *fgets (char *str, int tamanho,FILE *fp);

- '\n' caso for lido, fará parte da string, o que não acontece com gets
- A string resultante sempre terminará com '\0' (por isto somente *tamanho-1* caracteres, no máximo, serão lidos)
- Retorna EOF se um erro ocorrer ou um valor não-negativo caso a gravação tenha sido realizada com sucesso.



## Outros Comandos de Acesso a Arquivos

- fputs, Função escreve uma string num arquivo
- Protótipo:

  int fputs (char *str, FILE *fp);
- str, string a ser escrita
- fp, ponteiro para FILE, que está associado ao arquivo de onde a string será escrita
- '\n' que termina a string não é gravada no arquivo
- Retorna EOF se um erro ocorrer e um valor não-negativo caso a gravação tenha sido realizada com sucesso

## Outros Comandos de Acesso a Arquivos

- Ferror, A função retorna zero, se nenhum erro ocorreu e um número diferente de zero se algum erro ocorreu durante o acesso ao arquivo.
- Protótipo de ferror:

```
int ferror (FILE *fp);
```

- Na maioria dos casos, se um arquivo pode ser aberto, ele pode ser lido ou gravado. Porém, existem situações em que isto não ocorre. Por exemplo, pode acabar o espaço em disco enquanto gravamos, ou o disco pode estar com problemas e não conseguimos ler, etc.
- Uma função que pode ser usada em conjunto com **ferror()** é a função **perror()** (print error), cujo argumento é uma string que normalmente indica em que parte do programa o problema ocorreu



## Outros Comandos de Acesso a Arquivos

- No exemplo a seguir, fazemos uso de ferror, perror e fputs:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
        FILE *pf;
        char string[100];
        if  (   (pf = fopen("arquivo.txt","w") ) ==  NULL)
        {
           printf("\nNao consigo abrir o arquivo ! ");
           exit(1);
        }
        do
        {
           printf("\nDigite uma nova string. Para terminar, digite <enter>: ");
           gets(string);
           fputs(string, pf);
           putc('\n', pf);
           if(ferror(pf))
           {
              perror("Erro na gravacao");
              fclose(pf);
              exit(1);
           }
        } while (strlen(string) > 0);
        fclose(pf);
}
```