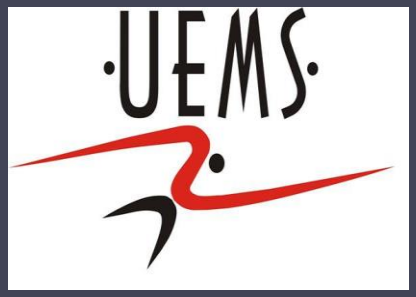


PCI- Funções e Procedimentos

Profa. Mercedes Gonzales Márquez

Funções

- Um ponto chave na resolução de um problema complexo é conseguir quebrá-lo em subproblemas menores, fáceis de serem entendidos e administrados.

- Problemas simples: Resolução direta.
- Problemas mais complexos: Quebrar em problemas menores. Cada problema menor pode se tornar uma função.

Funções

- Evitar extensos blocos de programa e portanto difíceis de ler e entender.
- Dividir o programa em partes que possam ser logicamente compreendidas de forma isolada.
- Reaproveitar blocos de programa já construídos (por você ou por outros programadores), minimizando erros e facilitando alterações.

Definição de uma função

- Uma função é definida da seguinte forma:

```
tipo nome(tipo parâmetro1, ..., tipo parâmetroN) {
        comandos;
return valor de retorno;
}
```

- Toda função deve ter um tipo válido em C, seja de tipos pré-definidos (int, char, float) ou de tipos definidos pelo usuário. Esse tipo determina qual será o tipo de seu valor de retorno.
- Os parâmetros são variáveis que são inicializadas com valores indicados durante a invocação da função.
- O comando return devolve para o invocador da função o resultado da execução desta.

Exemplos de função

Exemplo 1. Função que determina o fatorial de um número inteiro n.

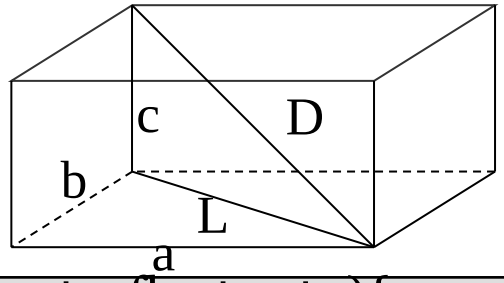
```
int fatorial(int x){
    int fat=1, i;
    for (i=x; i > 1; i--)
            fat = fat * i;
    return(fat);
}
```

Exemplo 2. Função que, dados os catetos de um triângulo retângulo, determine a sua hipotenusa.

```
float hipotenusa(float cat1, float cat2){
    float hip;
    hip =sqrt(cat1*cat1+cat2*cat2)
    return(hip)
}
```

Exemplos de função

Exemplo 3. Faça um programa que use a função hipotenusa para calcular o valor da diagonal D do seguinte paralelepípedo, após a leitura dos seus lados a, b e c.




```
float hipotenusa(float cat1, float cat2){
   float hip;
   hip =sqrt(cat1*cat1+cat2*cat2);
   return(hip);
}
int main(){
      float a,b,c,d;
      printf (“Informe os lados a,b e c do paralelepipedo”);
      scanf (“%f %f %f”,&a,&b,&c);
      d=hipotenusa(hipotenusa(a,b),c);
      printf (“O valor da diagonal eh: %.2f”,d);
}
```

Exemplos de função

Exemplo 4. Suponha um programa para calcular o numero de combinações de n eventos em conjuntos de p eventos, p<=n.

$$C(n,p) = \frac{n!}{p!(n-p)!}$$

- Sem o conceito de função, teríamos que repetir três vezes as instruções para calculo do fatorial de um numero x.

- Com o conceito de função, precisamos apenas escrever essas instruções uma única vez e substituir x por n, p, e (n-p) para saber o resultado de cada calculo fatorial.

Exemplos de função

```
#include <stdio.h>
int fatorial(int x){
    int fat=1, i;
    for (i=x; i > 1; i--)
            fat = fat * i;
    return(fat);
}
/* funcao principal */
int main(){
    int n,p,C;
    scanf("%d %d",&n,&p);
    if (p >= 0&&n >= 0&&p <= n){ /* chamada da funcao */
            C = fatorial(n)/(fatorial(p)*fatorial(n-p));
            printf("%d \n",C);
    }
}
```

Protótipo de uma função

- Até agora declaramos a função antes da *main()*. Mas, quando o código for extenso é conveniente deixarmos à mostra logo no começo a função main() e logo após declararmos as funções.
- Uma técnica usada é declarar **antes** da *main()* apenas o protótipo de cada função, e a função em si é declarada **após** a *main()*.
- O exemplo 4 usando protótipo será:

Exemplo de função com protótipo

```
#include <stdio.h>
int fatorial(int x); /* prototipo da funcao */
/* funcao principal */
int main(){
    int n,p,C;
    scanf("%d %d",&n,&p);
    if (p >= 0&&n >= 0&&p <= n){ /* chamada da funcao */
            C = fatorial(n)/(fatorial(p)*fatorial(n-p));
            printf("%d \n",C);
    }
}
int fatorial(int x){
    int fat=1, i;
    for (i=x; i > 1; i--)
            fat = fat * i;
    return(fat);
}
```

Variáveis locais e globais

- Uma variável é chamada **local** se ela foi declarada dentro de uma função. Nesse caso ela existe somente dentro da função, e após o término da execução desta, a variável deixa de existir. Variáveis parâmetros também são variáveis locais.

- Uma variável é chamada **global** se ela for declarada fora de qualquer função. Essa variável é visível em todas as funções. Qualquer função pode alterá-la e ela existe durante toda a execução do programa.

Passagem de parâmetros

- No exemplo 1, o valor de n na chamada fatorial(n) é passado para uma cópia x da variável n. Qualquer alteração em x não afeta o conteúdo de n no escopo da função principal. Dizemos então que o parâmetro e passado **por valor.**

Passagem de parâmetros

- Porém, pode acontecer de desejarmos alterar o conteúdo de uma ou mais variáveis no escopo da função principal. Neste caso, os parâmetros devem ser passados **por referência**.
- Ou seja, a função cria uma **cópia do endereço** da variável correspondente na função principal **em vez de uma cópia do seu conteúdo**. Qualquer alteração no conteúdo deste endereço é uma alteração direta no conteúdo da variável da função principal.

Passagem de parâmetros

No exemplo 4 foi requerido que p <= n. Caso forem informadas p e n, tal que não satisfaçam essa condição iremos trocar o conteúdo dessas variáveis para garantir essa condição. Usamos a seguinte função troca que é do tipo void ou também chamada de procedimento, a qual será explicada em breve.

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

Passagem de parâmetros

```
int main(){
    int n,p,C;
    scanf("%d %d",&n,&p);
    if (p > n)
            troca(&p,&n); /* passa os enderecos de p e de n */
    if (p >= 0&&n >= 0){
            C = fatorial(n)/(fatorial(p)*fatorial(n-p));
            printf("%d \n",C);
   }
}
```

Procedimentos ou funções tipo void

- O procedimento corresponde a uma função do tipo *void* (tipo de dado indefinido) que não possui o comando return devolvendo algum valor para a função chamadora.
- Por exemplo, a função abaixo imprime o número que for passado para ela como parâmetro:

```
void imprime (int numero) {
    printf ("Número %d\n", numero);
}
```

Procedimentos ou funções tipo void

```
# include <stdio.h>
void imprime (int numero) {
     printf ("Numero %d\n", numero);
}
int main () {
     int n,i;
     printf (“Informe um numero inteiro:”);
     scanf (“%d”,&n);
     printf (“Imprime os n primeiros multiplos de 10”);
     for (i=1;i<=n;i++)
          imprime (i*10);
}
```

Funções com vetores como parâmetros

- Um vetor passado como parâmetro é implicitamente passado por referência.
- o acesso aos elementos de v não precisa do modificador *
- Qualquer alteração de valor de um elemento de v é uma alteração no valor do vetor ‘original’. Exemplo: O seguinte procedimento lê um vetor de tamanho tam.

```
void LeVetor(int vet[], int tam){
   int i;
   for(i = 0; i < tam; i++){
            printf("Digite numero:");
            scanf("%d",&vet[i]);
   }
}
```

Funções com vetores como parâmetros

E no programa principal dois vetores são declarados e passados por referência para fazer a leitura dos seus dados.

```
int main(){
	int vet1[5], vet2[5];
	printf(" Lendo Vetor 1 --------\n");
	LeVetor(vet1,5);
	printf(" ------ Lendo Vetor 2 --------\n");
	LeVetor(vet2,5);
}
```

Acrescente um procedimento que faça a troca dos elementos entre dois vetores e outro procedimento que imprima os valores dos vetores. Faça a chamada do procedimento imprima antes e depois da troca.

Funções com vetores como parâmetros

- Ao passar um vetor como parâmetro, não é necessário fornecer o seu tamanho na declaração na função.
- Quando se trata de uma matriz ou de um vetor multi-dimensional somente podemos deixar de informar o tamanho da primeira dimensão, as outras dimensões devem ser informadas. Exemplo:

```
void LeMatriz(int mat[][10], int n) {
                    ...
}
```

- Ou então pode-se criar uma função indicando todas as dimensões. Exemplo:

```
void LeMatriz(int mat[10][10], int n) {
                    ...
}
```

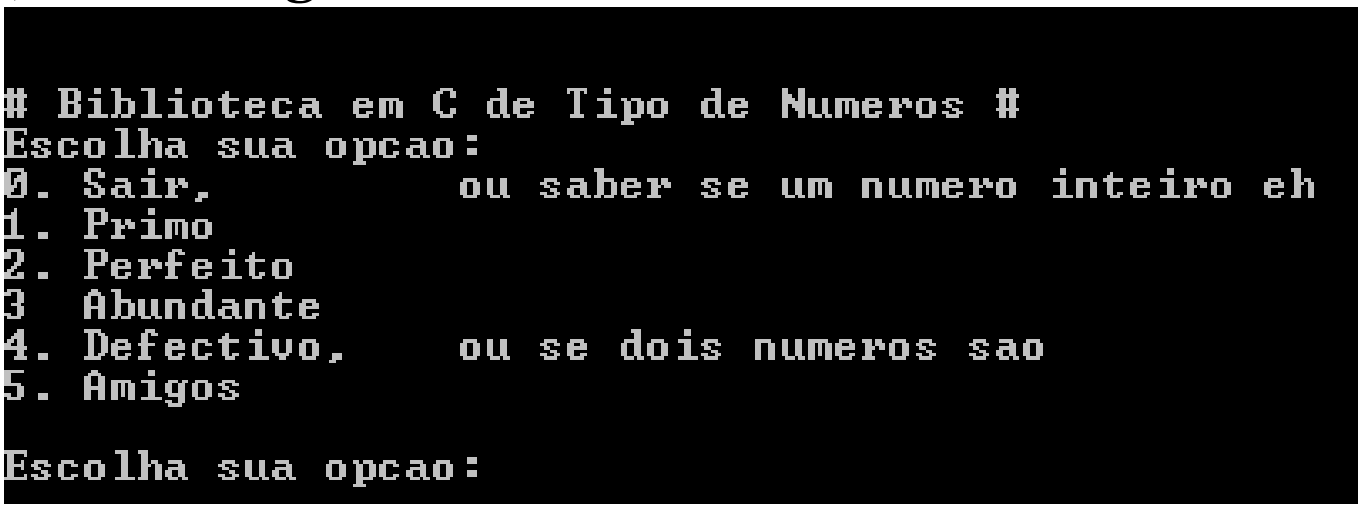
Funções com vetores como parâmetros

- Vejamos o caso no qual temos um parâmetro do tipo string, ou seja, um vetor de char.

```
int contaVogais (char s[]) {
   int numVogais, i;
   char *vogais;
   vogais = "aeiouAEIOU";
   numVogais = 0;
   for (i = 0; s[i] != '\0'; ++i) {
      char ch = s[i];
      int j;
      for (j = 0; vogais[j] != '\0'; ++j) {
            if (vogais[j] == ch) {
                  numVogais += 1;
                  break;
            }
      }
   }
return numVogais;
```

Exemplo um pouco extenso

- Vamos fazer um programa para determinar que tipo de número é um inteiro lido pelo usuário. As opções são primo, perfeito, abundante e defectivo. Ou seja: um programa de 5 funções, com a seguinte tela inicial:



- Segue-se o conceito de cada tipo de número.
- **Primo:** todo número natural maior que 1 cujos únicos divisores são o 1 e o próprio número. Exemplos: 2, 3, 5,...

Exemplo um pouco extenso

- **Perfeito**: todo número natural que é igual à soma dos seus divisores próprios DP(todos seus divisores exceto ele mesmo). Exemplo, o 6 é um número perfeito pois seus DP são 1, 2, e 3 e se cumpre que 1+2+3=6.
- **Abundante**: todo número natural cuja soma dos seus DP é maior que o próprio número. Por exemplo, 12 é abundante já que seus DP são 1, 2, 3, 4 e 6 e se cumpre que 1+2+3+4+6=16, que é maior que 12.
- **Defectivo**: todo número natural cuja soma dos seus DP é menor que o próprio número. Por exemplo, 16.
- **Números amigos**: pares (A,B) de números cuja soma dos DP de A é igual a B, e a soma dos DP de B é igual a A. Exemplo 220 e 284 são amigos.

Exemplo um pouco extenso

- Vamos criar as seguintes funções:
- main() que apenas chame o procedimento menu()

1. menu() que irá exibir as opções para o usuário escolher,
2. encaminha() que irá encaminhar a execução às funções correspondentes para a determinação dos tipos de números. Esta função considera os 'case' necessários.
3. leitura() que fará a leitura e validação da entrada do(s) número(s).
4. primo() que determina se um número é primo ou não.
5. soma_DP() que determinará a soma dos DP de um inteiro n, esta função será necessária para determinar se um número é perfeito, abundante, defectivo, e se dois números são amigos.

Exemplo um pouco extenso

6. Perfeito_ou_nao() que determinará, nesta única função, se o número é perfeito, é abundante ou defectivo
7. Amigos() que determinará se dois números são amigos.

Ou seja: um programa de 7 funções além da main().

- Veja a implementação descrita no programa TiposNumeros.c no site da disciplina. Ele possui 152 linhas.

- Quando um programa é extenso precisamos organizá-lo em mais de um arquivo para facilitar a sua compreensão.
- Mostraremos que podemos criar um projeto para organizar TiposNumeros.c em dois arquivos: Um que contenha o programa principal (main.c) e outro que contenha somente as funções de um programa (funcoes.c).

Criando um projeto no Code::Blocks

1. Selecione *File -> New -> Project*
   Como vamos criar um projeto para rodar no terminal, escolha *"Console application"*.
2. Em seguida, escolha a linguagem C
3. E dê um nome ao seu projeto.
   Evite usar espaços em branco e acentos. Use “TiposNumeros“ e um local para salvar seu projeto.
4. Continue clicando em Next, até ter criado o projeto.
5. Agora editamos o arquivo de código fonte de nome "main“ que foi gerado na pasta Sources ao lado. Nesse arquivo main.c vamos deixar a função main(), os cabeçalhos das funções, e uma função interna, a menu(), que vai exibir o menu de nossa aplicação.

Criando um projeto no Code::Blocks

- **Criando um projeto no Code::Blocks**

6. Agora vamos criar o arquivo de nome funcoes.c que contém a implementação das funções.
   Selecionamos *File -> New -> Empty File*
   Escolhendo salvar esse arquivo dentro do projeto com o nome funcoes.c, o qual será adicionado ao projeto no menu esquerdo.
7. Veja o programa main.c e funcoes.c no site da disciplina.

Agora basta compilar e rodar nosso projeto!

O Code::Blocks vai buscar dentro desses arquivos (main.c e funcoes.c) aquele que possui a função 'main()', e iniciar a execução do código.

Criando um arquivo header.h

- Vamos usar o nosso projeto com os arquivos 'main.c' e o ‘funcoes.c‘ e criar um arquivo cabeçalho.

  Escolha *File -> New -> Empty File*

- Vai ser perguntado se deseja adicionar esse arquivo ao seu projeto. Responda que sim e dê o nome ‘TiposNumeros.h‘

- Veja que escolhemos o mesmo nome de TiposNumeros.c, pois este e o header trabalham sempre em conjunto.
  Um vai explicar tudo (.h) e o outro vai implementar (.c).

- Note que ao dar um nome qualquer, com a extensão .h, o Code::Blocks vai criar uma seção chamada 'Headers', no menu direito, e vai colocar seu header lá.

Criando um arquivo header.h

- Seguidamente vamos pegar todas as declarações de funções que estão na 'main.c', que são os protótipos de todas as funções contidas no arquivo ‘TiposNumeros.c' e colocar no nosso header ‘TiposNumeros.h'.

  Agora precisamos que o módulo 'main.c' saiba da existência desses protótipos.
  Para isso, vamos incluir o header, adicionando:
  #include “TiposNumeros.h“

- Pronto, agora você pode compartilhar com qualquer pessoa as funções que você criou e ela não precisa saber como você implementou isso. Isso é muito comum quando pegamos bibliotecas de outros para usar em nossos projetos.