

Estrutura de Dados

Prof. Maurício Neto

O que vamos ver?

01

Funções

02

Modularização

01

Funções

Funções

A modularização em C começa com o uso eficiente de funções. As funções são uma das principais maneiras de organizar e modularizar o código. Elas são tipicamente utilizadas para realizar tarefas repetitivas durante a execução de um programa. Esse processo envolve associar um nome a uma sequência de comandos, o que chamamos de Declaração de Função. Posteriormente, o nome da função pode ser utilizado no corpo do programa sempre que for necessário executar o conjunto de comandos definidos, conhecido como Chamada de Função ou Procedimento.

Funções básicas

Em matemática, fazemos uso das funções, como:

- Função quadrado: $f(x) = x^2$ Exemplo: $f(2) = 4$
- Função reverso: $f(x) = 1/x$ Exemplo: $f(2) = 0,5$
- Função dobro: $f(x) = x * 2$ Exemplo: $f(3) = 6$

Essas funções possuem um parâmetro, que também podemos definir em funções com mais de um parâmetro:

- Função soma dois números: $f(x, y) = x + y$
- Função hipotenusa: $f(x, y) = \sqrt{x^2 + y^2}$

O resultado desta função para $x = 3$ e $y = 4$ é 5.

Portanto, $f(3,4)$ da função hipotenusa **retorna** 5.

Dizemos também que x e y são os **parâmetros da função**.

Os valores 3 e 4 são os **argumentos** da função hipotenusa.

Funções básicas

Podemos considerar uma função como uma sequência de comandos com um nome específico. É uma espécie de agrupamento de instruções.

Exemplo:

- Mensagem
- Criar 3 variáveis
- Iniciar as três variáveis
-
-
-

```
void main(){  
    println("Olá");  
    println("Bem vindo!");  
    int var1, var2, var3;  
    .  
    .  
    .  
}
```

Funções básicas

Em C, uma função **void** é uma função que não retorna nenhum valor ao seu chamador. O tipo de retorno void indica que a função executa uma ação, mas não fornece um resultado que possa ser utilizado posteriormente. Essa característica é útil para funções que realizam operações como imprimir dados, modificar variáveis por referência ou realizar tarefas que não necessitam de um retorno.

```
void nomeDaFuncao(parâmetros) {  
    // Corpo da função  
}
```

Funções básicas

Exemplo simples de uma função void que imprime uma mensagem na tela:

```
#include <stdio.h>

// Função que não retorna nenhum valor
void imprimirMensagem() {
    printf("Olá, esta é uma função void!\n");
}

int main() {
    // Chamando a função void
    imprimirMensagem();
    return 0;
}
```


Funções básicas

Definição da Função:

A função `imprimirMensagem()` é definida com o tipo de retorno `void`, indicando que não retornará nenhum valor.

Dentro da função, usamos `printf` para exibir uma mensagem na tela.

Chamada da Função:

No `main()`, chamamos `imprimirMensagem()`, que executa seu corpo, mas não retorna nenhum valor.

Funções básicas

Quando Usar Funções **void**

As funções void são frequentemente utilizadas em situações como:

Impressão de Saídas: Para exibir informações ao usuário.

Modificação de Dados: Para alterar o valor de variáveis passadas como argumentos por referência.

Execução de Tarefas: Para realizar operações que não requerem um valor de retorno, como inicializações ou configurações.

Funções básicas

Nas linguagens de programação, também existem funções, que são bastante similares às funções matemáticas. Uma função é um tipo específico de sub-rotina que retorna um resultado para o ponto de onde foi invocada.

```
#include <stdio.h>

int soma (int x, int y)
{
    int s;
    s = x + y;
    return (s);
}

int main(void)
{
    int c;

    c = soma (3 , 5);
    printf ("Resultado: %i\n",c);
    return 0;
}
```

Funções básicas – Exemplo 1

Observe que a função soma foi definida fora da função main(). Em C, main() também é uma função e é a primeira a ser executada quando o programa é iniciado.

Nesse programa, os parâmetros da função soma são x (do tipo int) e y (também do tipo int). O tipo de retorno da função soma também é int.

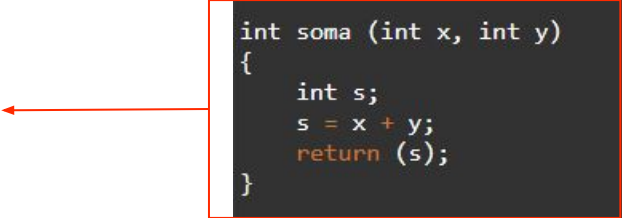
No corpo da função, os parâmetros x e y são usados como variáveis comuns.

```
#include <stdio.h>

int soma (int x, int y)
{
    int s;
    s = x + y;
    return (s);
}

int main(void)
{
    int c;

    c = soma (3 , 5);
    printf ("Resultado: %i\n",c);
    return 0;
}
```



Funções básicas – Exemplo 1

No corpo do programa, chamamos a função soma, passando os valores 3 e 5 como argumentos.

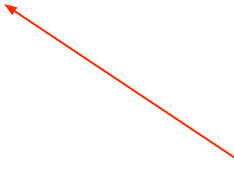
Portanto, para criar funções significativas em C, é importante escolher nomes expressivos, definir os parâmetros e tipo de retorno adequadamente, utilizar o return para retornar valores e chamar a função passando os argumentos corretos.

```
#include <stdio.h>

int soma (int x, int y)
{
    int s;
    s = x + y;
    return (s);
}

int main(void)
{
    int c;

    c = soma (3 , 5);
    printf ("Resultado: %i\n", c);
    return 0;
}
```



Funções básicas – Exemplo 1


A função main deve obrigatoriamente retornar um valor do tipo inteiro. Esse valor pode ser utilizado pelo sistema operacional para verificar o resultado da execução do programa. A convenção mais comum estabelece que a função main retorne zero quando a execução for bem-sucedida, enquanto um valor diferente de zero indica que houve problemas durante a execução.

```
#include <stdio.h>

int soma (int x, int y)
{
    int s;
    s = x + y;
    return (s);
}

int main(void)
{
    int c;

    c = soma (3 , 5);
    printf ("Resultado: %i\n",c);
    return 0;
}
```



Funções básicas – Exemplo 2

O compilador trata cada uma dessas variáveis de forma independente. Isso significa que elas possuem escopos diferentes.

Uma variável tem como escopo a região do programa em que ela é visível e pode ser acessada. Neste caso, uma variável `i` tem como escopo o programa principal, enquanto a outra variável `i` tem como escopo apenas a função em que foi declarada, ou seja, ela só pode ser acessada dentro dessa função específica.


```
#include <stdio.h>

long int fatorial (int n)
{
    int i;
    long int fat;
    fat = 1;
    for (i=1; i <= n; i++)
        fat = fat * i;
    return fat;
}

main(void)
{
    int i;
    for (i = 1; i <= 10; i++)
        printf("%i! = %li\n",i,fatorial(i));
}
```

Funções básicas – Exemplo 2

Note que tanto o retorno da função fatorial quanto a variável **fat** foram declarados como **long int**. A linguagem C disponibiliza uma variedade de tipos que permitem trabalhar com números de diferentes precisões.



```
#include <stdio.h>

long int fatorial (int n)
{
    int i;
    long int fat;
    fat = 1;
    for (i=1; i <= n; i++)
        fat = fat * i;
    return fat;
}

main(void)
{
    int i;
    for (i = 1; i <= 10; i++)
        printf("%i! = %li\n",i,fatorial(i));
}
```


Funções

Em C, existe uma série de tipos, que podem trabalhar com tipos diferentes:

Tipo	Bytes	Formato	Inicio	Fim
char	1	%c	-128	127
unsigned char	1	%c	0	255
int	2	%i	-32.768	32.767
unsigned int	2	%u	0	65.535
long int	4	%li	-2.147.483.648	2.147.483.647
unsigned long int	4	%lu	0	4.294.967.295
float	4	%f	3,4E-38	3,4E+38
double	8	%lf	1,7E-308	1,7E+308
long double	10	%Lf	3,4E-4932	3,4E+4932

Atividade

Desenvolver um programa em C que receba um **caractere**, um **número inteiro** longo e um número de **ponto flutuante longo**, e imprima esses valores na tela.

```
Digite um caractere: M
Digite um número inteiro longo: 172178278
Digite um número de ponto flutuante longo: 3.781287182

Você digitou:
Caractere: M
Número inteiro longo: 172178278
Número de ponto flutuante longo: 3.78
```

02

Modularização

Modularização

O que é?

Em virtude das funções e procedimentos, temos a capacidade de modularizar a construção do nosso software. Isso significa que podemos segmentar o sistema em partes menores, cada uma com tarefas específicas. A utilização desse recurso oferece diversas vantagens, como a reutilização do código, a simplificação da manutenção e uma melhor legibilidade, o que facilita a compreensão do código por outros desenvolvedores, entre outros benefícios. Assim, o conceito de dividir o código em blocos que se comunicam entre si, onde cada parte possui uma responsabilidade específica, é conhecido como modularização.

Modularização

Quando se trata de desenvolver e testar aplicações mais complexas, que exigem funcionalidades adicionais, é fundamental adotar técnicas que nos ajudem a organizar o código-fonte de maneira eficaz. Alguns pontos importantes a serem considerados incluem:

Programas complexos são formados por um conjunto de segmentos de código que, por si só, não são tão complicados.

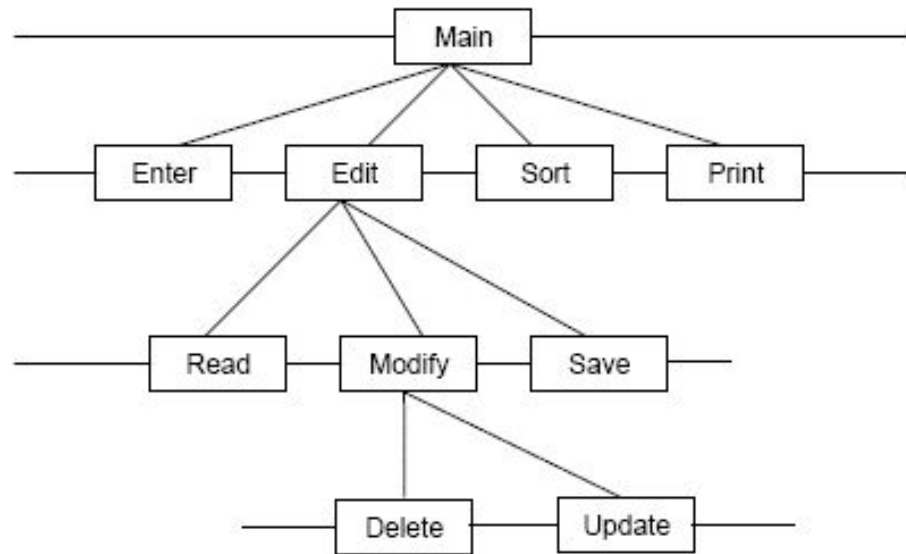
Frequentemente, reutilizamos trechos de código que já foram criados em outros projetos.

É comum na programação decompor aplicações complexas em partes menores e, em seguida, integrá-las para formar o produto final. Essa abordagem é conhecida como programação modular.

Modularização

O desenvolvimento de programas em C segue uma abordagem **Top-Down**, que se caracteriza por estruturar o projeto a partir da rotina de nível mais alto até as rotinas de nível mais baixo.

Essa abordagem começa com uma descrição geral do programa e caminha em direção da particularização, dividindo o código em módulos menores e mais específicos.



Características da Abordagem Top-Down

O projeto do programa é estruturado a partir da rotina principal, que chama as principais funções necessárias.

Posteriormente, são definidos os requisitos de cada uma dessas funções e o processo é repetido, dividindo-as em sub-rotinas cada vez mais específicas.

Essas sub-rotinas compartmentadas eventualmente realizarão ações simples que podem ser facilmente codificadas.

Ao definir como o aplicativo se integra em um nível elevado, o trabalho de nível inferior pode ser autossuficiente.

Define

É uma diretiva de pré-processamento que permite definir macros e constantes. Ele é usado para substituir ocorrências de um identificador por uma sequência de tokens especificada.

```
#define identificador token-string
```

identificador é o nome da macro ou constante que será substituído pelo pré-processador.

token-string é a sequência de tokens que substituirá o identificador.

Define - Exemplo

```
#define PI 3.14159  
#define area_circulo(r) (PI * (r) * (r))
```

PI é definido como a constante 3.14159.

area_circulo é definido como uma macro que calcula a área de um círculo com raio *r*.

Define e Const

Embora ambos **#define** e **const** sejam usados para definir constantes, existem algumas diferenças:

#define é uma diretiva de pré-processamento, enquanto **const** é uma palavra-chave da linguagem.

#define substitui o identificador por uma sequência de tokens, enquanto **const** cria uma variável constante na memória.

#define não possui tipo, enquanto **const** requer um tipo de dados.

#define não permite verificação de tipo, enquanto **const** permite.

Em geral, **#define** é usado para definir constantes simbólicas e macros simples, enquanto **const** é usado para definir constantes tipadas que precisam de verificação de tipo ou inicialização dinâmica.

Exemplo Top-Down

Vamos considerar um exemplo simples de um programa que calcula a média de notas de um aluno.

```
#include <stdio.h>

#define num_notas 3

int main() {
    float notas[num_notas];
    float soma = 0.0;
    float media;

    for (int i = 0; i < num_notas; i++) {
        printf("Digite a nota %d: ", i + 1);
        scanf("%f", &notas[i]);
        soma += notas[i];
    }

    media = soma / num_notas;

    printf("A média das notas é: %.2f\n", media);

    return 0;
}
```

Exemplo Top-Down

Desenvolvimento de um programa a partir da visão geral e, em seguida, detalhar as partes específicas. Vamos considerar um exemplo simples de um programa que calcula a média de notas de um aluno.

Estrutura Geral do Programa

Função Principal: `main()`

Função para Capturar Notas: `capturarNotas()`

Função para Calcular a Média: `calcularMedia()`

Função para Exibir a Média: `exibirMedia()`

Exemplo Top-Down

```
#include <stdio.h>

#define num_notas 3

float* capturarNotas() {
    static float notas[num_notas];

    for (int i = 0; i < num_notas; i++) {
        printf("Digite a nota %d: ", i + 1);
        scanf("%f", &notas[i]);
    }

    return notas;
}

float calcularMedia(float notas[]) {
    float soma = 0.0;
    for (int i = 0; i < num_notas; i++) {
        soma += notas[i];
    }
    return soma / num_notas;
}

void exibirMedia(float media) {
    printf("A média das notas é: %.2f\n", media);
}

int main() {
    float* notasPtr;
    float media;

    notasPtr = capturarNotas();
    media = calcularMedia(notasPtr);
    exibirMedia(media);

    return 0;
}
```

Exemplo Top-Down

Essa função é responsável por solicitar ao usuário que insira as notas.

Ela aloca um array estático de float chamado notas com tamanho num_notas.

Dentro de um loop, a função usa printf para exibir uma mensagem solicitando cada nota e scanf para ler o valor digitado pelo usuário, armazenando-o no array notas.

Após capturar todas as notas, a função retorna um ponteiro para o array notas.

```
float* capturarNotas() {  
    static float notas[num_notas];  
  
    for (int i = 0; i < num_notas; i++) {  
        printf("Digite a nota %d: ", i + 1);  
        scanf("%f", &notas[i]);  
    }  
  
    return notas;  
}
```

Exemplo Top-Down

Essa função recebe um array de float chamado notas como argumento.

Ela inicializa uma variável soma com 0.0.

Em seguida, a função percorre o array notas usando um loop, somando cada valor à variável soma.

Após a soma, a função calcula a média dividindo soma por num_notas.

Finalmente, a função retorna o valor da média.

```
float calcularMedia(float notas[]) {  
    float soma = 0.0;  
    for (int i = 0; i < num_notas; i++) {  
        soma += notas[i];  
    }  
    return soma / num_notas;  
}
```


Exemplo Top-Down

Essa função recebe um valor float chamado media como argumento.

Ela usa printf para exibir uma mensagem formatada, imprimindo o valor da média com duas casas decimais.

```
void exibirMedia(float media) {  
    printf("A média das notas é: %.2f\n", media);  
}
```

Exemplo Top-Down

Ela declara um ponteiro `notasPtr` do tipo `float*` para armazenar o array de notas e uma variável `media` do tipo `float` para armazenar a média.

Chama a função **`capturarNotas()`**, que retorna um ponteiro para o array de notas preenchido. Esse ponteiro é armazenado em `notasPtr`.

Chama a função **`calcularMedia()`**, passando `notasPtr` como argumento. A média calculada é armazenada na variável `media`.

Chama a função **`exibirMedia()`**, passando `media` como argumento, para exibir a média na tela.

Retorna 0 para indicar que o programa foi executado com sucesso.

```
int main() {  
    float* notasPtr;  
    float media;  
  
    notasPtr = capturarNotas();  
    media = calcularMedia(notasPtr);  
    exibirMedia(media);  
  
    return 0;  
}
```

Exemplo Top-Down

Esse código demonstra a abordagem Top-Down, com cada função realizando uma tarefa específica.

A função **capturarNotas()** captura as notas, **calcularMedia()** calcula a média e **exibirMedia()** exibe o resultado.

A função **main()** coordena a execução dessas funções para obter o resultado final.

Atividade

Faça uma calculadora simples, com as quatro operações, o usuário deve digitar dois números para a operação escolhida, o resultado deve ser exibido na tela, e o programa deve ter uma opção para finalizar.

```
Calculadora Simples
Escolha uma operação (+, -, *, /) ou 's' para sair: +
Digite o primeiro número: 10
Digite o segundo número: 5
Resultado: 15.00
Calculadora Simples
Escolha uma operação (+, -, *, /) ou 's' para sair:
```