

Programação de Computadores

CRIAÇÃO DE FUNÇÕES

Introdução

- As funções são importantes para:
 - **Dividir** o código em blocos
 - **Reaproveitar** código existente
- A modularização de programas é a principal característica da **programação estruturada**
 - Facilita a manutenção
 - Encapsula a solução
 - Cria uma interface

Saída ← `double` sqrt(`double`); → Entrada

Introdução

- A biblioteca padrão da linguagem C/C++ possui mais de **140 funções** predefinidas
 - Prefira utilizar as funções existentes
 - Não reinvente a roda
- Para **criar uma função** é preciso:
 - **Declarar** a função (fornecer um protótipo)
 - **Definir** a função (fornecer um corpo)
 - **Chamar** a função

Definindo Funções

- A **definição de uma função** obedece ao modelo:



Funções Sem Retorno

- Funções que **não retornam valores** são funções de tipo **void**
 - A instrução de **retorno é opcional**
 - Se utilizada, deve ficar vazia

Tipo de retorno void

```
void função(parâmetros)
{
    instrução 1;
    instrução 2;
    ...
    return;      // opcional
}
```

Funções Sem Retorno

- São normalmente usadas para modularizar o programa

```
void tchau(int n)
{
    cout << "Finalizando sessão número " << n;
    cout << endl;
}
```

- A chamada da função passa um valor inteiro

```
int main()
{
    cout << "Encerrar sessão: ";
    int sessao;
    cin >> sessao;
    tchau(sessao); // chamada da função
}
```

Funções Com Retorno

- Funções que **retornam valores** têm a forma geral abaixo:
 - O **valor de retorno é obrigatório** e pode ser uma constante, uma variável ou uma expressão

Tipo de retorno diferente de void (int, double, char, etc.)

```
tipo função(parâmetros)
{
    instrução 1;
    instrução 2;
    ...
    return valor;
}
```

Funções Com Retorno

- São muito usadas para **encapsular cálculos**:

```
double media(double a, double b)
{
    // media aritmética entre a e b
    double m = (a + b)/2;
    return m;
}
```

- A chamada da função retorna um resultado

```
int main()
{
    double quant;
    quant = media(12,8); // chamada da função
    cout << "Resultado = " << quant << endl;
}
```


Definindo Funções

- Uma função pode ter **vários retornos**
 - Mas apenas um deles será executado

```
int maior (int a, int b)
{
    if (a > b)
        return a;
    else
        return b;
}

int main()
{
    cout << "Digite dois números: ";
    int num1, num2;
    cin >> num1 >> num2;
    cout << "O maior deles = " << maior(num1,num2) << endl;
}
```

Definindo Funções

- Ao contrário de linguagens como PASCAL, a linguagem C++ **não permite a criação de uma função dentro de outra**

```
int main()
{
    void flexao(int n)
    {
        cout << "Faça " << n << " flexões." << endl;
    }

    flexao(3);    // chama a função flexao
    ...
}
```

Parâmetros e Argumentos

```
#include <iostream>
using namespace std;

void flexao(int);          // protótipo da função flexao

int main()
{
    flexao(3);             // chama a função flexao
    cout << "Escolha um número inteiro: ";
    int cont;
    cin >> cont;
    flexao(cont);          // chama flexao novamente
    return 0;
}

void flexao(int n)
{
    cout << "Faça " << n << " flexões." << endl;
}
```

Parâmetros e Argumentos

- Saída do programa:

Faça 3 flexões.

Escolha um número inteiro: **512**

Faça 512 flexões.

- Uma função pode ser **chamada várias vezes**
 - A função `flexao()` é chamada duas vezes:
 - Uma com o argumento sendo o **valor 3**
 - Outra com o argumento sendo o **valor de uma variável**

Parâmetros da Função

- Apenas na definição de uma função é preciso **dar nome aos parâmetros**

```
void flexao(int);    // protótipo da função
...

void flexao(int n)   // definição da função
{
    cout << "Faça " << n << " flexões." << endl;
}
```

- Os parâmetros de uma função são **declarações de novas variáveis** que receberão o valor dos argumentos

- A definição de várias funções em um arquivo é feita de **forma sequencial**

```
#include <iostream>

void flexao(int);
void abdominal(int);

int main()
{
    ...
    return 0;
}

void flexao(int n)
{
    ...
}

void abdominal(int n)
{
    ...
}
```

Funções e Arquivos

- Funções podem estar em arquivos diferentes

```
// Arquivo de inclusão:  
// ginastica.h
```

```
void flexao(int);  
void abdominal(int);
```

```
// Arquivo fonte:  
// ginastica.cpp
```

```
void flexao(int n)  
{  
    ...  
}  
  
void abdominal(int n)  
{  
    ...  
}
```

```
// Arquivo principal:  
// malhando.cpp
```

```
#include <iostream>  
#include "ginastica.h"  
  
int main()  
{  
    cout << "Exercícios "  
         << "de hoje:"  
         << endl;  
  
    flexao(10);  
    abdominal(20);  
  
    return 0;  
}
```

Funções e Arquivos

```
#include <iostream>
using namespace std;

float media(float, float);

int main()
{
    float a = media(8,10);
    float b = 12 + media(15, media(4,2)) + a;
    cout << "As aulas tem " << b + media(20,40) << " horas.\n";
}

float media(float x, float y)
{
    float num = (x + y)/2;
    return num;
}
```




Funções e Arquivos



- Saída do programa:

As aulas tem 60 horas

- Uma **função** pode ser **usada como argumento** se seu valor de retorno for compatível com o tipo esperado pelo parâmetro

```
float media(float, float);
```

```
float b = 12 + media(, ) + a;
```

 
float float

Inicialização com Funções

```
// converte metros em centímetros
#include <iostream>
using namespace std;

int converte(int);    // protótipo da função

int main()
{
    cout << "Entre com a distância em metros: ";
    int num;
    cin >> num;

    int cent = converte(num);    // inicializando com uma função

    cout << num << " metros = " << cent << " centímetros.\n";
    return 0;
}

int converte(int n)    // definição da função
{
    int val = 100 * n;
    return val;
}
```

Inicialização com Funções

- Saída do programa:

```
Entre com a distância em metros: 30  
30 metros = 3000 centímetros.
```

- O programa usa `cin` para obter um valor para a variável `num` e este valor é `passado para a função` `converte()`

```
cout << "Entre com a distância em metros: ";  
cin >> num;
```

```
int cent = converte(num);
```

Retorno de Funções

- Funções com retorno diferente de void **devem usar a instrução return** para prover o valor de retorno e finalizar a função

```
int main()                // definição da função main
{
    ...
    return 0;
}

int converte(int n)       // definição da função converte
{
    ...
    return val;
}
```

Retorno de Funções

- Uma **instrução de retorno** pode conter uma expressão

```
int converte(int n)
{
    int val = 100 * n;
    return val;
}
```

```
int converte(int n)
{
    return 100 * n;
}
```

- Uma **função que retorna valor** pode ser usada no lugar de uma variável ou constante

```
int a = converte(10);
int b = 20 + converte(15);
cout << "O tamanho é " << converte(10) << " centímetros." << endl;
```

Diretiva using com Funções

- A diretiva **using** pode ser usada dentro ou fora da definição

```
#include <iostream>
void flexao(int);

int main()
{
    using namespace std;
    cout << "Escolha um número inteiro: ";
    ...
}

void flexao(int n)
{
    std::cout << "Faça " << n << " flexões." << std::endl;
}
```

Variável Local vs Global

- Uma **variável declarada fora de uma função** é chamada de **variável global** e visível em todo o código
 - Variáveis globais não inicializadas recebem o valor zero

```
#include <iostream>
using namespace std;

int x;      // variável global com valor 0
int y = 1;  // variável global com valor 1

int main()
{
    ...
}
```

Variável Local vs Global

- Uma **variável declarada dentro de uma função** é chamada de **variável local** e é visível apenas dentro da função
 - Os parâmetros da função são variáveis locais
 - Não são inicializadas automaticamente para zero
 - São alocadas na entrada e liberadas na saída da função

```
int converte(int n)           // n é uma variável local
{
    int x;                   // x contém lixo da memória
    int y = 1;               // y foi inicializada para o valor 1

    ...

}
```


Variável Local vs Global

```
#include <iostream>
using namespace std;

void local(void);
int x = 1, y = 2; // variáveis globais

int main()
{
    cout << "x antes: " << x << ", y antes: " << y << endl;
    local();
    cout << "x depois: " << x << ", y depois: " << y << endl;
    return 0;
}

void local(void)
{
    int y; // variável local
    x = 3;
    y = 3;
    cout << "x dentro: " << x << ", y dentro: " << y << endl;
}
```

Variável Local vs Global

- Saída do programa:

```
x antes: 1, y antes: 2  
x dentro: 3, y dentro: 3  
x depois: 3, y depois: 2
```

- A declaração de uma variável local **esconde uma variável global de mesmo nome**
 - A variável local deixa de existir ao final da função
 - A global se torna visível novamente

Nomes de Funções

- Programadores C++ tem muita flexibilidade na **escolha de nomes para funções**:

MinhaFuncao()	minha_funcao()
minhafuncao()	minha_func()
minhaFuncao()	mf()

- Cada programador tem **preferência por um estilo**
 - Não existe um estilo errado
 - O importante é **manter o mesmo padrão** em todo o código

Nomes de Funções

- **Palavras-chave** são o vocabulário de uma linguagem de programação e **não podem ser usadas** para dar nome a uma função
- Até o momento foram utilizadas as seguintes palavras-chave:

int	void	if
double	using	else
float	namespace	return

Resumo

- Um **programa C++** consiste de um ou mais módulos, chamados funções
- Existem dois tipos de funções:
 - Funções que retornam valor – **retorno é obrigatório**
 - Funções que não retornam valor - **tipo void**
- Os parâmetros de uma função informam:
 - A **quantidade** de argumentos
 - Os **tipos** dos argumentos

Resumo

- Um programa pode ser quebrado em vários arquivos
 - Arquivos **de inclusão** (.h) - contêm o protótipo das funções
 - Arquivos **fonte** (.cpp) - contêm a definição das funções
- As variáveis podem ser criadas dentro ou fora das funções:
 - Variáveis **globais** - inicializadas para zero
 - Variáveis **locais** - contém lixo da memória
 - Evite o uso de variáveis globais
 - Dificultam a manutenção do código