Unidade 7: Modularização de Código

Prof. Daniel Caetano

Objetivo: Modularização de Código com Construção de Funções.

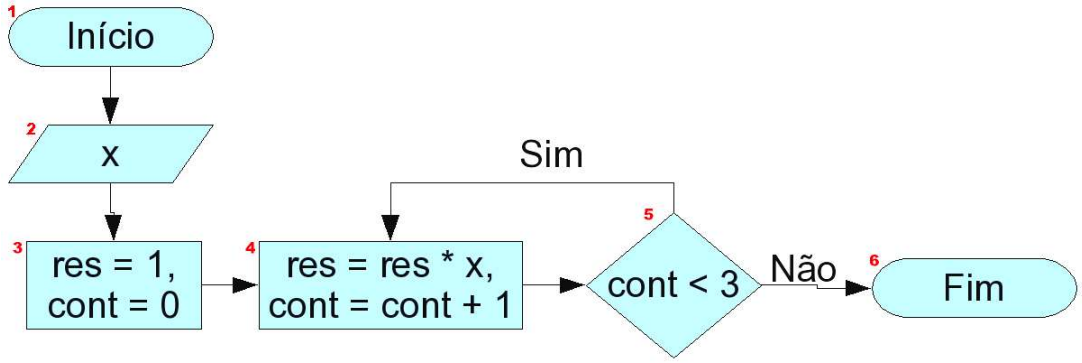
Bibliografia: ASCENCIO, 2007; MEDINA, 2006; SILVA, 2010; SILVA, 2006.

INTRODUÇÃO

Nas aulas anteriores aprendemos a fazer algoritmos simples, com um único bloco de texto. Isso é prático para programas mais simples como os que temos feito, mas para programas mais complexos, a situação muda.

Imaginemos, por exemplo, que um dia tenhamos que fazer um programa que calcule uma infinidade de operações de derivação (derivadas). Se a cada vez que precisarmos programar uma derivação tivermos que escrever todo o código de derivação (que é enorme e complexo) estaríamos com a nossa vida bem complicada!

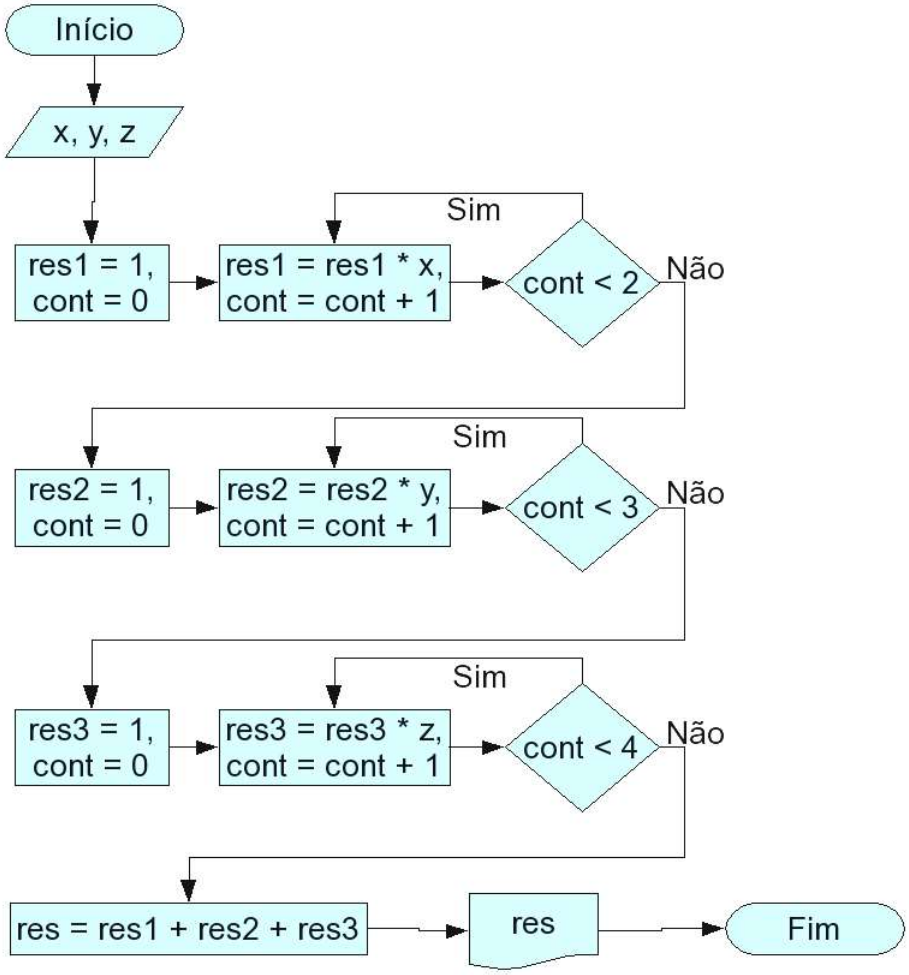
Vamos exemplificar. Considere o algoritmo do fluxograma abaixo, que resolve a potência x^3, usando a lógica de que $x^3 = x * x * x$.



A lógica do algoritmo é simples: iniciamos o resultado res = 1 e, para cada , valor da contagem cont, multiplicamos res por x (res = res * x). A tabela a seguir mostra os valores das variáveis em cada passo, supondo que o usuário tenha digitado o valor 3 para x:

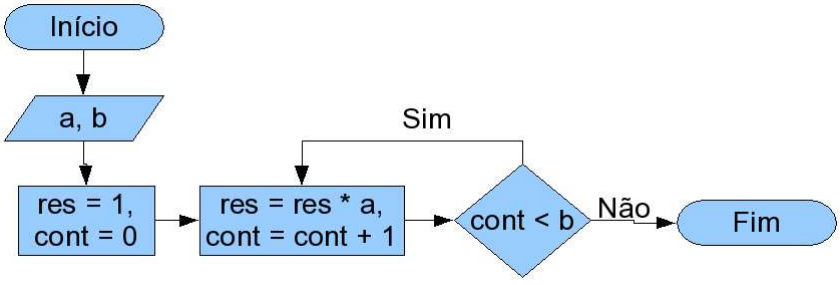
Passo	Bloco	x	res	cont	cont < 3
1	1	-	-	-	-
2	2	3	-	-	-
3	3	3	1	0	-
4	4	3	3	1	-
5	5	3	3	1	SIM
6	4	3	9	2	-
7	5	3	9	2	SIM
8	4	3	27	3	-
9	5	3	27	3	NÃO
10	6	3	**27**	3	-

Observe que, ao final, o resultado res = 27, que é exatamente o valor para 3^3. Agora, veja só como ficaria um fluxograma completo para o cáculo da expressão $x^2+y^3+z^4$:



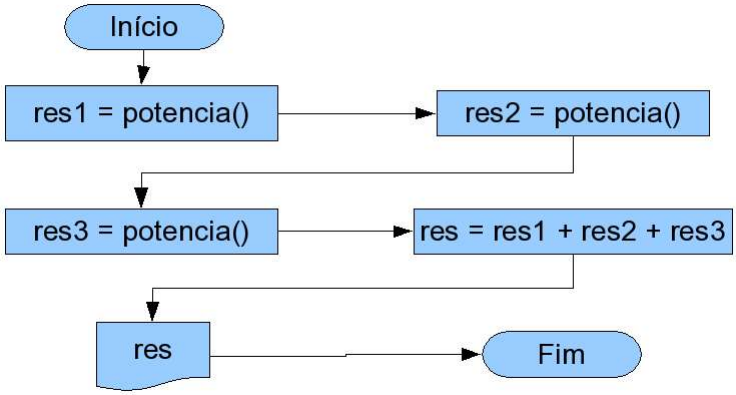
Observe que, neste fluxograma, temos uma porção de blocos repetidos, com poucas variações internas. Isso ocorre porque, obviamente, a forma de calcular a potência de um valor por outro (a^b) é sempre parecido, mudando apenas o valor que é usado na multiplicação (a) e o limite do cálculo lógico (b).

Que tal se pudéssemos dar um nome para esse bloco e, ao invés de repetí-lo, pudéssemos apenas indicar seu uso? Imagine que possamos definir o bloco chamado **potencia()**, definido conforme a seguir:



Esse é um algoritmo que recebe dois números, **a** e **b**, e calcula a potência: $\mathbf{a^b}$.

Vamos reconstruir o programa anterior, que calcula **res =** $\mathbf{x^2+y^3+z^4}$ apenas com o uso da expressão **potencia()** para que todo o processo apresentado acima seja executado, resultando no valor do cálculo:

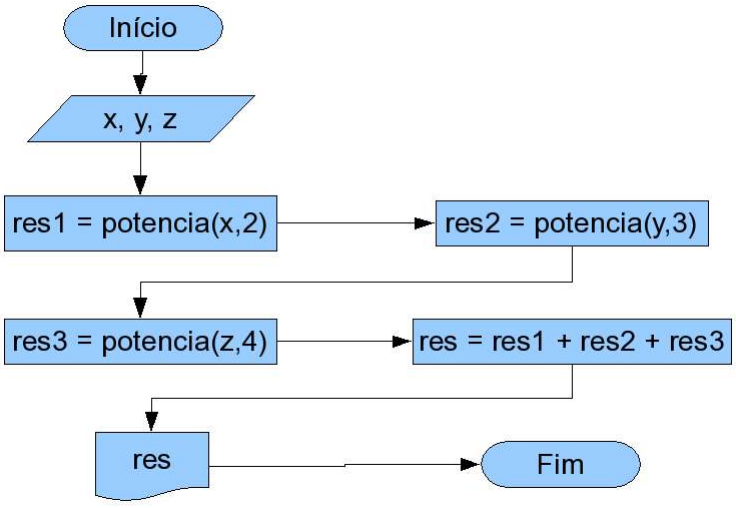


Bem mais simples de entender, não? Entretanto, esse resultado não é tão bom; o programa original pedia apenas **três** números; esse pede **seis**, porque o algoritmo que chamamos de **potencia()** considera que o usuário deve digitar os dois (o valor da base e o do expoente) para poder ser usado.

Podemos indicar, entretanto, para o computador, que os valores recebidos pelo algoritmo potência serão repassados pelo algoritmo principal, usando uma notação do seguinte tipo:

potencia(a,b)

Isto é: dentro dos parênteses, colocaremos as informações necessárias para o cálculo, já no programa principal, de maneira que o algoritmo não tenha que solicitá-las ao usuário. Observe a próxima versão do algoritmo:



No fundo, o que fizemos foi criar **dois** algoritmos: um deles, chamado **potencia()** não é usado diretamente pelo usuário, mas sim pelo nosso algoritmo principal.

Aos algoritmos que são usados por outros algoritmos damos o nome de **função**.

1. USO DE FUNÇÕES

Até o presente momento, o conceito de função foi apresentado de maneira lúdica, com o uso de fluxogramas. Agora veremos, na prática, como definimos uma função. Vamos verificar o código que fizemos anteriormente, em C/C++ para executar uma média entre dois números, a seguir. Observe as linhas destacadas em verde e vermelho... o que você percebe?

```
#include <iostream>
using namespace std;
int main(void) {

      float n1, n2, m;
      cout << "Calcula a média de dois números" << endl;
      cout << "Digite o 1o. número: ";
      cin >> n1;
      cout << "Digite o 2o. número: ";
      cin >> n2;
      m = (n1+n2)/2;
      cout << "A média é: " << m;
}
```

Elas são bem parecidas, não? Vamos criar, então, uma função chamada **lenumero()**, que receberá qual é o número de que deve ser lido (1 para 1o., 2 para 2o., 3 para 3o.... e assim por diante) e nos retornará o valor lido, de maneira que possamos reescrever esse programa da seguinte forma:

```
#include <iostream>
using namespace std;
int main(void) {

      float n1, n2, m;
      cout << "Calcula a média de dois números" << endl;
      n1 = lenumero(1);
      n2 = lenumero(2)
      m = (n1+n2)/2;
      cout << "A média é: " << m;
}
```

Se digitarmos o programa acima, entretanto, teremos um erro... o computador irá reclamar que não sabe o que significa **lenumero()**, é claro! Temos, então, de explicar para ele o que é esse tal de **lenumero()**.

Podemos fazer isso assim (não se preocupe, por enquanto, com a forma de escrever, isso será explicado mais adiante):

```
#include <iostream>
using namespace std;
int main(void) {

      float n1, n2, m;
      cout << "Calcula a média de dois números" << endl;
      n1 = lenumero(1);
      n2 = lenumero(2)
      m = (n1+n2)/2;
      cout << "A média é: " << m;
}

// Função que lê um número
float lenumero(int pos) {
      // Aqui vai o código da função
}
```

A linguagem ainda não ficará feliz, porque ainda não escrevemos o código da função. Vamos escrevê-lo, então.

```
#include <iostream>
using namespace std;
int main(void) {
      float n1, n2, m;
      cout << "Calcula a média de dois números" << endl;
      n1 = lenumero(1);
      n2 = lenumero(2)
      m = (n1+n2)/2;
      cout << "A média é: " << m;
}

// Função que lê um número
float lenumero(int pos) {
      // Declara variável de entrada
      double num;
      // Imprime a mensagem para digitar um número
      cout << "Digite o " << pos << "o. número: ";
      // Lê o número
      cin >> num;
}
```

O C/C++ ainda vai reclamar. Isso ocorre porque, no programa, usamos a função **lenumero** da seguinte forma:

n1 = lenumero(1);

Ou seja: eu estou pegando o resultado de **lenumero** e guardando em uma variável. Ocorre que o computador não tem bola de cristal e ele **não sabe** qual valor ele deve responder, isto é, qual é o valor que vai ser fornecido por **lenumero** para que seja armazenado em **n1**!

Para ajudá-lo, precisamos usar a instrução **return**. É a instrução **return** que indica para uma função qual é o valor que deve ser respondido para quem a chamou. Isso é feito da seguinte forma:

```
#include <iostream>
using namespace std;
int main(void) {
	float n1, n2, m;
	cout << "Calcula a média de dois números" << endl;
	n1 = lenumero(1);
	n2 = lenumero(2)
	m = (n1+n2)/2;
	cout << "A média é: " << m;
}

// Função que lê um número
float lenumero(int pos) {
	// Declara variável de entrada
	double num;
	// Imprime a mensagem para digitar um número
	cout << "Digite o " << pos << "o. número: ";
	// Lê o número
	cin >> num;
	// Retorna o valor lido
	return num;
}
```

Tudo pronto, você executa e... o C/C++ **ainda** reclama! Que coisa chata!

Na verdade, o C/C++ está reclamando porque o programa é executado do início para o fim, ou seja, de cima para baixo. Por essa razão, quando o programa encontra pela primeira vez a instrução:

n1 = lenumero(1);

Ele não sabe o que é isso ainda. Como o computador, ao executar um programa em C/C++ é um bichinho ansioso, esse problema pode ser resolvido de duas formas.

A primeira maneira de resolver isso é a mais simples: simplesmente vamos colocar a função **lenumero** *antes* do algoritmo principal. Mas atenção: antes do algoritmo principal **não é no início do arquivo...** é aqui, observe:

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

// Função que lê um número
float lenumero(int pos) {
	// Declara variável de entrada
	double num;
	// Imprime a mensagem para digitar um número
	cout << "Digite o " << pos << "o. número: ";
	// Lê o número
	cin >> num;
	// Retorna o valor lido
	return num;
}
```

```
// Programa Principal
int main(void) {
      float n1, n2, m;
      cout << "Calcula a média de dois números" << endl;
      n1 = lenumero(1);
      n2 = lenumero(2)
      m = (n1+n2)/2;
      cout << "A média é: " << m;
}
```

Alguns programadores não gostam de fazer isso, então eles preferem fazer uma outra coisa: deixar a função nova no final do arquivo, mas **explicar com antecedência** para o C/C++ que ela vai estar lá, que ele não precisa ficar ansioso:

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

// Declarações de Funções (Protótipos)
float lenumero(int pos);

// Programa Principal
int main(int argc, char *argv[]) {
      float n1, n2, m;
      cout << "Calcula a média de dois números" << endl;
      n1 = lenumero(1);
      n2 = lenumero(2)
      m = (n1+n2)/2;
      cout << "A média é: " << m;
}

// Função que lê um número
float lenumero(int pos) {
      // Declara variável de entrada
      double num;
      // Imprime a mensagem para digitar um número
      cout << "Digite o " << pos << "o. número: ";
      // Lê o número
      cin >> num;
      // Retorna o valor lido
      return num;
}
```

Com isso, todos ficam satisfeitos: o compilador, que ao encontrar o primeiro uso de **lenumero** já saberá que, em algum ponto do programa, isso será explicado... e o programador, que pode deixar a rotina principal de seu programa no topo do arquivo.

2. EXPLICANDO FUNÇÕES

Bem, já vimos como escrever uma função: basicamente é escrever um outro pequeno programa. Já vimos até como retornar um resultado. Mas como é construída aquela primeira linha "feia" do nome da função?

Observe abaixo:

```
// Função que lê um número
float lenumero(int pos) {
      // Código da Função
}
```

A primeira linha, especificamente, é: **float lenumero(int pos)** . O que isso significa?

Essa primeira linha se chama **declaração** da função ou **assinatura** da função. Essa linha diz para o computador (e para qualquer programador que vá usá-la) **como** ela deve ser usada. Essa declaração é composta por três partes principais:

tipo_de_retorno nome_da_função (parâmetro_da_funcao)

O tipo de retorno indica o tipo de valor que essa função retorna, com o return. Os tipos podem ser quaisquer tipos válidos no C/C++, como por exemplo: **int**, **float**, **double**, **long**, **bool**... adicionalmente existe um tipo_de_retorno chamado **void** que é usado sempre que uma função executar uma tarefa **sem retornar nada**.

O nome da função é o nome que usaremos para executá-la, isto é, sempre que quisermos executar esse trecho de código, usaremos este nome, como já foi visto no programa. As regras para esse nome são as mesmas usadas para variáveis: só letras simples e *underline*, sem acentuação, sem espaço. Números são permitidos, **desde que o nome não comece com número!**

E o parâmetro_da_função?

Bem, o parâmetro_da_função descreve a informação que é necessário fornecer àquela função para que ela possa ser executada. No caso da nossa função **lenumero**, o parâmetro é usado simplesmente para imprimir a mensagem:

"Digite o Xo. número: "

Onde X é substituído pelo valor que o programador forneceu ao desenvolver o programa. Assim:

Código:	**Texto impresso:**
lenumero(654);	"Digite o 654o. número: "
lenumero(2);	"Digite o 2o. número: "
lenumero(76);	"Digite o 76o. número: "

O parâmetro funciona, dentro da função, como se fosse uma variável com um valor pré-definido. Assim, precisamos dar um **tipo** e um **nome** para o parâmetro, como para qualquer outra variável.

Assim, observe novamente a declaração da função:

float lenumero(int pos)

Essa declaração significa que:

a) Essa função retorna um valor do tipo **float** (número com vírgula)

b) Essa função pode ser acessada pelo nome **lenumero**.

c) Essa função recebe um parâmetro que, dentro da função, será acessado por uma variável chamada **pos** do tipo **int**.

Uma função pode ter vários parâmetros. Para indicar isso, usamos a vírgula separando cada um deles. Por exemplo: no caso da função **potencia()** do início da aula, tínhamos dois parâmetros: **a** e **b**, para que ela pudesse executar o cálculo $\mathbf{a^b}$... ambos tinham que ser inteiro (**int**) e o resultado também é um inteiro (**int**). Assim, aquela função poderia ser declarada da seguinte forma:

int potencia(int a, int b)

Eventualmente podemos criar uma função que não precise de parâmetros. Nesse caso, indicamos isso com a palavra **void**. Consideremos, por exemplo, uma função chamada **imprimeajuda**, que não recebe parâmetros e nem retorna nenhum valor. Ela seria declarada desta forma:

void imprimeajuda(void)

NOTA: Uma função pode ter vários parâmetros, mas apenas UM valor pode ser retornado com o **return**.

2.1. Corpo da Função

Depois da declaração, o código da função deve ser especificado. Observe que, logo após a declaração, é sempre indicado um trecho entre chaves: { ... }. Observe:

```
// Função que lê um número
float lenumero(int pos) {
      // Código da Função
}
```

As chaves indicam onde se inicia e onde finaliza o corpo da função. Tudo que estiver entre o { ... } será executado quando a função for chamada. Se a função for declarada com um tipo de retorno (**float**, no exemplo), ela DEVE ter um comando **return** ao seu final, retornando um valor do tipo correto.

Caso a função seja declarada do tipo **void**, o return é opcional ao final, mas pode ser indicado simplesmente como **return;** (sem nenhum valor entre o return e o ;), se for desejado.

3. ESCOPO DAS VARIÁVEIS

Assim como no programa principal, é possível usar variáveis. Além daquelas recebidas como parâmetros, é possível declarar novas variáveis. A declaração se faz como no programa principal. Por exemplo:

```
// Função que lê um número
float lenumero(int pos) {
      // Declara variável de entrada
      double num;
      // Imprime a mensagem para digitar um número
      cout << "Digite o " << pos << "o. número: ";
      // Lê o número
      cin >> num;
      // Retorna o valor lido
      return num;
}
```

Os pontos importantes... IMPORTANTES e que não devem ser esquecidos são:

a) A variável declarada em uma função **NÃO EXISTE** em outra função, isto é, a variável só vale dentro da função em que ela foi declarada.

b) Alterar valor das variáveis-parâmetro é possível, mas **A ALTERAÇÃO SERÁ PERDIDA** quando a função terminar sua execução.

Observe os exemplos a seguir:

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

// Declarações de Funções (Protótipos)
void funcao1(void);
void funcao2(void);

// Programa Principal
int main(void) {

      int a;
      a = 1;
      cout << "A na principal: " << a << endl;
      funcao1();
      funcao2();
      cout << "B na principal: " << b << endl;
}

// Função 1
void funcao1(void) {
      int b;
      b = 1;
      cout << "B na Funcao1: " << b << endl;
      cout << "A na Funcao1: " << a << endl;
}
```

```
// Função 2
void funcao2(void) {
      b = b + 1;
      cout << "B na Funcao2: " << b << endl;
      cout << "A na Funcao2: " << a << endl;
}
```

Esse programa irá dar erros em várias das linhas. Na rotina principal (**main**), ele irá reclamar que não sabe o que é a variável **b**. E ele reclama com razão, a variável **b** não foi declarada na rotina principal.

Na **funcao1** ele irá reclamar que não sabe o que é a, e com razão, porque **a** não foi declarado nessa função. Finalmente, na **funcao2** ele irá reclamar de tudo, porque dentro dessa função não existe nem **a** nem **b**.

Assim, reforçando a primeira regra: **as variáveis só têm validade dentro das funções em que foram declaradas!**

Observe, agora, o programa a seguir:

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

// Declarações de Funções (Protótipos)
void funcao(int a);


// Programa Principal
int main(void) {

      int a;
      a = 1;
      cout << "A na principal: " << a << endl;
      funcao(a);
      cout << "A na principal: " << a << endl;
}

// Função 1
void funcao(int a) {
      cout << "A no início da Funcao: " << a << endl;
       a = a + 1;
      cout << "A no fim da Funcao: " << a << endl;
}
```

Observe que a **funcao** recebe o valor correto, atualiza este valor, mas ao voltar para a rotina principal, a alteração se perdeu.

Para reforçar: **é possível modificar variáveis-parâmetro, mas a modificação se perde quando a função termina**. Para retornar valores é preciso usar a instrução **return**.

4. FUNÇÃO PARA CÁLCULO DE ARREDONDAMENTO

Ainda que não exista uma função pronta no padrão C/C++ para arredondamento de acordo com a regra usual, podemos produzir uma função de arredondamento usando aquelas disponíveis, em especial a função **floor**. A idéia é conseguir esse tipo de arredondamento:

Número	Arrendondamento
1,4	1,0
1,5	2,0

Vamos construir a função **round(x)**, que pode construída e usada da seguinte forma:

```
#include <iostream>
#include <math.h>

using namespace std;

int round(float x) {

	int res;

	res = floor(x+0.5);
	return res;
}

int main(void) {

	float num, arred;

	// Pega número fracionário
	cout << "Digite um número fracionário (ex: 1.578): ";
	cin >> num;

	// Arredonda
	arred = round(num);

	// Imprime resultado:
	cout << "O valor arredondado é: " << arred << endl;
}
```

Neste momento você pode estar se perguntando: como é que eu posso arredondar com um determinado número de casas decimais? A solução é usar um truque simples, como o indicado abaixo, para arredondar com DUAS casas decimais:

```
#include <iostream>
#include <math.h>

using namespace std;
```

```
int round(float x) {

	int res;

	res = floor(x+0.5);
	return res;
}

int main(void) {

	float num, arred;

	// Pega número fracionário
	cout << "Digite um número fracionário (ex: 1.578): ";
	cin >> num;

	// Arredonda com duas casas decimais
	num = num * 100;
	arred = round(num);
	arred = arred / 100;

	// Imprime resultado:
	cout << "O valor arredondado é: " << arred << endl;
}
```

Observe o truque: o número foi multiplicado por 100, para que as duas casas decimais desejadas venham para a parte inteira do número; após isso, é feito o arredondamento com o round e, finalmente, o número resultante é dividido por 100, para que volte a ter duas casas decimais. Observe a regra:

Número de Casas Decimais	**Multiplicar e Dividir por...**	**Ou...**
0	1	10^0
1	10	10^1
2	100	10^2
3	1000	10^n

Ou seja, para arredondar com **n** casas decimais, eu tenho que, antes de arredondar, multiplicar por $\mathbf{10^n}$ e, depois de arredondar, dividir por $\mathbf{10^n}$. Podemos usar a função **pow** para realizar a potência de 10, e o código anterior, adaptado para qualquer número de casas decimais, fica assim:

```
#include <iostream>
#include <math.h>

using namespace std;

int round(float x) {

	int res;

	res = floor(x+0.5);
	return res;
}
```

```
int main(void) {

	float num, arred;
	int casas;

	// Pega número fracionário
	cout << "Digite um número fracionário (ex: 1.578): ";
	cin >> num;
	cout << "Digite o número de casas desejado: ";
	cin >> casas;

	// Arredonda com duas casas decimais
	num = num * pow(10,casas);
	arred = round(num);
	arred = arred / pow(10,casas);

	// Imprime resultado:
	cout << "O valor arredondado é: " << arred << endl;

}
```

5. A FUNÇÃO MAIN (ROTINA PRINCIPAL)

Na aula passado já descobrimos o que significavam aquelas linhas "include" no início do código: elas significam quais são os tipos de extensões que vamos usar em nosso programa. Lembra-se que para usar as funções matemáticas precisávamos de incluir math.h?

Bem, hoje entenderemos outra parte do programa: a declaração da rotina principal.

A rotina principal, chamada sempre de **main** é uma função como outra qualquer; a única diferença é que esta rotina é a escolhida pelo computador para **iniciar a execução do programa**. Para tanto, ela tem uma declaração padronizada e fixa:

```
// Programa Principal
int main(void) {

	// Seu código vai aqui

}
```

Observe que ela retorna um valor inteiro (int) e não recebe nenhum parâmetro, o que é indicado com a palavra **void**.

5.1. Função Main Completa (OPCIONAL)

Vez ou outra você encontrará a função **main** declarada da seguinte forma:

```
// Programa Principal
int main(int argc, char *argv[]) {

	// Seu código vai aqui
}
```

Observe que, neste caso, ela ainda retorna um valor inteiro (int)... mas agora recebe dois parâmetros: um inteiro chamado **argc** e um negócio estranho e complicado chamado **argv** que, acredite, é a declaração de uma "tabela" de caracteres.

O aluno mais curioso certamente vai se perguntar: "mas, professor, se essa é a primeira coisa que o computador executa, quem vai colocar valores nesses parâmetros? Bem, existem diversas maneiras de colocar valores nesses parâmetros; vamos aqui explicar duas, para que possamos entender as funções deles.

Atualmente, estamos acostumamos a abrir um documento do Word simplesmente clicando no ícone de um documento. Ao clicar, por exemplo, em **arquivo.doc**, o Windows automaticamente executa o **Word.exe** e abre aquele arquivo indicado. Você já parou para pensar como é que isso acontece? Como o Word fica sabendo qual arquivo ele deve abrir?

Bem, "por baixo dos panos", o que o Windows faz, quando você clica no **arquivo.doc**, é executar o seguinte comando:

word.exe arquivo.doc

Em outras palavras, é como se ele abrisse um prompt de comandos do DOS e digitasse essa linha de comando indicada acima. A primeira palavra instrui o computador a carregar o programa **word.exe** na memória e executar a função **main** dele. A segunda palavra será passada como parâmetro da função **main**. Se selecionarmos vários arquivos para que sejam abertos simultaneamente, o Windows executa isso:

word.exe arquivo.doc arquivo2.doc arquivo3.doc

Relembremos, agora, a declaração da função main:

```
// Programa Principal
int main(int argc, char *argv[]) {

	// Seu código vai aqui

}
```

Como os parâmetros são guardados aí nesse "argc/argv"?

Bem, **argc** indica o **número** de parâmetros e **argv** é uma tabela com todos esses parâmetros. Vamos ver como isso funciona?

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {

      cout << "Número de Parâmetros: " << argc << endl;
}
```

Observe que, no mínimo, um programa tem sempre um parâmetro (não passamos nenhum e, ainda assim, argc vale 1!). Que parâmetro é esse?

Para saber, vamos imprimir o primeiro valor da tabela **argv**. Lembrando que em programação as contagens começam sempre em 0 (zero), a primeira posição da tabela é a posição 0. A impressão pode ser feita assim:

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {

      cout << "Número de Parâmetros: " << argc << endl;
      cout << "Primeiro Parâmetro: " << argv[0] << endl;
}
```

Observe! O primeiro parâmetro é exatamente o nome do programa, com o caminho completo! Modifique o programa como indicado abaixo:

```
#include <iostream>
using namespace std;
int main(int argc, char *argv[]) {

      cout << "Número de Parâmetros: " << argc << endl;
      cout << "Primeiro Parâmetro: " << argv[0] << endl;
      cout << "Segundo Parâmetro: " << argv[1] << endl;
}
```

Execute o programa. Você verá que ocorre um pequeno problema! Este problema tem origem no fato que estamos tentanto imprimir um parâmetro que não existe: há um parâmetro apenas, e estou tentando imprimir o segundo! Vamos ver como passar dois parâmetros?

Abra um prompt e, usando o comando CD, vá até o diretório onde a aplicação foi criada (o diretório do projeto). Observe a orientação do professor.

No prompt, digite o nome do programa, seguido de um espaço e, em seguida, uma palavra qualquer. Por exemplo, se o executável do seu programa se chama **Project1.exe**, execute-o assim:

Project1.exe professor

E veja o que acontece!

6. BIBLIOGRAFIA

ASCENCIO, A.F.G; CAMPOS, E.A.V. **Fundamentos da Programação de Computadores**. 2ed. Rio de Janeiro, 2007.

MEDINA, M; FERTIG, C. **Algoritmos e Programação: Teoria e Prática**. 2ed. São Paulo: Ed. Novatec, 2006.

SILVA, I.C.S; FALKEMBACH, G.M; SILVEIRA, S.R. **Algoritmos e Programação em Linguagem C**. 1ed. Porto Alegre: Ed. UniRitter, 2010.