Unidade 1: Funções

Prof. Daniel Caetano

Objetivo: Modularização de Código com Construção de Funções.

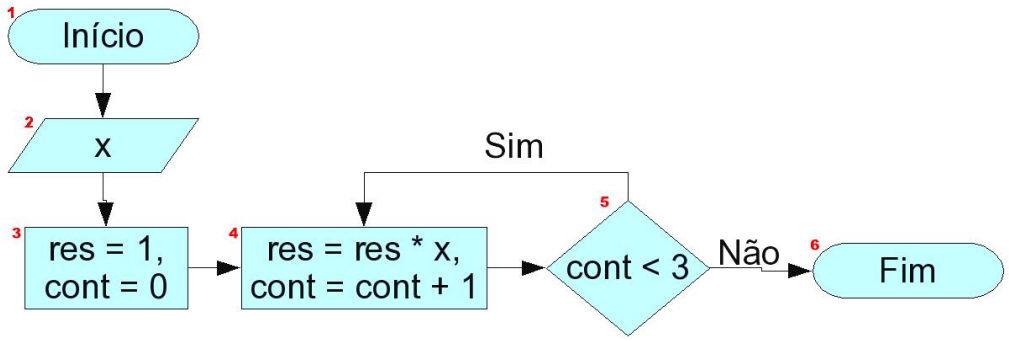
Bibliografia: ASCENCIO, 2007; MEDINA, 2006; SILVA, 2010; SILVA, 2006.

INTRODUÇÃO

No curso de Algoritmos aprendemos a construir algoritmos simples, com um único bloco de texto. Isso é prático para programas mais simples como todos aqueles criados, mas, para programas mais complexos, a situação muda.

Imaginemos, por exemplo, que um dia tenhamos que fazer um programa que precise realizar uma tarefa em diversos pontos diferente, como, por exemplo, inserir um cliente em um banco de dados. Como cada inserção implica em um longo e tedioso processo de verificação de erros e possíveis medidas corretivas, se formos reprogramar "tudo isso" a cada vez que precisarmos inserir um cliente no banco de dados, nossa vida ficará realmente complicada!

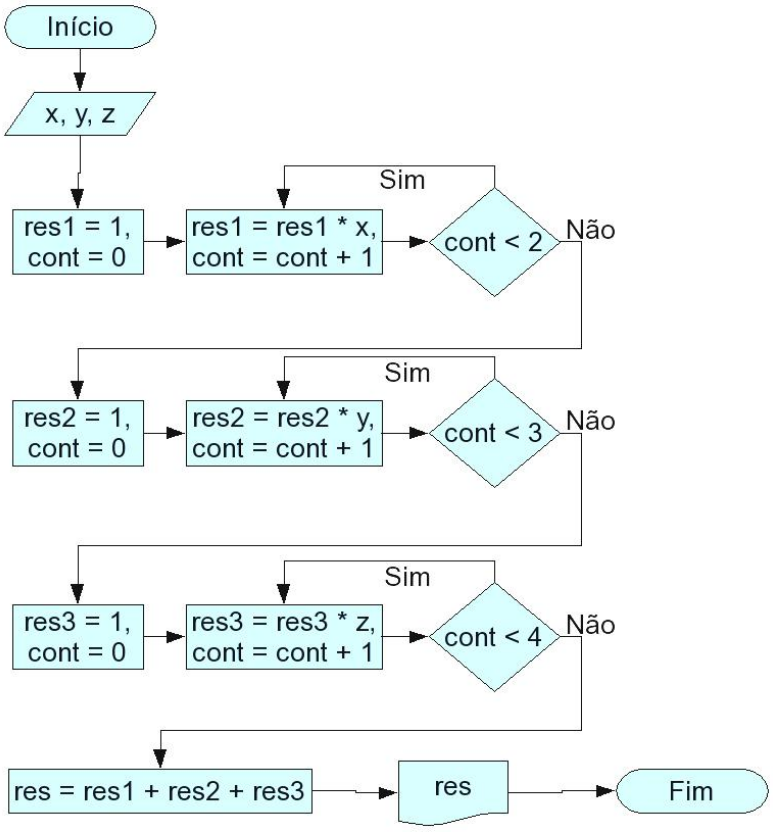
Vamos exemplificar com um caso mais concreto e simples. Considere o algoritmo do fluxograma abaixo, que resolve a potência x^3, usando a lógica de que $x^3 = x * x * x$.



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:

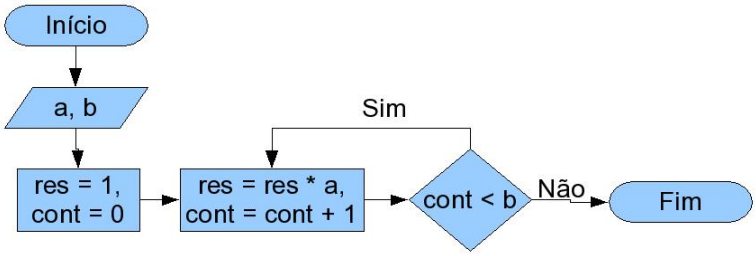
Passo	Bloco	x	res	cont	cont < 3
1	1	-	-	-	-
2	2	3	-	-	-
3	3	3	1	0	-
4	4	3	3	1	-
5	5	3	3	1	SIM
6	4	3	9	2	-
7	5	3	9	2	SIM
8	4	3	27	3	-
9	5	3	27	3	NÃO
10	6	3	**27**	3	-

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 $x^2+y^3+z^4$:



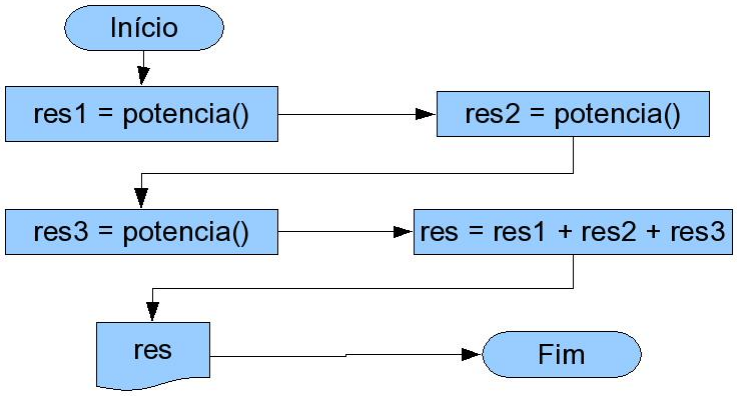
Observe que, neste fluxograma, temos uma porção de blocos repetidos, com poucas variações internas. Isso ocorre porque, obviamente, a forma de calcular a potência de um valor por outro (a^b) é sempre parecido, mudando apenas o valor que é usado na multiplicação (a) e o limite do cálculo lógico (b).

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** = $x^2+y^3+z^4$ apenas com o uso da expressão **potencia()** para que todo o processo apresentado acima seja executado, resultando no valor do cálculo:

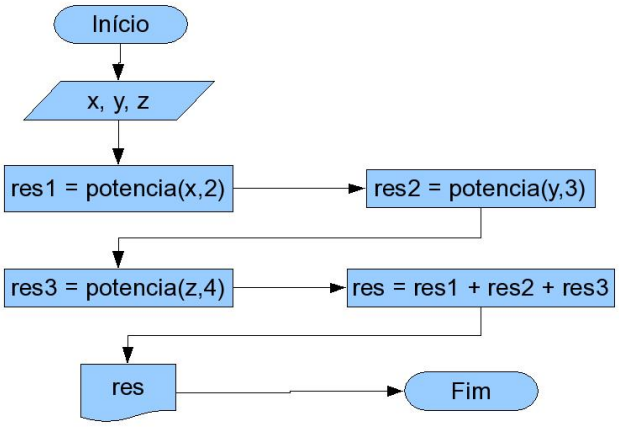


Bem mais simples de entender, não? Entretanto, esse resultado não é tão bom; o programa original pedia apenas **três** números; esse pede **seis**, porque o algoritmo que chamamos de **potencia()** considera que o usuário deve digitar os dois (o valor da base e o do expoente) para poder ser usado.

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**.

3.1. Parâmetros por Referência

Se, eventualmente, precisarmos que a alteração em um parâmetro seja refletida na variável original (usada na chamada da função), devemos indicar isso para o programa. Isso pode ser feito usando o símbolo **&** ("e" comercial) na frente do nome da variável. Analise o programa a seguir, execute-o e veja o que ocorre. A unica mudança com relação ao programa anterior é o & (que foi destacado em negrito e em vermelho na listagem):

```
#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 agora a mudança no valor do parâmetro **a** na funcao() faz com que o valor da variável **a** na main() também se altere.

NOTA: se um parâmetro é passado por referência, é um ERRO fornecer um **número** no local deste parâmetro. Observe o código abaixo:

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

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

// Programa Principal
int main(void) {
      funcao(1);          // ISSO É UM ERRO!
}

// Função 1
void funcao(int &a) {
       a = a + 1;
```

```
}
```

3.2. Variáveis Globais

Se, eventualmente, precisarmos de alguma variável que seja acessível em **qualquer parte do programa**, podemos declarar uma variável global. Isso é feito declarando a variável normalmente, mas fazendo isso no topo do programa, fora de qualquer função. Observe o código abaixo:

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

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

// Programa Principal
int main(void) {

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

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

EVITE usar variáveis globais, pois elas são muito problemáticas: não há controle algum sobre quais partes do programa podem ler ou escrever sobre elas e isso pode trazer muitos problemas de difícil solução. Além disso, programas que usam variáveis globais são muito mais difíceis de compreender, já que a explicação e definição da variável está, normalmente, muito longe do local em que a variável é usada.

Uma forma menos traumática de usar as variáveis globais é usá-las como números constantes, ou seja, o valor dela só é definido na declaração e nunca mais é mexido. Para isso, usa-se a palavrinha **const** conforme o exemplo abaixo:

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

// Declarações de Funções (Protótipos)
void funcao(void);
// Declarações de Variáveis Globais
const float pi = 3.141592;

// Programa Principal
int main(void) {
      cout << "PI na principal: " << pi << endl;
```

```
        funcao();
}

// Função 1
void funcao() {
        cout << "PI na função: " << pi << endl;
}
```

NOTA: NUNCA tente mudar o valor de uma variável **const**. A tentativa em fazê-lo causará um erro de compilação.

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.

NOTAS DE AULA 10 – Estruturas

1. Estruturas de Dados

A linguagem C/C++ fornece uma porção de tipos de variáveis: int, float, long, double, boolean, char... dentre outros. Mas todos esses tipos são chamados de "tipos simples" ou "tipos elementares" porque armazenam apenas uma informação: um número.

Algumas vezes precisamos representar tipos de dados mais complexos como um cliente ou um produto. Um produto, por exemplo, pode ser representado como um **id**, um **nome** e um **preço**. Como fazemos para criar uma lista de produtos?

Uma das alternativas seria criar uma lista "maluca", com vários vetores, cada um armazenando um tipo de informação diferente, mas o gerenciamento da mesma seria muito complicado!

Uma outra alternativa, mais simples, é **criar nosso próprio tipo de dado**: o tipo **produto**. Para fazer isso, usamos a instrução **struct**:

```
struct produto {
        int id;
        char nome[60];
        float preco;
};
```

Só isso? Só! Mas como usamos isso? Do mesmo jeito que qualquer outro tipo de dado. Por exemplo, se criamos uma pilha para inteiros assim:

```
int pilha[50];
int tpilha = -1;
```

Para criarmos uma pilha de produtos fazemos assim:

```
produto pilha[50];
int tpilha;
```

Observe como é a mesma coisa... só que o tipo de dado do vetor não é mais "int" e sim "produto". Ok, mas como é que eu acesso os dados de um produto? Por exemplo, como eu defino o "**id**" de um produto? Assim:

```
produto x;      // Declara o produto "x"
x.id = 10;      // Altera o id do produto "x"
```

O "." Indica que eu quero alterar o "id" que está dentro do produto "x". Mas... Como definir o nome do produto?

Bem, como o nome é um vetor, podemos fazer assim:

```
produto x;      // Declara o produto "x"
x.nome[0] = 'S';        // Guada primeira letra do nome
x.nome[1] = 'a';        // Guada segunda letra do nome
x.nome[2] = 'b';        // Guada terceira letra do nome
x.nome[3] = 'ã';        // Guada quarta letra do nome
x.nome[4] = 'o';        // Guada quinta letra do nome
x.nome[5] = '\0';       // Indica que o nome acabou
```

Como isso é muito chato e desagradável, os criadores do C/C++ elaboraram algumas funções para trabalhar com textos. Essas funções foram colocadas dentro da biblioteca **string.h**. Uma dessas funções é a **strcpy**, que copia um texto para dentro de uma variável:

```
produto x;
strcpy(x.nome,"Sabão");
```

Essa função funciona bem, mas para isso é preciso, no topo do código, acrescentar essa linha:

```
#include <string.h>
```

Experimente o programa abaixo:

```
#include <iostream>
#include <string.h>
using namespace std;
struct produto {
        int id;
        char nome[60];
        float preco;
};
int main(void) {
        produto p;
        p.id = 1;
        strcpy(p.nome, "Mouse");
        p.preco = 35.0;
        cout << p.id << ":";
        cout << p.nome << "-";
        cout << p.preco << endl;
}
```

Podemos ainda copiar o conteúdo de um dado complexo para outro, normalmente:

```
#include <iostream>
#include <string.h>
using namespace std;
struct produto {
        int id;
        char nome[60];
        float preco;
};
int main(void) {
        produto p, q;
        p.id = 1;
        strcpy(p.nome, “Mouse”);
        p.preco = 35.0;
        q = p;
        cout << q.id << “:”;
        cout << q.nome << “-“;
        cout << q.preco << endl;
}
```

NOTAS DE AULA 11 – Ponteiros

1. Antes dos Ponteiros: Variáveis e a Memória

Antes de tentar compreender o que é um ponteiro, o aluno precisa entender o que é, exatamente, uma variável.

Quando se estuda arquitetura e organização de computadores, compreendemos que o computador é composto de CPU, memória e unidades de entrada e saída. Qualquer informação que precise ser armazenada, será colocada na memória.

Simplificadamente, cada byte da memória é chamado de **posição de memória**. Algumas informações cabem em uma única posição de memória (como uma letra, do tipo de dado **char**), outras exigem várias posições de memória (como um **int** que, por ter 32 bits, exige 4 posições de memória).

Cada posição de memória tem um **endereço**, o chamado **endereço de memória**. O primeiro endereço de memória é o **0** (zero), o segundo é o **1**, o terceiro é o **2**... e assim por diante.

Quando declaramos uma variável do tipo **char**, por exemplo, na verdade solicitamos que o computador **reserve uma posição de memória**, para que possamos guardar uma letra na memória. Essa posição de memória, é claro, possui um **endereço**, que fica associado ao **nome da variável**, de maneira que não precisamos saber o endereço numérico, já que podemos nos referir a ele pelo **nome da variável**.

Assim, quando escrevemos esse código:

```
char    letra;
letra = 'a';
```

No fundo estamos dizendo ao computador que "reserve um byte na memória e dê o nome a essa posição de 'letra'"... e depois pedimos que ele guarde o valor referente à letra 'a' nessa posição de memória.

No caso dos valores inteiros, para citar outro exemplo:

```
int     num;
num = 10;
```

No fundo estamos dizendo ao computador que "reserve 4 bytes (espaço para um inteiro) e dê o nome de 'num' à posição do primeiro desses quatro bytes"... e depois disso solicitamos que o computador armazene o valor '10' nessas posições de memória.

Se quisermos saber o **e**ndereço de memória em que uma variável foi armazenada, podemos usar o operador &:

```
int num;
cout << &num;
```

Execute esse código e veja que número estranho aparece. Esse "número estranho" é o endereço de memória em que a variável foi alocada.

Agora, experimente o código a seguir:

```
int num1, num2;
cout << &num1 << endl;
cout << &num2;
```

Observe que o endereço de **num2** é exatamente 4 posições após **num1**. Isso ocorre porque, como já dito, variáveis inteiras ocupam 4 bytes (ou 4 posições de memória).

Se quisermos saber quantas posições de memória uma variável ocupa, podemos usar a função **sizeof()**. Experimente o código abaixo para descobrir quantos bytes uma variável do tipo float e uma do tipo double ocupam:

```
cout << sizeof(float);
cout << sizeof(double);
cout << sizeof(produto);          // Pressupõe que a estrutura produto foi declarada!
```

Note, porém, que não é "direto" guardar o endereço de uma variável em outra variável. O exemplo abaixo, por exemplo, irá causar algumas reclamações por parte do compilador:

```
int i, end;
end = &i;
```

2. Enfim, os Ponteiros

Embora nem sempre seja a única solução, algumas vezes pode ser útil trabalhar diretamente com os endereços de memória. Um exemplo clássico é quando criamos uma função que retorna uma resposta de tamanho não definido previamente (como um texto).

Para passarmos uma variável por referência (para que o texto fosse colocado nela) precisaríamos saber, com antecedência, o tamanho máximo do texto, mas isso nem sempre é possível. O que fazer?

Bem, a função que vai "devolver" o texto poderia devolver apenas o endereço onde se encontra o texto. E como saber onde começa e termina o texto? Simples: ele começa no endereço retornado e vai até o byte com valor '\0' (o byte de terminação é uma convenção).

Já que é útil, como fazemos para lidar direto com os endereços? Simples: vamos declarar um tipo especial de variável chamado "ponteiro", o que pode ser feito com um ***** antes do nome da variável:

```
int *end;
```

Essa declaração permite que eu armazene o **endereço de um inteiro** na variável de nome "**end**".

Observe o código (e o execute para ver o resultado):

```
int i, *end;
end  = &i;
cout << &i;
cout << end;
```

Observe que o tipo de dado é importante para os ponteiros. O código abaixo dá erro:

```
int *a;
char b;
a = &b;
```

Não podemos guardar o endereço de um **char** em um ponteiro para inteiros.

Ok, interessante... temos uma variável que guarda um endereço de memória... mas como ler o valor lá armazenado? Simples: usamos o operador *****conforme indicado**:**

```
int *a, b;
b = 10;
a = &b;
cout << &b;      // Imprime o endereço de 'b'
cout << b;       // Imprime o valor de 'b'
cout << a;       // Imprime o endereço de 'b' (armazenado na variável 'a')
cout << *a;      // Imprime o valor de 'b' (o valor armazenado no endereço apontado por a)
```

O nome dessas variáveis é **ponteiro** porque elas **apontam uma posição de memória**. Para saber o valor armazenado no endereço apontado, é preciso usar o operador de desreferenciamento *****.

Nas próximas aulas veremos a utilidade deste conceito.

3. Ponteiros para Estruturas

É possível criar ponteiros para estruturas, sem problemas. Considere a estrutura abaixo:

```
struct produto {
        int id;
        char nome[60];
        float preco;
};
```

Para criar um ponteiro para ela do mesmo jeito que já vimos para os tipos de dados básicos:

```
produto *p;
produto q;
p = &q;
```

A grande diferença, nesse caso, é como fazemos para acessar os valores da estrutura. Quando usamos o ponteiro, temos de desreferenciá-lo. Observe no código a seguir.

```
produto *p;
produto q;
p.id = 10;
p = &q;
cout << p.id;
cout << (*q).id;
```

É muito importante colocar o (*q) entre parênteses, caso contrário não irá funcionar corretamente. Para evitar confusão, entretanto, existe um "substituto" para o "." Que vale para o caso de ponteiro, que é o **->** :

```
produto *p;
produto q;
p.id = 10;
p = &q;
cout << p.id;
cout << (*q).id;
cout << q->id;          // q->id  Substitui a forma (*q).id
```