# Linguagem C++ - Notas de Aula

Prof. Armando Luiz N. Delgado
baseado em revisão sobre material de Profª Carmem Hara e Prof. Wagner Zola

Março 2018

## Parte I
# Programação Básica em C++

Estas notas de aula apresentam os conceitos básicos da Linguagem C++ e se propõe a abordar apenas o que é importante para a compreensão básica de programas de computadores. Assim, conceitos de C++ como objetos, classes, templates e outros conceitos relacionados à programação orientada a objetos não são abordados aqui.

# 1 Programas C++

Essencialmente, um programa C++ consiste de uma ou mais partes chamadas funções[1]. Além disso, um programa em C++ deve definir pelo menos uma função chamada **main**. Esta função marca o ponto de início de execução do programa.

Programas C++ tem a seguinte estrutura geral:

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

definição de constantes

funções

int main()
{
   declaração de variáveis
   ....
   sentenças
   ....
}
```

## 1.1 Sentenças: simples e compostas

Cada instrução em C++ é chamada de sentença. **Sentenças simples** são terminadas com um ponto e vírgula. Usando chaves, podemos agrupar sentenças em blocos, chamados de **sentenças compostas**.

Exemplos de sentenças incluem:

- Simples:

```
x = 3;
```

- Composta:

```
{
   i = 3;

   cout << i << endl;

   i = i + 1;
}
```

O corpo da função main() é um exemplo de sentença composta.

## 1.2 Variáveis em C++

Uma variável é uma informação que você pode usar dentro de um programa C++ . Esta informação está associada com um lugar específico da memória (isso é feito pelo compilador). O **nome** da variável e o endereço da memória onde a informação está armazenada estão associados. O nome e o endereço não mudam. Mas, o valor da informação pode mudar (o valor do que está dentro da caixa pode mudar, embora o

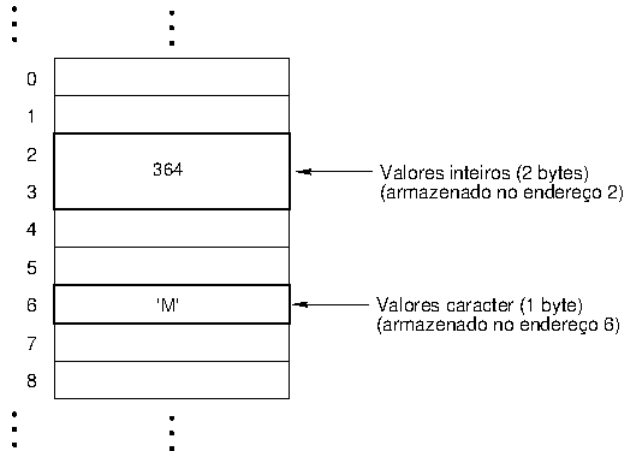
[1]Na verdade, um programa C++ é composto pela definição de funções e de elementos estruturais denominados *classes*. Estes são tema de estudo em cursos avançados de programação orientada a objetos.

tipo seja sempre o mesmo). Cada variável tem um tipo associado. Alguns tipos de variáveis que discutiremos incluem int, char e float.

Cada variável usa uma determinada quantidade de armazenamento em memória. A maneira como sabemos quantos bytes são utilizados é pelo tipo da variável. Variáveis do mesmo tipo utilizam o mesmo número de bytes, não interessando qual o valor que a variável armazena.

Um dos tipos utilizados para armazanar números é o int. Ele é usado para armazenar números inteiros.

Outro tipo é o char, usado para armazenar caracteres. Um caracter é um símbolo (uma letra do alfabeto, um dígito, um símbolo de pontuação, etc). Um char é armazenado em 1 byte de memória. Cada caracter é associado com um valor entre 0 e 255. O compilador C++ faz a tradução para você, portanto você não precisa saber estes números. Em C++ , um caracter é representado entre apóstrofes (’). Por exemplo, ’C’, ’a’, ’5’, ’$’. Note que ’5’ é um caracter, e não o inteiro 5.




A figura acima mostra como um int e um char são armazenados na memória.

Outro tipo existente é o float, usado para armazenar números reais (números com o ponto decimal). Este números são armazenados em duas partes: a *mantissa* e o *expoente*. Eles são armazenados de uma maneira que se assemelha a notação exponencial. Por exemplo, o número $6.023 \times 10^{23}$ é escrito como $6.023e23$. Neste caso, a mantissa é 6.023 e o expoente 23.

Estes números são armazenados de uma forma padrão, tal que a mantissa tem apenas um dígito para a esquerda do ponto decimal. Desta forma, 3634.1 é escrito como 3.6341e3, e 0.0000341 é escrito 3.41e-5. Note também que a precisão é limitada pela mantissa. Somente os 6 dígitos mais significativos são armazenados. Em Code::Blocks um float ocupa 4 bytes de memória. Há muitos outros tipos ( short, long, double), que serão descritos no futuro.

## 1.3 Definição de Variável em C++

Se você usa variáveis no programa, você deve defini-las. Isto envolve especificar o tipo da variável e o seu nome. As regras para formar nomes de variáveis em C++ são:

- qualquer sequência de letras, digitos, e ’_’, MAS DEVE COMEÇAR com uma letra ou com ’_’. Por exemplo, hora_inicio, tempo, var1 são nomes de variáveis válidos, enquanto 3horas, total$ e azul-claro não são nomes válidos;
- Maiúsculas $\neq$ Minúsculas;
- Não são permitidos nomes ou palavras reservadas da linguagem.

É sempre uma boa idéia ter certas regras (para você mesmo) para nomear variáveis para tornar o programa mais legível:

- Dê nomes significativos as variáveis (mas não muito longos);

| | | | | | |
|---|---|---|---|---|---|
| auto | break | case | char | const | continue |
| default | do | double | else | enum | extern |
| float | for | goto | if | int | long |
| main | register | return | short | signed | sizeof |
| static | struct | switch | typedef | union | unsigned |
| void | volatile | while | | | |

Tabela 1: Palavras Reservadas da Linguagem C++

- Use nomes de variáveis do tipo i, j, k somente para variáveis tipo contadores;
- Pode-se usar letras maiúsculas ou '_' para juntar palavras. Por exemplo, horaInicio ou hora_inicio. Use o que você preferir, mas SEJA **CONSISTENTE** em sua escolha.

Os tipos básicos de dados existentes em C++ são:

| Tipo de Dado | Bits | Faixa de Valores |
|---|---|---|
| **char** | 8 | -128 a 127 |
| **bool** | 8 | true ou false |
| **int** | 32 | -2.147.483.647 a 2.147.483.647 |
| **float** | 32 | 7 dígitos significativos |
| **double** | 64 | 15 dígitos significativos |

Abaixo está um exemplo de um programa com diversas definições de variáveis:

```
int main()
{
    int pera;
    char qualidade;
    float peso;

    pera = 3;
    qualidade = 'A';
    peso = 0.653;
    ...
}
```

Quando variáveis são definidas, elas não possuem valores ainda. Nós damos valores às variáveis usando o *operador de atribuição* (=). Variáveis também podem ser inicializadas para conter valores quando são definidas. Usando esta forma, o program acima ficaria:

```
int main()
{
    int pera = 3;
    char qualidade = 'A';
    float peso = 0.653;

    ...
}
```

Para resumir: quando um programa é executado, **uma variável** é associada com:

- **um tipo:** diz quantos bytes a variável ocupa, e como ela deve ser interpretada.
- **um nome:** um identificador.
- **um endereço:** o endereço do byte menos significativo do local da memória associado a variável.
- **um valor:** o conteúdo real dos bytes associados com a variável; o valor da variável depende do tipo da variável; a definição da variável não dá valor a variável; o valor é dado pelo operador de atribuição, ou usando a função cin. Nós veremos mais tarde que a função cin atribui a uma variável um valor digitado no teclado.
- Em C++ , nomes de variáveis devem ser declarados antes de serem usados. Se não for declarado, ocorrerá um erro de compilação.
- Devem ser dados valores às variáveis antes que sejam utilizadas. Se você tentar utilizar a variável antes de especificar o seu valor, você obterá "lixo" (o que quer que esteja armazenado no endereço da variável na memória quando o programa começa sua execução), culminando com falha na execução do programa.

## 1.4 Constantes

Em C++ , além de variáveis, nós podemos usar também números ou caracteres cujos valores não mudam. Eles são chamados de constantes. Constantes não são associados a lugares na memória.

Assim como variáveis, constantes também têm tipos. Uma constante pode ser do tipo int, char, etc. Você nao tem que declarar constantes, e pode utilizá-las diretamente (o compilador reconhece o tipo pela maneira que são escritos). Por exemplo, 2 é do tipo int, e 2.0 é do tipo double. Por convenção, todas as constantes reais são do tipo double.

## 1.5 Caracteres Constantes

Um constante caracter é escrita entre apóstrofes, como em 'A'. Todas as letras, números e símbolos que podem ser impressos são escritos desta forma em C++ . Às vezes precisamos de caracteres que não podem ser impressos, por exemplo, o caracter de "nova linha", que não tem uma tecla específica no teclado. Neste caso, usa-se *caracteres de escape*. Tais caracteres são escritos não somente como um símbolo entre apóstrofes, mas como um sequência de caracteres entre apóstrofes. Por exemplo, `'\n'` é o caracter para nova linha (uma sequência que inicia com a barra invertida é chamada de *sequência de escape*). Se quisermos representar o caracter de barra invertida, temos que escrever `'\\'`. Note que `\n` é o caracter de nova linha - embora use-se dois símbolos para representá-lo. A barra invertida é chamada de *escape*. Ele diz ao compilador que o n que segue não é a letra n, mas que a sequência completa de caracteres deve ser interpretada como o caracter de "nova linha".

Cada caracter constante tem um valor inteiro igual ao seu valor numérico do seu código ASCII. Por exemplo, considere a constante 'A', que tem código ASCII 65, e 'B' que tem código 66. Nós podemos usar a expressão 'A' + 1. O resultado é o valor 66. E se o tipo da expressão resultante for char, então o resultado da expressão é 'B'.

## 1.6 Entrada e Saída

Se quisermos que um programa C++ mostre alguns resultados, ou se quisermos que o programa peça ao usuário que entre com alguma informação, nós podemos usar os elementos cout e cin[2]. Se você quiser usar estes elementos em seu programa, voce deve incluir as seguintes linhas no início do seu código fonte:

[2] cout e cin são na verdade objetos das classes ostream e istream. Mas este detalhe não é abordado nestas notas de aula. Será visto apenas o uso destes objetos como primitivas simples para Entrada e Saída de dados.

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

Isto faz com que o arquivo *header* chamado `iostream` seja incluído no seu arquivo fonte durante a compilação. Este arquivo contém definições de diversas funções e classes (por exemplo, `cout` e `cin`). Ele declara ao compilador o nome das funções e algumas informações adicionais necessárias para que as instruções sejam executadas corretamente.

### 1.6.1 Exibindo informações na tela: `cout`

`cout` pode ser utilizado para imprimir mensagens e valores em uma variedade de formatos. Por enquanto, `cout` é melhor descrito através de exemplos.

```
cout << "Alô todo mundo" << endl;
```

Imprimirá `Alô todo mundo` em uma linha na tela do computador. O valor `endl` representa a mudança de linha.

Para o comando `cout` fazer o que deve, nós devemos especificar o que será impresso. Nós devemos dar ao comando o que chamamos de *argumentos*. No exemplo acima, `Alô todo mundo` e `endl` são argumentos para `cout`.

Os argumentos de `cout` podem ser uma variável, uma expressão ou um *string* (uma série de caracteres entre aspas (")).

Nós também podemos colocar caracteres de *escape* no *string* para imprimir caracteres especiais. Por exemplo, colocando `\n` no *string* causa que o restante do string seja impresso na linha seguinte. Outros caracteres de escape serão apresentados no futuro.

Considere o seguinte programa:

```
#include <iostream>
using namespace std;
#define PRECO 1.99

int main()
{
      int pera = 3;
      char qualidade = 'A';
      float peso = 2.5;

      cout << "Existem " << pera << "peras de qualidade " << qualidade
            << "pesando " << peso << "quilos." << endl;
      cout << "O preco por quilo eh R$" << PRECO
            << ", o total eh R$" << peso * PRECO << endl;
}
```

A saída do programa será:

```
Existem 3 peras de qualidade A pesando 2.5 quilos.
O preco por quilo eh 1.99, o total eh 4.975
```

A linha `#define PRECO 1.99` no início do programa define uma *macro*. Ou seja, definimos que `PRECO` é um sinônimo para `1.99` e, portanto, toda ocorrência de `PRECO` no programa é substituído por `1.99` antes que ele seja compilado.

### 1.6.2 Lendo informação: cin

cin pode ser usado para ler valores digitados no teclado.

Considere o seguinte programa:

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

int main()
{
      int idade;

      cout << "Entre sua idade: ";
      cin >> idade

      cout << "Voce tem " << idade << "anos." << endl;
}
```

Este programa mostrará no monitor: Entre sua idade: e aguardará que um número seja digitado e a tecla ENTER. Depois disso, a variável idade conterá o valor digitado pelo usuário.

Mais de um valor pode ser lido por um mesmo cin. Considere o seguinte exemplo:

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

int main()
{
      int dia, mes, ano;

      cout << "Entre com a data do seu aniversario (dd mm aa): ";
      cin >> dia >> mes >> ano;

      cout << "Voce nasceu em " << dia << "/" << mes << "/" << ano << endl;
}
```

Este exemplo funciona exatamente como o exemplo anterior. Um único cin lê os 3 números quando estes números são separados por espaços (espaços em branco, tabulação, novas linhas). Então você pode teclar ENTER depois de cada número, ou colocar espaços ou tabulações entre os números. Os espaços são ignorados pelo cin.

## 1.7 Algoritmo X Programa

```
ALGORITMO PERIMETRO_AREA

  /* Calcula o perímetro e a area de uma circunferencia
     de raio R (fornecido pelo usuario) */

  /* Definir variaveis */
        int Raio;
        float Perim, Area, PI;
        PI = 3.14159;
```

```
     /* Obter Raio da circunferencia */
           Escreva("Entre com o valor do raio:");
           Leia(Raio);

     /* Calcular Perimetro do Circulo */
           Perim = 2 * PI * Raio;

     /* Calcular Area da Circunferencia */
           Area = PI * Raio ** 2;

     /* Exibir Resultados */
           Escreva("O perimetro da circunferencia de raio", Raio, "eh", Perim);
           Escreva("e a area eh ",Area);

     /* Terminar Programa */

   FIM_ALGORITMO PERIMETRO_AREA
```

**Programa em C++**

```
/* programa que calcula o perímetro e a área de uma
   circunferência de raio R (fornecido pelo usuário) */

#include <iostream>  /* inclui diretivas  de     entrada-saída */
#include <cmath>  /*  inclui diretivas das funções matemáticas */

using namespace std;

#define  PI  3.14159

int main( )
{
     /* Definir variaveis */
        int Raio;
        float  Perim, Area;

     /* Obter Raio da circunferencia */
        cout << "Entre com o valor do raio: ";
        cin >> Raio;

     /* Calcular Perimetro do Circulo */
      Perim = 2 * PI * Raio;

     /* Calcular Area da Circunferencia */
        Area  = PI * pow(Raio, 2);

     /* Exibir Resultados */
        cout << "O perimetro da circunferencia de raio " << Raio
             << " eh  " << Perim << endl;
        cout << "e a area eh " << Area << endl;
```

```
}
```

# 2 Operações Aritméticas e Expressões. Operações Relacionais.

## 2.1 Operações Aritméticas

Em C++ , nós podemos executar operações aritméticas usando variáveis e constantes. Algumas operações mais comuns são:

**+** adição

**-** subtração

***** multiplicação

**/** divisão

**%** resto (módulo)

Estas operações podem ser usadas como mostram os exemplos abaixo, assumindo que as variáveis necessárias já estão declaradas:

```
celsius = (fahrenheit - 32) * 5.0 / 9.0;

forca =  massa * aceleracao;

i = i + 1;
```

### 2.1.1 Precedência de Operadores

Em C++ , assim como em álgebra, há uma ordem de precedência de operadores.

Assim, em $(2 + x)(3x^2 + 1)$, expressões em parêntesis são avaliadas primeiro, seguidos por exponenciação, multiplicação, divisão, adição e subtração.

Da mesma forma, em C++ , expressões entre parêntesis são executadas primeiro, seguidas de `*`, `/` e `%` (que tem todos a mesma precedência), seguido de + e − (ambos com a mesma precedência).

Quando operações adjacentes têm a mesma precedência, elas são associadas da esquerda para a direita. Assim, `a * b / c * d % e` é o mesmo que `(((a * b) / c) * d) % e)`.

### 2.1.2 A Operação de Resto (%)

Esta operação é usada quando queremos encontrar o resto da divisão de dois inteiros. Por exemplo, 22 dividido por 5 é 4, com resto 2 ($4 \times 5 + 2 = 22$).

Em C++ , a expressão `22 % 5` terá valor `2`.

Note que `%` só pode ser utilizados entre dois inteiros. Usando ele com um operando do tipo `float` causa um erro de compilação (como em `22.3 % 5`).

### 2.1.3 Expressões e Variáveis

Expressões aritméticas podem ser usadas na maior parte dos lugares em que uma variável pode ser usada.

O exemplo seguinte é válido:

```
int raio = 3 * 5 + 1;

cout << "circunferencia = " << 2 * 3.14 * raio << endl;
```

Exemplos de lugares onde uma expressão aritmética NÃO pode ser usada incluem:

```
int yucky + 2 = 5;

cin >> oops * 5;
```

Este exemplo é ilegal e causará erro de compilação.

## 2.2 Operadores Relacionais

Em C++ , há operadores que podem ser usados para comparar expressões: os operadores relacionais.
Há seis operadores relacionais em C++ :

$<$ menor que

$>$ maior que

$<=$ menor ou igual que ($\leq$)

$>=$ maior ou igual que ($\geq$)

**==** igual a

**!=** não igual a ($\neq$)

Os resultados deste operadores é `0` (correspondendo a *falso*), ou `1` (correspondendo a *verdadeiro*). Valores como esses são chamados valores *booleanos*. Algumas linguagens de programação como Pascal tem um tipo de variável distinto para valores booleanos. Este não é o caso do C++ , onde valores booleanos são armazenados como variáveis numéricas tais como o `int`.

Considere o seguinte programa:

```
int main()
{
    int idade;

    idade = 17;
    cout << "Pode tirar carteira de motorista? " << (idade >= 18) << endl;
    idade = 35;
    cout << "Pode tirar carteira de motorista? " << (idade >= 18) << endl;
}
```

A saída deste programa será:

```
Pode tirar carteira de motorista? 0
Pode tirar carteira de motorista? 1
```

Na primeira linha, `idade` é `17`. Logo, `17 >= 18` é falso, que é `0`.
Depois disso, `idade` é `35`. Logo, `35 >= 18` é verdadeiro, que é `1`.

Note também que o operador de igualdade é escrito com "sinais de igual duplo", ==, não =. Tenha cuidado com esta diferença, já que colocar = no lugar de == não é um erro sintático (não gera erro de compilação), e não significa o que você espera.

### 2.2.1 Precedência dos operadores relacionais

Operadores aritméticos tem precedência maior que os operadores relacionais. Por exemplo, a expressão `3 + 5 < 6 * 2` é o mesmo que `(3 + 5) < (6 * 2)`.

Se por alguma razão você quer que o resultado do uma operação relacional em uma expressão aritmética, é necessário usar parêntesis. Por exemplo, a expressão `score + (score == 0)` será sempre igual ao valor de `score`, exceto quando o valor de `score` seja `0`. Neste caso, o valor da expressão é `1` (porque `(score == 0)` é igual a 1).

Uma observação sobre valores booleanos – embora você possa assumir que o valor de uma operação relacional é `0` ou `1` em C++ , **qualquer valor diferente de zero é considerado verdadeiro**. Falaremos sobre isso mais tarde durante o curso.

## 2.3 Revisão de Expressões:

O que é impresso pelos dois programas abaixo?

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

int main() {
  int score = 5;

  cout << 5 + 10 * 5 % 6;          // 7
  cout << 10 / 4;                  // 2
  cout << 10.0 / 4.0;              // 2.5
  cout << 'A' + 1                  // B
  cout << score + (score == 0);    // 5
}
```

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

int main() {
  int n1, n2, n3;

  cout << "Entre com um numero inteiro: ";
  cin >> n1;
  n2 = n1 / 5;
  n3 = n2 % 5 * 7;
  cout << n2 << " " << n3 << " " << (n2 != n3 + 21) << endl;
}
```

Como é a seguinte expressão completamente parentizada ?

```
a * b / c + 30 >= 45 + d * 3 ++e == 10
```

## 2.4 Exemplo de programas

**Exemplo 1:** escreva um programa que leia um número inteiro e imprima 0 se o número for par e 1 se o número for ímpar.

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

int main() {
  int numero;

  cout << "Entre com um numero inteiro: ";
  cin >> numero;
  cout << "\nPar? " << numero % 2 << endl;
}
```

**Exemplo 2:** escreva um programa que leia 3 números inteiros e calcule a soma, média, e produto.

```
#include <iostream>
#include <iomanip>     // necessario para usar setw() e setf() em cout
using namespace std;

int main() {
  int n1, n2, n3;
  int soma;

  cout <<  "Entre com 3 numeros inteiros: ";
  cin >> n1 >> n2 >> n3;
  soma = n1 + n2 + n3;
  cout <<  "Soma = " << soma << endl;
  cout.setf (ios::fixed | ios::showpoint); // reais em ponto fixo
  cout.precision(2);                       // 2 casa decimais

  // setw(8) fixa tamanho da representação em 8 digitos
  cout << "Media = " << setw(8) << soma / 3.0 << endl;
  cout <<  "Produto = " << (unsigned) n1 * n2 * n3 << endl;
}
```

## 2.5 Precedência e associatividade de operadores

| Operador | Associatividade |
|---|---|
| `()` | esquerda para direita |
| `-` (unários) | direita para esquerda |
| `*  /  %` | esquerda para direita |
| `+  -` | esquerda para direita |
| `<  <=  >  >=` | esquerda para direita |
| `==  !=` | esquerda para direita |

# 3 Expressões como valores

Em C++ , todas as expressões são avaliadas. O resultado da avaliação é um valor e pode ser usado em quaisquer lugares.

## 3.1 Expressões aritméticas, relacionais e lógicas

Como você já sabe, expressões usando operadores aritméticos, relacionais e lógicos[3] são avaliados. O valor resultante é um número. Para os operadores relacionais e lógicos, este número pode ser 0 (que significa falso) ou 1 (que significa verdadeiro). Por exemplo:

| | |
|---|---|
| `3 + 5 * 4 % (2 + 8)` | tem valor 3; |
| `3 < 5` | tem valor 1; |
| `x + 1` | tem valor igual ao valor da variável `x` mais um; |
| `(x < 1) \|\| (x > 4)` | tem valor 1 quando o valor da variável `x` é fora do intervalo [1,4], e 0 quando `x` está dentro do intervalo. |

## 3.2 Expressões envolvendo o operador de atribuição (=)

O formato do operador de atribuição é:

$$lvalue = expressao \tag{1}$$

Um $lvalue$ (do inglês "left-hand-side value" - valor a esquerda) é um valor que se refere a um endereço na memória do computador. Até agora, o único "lvalue" válido visto no curso é o nome de uma variável. A maneira que a atribuição funciona é a seguinte: a expressão do lado direito é avaliada, e o valor é copiado para o endereço da memória associada ao "lvalue". O tipo do objeto do "lvalue" determina como o valor da $expressao$ é armazenada na memória.

Expressões de atribuição, assim como expressões, têm valor. O valor de uma expressão de atribuição é dado pelo valor da expressão do lado direito do =. Por exemplo:

| | |
|---|---|
| `x = 3` | tem valor `3`; |
| `x = y+1` | tem o valor da expressão `y+1`. |

Como consequência do fato que atribuições serem expressões que são associadas da direita para esquerda, podemos escrever sentenças como:

```
i = j = k = 0;
```

Que, usando parênteses, é equivalente a `i = (j = (k = 0))`. Ou seja, primeiro o valor 0 é atribuído a `k`, o valor de `k = 0` (que é zero) é atribuído a `j` e o valor de `j = (k = 0)` (que também é zero) é atribuído a `i`.

Uma característica muito peculiar de C++ é que expressões de atribuição podem ser usados em qualquer lugar que um valor pode ser usado. Porém você deve saber que usá-lo dentro de outros comandos produz um efeito colateral que é alterar o valor da variável na memória. Portanto, a execução de:

```
int quadrado, n = 2;

cout << "Quadrado de " << n << " eh menor que 50? " << ((quadrado = n * n) <
```

[3] Operadores lógicos `&&` e `||` serão vistos na próxima aula.

causa não apenas que o valor `4` seja impresso, como a avaliação da expressão relacional dentro do cout faz com que o número `4` seja copiado para o endereço de memória associado com a variável `quadrado`. Note que é necessário usar parênteses em `quadrado = n * n` já que = tem menor precedência que o operador relacional <.

Agora compare o exemplo anterior com o próximo, no qual o valor `4` é impresso, mas sem nenhum efeito colateral:

```
int quadrado, n = 2;

cout << "Quadrado de " << n << " eh menor que 50? " << (n * n < 50) << endl;
```

Note que agora não há necessidade de parênteses para a expressão `n * n` porque `*` tem maior precedência que o operador relacional <.

# 4 Ordem sequencial de execução de sentenças o comando condicional: `if` e `if - else`

A execução de um programa C++ começa com a função `main()`. Em todos os exemplos que vimos até este momento, sentenças são executadas sequencialmente. A ordem sequencial de execução de senteças pode ser alterada se certas condições forem satisfeitas durante a execução do programa. Isto é chamado *desvio condicional*.

Todas as linguagens de programação oferecem comandos para o desvio condicional. O mais simples é a senteça `if`. Em C++ , ele tem o formato:

`if` ($expressao$)
$corpododesvio$

O *corpo do desvio*, por sua vez, pode ser uma sentença simples ou composta (veja Seção 1.1).

Quando uma sentença `if` é encontrada em um programa,

1. O teste na $expressao$ em parênteses é avaliada.
2. Se o valor da expressão de teste for DIFERENTE de zero, as sentenças que compõem o *corpo do desvio* que segue a expressão de teste são executadas.

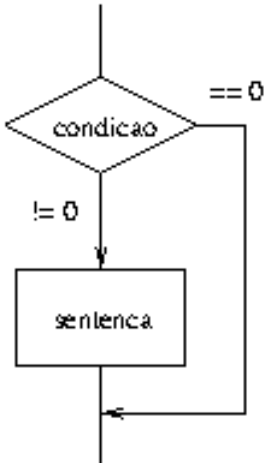



Figura 1: O comando `if`

Considere o seguinte exemplo que converte uma fração digitada pelo usuário (numerador e denominador) em decimal e imprime o resultado:

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

int main( ){

  int a, b;

  cout << "Entre com uma  fracao (numerador and denominador): ";
  cin >> a >> b;

  cout << "A fracao em decimal eh " << 1.0 * a / b << endl;
}
```

No exemplo acima, escrevemos `1.0 * a / b`, já que `a` e `b` são do tipo `int`, e portanto `a / b` é uma divisão de inteiros e a parte fracional do resultado seria truncado, o que certamente não é o que desejamos.

Voce vê algo errado neste programa ? Uma coisa a ser notada é que se o usuário digitar um denominador igual a `0`, nós teremos um erro de execução, já que o programa tentaria executar uma divisão por zero. O que é necessário fazer é testar se o denominador é igual a zero e dividir só no caso dele for diferente de zero. Poderíamos reescrever o programa acima da seguinte forma:

**Exemplo 1:**

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

int main( ){

  int a, b;

  cout << "Entre com uma  fracao (numerador e denominador): ";
  cin >> a >> b;

  if (b != 0)
    cout << "A fracao em decimal eh " << 1.0 * a / b << endl;
}
```

**Exemplo 2:** Programa que lê dois números e ordena o par caso o primeiro número digitado for maior que o segundo.

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

int main( ){

  int num1, num2, aux;

  cout << "Entre com dois numeros inteiros: ";
  cin >> num1 >> num2;

  if (num1 > num2) {
    aux = num1;
    num1 = num2;
    num2 = aux;
    cout << "Trocou \n";
  }

  cout << "Os numeros ordenados: " << num1 << " " << num2 << endl;

}
```

O programa do Exemplo 1 acima ficaria ainda melhor se ao invés de não fazer nada no caso do denominador ser zero, imprimirmos uma mensagem de erro ao usuário, explicando o que há de errado.

A sentença em C++ que permite fazermos isso é o `if - else`. O formato do `if-else` é:

```
if (expressao)
        sentenca_1
else
        sentenca_2
```

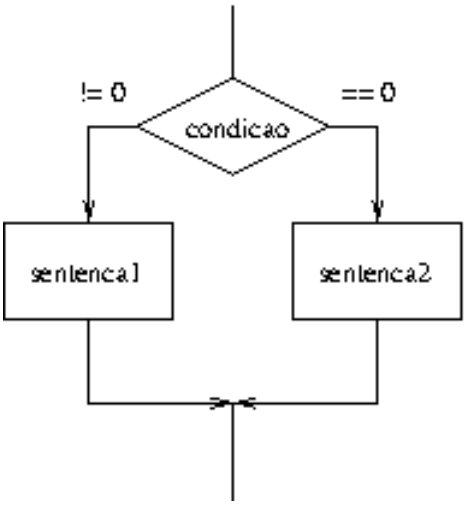



Figura 2: O comando `if-else`

Primeiro, a *expressao* (que usualmente chamamos de condição) é avaliada. Caso a condição seja verdadeira (o que é equivalente a dizer que o valor é diferente de zero), entao a $sentenca_1$ é executada. Caso contrário, a $sentenca_2$ é executada.

Note que uma *sentença* pode ser simples ou composta. Se você quiser agrupar diversas sentenças para serem executadas, você pode colocá-las entre chaves (`{` e `}`).

Por hora, vamos continuar com nosso exemplo simples e torná-lo mais explicativo:

**Exemplo 3:**

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

int main( ){

  int a, b;

  cout << "Entre com uma fracao (numerador and denominador): ";
  cin >> a >> b;

  if (b != 0)
    cout << "A fracao decimal eh " << 1.0 * a / b << endl;
  else
    cout << "Erro: denominador zero!\n";
}
```

**Exemplo 4:** Considere agora o exemplo já visto que pede que um usuário entre com um número e verifique se o número é par. Porém agora, queremos que o programa imprima "o numero e par" ou "o numero e impar".

```
#include <iostream>
```

```
using namespace std;

int main( ){

  int num;

  // obtem um numero do usuario
  cout << "Entre com um inteiro: ";
  cin >> num;

  // imprime uma mensagem dizendo se o numero e par ou impar
  if (num % 2 == 0)
    cout << "O numero eh par.\n";
  else
    cout << "O numero eh impar.\n";

}
```

## 4.1 Um erro comum

É muito frequente utilizar o operador relacional == em expressões condicionais da sentença `if`. Por exemplo:

```
int  saldo = 2000;

if (saldo == 1)
   cout << "Voce esta quebrado! " << endl;
else
   cout << "Seu saldo eh " << saldo << endl;
```

Como a sentença `saldo = 2000` inicializa o valor da variável `saldo` com `2000`, a expressão `saldo == 1` tem valor `0`. Portanto, a senteça que segue o `else` será executada, e a mensagem

```
Seu saldo e 2000
```

será impressa.

Agora, suponha que, devido a um erro, você tenha colocado = ao invés de ==:

```
int saldo = 2000;

if (saldo = 1)
   cout << "Voce esta quebrado! " << endl;
else
   cout << "Seu saldo eh " << saldo << endl;
```

Agora, a expressão `saldo = 1` tem valor `1`. Portanto, a sentença que segue o `if` será executada, e a mensagem

```
Voce esta quebrado!
```

será impressa. Além disso, a atribuição causará um efeito colateral, e alterará o valor de `saldo` para `1`.

Tal uso do operador de atribuição não é ilegal, e não será detectado pelo compilador como erro. Portanto, *tome cuidado com o uso de atribuição no lugar de igualdade*. Tal erro é muito comum, e não é fácil de achar.

Como regra geral, NÃO utilize atribuições dentro de outras sentenças.

# 5 Aninhando senteças `if` e `if-else`

Como era de se esperar, é possível colocar uma sentença condicional dentro de outra. Por exemplo, se quisermos imprimir uma mensagem apropriada caso um número seja positivo ou negativo e par ou ímpar, nós poderíamos escrever o seguinte:

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

int main( ){

  int num;

  // Obtem um numero do usuario
  cout << "Entre com um inteiro: ";
  cin >> num;

  // Imprime uma mensagem dizendo se o numero e positivo ou negativo,
  // positivo ou negativo.
  if (num >= 0) {
    if (num % 2 == 0)
      cout << "O numero e par e positivo\n";
    else
      cout << "O numero e impar e positivo\n";
  }
  else {
    if (num % 2 == 0)
      cout << "O numero e par e negativo\n";
    else
      cout << "O numero e impar e negativo\n";
  }
}
```

## 5.1 A ambigüidade do `else`

O aninhamento de sentenças `if-else` sem usar chaves (`{` e `}`) para delimitar o bloco de senteças a ser executado pode trazer efeitos indesejados.

Há uma regra simples para determinar qual `if` está associado a qual `else`.

**Regra de associação:** Um `else` está associado com a última ocorrência do `if` sem `else`.

O exemplo seguinte está errado porque associa o `else` ao `if` "incorreto":

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

int main( ){
```

```
  int num;

  // Obtem um numero do usuario
  cout << "Entre com o numero de peras: ";
  cin >> num;

  // Imprime uma mensagem dizendo se o numero de peras e 0 ou 1
  //  (*** isto esta' errado !!  ***)
  if (num != 0)
    if (num == 1)
      cout << "Voce tem uma pera.\n";
  else
    cout << "Voce nao tem nenhuma pera.\n";
}
```

Neste exemplo, o `if` tem o seguinte significado, segundo a regra de associação:

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

int main( ){

  int num;

  // Obtem um numero do usuario
  cout << "Entre com o numero de peras: ";
  cin >> num;

  // Como a sentenca if e' vista pelo compilador
  if (num != 0)
    if (num == 1)
      cout << "Voce tem uma pera.\n";
    else
      cout << "Voce nao tem nenhuma pera.\n";
}
```

Para evitar este problema, chaves (`{` e `}`) devem ser usadas para tirar a ambiguidade. O exemplo abaixo mostra como as chaves podem ser inseridas para corrigir o programa acima.

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

int main( ){

  int num;

  // Obtem um numero do usuario
  cout << "Entre com o numero de peras: ";
  cin >> num;
```

```
  // Como corrigir o problema (este programa funciona)
  if (num != 0) {
    if (num == 1)
      cout << "Voce tem uma pera.\n";
  }
  else
    cout << "Voce nao tem nenhuma pera.\n";
}
```

**Exercício 1:** Faça um programa que leia 3 números e imprima o maior.

# 6 Operadores Lógicos

Todos os programas até agora consideraram `if` com condições de teste simples. Alguns exemplos de testes simples: `b != 0`, `contador <= 5`. Estas expressões testam uma condição. Portanto, quando mais de uma condição precisa ser testada, precisamos usar sentenças `if` e `if-else` aninhadas.

A linguagem C++ , assim como a maioria das linguagens de programação de alto nível suportam *operadores lógicos* que podem ser usados para criar *operações lógicas* mais complexas, combinando condições simples. O valor de uma expressão lógica é ou VERDADEIRO ou FALSO. Lembre que não há constantes lógicas VERDADEIRO e FALSO em C++ ; em expressões lógicas `0` é interpretado como FALSO, e qualquer valor diferente de zero é interpretado como VERDADEIRO.

Os operadores lógicos são

`!` NÃO lógico, operação de negação (operador unário)
`&&` E lógico, conjunção (operador binário)
`||` OU lógico, disjunção (operador binário).

Por exemplo, se quisermos testar se um número `num` é positivo e par, e imprimir uma mensagem como no exemplo anterior, podemos escrever:

```
if (num >= 0)
   if (num % 2 == 0)
      cout << "Numero par nao negativo." << endl;
```

Com os operadores lógicos isso pode ser simplificado:

```
if ((num>=0) && (num%2 == 0))
     cout << "Numero par nao negativo." << endl;
```

A operação de negação, `!`, pode ser usado da seguinte forma:

`!`*expressão lógica*: O valor é a negação lógica da expressão dada. Por exemplo:
`!0` é `1`
`!1` é `0`

Nós podemos usar o operador de negação lógica e escrever o exemplo acima como:

```
if (num>0 && !(num%2))
      cout << "Numero par nao negativo." << endl;
```

Os dois operadores binários operam sobre duas expressões lógicas e tem o valor `1` (verdadeiro) or `0` (falso). Os exemplos abaixo mostram o seu uso:

`a==0 && b==0` (verdadeiro se ambos `a == 0` e `b == 0`, portanto se `a` e `b` são `0`)
`a==0 || b==0` (verdadeiro se pelo menos uma das variáveis `a` or `b` for `0`)

**Uma expressão usando `&&` é verdadeira somente se ambos os operadores forem verdadeiros (não zero).**

**Uma expressão usando `||` é falsa somente se ambos os operadores forem falsos (zero).**

Verifique na Tabela 2 o resultado do uso de operadores lógicos:

| $expr_1$ | $expr_2$ | $expr_1$ `&&` $expr_2$ | $expr_1$ `||` $expr_2$ |
|---|---|---|---|
| $verdadeiro$ | $verdadeiro$ | $verdadeiro$ | $verdadeiro$ |
| $verdadeiro$ | $falso$ | $falso$ | $verdadeiro$ |
| $falso$ | $verdadeiro$ | $falso$ | $verdadeiro$ |
| $falso$ | $falso$ | $falso$ | $falso$ |

Tabela 2: Resultado de uso de Operadores Lógicos

A precedência do operador de negação lógica é a mais alta (no mesmo nível que o "-" unário). A precedência dos operadores lógicos binários é menor que a dos operadores relacionais, e mais alta que a operação de atribuição. O `&&` tem precedência mais alta que o `||`, e ambos associam da esquerda para a direita (como os operadores aritméticos).

Como a precedência dos operadores lógicos é menor que a dos operadores relacionais, não é necessário usar parênteses em expressões como:

```
x >= 3 && x <= 50
x == 1 || x == 2 || x == 3
```

A Tabela 3 mostra o quadro completo de precedência de operadores aritméticos, relacionais e lógicos.

| Operador | Associatividade |
|---|---|
| `()` | esquerda para direita |
| `!  -` (unários) | direita para esquerda |
| `*  /  %` | esquerda para direita |
| `+  -` | esquerda para direita |
| `<  <=  >  >=` | esquerda para direita |
| `==  !=` | esquerda para direita |
| `&&` | esquerda para direita |
| `||` | esquerda para direita |

Tabela 3: Precedência e associatividade de operadores

No próximo exemplo, o programa verifica se as três variáveis `lado1`, `lado2`, e `lado3`, podem ser lados de um triângulo reto. Nós usamos o fato que os três valores devem ser positivos, e que o quadrado de um dos lados deve ser igual a soma dos quadrados dos outros lados (Teorema de Pitágoras) para determinar se o triângulo é reto.

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

int main( ){

  int lado1, lado2, lado3;
  int s1, s2, s3;

  cout << "Entre com o tamanho dos lados do triangulo: ";
  cin >> lado1 >> lado2 >> lado3;

  // calcula o quadrado dos lados
  s1 = lado1*lado1;
  s2 = lado2*lado2;
  s3 = lado3*lado3;

  // testa a condicao para um triangulo reto

  if ( lado1>0 && lado2>0 && lado3 > 0 ) {
    if (s1==s2+s3 || s2==s1+s2 || s2==s1+s3) ) {
    cout << "Triangulo reto!\n";
  }
  else {
    cout << "Nao pode ser um triangulo!\n";
  }
}
```

Na utilização de expressões lógicas, as seguintes identidades são úteis. Elas são chamadas de *Lei de DeMorgan*:

`!(x && y)` é equivalente a `!x || !y`

e

`!(x || y)` é equivalente a `!x && !y`

# 7 Exemplos

## 7.1 IF - ELSE

Assuma as seguintes declaraçõoes de variáveis:

```
int x = 4;
int y = 8;
```

O que é impresso pelos seguintes programas ?

1.
```
if (y = 8)
    if (x = 5)
        cout <<  "a ";
    else
        cout <<  "b ";
cout <<  "c ";
cout <<  "d" << endl;

==> a c d
```

2. mude = para ==

```
==> b c d
```

3. altere o programa acima para produzir a seguinte saida:

   - Assuma `x = 5` e `y = 8`
     (a) `a`
     (b) `a d`
   - Assuma `x = 5` e `y = 7`
     (a) `b c d`

## 7.2 Operadores lógicos

O que é impresso pelas seguintes sentenças?

1. Assuma `x = 5` e `y = 8`.

```
if (x == 5 && y == 8)
    cout <<  "a" << endl;
else
    cout <<  "b" << endl;      ==> a
```

2. Assuma `x = 4` e `y = 8`.

```
if (x == 5 || y == 8)
    cout <<  "a" << endl;
else
    cout <<  "b" << endl;      ==> a


if !(x == 5 || y == 8)     // equiv. (x != 5 && y != 8)
    cout <<  "a" << endl;
else
    cout <<  "b" << endl;      ==> b


if !(x == 5 && y == 8)    // equiv. (x != 5 || y != 8)
    cout <<  "a" << endl;
else
    cout <<  "b" << endl;      ==> a
```

3. Precedência: `!  > && > ||`

```
if (x == 5 ||  y == 8 && z == 10)
```

equiv.

```
if (x == 5 || (y == 8 && z == 10))
```

## 8 A construção `else-if`

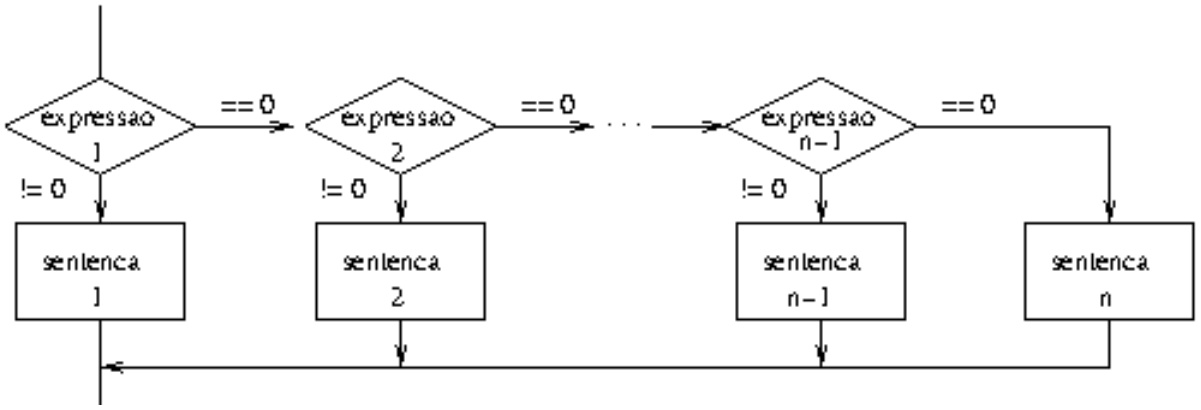
Embora ela não seja um tipo diferente de sentença, a seguinte construção é bastante comum para programar decisões entre diversas alternativas:

```
if (expressao1)
        sentenca1
else if (expressao2)
        sentenca2
else if (expressao3)
        sentenca3
⋮
else if (expressaon−1)
        sentencan−1
else
        sentencan
```

As expressões lógicas são avaliadas em ordem, começando com a $expressao_1$. Se uma das expressões for verdadeira, a sentença associada será executada. Se nenhuma for verdadeira, então a sentença, $sentenca_n$, do último `else` será executada como opção *default*. Se a opção *default* não for necessária, então a parte

```
else
        sentencan
```

pode ser removida.




**Exemplo 9:** O seguinte exemplo mostra um `else-if` de três opções. O programa lê dois números e diz se eles são iguais ou se o primeiro número é menor ou maior que o segundo.

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

int main( ){

  int num1, num2;

  // obtem 2 numeros do usuario
  cout << "Entre um numero: ";
```

```
  cin >> num1;
  cout << "Entre com um outro numero: ";
  cin >> num2;

  // mostra a mensagem de comparacao
  if (num1 == num2)
    cout << "Os numeros sao iguais\n";
  else if (num1 < num2)
    cout << "O primeiro numero e menor\n";
  else
    cout << "O primeiro numero e maior\n";
}
```

No programa acima, se `(num1 == num2)` for verdadeiro, então os números são iguais. Senão, é verificado se `(num1 < num2)`. Se esta condição for verdadeira, então o primeiro número é menor. Se isso não for verdadeiro, então a única opção restante é que o primeiro número é maior.

**Exemplo 10:** Este programa lê um número, um operador e um segundo número e realiza a operação correspondente entre os operandos dados.

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

int main( ){

  float num1, num2;
  char op;

  // obtem uma expressao do usuario
  cout << "Entre com numero operador numero\n";
  cin >> num1 >> op >> num2;

  // mostra o resultado da operacao
  if (op == '+')
    cout << " = " << setprecision(2) << num1 + num2;
  else if (op == '-')
    cout << " = " << setprecision(2) << num1 - num2;
  else if (op == '/')
    cout << " = " << setprecision(2) << num1 / num2;
  else if (op == '*')
    cout << " = " << setprecision(2) << num1 * num2;
  else
    cout << " Operador invalido.";
  cout << endl;
}
```

Exemplos da execução deste programa:

```
 Entre com numero operador numero:
 5 * 3.5
  = 17.50
```

```
Entre com numero operador numero:
10 + 0
 = 10.00

Entre com numero operador numero:
10 x 5.0
 Operador invalido.
```

# 9 Estruturas de Repetição

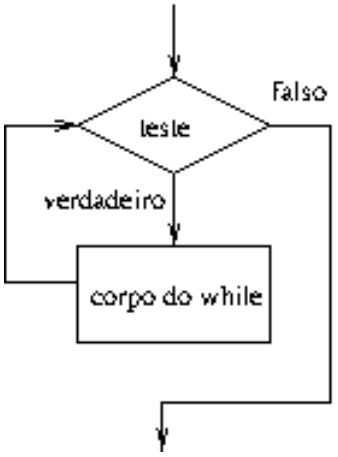
A linguagem C++ possui comandos para repetir uma sequência de instruções. Estas **estruturas de repetição**, também conhecidas como **laços** (do inglês ***loops***). Nesta seção veremos a estrutura while, sendo que as demais estruturas de repetição em C++ , for e do ... while serão vistas nas Seções 13.2 e 19.

## 9.1 O comando de repetição while

O comando de repetição while tem duas partes: a *expressão de teste* e o *corpo da repetição*. O formato do while é:

while (*expressão teste* )
    *corpo da repetição*

A *expressão teste* é inicialmente avaliada para verificar se o laço deve terminar. Caso a expressão seja verdadeira (isto é, diferente de 0 (zero)), o *corpo da repetição* é executado. Depois desta execução, o processo é repetido a partir da *expressão teste*. O *corpo do laço*, por sua vez, pode ser uma sentença simples ou composta (veja Seção 1.1).




O exemplo abaixo mostra o uso do comando de repetição while:

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

int contador;

contador = 0;
while( contador < 5 )
{
    cout << "contador = " << contador << endl;
    contador = contador + 1;
}

cout << "ACABOU !!!!" << endl;
```

**Saída:**

```
contador = 0
contador = 1
contador = 2
contador = 3
```

```
contador = 4
ACABOU !!!!
```

Neste exemplo, a expressão de teste é contador < 5, e o corpo do laço é a sentença cout.

Se examinarmos cuidadosamente este exemplo, veremos que a variável contador é inicializada com 0 (zero). Depois disso, a expressão de teste é verificada e, como 0 < 5 é verdadeiro, o corpo da repetição é executado. Assim, o programa imprime contador = 0, e incrementa contador. Em seguida, a expressão de teste é verificada novamente e todo o processo se repete até que contador seja 4 e contador = 4 seja impresso.

Depois disso, contador é incrementado para 5 e o teste é executado. Mas desta vez, 5 < 5 é falso, então a repetição não continua. A execução do programa continua na sentença que segue o laço (no caso, imprimir a frase ACABOU !!!).

Imediatamente após a execução do while, a variável contador tem valor 5.

O exemplo seguinte mostra um uso mais apropriado do comando while: Em situações onde o número de repetições não é conhecido antes do inicío do comando while.

**Exemplo 1:** Este programa pede números ao usuário até que a soma de todos os números digitados for pelo menos 20.

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

int main( ){

  int total, num;

  total = 0;
  while( total < 20 ) {
    cout << "Total = " << total << endl;
    cout << "Entre com um numero: ";
    cin >> num;

    total = total + num;
  }

  cout << "Final total = " << total << endl;
}
```

**Exemplo de saída:**

```
 Total = 0
 Entre com um numero: 3
 Total = 3
 Entre com um numero: 8
 Total = 11
 Entre com um numero: 15
 Final total = 26
```

Inicialmente, é dado o valor 0 à variável total, e o teste é verdadeiro ( 0 < 20). Em cada iteração, o total é impresso e o usuário digita um número que é somado a total. Quanto total for maior ou igual a 20, o teste do while torna-se falso, e a repetição termina.

## 9.2 Estilo de formatação para estruturas de repetição

A regra principal é ser **consistente**. Assim, seu programa será mais legível.

### 9.2.1 Colocação das chaves

Há três estilos comuns de colocar as chaves:

```
while (expressao)
   {
   sentenca;
   }

while (expressao)
{
   sentenca;
}

while (expressao) {
   sentenca;
}
```

APENAS UM DESTES ESTILOS deve ser consistentemente usado para as sentenças de repetição ( for, while e do ... while). Use o estilo com o qual você se sentir mais confortável.

### 9.2.2 Necessidade ou não das chaves

Foi mencionado anteriormente que o corpo da repetição pode ser uma sentença composta (conjunto de sentenças delimitadas por chaves ( { e }) ou ums sentença simples. Por exemplo:

```
while( i < 5 )
   i = i + 1;
```

Embora as chaves possam ser omitidas, há uma única razão para colocá-las sempre. Considere o caso simples abaixo:

```
while( i < 5 ) {
   i = i + 1;
}
```

Quando você adicionar algo ao programa, você poderá adicionar uma sentença para um laço com apenas uma sentença. Se você fizer isso, é vital que você também adicione chaves. Se você não fizer isso, a segunda sentença do laço não será considerada como parte do laço. Por exemplo:

```
while( i < 5 )
   i = i + 1;
   j = j + 1;
```

é na verdade o mesmo que:

```
while( i < 5 )
   i = i + 1;
j = j + 1;
```

enquanto a intenção era na realidade:

```
while( i < 5 ) {
   i = i + 1;
   j = j + 1;
}
```

### 9.2.3 Uso de espaço em branco

A outra questão de formato é se deve ser colocado um espaço em branco depois do while e antes do abre parênteses ( ). Por exemplo:

```
while (i<5)
```

ou

```
while (i<5)
```

ou

```
while( i < 5 )
```

Isto também é uma escolha pessoal. Porém seja consistente em sua escolha !

### 9.2.4 Laços aninhados

É possível colocar um laço dentro de outro (laço aninhado).

**Exemplo 2:**

```
#include <iostream>
#include <iomanip>  // NecessÃ¡rio para se usar a funÃ§Ã£o setw() em cout

using namespace std;

int main( ){

  int linha, coluna;

  linha = 1;
  while (linha < 5)
  {
    coluna = 1;
    while (coluna < 5)
    {
      cout << setw(3) << linha * coluna;
      coluna = coluna + 1;
    }
    linha = linha + 1;
  }
  cout << endl;
}
```

**Saída:**

```
1  2  3  4
2  4  6  8
3  6  9 12
4  8 12 16
```

No exemplo acima, para cada iteração do laço externo, o laço interno imprime uma linha com números e depois pula de linha.

**Exemplo 3:** Este exemplo é parecido com o anterior, exceto que o cout que produz a mudança de final de linha é colocado dentro do laço interno. Como era de se esperar uma nova linha é impressa após cada valor ao invés de ser depois de 4 valores.

```
#include <iostream>
#include <iomanip>  // NecessÃ¡rio para se usar a funÃ§Ã£o setw() em cout

using namespace std;

int main( ){

  int linha, coluna;

  linha = 1;
  while (linha < 5)
  {
    coluna = 1;
    while (coluna < 5)
    {
      cout << setw(3) << linha * coluna;
      cout << endl;
      coluna = contador + 1;
    }

    linha = linha + 1;
  }
}
```

**Saída:**

```
  1
  2
  3
  4
  2
  4
  6
  8
  3
  6
  9
 12
  4
  8
```

```
12
16
```

**Exemplo 4:** Este exemplo imprime um triângulo de asteriscos, de forma que a quantidade de asteriscos em uma linha é igual à ordem da linha (na linha 1, 1 asterisco, na linha 2, 2 asteriscos, etc.)

```
#include <iostream>

using namespace std;

int main( ){

  int linha, coluna;

  cout << endl;
  linha = 1;
  while (linha < 8)
  {
    cout << "\t";
    coluna = 1;
    while (coluna < linha)
    {
      cout << "*";
      coluna = coluna + 1;
    }
    cout << endl;
    linha = linha + 1;
  }
}
```

**Saída:**

```
*
**
***
****
*****
******
*******
********
```

# 10 Funções

## 10.1 Funções: o que são e por que usá-las

Quando queremos resolver um problema, em geral tentamos dividi-lo em subproblemas mais simples e relativamente independentes, e resolvemos os problemas mais simples um a um. A linguagem C++ dispõe de construções (abstrações) que auxiliam o projeto de programas de maneira *top-down*. Uma função cria uma maneira conveniente de encapsular alguns detalhes de "processamento", ou seja, como algum resultado é obtido. Quando esta "computação" é necessária, a função é *chamada*, ou *invocada*. Desta forma, quando uma função é chamada o usuário não precisa se preocupar como a computação é realizada. É importante saber o que a função faz (qual o resultado da execução de uma função) e também como se usa a função. Criando funções, um programa C++ pode ser estruturado em partes relativamente independentes que correspondem as subdivisões do problema.

Você já viu algumas funções: `cin.get()`, `sqrt()`. Elas são funções de uma biblioteca padrão (do C++ ). Você não sabe como elas foram escritas, mas já viu como utilizá-las. Ou seja, você sabe o *nome* das funções e quais informações específicas você deve fornecer a elas (valores que devem ser *passados* para as funções) para que a função produza os resultados esperados.

Quando nos referirmos a uma função neste texto usaremos a maneira frequentemente utilizada que é o nome da função seguido de `()`.

Tomemos como exemplo o programa abaixo, que recebe 2 conjuntos de 3 números e soma o maior valor de cada conjunto:

```
/* ------------------------------------------------------------------
   Recebe 2 conjuntos de 3 números e soma o maior valor de cada
   conjunto.
   ------------------------------------------------------------------
*/

#include <iostream>
#include <cmath>

using namespace std;

int main()
{
  int a, b, c, maior_1, maior_2;

  /* Lê o primeiro conjunto de 3 valores */
  cin >> a >> b >> c;

  /* Verifica o maior dos três valores informados */
  /* Assume inicialmente que a variável 'a' tem o maior valor */
  maior_1 = a;

  /* Somente muda o valor de 'maior' se
     os valores em 'b' ou 'c' forem maiores */
  if (b > maior_1)
    {
      maior_1 = b;
    }
```

```
  if (c > maior_1)
    {
      maior_1 = c;
    }

  /* Neste ponto do programa, maior_1 contém o maior valor
     dentre os 3 primeiros valores informados
   */

  /* Lê o segundo conjunto de 3 valores */
  cin >> a >> b >> c;

  /* Verifica o maior dos três valores informados */
  /* Algoritmo igual ao  acima, exceto pela variável que  recebe o maior
     valor */
  /* Assume inicialmente que a variável 'a' tem o maior valor */
  maior_2 = a;

  /* Somente muda o valor de 'maior' se
     os valores em 'b' ou 'c' forem maiores */
  if (b > maior_2)
    {
      maior_2 = b;
    }

  if (c > maior_2)
    {
      maior_2 = c;
    }

  /* Neste ponto do programa, maior_2 contém o maior valor
     dentre os 3 valores informados no 2o. conjunto de entrada
   */

  /* Calcula e exibe a soma solicitada */
  cout << endl << maior_1 << " + " << maior_2
       << " = " << maior_1 + maior_2 << endl;

}
```

Observe que o código que verifica o maior valor dentre 3 números teve que ser reproduzido dentro do programa por duas vezes (para descobrir o maior valor de dois conjuntos diferentes de 3 números).

Um dos benefícios mais óbvios de usar funções é que podemos evitar *repetição de código*. Em outras palavras, se você quiser executar uma operação mais de uma vez, você pode simplesmente escrever a função uma vez e utilizá-la diversas vezes ao invés de escrever o mesmo código várias vezes. Outro benefício é que se você desejar alterar ou corrigir alguma coisa mais tarde, é mais fácil alterar em um único lugar.

O exemplo acima poderia ser simplificado pela criação de uma função chamada `maior`, que dados três números $a$, $b$, e $c$, dá como resultado o maior valor dentre os três valores fornecidos:

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

```
int main()
{
  int a, b, c, maior;

  cout << "Entre com os tres numeros: ";
  cin >> a >> b >> c;

  if (a > b)
    maior = a;
  else
    maior = b;

  if (maior < c)
    maior = c;

  cout << "O Maior numero eh " << maior << endl;

}
```

O exemplo pode ser então alterado e simplificado com o uso da função `maior()`:

```
/* ---------------------------------------------------------------------
   Recebe 2 conjuntos de 3 números e soma o maior valor de cada
   conjunto.
   ---------------------------------------------------------------------
*/

#include <iostream>
#include <cmath>

using namespace std;

/* Função que retorna o maior valor entre
   x, y e z
*/
int acheMaior (int x, int y, int z)
{
  int maior;

  /* Assume inicialmente que a variável 'x' tem o maior valor */
  maior = x;

  /* Somente muda o valor de 'maior' se
     os valores em 'y' ou 'z' forem maiores */
  if (y > maior)
    {
      maior = y;
    }
```

```
  if (z > maior)
    {
      maior = z;
    }


  return maior;

}


int main()
{
  int a, b, c, maior_1, maior_2;

  /* Lê o primeiro conjunto de 3 valores */
  cin >> a >> b >> c;

  /* Verifica o maior dos três valores informados */
  maior_1 = acheMaior(a,b,c);

  /* Neste ponto do programa, maior_1 contém o maior valor
     dentre os 3 primeiros valores informados
   */

  /* Lê o segundo conjunto de 3 valores */
  cin >> a >> b >> c;

  /* Verifica o maior dos três valores informados */
  /* Usa  a mesma  função, pois  o procedimento  para encontrar  o maior
     valor de 3 números é o mesmo.
  */
  maior_2 = acheMaior(a,b,c);

  /* Neste ponto do programa, maior_2 contém o maior valor
     dentre os 3 valores informados no 2o. conjunto de entrada
   */

  /* Calcula e exibe a soma solicitada */
  cout << endl << maior_1 << " + " << maior_2
       << " = " << maior_1 + maior_2 << endl;

}
```

Como pode ser observado, sejam quais forem os conjuntos de 3 números fornecidos, não precisa escrever um código similar ao mostrado na função `maior` acima para cada número. Basta `chamar` a função `maior()`, passar os valores necessários para verificar o maior valor de cada conjunto, e utilizar os resultados.

Evitar repetição de código é a razão histórica que funções foram inventadas (também chamado de procedimento ou subrotinas em outras linguagens de programação). A maior motivação para utilizar funções nas linguagens contemporâneas é a redução da complexidade do programa e melhoria da modularidade do pro-

grama. Dividindo o programa em funções, é muito mais fácil projetar, entender e modificar um programa. Por exemplo, obter a entrada do programa, realizar as computações necessárias e apresentar o resultado ao usuário pode ser implementado como diferentes funções chamadas por `main()` nesta ordem.

Funções podem ser escritas *independentemente* uma da outra. Isto significa que, em geral, variáveis usadas dentro de funções não são compartilhadas pelas outras funções. Assim sendo, o comportamento da função é previsível. Se não for assim, duas funções completamente não relacionadas podem alterar os dados uma da outra. Se as variáveis são *locais* a uma função, programas grandes passam a ser mais fáceis de serem escritos. A comunicação entre funções passa a ser controlada – elas se comunicam somente através pelos valores passados as funções e os valores retornados.

## 10.2 Definindo funções

Um programa C++ consiste de uma ou mais definições de funções (e variáveis). Há sempre uma função chamada `main`. Outras funções também podem ser definidas. Cada uma pode ser definida separadamente, mas nenhuma função pode ser definida dentro de outra função. Abaixo, mostramos um exemplo simples de um programa que consiste de duas funções: `main()` e `alo()`. Quando executado, este programa imprimirá a mensage `Alo!` três vezes.

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

// definicao da funcao alo()
void alo(void)
{
   cout << "Alo!" << endl;
}

// definicao da funcao main()
int main ()
{
   int i;

   i = 1;
   while (i <= 3)
   {
       alo();
       i = i + 1;
   }
}
```

Todas as funções devem ser declaradas ou definidas antes de serem usadas. As funções da biblioteca padrão, tais como `cin.get()`, são pré-definidas, mas mesmo assim devem ser declaradas (deve ser anunciado ao compilador que elas existem). É por isso que incluímos a linha `#include <iostream>` no início do código fonte.

O formato geral da definição de uma função é

*tipo-do-resultado nome-da função* (*lista-de-argumentos*)
{
    *declarações e sentenças*
}

A primeira linha da definição é o cabeçalho da função. Ela têm três partes principais: o nome da função, o tipo do resultado (que é um valor) que a função computa e retorna, e entre parênteses uma lista de *parâmetros* (também chamado de *argumentos formais*). Se a função não retorna nenhum valor, o tipo é chamado de `void`, e esta palavra é escrita no cabeçalho na frente do nome da função. Se a função não tiver argumentos formais, a palavra `void` pode ser escrita no lugar da lista de argumentos formais entre os parênteses. Para simplificar a exposição, falaremos sobre o *tipo do retorno* e os *argumentos formais* mais tarde. Eles servem para permitir que as funções troquem informações entre si.

## 10.3 Funções simples

Para começar, vamos utilizar funções na seguinte forma:

```
void nome-da-função (void)
{
        declarações e senteças (corpo da função)
}
```

O primeiro `void` significa que esta função não tem tipo de retorno (não retorna um valor), e o segundo significa que a função não tem argumentos (ela não precisa de nenhuma informação externa para ser executada). Isso não significa que a função não faz nada. Ela pode realizar alguma ação, como imprimir uma mensagem. O exemplo abaixo mostra um programa que usa uma função como essa:

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

// DEFINIÇÃO da função alo()
void alo(void)
{
    cout << "Alo." << endl;
}

// Programa Principal
int main()
{
    alo();
}
```

Neste exemplo, o programa consiste de duas funções, `main()` e `alo()`.

A função `alo()` imprime a mensagem `Alo.` quando chamada. A sentença `cout` é o corpo da função. Dentro da função `main()` há uma *chamada* a função `alo()`. A função é chamada pelo seu nome seguido de `()` (já que a função `alo` não tem argumentos, nenhuma expressão é escrita dentro dos parênteses). A função `alo()` não retorna um valor, ela é chamada simplesmente para realizar uma ação (imprimir a mensagem). A chamada de função é uma sentença válida em C++ , portanto deve ser terminada por ponto e vírgula (`;`).

```
alo();
```

Observe que a ordem em que as funções são definidas dentro do código-fonte é importante, sendo que uma função deve sempre ser definida ANTES das funções em que ela é CHAMADA. No nosso exemplo, como a função `alo()` é chamada pela função `main()`, então a DEFINIÇÃO da função `alo()` deve vir

antes da definição da função `main()`. O uso de protótipos pode ser usado para definir as funções em qualquer ordem dentro do código-fonte, o que será visto na Seção 10.9.

Outra coisa que você deve ter notado é que `main()` também é uma função. A função `main()` não difere em nada das demais funções, com a exceção de que contém o programa principal, isto é, ao se executar um programa, ela á a primeira função a ser executada. As demais funções são executadas somente quando chamadas a partir da execução da função `main()`.

### 10.3.1 Argumentos

Nosso próximo exemplo pede que o usuário digite suas iniciais, e então chama a função `cumprimenta()` para imprimir a mensagem "Ola" junto com as iniciais digitadas. Estas iniciais (seus valores) são passadas para a função `cumprimenta()`. A função `cumprimenta()` é definida de forma que ela imprimirá a mensagem incluindo quaisquer iniciais passadas.

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

void cumprimenta(char inic1, char inic2)
{
    cout << "Ola, " << inic1 << inic2 << "!" << endl;
}

int main()
{
   char primeiro, segundo;

   cout << "Entre com duas iniciais (sem separacao): ";
   cin >> primeiro >> segundo ;
   cumprimenta(primeiro, segundo);
}
```

A função `main()` chama a função `cumprimenta()`. Ao fazer esta chamada, `main()` passa para `cumprimenta()` os valores dos dois caracteres para serem impressos. Veja um exemplo de execução do programa:

```
Entre com duas iniciais (sem separacao): YK
Alo, YK!
```

Note que há uma correspondência entre a quantidade, a ordem e tipo dos valores que `main()` passa (estes são chamados de *parâmetros reais* ou *argumentos reais*) e os argumentos listados no cabeçalho da função `cumprimenta()` (denominados *argumentos formais*).

## 10.4 Funções que retornam um valor

Funções que não retornam nenhum valor (como `alo()`, `main()`) possuem tipo `void`.

Além de executarem ações (como imprimir) uma função também pode *retornar* um valor para o programa que o chamou. Uma função que retorna um valor tem no cabeçalho o nome do tipo do resultado. O valor retornado pode ser de qualquer tipo, incluindo `int`, `float` e `char` (é claro que uma vez definida, a função só é de um tipo específico). Uma função que retorna um tipo diferente de `void` executa alguns cálculos, e retorna o resultado (que é um único valor) para quem a chamou. A função chamadora pode então usar o resultado. Para retornar um valor para a função chamadora, a função usa a sentença `return`.

O formato da sentença `return` é a seguinte:

`return` *expressão*;

A *expressão* é avaliada e o seu valor é convertido ao tipo de retorno da função (o tipo da função é dado no cabeçalho da função antes do nome da função).

Considere o seguinte exemplo. O programa consiste de duas funções: `main()` e `quadrado`. O programa pede que o usuário digite três números e verifica se eles podem ser os lados de um triângulo reto.

```
// programa que verifica se 3 numeros podem ser os lados de um
// triangulo reto.
//
#include <iostream>
using namespace std;

// funcao que calcula o quadrado de um numero
int quadrado(int n)
{
    return n * n;
}


int main()
{
    int s1, s2, s3;

    cout << "Entre tres inteiros: ";
    cin >> s1 >> s2 >> s3;

    if ( s1 > 0 && s2 > 0 && s3 > 0 &&
       (quadrado(s1) + quadrado(s2) == quadrado(s3) ||
        quadrado(s2) + quadrado(s3) == quadrado(s1) ||
        quadrado(s3) + quadrado(s1) == quadrado(s2)) )
    {
          cout << " " << s1 << " " << s2 << " " << s3
               << " podem formar um triangulo reto\n";
    }
    else
    {
          cout << " " << s1 << " " << s2 << " " << s3
               << " nao podem formar um triangulo reto\n";
    }
}
```

Note que quando chamamos a função `quadrado()` passamos o valor no qual desejamos executar o cálculo, e também usamos o valor retornado pela função em expressões. O valor de `quadrado(s1)` é o valor que a função `quadrado()` retorna quando chamado com o valor do argumento sendo igual ao valor da variável `s1`.

Os valores retornados pelas chamadas de funções podem ser usados em todos os lugares onde valores podem ser usados. Por exemplo,

```
y = quadrado(3);
```

Aqui `quadrado(3)` tem o valor `9`, portanto `9` pode ser atribuído a variável `y`;

```
x = quadrado(3) + quadrado(4);
```
atribuirá `25` a variável `x`, e
```
area = quadrado(tamanho);
```
atribuirá a variável `area` o valor da variável `tamanho` elevado ao quadrado.

O próximo exemplo tem uma função chamada `cinco`:

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

int cinco(void)
{
    return 5;
}

int main()
{
    cout << "cinco = " << cinco() << endl;
}
```

A saída do programa será
```
cinco = 5
```
porque o valor de `cinco()` dentro da sentença `cout` é `5`. Olhando na sentença `return`, `5` é a expressão retornada para o chamador.

Outro exemplo:

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

int obtem_valor(void)
{
    int valor;

    cout << "Entre um valor: ";
    cin >> valor;

    return valor;
}


int main()
{
    int a, b;

    a = obtem_valor();
    b = obtem_valor();

    cout << "soma = " << a + b << endl;
}
```

Este programa obtém dois inteiros do usuário e mostra a sua soma. Ele usa a função `obtem_valor()` que mostra uma mensagem e obtém o valor do usuário.

Um exemplo de saída deste programa é:

```
Entre um valor: 15
Entre um valor: 4
soma = 19
```

## 10.5 Mais sobre o `return`

Quando uma função `return` é executada, a função imediatamente acaba – mesmo que haja código na função após a sentença `return`. A execução do programa continua após o ponto no qual a chamada de função foi feita. Sentenças `return` podem ocorrer em qualquer lugar na função – não somente no final. Também é válido ter mais de um `return` dentro de uma função. A única limitação é que `return` retorna um único valor.

O seguinte exemplo mostra uma função (uma versão para `int` da função `obtem_valor`) que pede para usuário um valor e se o usuário digitar um valor negativo, imprime uma mensagem e retorna um valor positivo.

```
int obtem_valor_positivo(void)
{
   int valor;

   cout << "Entre um valor: ";
   cin >> valor;

   if (valor >= 0)
      return valor;

   cout << "Tornando o valor positivo..." << endl;

   return -valor;
}
```

Em uma função `void`, `return;` (só com `;`) pode ser usado para sair de uma função. O exemplo seguinte, pede instruções ao usuário. Se o usuário reponder **nao**, a função termina. Do contrário, ele imprime as instruções e depois termina.

```
void instrucoes(void)
{
   int ch;

   cout << "Voce quer instrucos? (s/n): ";
   ch = cin.get();

   /* Termina se resposta for n */
   if (ch == 'n' || ch == 'N')
      return;

   /* Mostra instrucoes */
   cout << "As regras do jogo sao . . . ";
     .
     .
     .
   return;
}
```

O `return` final (antes de fechar as chaves do corpo da função) na função é opcional. Se omitido, a função atingirá o final da função e retornará automaticamente. Note que o `return` é opcional somente para funções `void`.

## 10.6 Mais sobre Argumentos

A comunicação entre uma função e o chamador pode ser nas duas direções. Argumentos podem ser usados pelo chamador para passar dados para a função. A lista de argumentos é definida pelo cabeçalho da função entre parênteses.. Para cada argumento você precisa especificar o tipo do argumento e o nome do argumento. Se houver mais de um argumento, eles são separados por vírgula. Funções que não possuem argumentos tem `void` como lista de argumento. No corpo da função os argumentos (também chamados de *argumentos formais* ou *parâmetros formais*) são tratados como variáveis. É erro defini-los dentro do corpo da função porque eles já estão definidos no cabeçalho. Antes da execução da função os valores passados pelo chamador são atribuídos aos argumentos da função.

Considere o seguinte programa com a função `abs()` que calcula o valor absoluto de um número.

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

/* Definicao da funcao abs */
int abs(int x)
{
   if (x < 0)
      x = -x;

   return x;
}


int main()
{
    int n;

    cout << "Entre um numero: ";
    cin >> n;

    cout << "Valor absoluto de " << n << " eh " << abs(n) << endl;
}
```

A função `abs()` tem um argumento do tipo `int`, e seu nome é `x`. Dentro da função, `x` é usado como uma variável `x`.

Uma vez que `abs()` tem um único argumento, quando ela é chamada, há sempre um valor dentro do parênteses, como em `abs(n)`. O valor de `n` é passado para a função `abs()`, e antes da execução da função, o valor de `n` é atribuído a `x`.

Aqui está um exemplo de uma função que converte uma temperatura de Farenheit para Celsius:

```
float fahr_para_cels(float f)
{
    return 5.0 / 9.0 * (f - 32.0);
}
```

Como você pode ver, esta função tem somente um argumento do tipo `float`. Um exemplo de chamada desta função poderia ser:

```
fervura = fahr_para_cels(212.0);
```

O resultado da função `fahr_para_cels(212.0)` é atribuído a `fervura`. Portanto, depois da execução desta sentença, o valor de `fervura` (que é do tipo `float`) será `100.0`.

O exemplo seguinte possui mais de um argumento:

```
float area(float largura, float altura)
{
    return largura * altura;
}
```

Esta função possui dois argumentos do tipo `float`. Para chamar uma função com mais de um argumento, os argumentos devem ser separados por vírgula. A ordem em que os argumentos são passados deve ser na mesma em que são definidos. Neste exemplo, o primeiro valor passado será a largura e o segundo a altura. Um exemplo de chamada seria

```
tamanho = area(14.0, 21.5);
```

Depois desta sentença, o valor de `tamanho` (que é do tipo `float`) será `301.0`.

Quando passar os argumentos, é importante ter certeza de passá-los na ordem correta e que eles são do tipo correto. Se isso não for observado, pode ocorrer erro ou aviso de compilação, ou resultados incorretos podem ser gerados.

Uma última observação. Os argumentos que são passados pelo chamador podem ser expressões em geral e não somente constantes e variávies. Quando a função é chamada durante a execução do programa, estas expressões são avaliadas, e o valor resultante passado para a função chamada.

## 10.7 Chamada por valor

Considere novamente a função `quadrado()`. Se esta função é chamada de `main()` como

```
p = quadrado(x);
```

somente o valor (não o endereço) de `x` é passado para `quadrado`. Por exemplo, se a variável tem valor 5, para a função `quadrado()`, `quadrado(x)` ou `quadrado(5)` são o mesmo. De qualquer forma, `quadrado()` receberá somente o valor `5`. `quadrado()` não sabe se na chamada da função o `5` era uma constante inteira, o valor de uma variável do tipon `int`, ou alguma expressão como `625/25 - 4 * 5`. Quando `quadrado()` é chamado, não interessa qual a expressão entre parênteses, ela será avaliada e o valor passado para `quadrado()`.

Esta maneira de passar argumentos é chamada de *chamada por valor*. Argumentos em C++ são passados por valor. Portanto, a função chamada não pode alterar o valor da variável passada pelo chamador como argumento, porque ela não sabe em que endereço de memória o valor da variável está armazenado.

## 10.8 Variáveis locais

Como você provavelmente já reparou em alguns exemplos, é possível definir variáveis dentro de funções, da mesma forma que temos definido variáveis dentro da função `main()`. A declaração de variáveis é feita no início da função.

Estas variáveis são *restritas* a função dentro da qual elas são definidas. Só esta função pode "enxergar" suas próprias variáveis. Por exemplo:

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

void obtem_int(void)
{
    int x;

    cout << "Entre um valor: ";
    cin >> x;

    cout << "Obrigado!\n";
}


int main()
{
    obtem_int();

    /* **** Isto esta' errado **** */
    cout << "Voce digitou " << x << endl;
}
```

A função `main()` usou um nome `x`, mas `x` não é definido dentro de `main`; ele é uma variável local a `get_int()`, não a `main()`. Este programa gera erro de compilação.

Note que é possível ter duas funções que usam variáveis locais com o mesmo nome. Cada uma delas é restrita a função que a define e não há conflito. Analise o seguinte programa (ele está correto):

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

int obtem_novo_int(void)
{
    int x;

    cout << "Entre um valor: ";
    cin >> x;

    cout << "Obrigado!\n";
    return x;
}

int main()
{
    int x;

    x = obtem_novo_int();

    /* ****Isto nao esta errado !! **** */
    cout << "Voce digitou " << x << endl;
}
```

A função `obtem_novo_int()` usa uma variável local chamada `x` para armazenar o valor digitado e retorna como resultado o valor de `x`. `main()` usa outra variável local, também chamada de `x` para receber o resultado retornado por `obtem_novo_int()`. Cada função tem sua própria variável `x`.

## 10.9 Protótipos

Os protótipos servem para dar ao compilador informações sobre as funções. Isso para que você possa chamar funções antes que o compilador tenha a definição (completa) das funções. O protótipo de uma função é idêntico ao cabeçalho da função, mas o nome dos argumentos podem ser omitidos e ele é terminado com um ponto e vírgula. Protótipos *declaram* uma função ao invés de *defini-las*. O formato de um protótipo é:

*tipo-de-retorno nome-da-função* (*lista-dos-tipos-dos-argumentos*) ;

Definindo protótipos, você não precisa se preocupar com a ordem em que define as funções dentro do código-fonte do programa. A principal vantagem de definir protótipos é que erros de chamada de funções (como chamar uma função com o número incorreto de argumentos, ou com argumentos de tipo errado) são detectados pelo compilador. Sem protótipos, o compilador só saberia que há erro depois de encontrar a definição da função.

Abaixo, mostramos a definição de duas funções e seus respectivos protótipos:

```
float volume(float, float, float);
float dinheiro(int, int, int, int);


float volume(float comprimento, float largura, float altura)
{
    return comprimento * largura * altura;
}

float dinheiro(int c25, int c10, int c5, int c1)
{
     return c25 * 0.25 + c10 * 0.10 +
            c5 * 0.05 + c1 * 0.01;
}
```

## 10.10 Documentação de funções

Você deve documentar as funções que escreve. Na documentação você deve especificar as seguintes informações:

**Ação** – o que a função faz

**Entrada** – descrição dos argumentos passados para a função

**Saída** – descrição do valor retornado pela função

**Suposições** – o que você assume ser verdade para que a função funcione apropriadamente

**Algoritmo** – como o problema é resolvido (método)

Estas informações devem ser colocadas como comentário antes da definição da função.

## 10.11 Comentários

Você pode colocar comentários no seu programa para documentar o que está fazendo. O compilador ignora completamente o que quer esteja dentro de um comentário.

Comentários em C++ são textos que começam com `//` em cada linha, ou são delimitados por `/*` e `*/`. Os símbolos `//` não possuem espaço entre si. Alguns exemplos:

```
// Este é um comentário sem graça


// Este é um comentário um pouco
// maior que tem diversas linhas


/* Este é um outro comentário maior
   ainda e que tem várias linhas
   explicando qualquer aspecto mais detalhado
   do programa
 */
```

**Regras para comentário**

É sempre uma boa idéia colocar comentários em seu programa das coisas que não são claras. Isto vai ajudar quando mais tarde você olhar o programa que escreveu já há algum tempo ou vai ajudar a entender programas escritos por outra pessoa.

Um exemplo de comentário útil:

```
/* converte temperatura de farenheit para celsius */
celsius = (fahrenheit - 32) * 5.0 / 9.0;
```

O comentário deve ser escrito em português e não em C++ . No exemplo abaixo

```
/* usando scanf, obter valor de idade e multiplicar por 365 para
 * obter dias */
cin >> idade;
dias = idade * 365;
```

o comentário é basicamente uma transcrição do código do programa. Em seu lugar, um comentário como

```
/* obtem idade e transforma em numero de dias */
```

seria mais informativo neste ponto.

Em outras palavras, você deve comentar o código, e não codificar o comentário.

Você também deve evitar comentários inúteis ou óbvios. Por exemplo:

```
// Incrementa i
i = i + 1;
```

Não há necessidade de comentários já que `i = i + 1` já é auto explicativo.

Abaixo está um exemplo de como você deve comentar uma função.

```
/* função instrucoes()
 *  acao:        mostra instrucoes do programa
 *  entrada:     nenhuma
 *  saida:       nenhuma
 *  suposicoes:  nenhuma
 *  algoritmo:   imprime as instrucoes
 */
void instrucoes(void)
{
   /* mostra instrucoes */
   cout << "O processo de purificacao do  Uranio-235 e' . . .  ";
     .
     .
}
```

# 11 Mais sobre funções: Quando `return` não é suficiente

Considere o programa abaixo que pede ao usuário dois inteiros, armazena-os em duas variáveis, troca seus valores, e os imprime.

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

int main()
{
  int a, b, temp;

  cout << "Entre dois numeros: ";
  cin >> a >> b;

  cout << "Voce entrou com " << a << " e " << b << endl;

  /* Troca a com b */
  temp = a;
  a = b;
  b = temp;

  cout << "Trocados, eles sao " << a << " e " << b << endl;
}
```

Aqui está um exemplo de execução do programa:

```
Entre dois numeros: 3 5
Voce entrou 3 e 5
Trocados, eles sao 5 e 3
```

O seguinte trecho do programa executa a troca de valores das variáveis `a` e `b`:

```
temp = a;
a = b;
b = temp;
```

É possível escrever uma função que executa esta operação de troca? Considere a tentativa abaixo de escrever esta função:

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

void troca(int x, int y)
{
    int temp;

    temp = x;
    x = y;
    y = temp;
}
```

```
int main()
{
  int a, b;

  cout << "Entre dois numeros: ";
  cin >> a >> b;

  cout << "Voce entrou com " << a << " e " << b << endl;

  // Troca a com b
  troca(a, b);

  cout << "Trocados, eles sao " << a << " e " << b << endl;
}
```

Se você executar este programa, verá que ele **não** funciona:

```
Entre dois numeros: 3 5
Voce entrou 3 e 5
Trocados, eles sao 3 e 5
```

Como você já viu nas notas anteriores, em C++ os argumentos são passados **por valor**. Uma vez que somente os valores das variáveis são passados, não é possível para a função `troca()` alterar os valores de `a` e `b` porque `troca()` não sabe onde na memória estas variáveis estão armazenadas. Além disso, `troca()` não poderia ser escrito usando a sentença `return` porque podemos retornar APENAS UM valor (não dois) através da sentença `return`.

## 11.1 Usando referência

A solução para o problema acima é ao invés de passar os valores de `a` e `b`, passar uma referência às variáveis `a` e `b`. Desta forma, `troca()` saberia que endereço de memória escrever, portanto poderia alterar os valores de `a` e `b`.

Lembre-se que em C++ a cada variável está associado: (i) um nome; (ii) um tipo; (iii) um valor; e (iv) um endereço. Assuma que existam as seguintes definições de variáveis.

```
int i = 5;
char c = 'G';
```

Na memória, eles podem estar armazenados da forma indicada na Figura 11.1:

A variável inteira `i` está armazenada no endereço `1342`. Ela usa dois bytes de memória (quando um objeto usa mais de um byte, seu endereço é onde ele começa – neste caso, `1342` e não `1343`). A variável do tipo `char c` está armazenada no endereço `1346` e usa um byte de memória. O compilador é que controla do local de armazenamento destas variáveis em memória.

## 11.2 Argumentos por referência em funções

Considere novamente o exemplo da função `troca()`. Quando `a` e `b` são passados como argumentos para `troca()`, na verdade, somente seus valores são passados. A função não podia alterar os valores de `a` e `b` porque ela não conhece os endereços de `a` e `b`. Mas se referências para `a` e `b` forem passados como argumentos ao invés de `a` e `b`, a função `troca()` seria capaz de alterar seus valores; ela saberia então

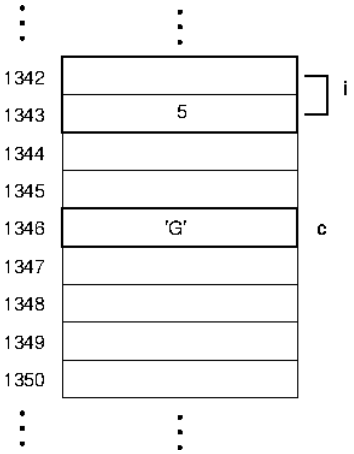


Figura 3: Variáveis em memória

em que endereço de memória escrever. Na verdade, a função não sabe que os endereços de memória são associados com `a` e `b`, mas ela pode modificar o conteúdo destes endereços. Portanto, passando uma variável por referência (ao invés do valor da variável), habilitamos a função a alterar o conteúdo destas variáveis na função chamadora.

A definição da função `troca()` deve ser alterada, e a lista de parâmetros formais deve ter argumentos não do tipo `int`, mas referências para `int`, ou seja, `int &` . Quando chamamos a função `troca()`, nós continuamos passando parâmetros reais `a` e `b`, mas desta vez, o compilador sabe que o que será passado para a função `troca()` são as referências a estas variáveis, e não seus valores. Dentro da função `troca()` não deverá haver mudanças. Uma vez que agora os parâmetros formais são referências, o acesso aos objetos deve ser escrito normalmente, mas deve-se ter em mente que qualquer alteração nos valores dos parâmetros formais da função implica em alterar o valor dos argumentos passados para a função no momento de sua chamada. Assim, a função `troca()` é capaz de alterar os valores de `a` e `b` "remotamente".

O programa abaixo é a versão correta do problema enunciado para a função `troca()`:

```
#include <iostream>

using namespace std;

/* função troca(px, py)
 *  ação:         troca os valores inteiros apontados por px e py
 *  entrada:      apontadores px e py
 *  saida:        valor de px e py trocados na origem da chamada da função
 *  suposições:   px e py sao apontadores validos
 *  algoritmo:    primeiro guarda o primeiro valor em um temporario e
 *                troca
 */
void troca(int & px, int & py)
{
    int temp;

    temp = px;
    px = py;
    py = temp;
}

int main()
```

```
{
  int a, b;

  cout << "Entre dois numeros: ";
  cin >> a >> b;

  cout << "Voce entrou com " << a << " e " << b << endl;

  // Troca a com b -- passa argumentos por referencia
  troca(a, b);

  cout << "Trocados, eles sao " << a << " e " << b << endl;
}
```

A saída deste programa é:

```
Entre dois numeros: 3 5
Voce entrou com  3 e 5
Trocados, eles sao 5 e 3
```

Basicamente, se a função precisa alterar o valor de uma variável no ponto de chamada da função, então passamos o nome da variável como parâmetro real, e escrevemos a função indicando os parâmetros como sendo de SAìDA ou PASSADOS POR REFERÊNCIA.

# 12 O pré-processador

O pré-processador é um programa que faz alguns processamentos simples antes do compilador. Ele é executado automaticamente todas as vezes que seu programa é compilado, e os comandos a serem executados são dados através de *diretivas do pré-processador*.

Estas diretivas são colocadas em linhas que contém somente a diretiva (elas não são código da linguagem C++ , portanto as regras para elas são um pouco diferentes). As linhas que começam com um # são comandos para o pré-processador. A linha inteira é reservada para este comando (nenhum código C++ pode aparecer nesta linha e comandos do pré-processador não podem estar separados em diversas linhas).

## 12.1 A diretiva `#define`

Uma diretiva que é usada frequentemente é o `#define`. Esta diretiva é usada para fazer *substituição de macros*. Por enquanto, mostraremos uma utilização simples do `#define`, que é simplestemente uma substituição no texto.

O uso mais frequente desta diretiva é dar nomes simbólicos a uma constante (você já viu outra maneira de definir contantes que é colocar a palvavra `const` antes da definição de uma variável). Por exemplo, seria conveniente usar `PI` em seus programas ao invés de digitar `3.1415926535` toda hora. Como outro exemplo, se você quiser escrever um programa sobre estudantes de uma turma de 81 alunos, você poderia definir `NUM_ALUNOS` como `81`. Assim, se o número de alunos mudar, você não precisaria modificar todo o seu programa onde o número de alunos (`81`) é utilizado, mas simplesmente alterar a diretiva `#define`. Estas duas diretivas são definidas da seguinte forma:

```
#define PI          3.1415926535
#define NUM_ALUNOS  81
```

Por convenção, nomes introduzidos por um `#define` são geralmente em letra maiúscula (e variáveis são em letra minúscula, ou uma mistura de letras minúsculas e maiúsculas). Assim, quando você vê um nome em um programa, você sabe se o nome refere-se a uma variável ou um nome definido por um `#define`.

Considere o seguinte programa exemplo que usa `PI`:

```
#define PI        3.14159265

int main()
{
   double raio;

   cout << "Entre com o raio: ";
   cin >> raio;

   cout << "Circunferencia = " << 2.0 * PI * raio << endl;
}
```

Lembre-se que o nome `PI` não é um nome de variável. Ele é um nome que o pré-processador substituirá pelo *texto* especificado pelo `#define` (mais ou menos da mesma forma que o comando pesquisa-e-substitui do editor de texto). O compilador nunca vê ou sabe sobre `PI`. O compilador vê o seguinte `cout` do programa acima depois do pré-processador ser executado:

```
cout << "Circunferencia = " << 2.0 * 3.14159265 * raio << endl;
```

## 12.2 A diretiva `#include`

Agora imagine que estamos escrevendo uma biblioteca geométrica: um conjunto de funções para calcular a área de cilindros, cones, esferas. Se diferentes pessoal estão escrevendo cada uma das funções, eles

provavelmente colocarão suas funções em diferentes arquivos. Mas todas as funções usam o numero $\pi$, e algumas outras constantes podem ser necessárias também. Ao invés de colocar o `#define` no início de cada arquivo, um único arquivo `geom.h` pode ser criado. Este arquivo conterá a linha

```
#define  PI    3.14159265
```

Assim, se todos os arquivos de funções geométricas puderem enxergar `geom.h`, eles compartilharão as mesmas definições. é para isso que usamos a diretiva `#include`, para incluir em seu programa, informações que estão em outro arquivo. Estas diretivas geralmente estão no início do programa fonte, antes da definição de funções e varáveis. Por exemplo, a diretiva

```
#include "geom.h"
```

colocada nos arquivos fontes que contêm as funções geométricas fará com que todos eles usem o nome simbólico `PI` ao invés de `3.14159265`. O fato do nome do arquivo estar em aspas significa que o arquivo `geom.h` está no mesmo diretório que os arquivos fontes (ao invés do diretório onde se encontram as bibliotecas padrão de C++ ).

A diretiva

```
#include <iostream>
```

é colocada no início do programa fonte para incluir informações (como protótipos de funções) que são necessários quando `cout` e `cin` são chamados dentro do programa. O arquivo entre `< >` está em algum diretório padrão conhecido pelo pré-processador. Este arquivo `iostream` é comum a todas as implementações da linguagem C++ e contém infomações necessárias para executar operações de entrada e saída da entrada e saída padrão (teclado e monitor).

## 12.3 Comentários

De um modo geral, o pré-processador dos compiladores existentes remove todos os comentários do arquivo fonte antes do programa ser compilado. Portanto, o compilador nunca vê realmente os comentários.

# 13 Vetores ou Arrays

Considere o seguinte programa. Este programa pede ao usuário notas de 4 estudantes, calcula a média e imprime as notas e a média.

```
int main()
{
   int nota0, nota1, nota2, nota3;
   int  media;

   cout << "Entre a nota do estudante 0: ";
   cin >> nota0;
   cout << "Entre a nota do estudante 1: ";
   cin >> nota1;
   cout << "Entre a nota do estudante 2: ";
   cin >> nota2;
   cout << "Entre a nota do estudante 3: ";
   cin >> nota3;

   media = (nota0 + nota1 + nota2 + nota3) / 4;

   cout << "Notas: " << nota0 << " " << nota1 << " " << nota2 << " "
        << nota3 << endl;
   cout << "Media: " << media << endl;
}
```

Este programa é bem simples, mas ele tem um problema. O que acontece se o número de estudantes aumentar ? O programa ficaria muito maior (e feio !!). Imagine o mesmo programa se existissem 100 estudantes.

O que precisamos é uma abstração de dados para agrupar dados relacionados. Este é o objetivo de *arrays* em C++ .

Um *array* é uma coleção de um ou mais objetos, do mesmo tipo, armazenados em endereços adjacentes de memória. Cada objeto é chamado de *elemento* do array. Da mesma forma que para variáveis simples, damos um nome ao array. O tamanho do array é o seu número de elementos. Cada elemento do array é numerado, usando um inteiro chamado de *índice*. Em C++ , a numeração começa com `0` e aumenta de um em um. Assim, o último índice é igual ao número de elementos do array menos um.

Por exemplo, podemos definir um array `nota` de tamanho `100` para armazenar as notas dos cem estudantes:

```
int nota[100];
```

Quando o compilador encontra esta definição, ele aloca 200 bytes consecutivos de memória (dois bytes – referente a cada `int` – para cada nota). Cada nota pode ser acessada dando o nome do array e o índice entre colchetes: como `nota[0]` (para a primeira nota), `nota[1]` para a segunda nota, e assim por diantes, até a última nota, `nota[99]`.

Mas antes de prosseguirmos com as propriedades e algoritmos de vetores, vamos introduzir operadores especiais de atribuição aritmética e uma nova estrutura de repetição que é bastante adequada na manipulação de vetores.

## 13.1 Operação de Atribuição Aritmética

É freqüente em programas C++ expressões do tipo:

```
tudo  = tudo + parte;

tamanho = tamanho * 2.5;

x = x * (y + 1);

j = j - 1;
```

C++ fornece operadores adicionais que podem ser usados para tornar estes tipos de atribuições mais curtos.

Há um operador de atribuição para cada operação aritmética listada anteriormente:

**+=** operação de atribuição de adição

**-=** operação de atribuição de subtração

***=** operação de atribuição de multiplicação

**/=** operação de atribuição de divisão

**%=** operação de atribuição de resto

Cada uma dessas operações podem ser usadas para tornar as expressões anteriores mais curtas:

```
tudo += parte;

tamanho *= 2.5;

x *= y + 1;

j -= 1;
```

## 13.2 A Estrutura de Repetição for

Considere o laço while abaixo:

```
int i;

i = 0; /* valor inicial da variável de controle da repetição */
while (i  <= 10) /* Testa variável de  controle para saber  se haverá
                    repetição ou não do corpo do "while" */
{
    ....
    ....

    i += 1; /* expressão de incremento da variável de controle da
               repetição */
}
```

Este laço pode ser expresso de forma diferente utilizando a estrutura de repetição for. A estrutura acima pode ser expressa de forma equivlente com um for da seguinte forma:

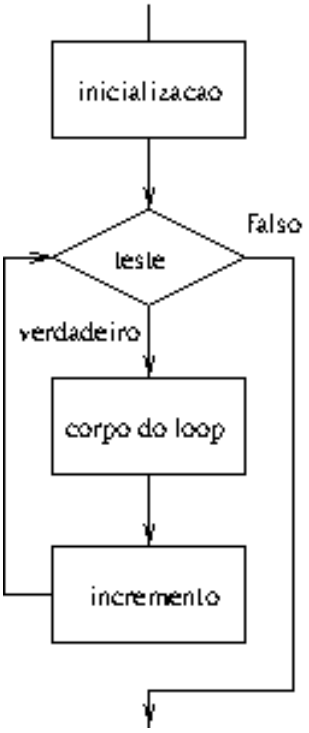
```
int i;

for (i = 0; i <= 10; i += 1)
{
     ....
     ....
}
```

Como pode-se observar, há 4 partes no laço for: *inicialização*, *expressão de teste*, *expressão de incremento* e o *corpo do laço*. O formato do laço for é:

for (*inicialização* ; *expressão de teste* ; *incremento* ){
*corpo da repetição*
}

A *inicialização* é executada uma única vez no início do laço. A *expressão teste* é então avaliada para verificar se o laço deve terminar. Caso a expressão seja verdadeira (isto é, diferente de Zero), o *corpo da repetição* é executado. Depois desta execução, a expressão de *incremento* é executada e o processo é repetido a partir da *expressão teste*. O *corpo da repetição*, por sua vez, pode ser uma sentença simples ou composta.




Veja abaixo um exemplo simples do laço for:

```
int contador;

for( contador = 0; contador < 5; contador += 1 )
    cout <<  "contador = " << contador << endl;

cout << "ACABOU !!!!\n";
```

**Saída do programa:**

```
contador = 0
```

```
contador = 1
contador = 2
contador = 3
contador = 4
ACABOU !!!!
```

Se você examinar cuidadosamente este exemplo, poderá ver precisamente o que está acontecendo. Primeiro, a inicialização é executada, que é a sentença `contador = 0`. Isso modifica o valor da variável `contador` para `0`. Então, o teste é executado. como `0 < 5` é verdadeiro, o laço continua. Assim, o corpo da repetição é executado, imprimindo a primeira linha da saída, `contador = 0`. Depois disso, o incremento é executado, que é a sentença `contador++`, que altera o valor da variável `contador` para `1`.

Esta é a 1ª iteração do laço. Então, o teste é executado novamente (como `1 < 5` é verdadeiro, o laço continua), o corpo da repetição mostra `contador = 1`, e `contador` é incrementado novamente.

Este processo continua até que `contador` seja `4` e `contador = 4` seja impresso. Depois disso, `contador` é incrementado para `5` e o teste é executado. Mas desta vez, `5 < 5` é falso, então o laço não continua. A execução do programa continua na sentença que segue o laço (no caso, imprimir a frase `ACABOU !!!`).

Após a execução do laço, a variável contador tem valor `5`.

Ao invés de usar o teste `contador < 5`, você poderia também ter usado a expressão `contador <= 4`. O resultado seria o mesmo. Use a expressão que você preferir. Outra expressão que também poderia ter sido usada é `contador != 5`. Porém esta expressão torna o programa menos legível (não é tão evidente que o valor de `contador` está sendo incrementado até atingir o valor `5`). Além disso, isso poderia causar problemas se mudássemos a inicialização para um valor maior que `5`. Por exemplo, se a inicialização for `contador = 25` e a expressão teste for `contador != 5` o laço nunca terminaria, pois o contador começa com `25` e a cada iteração o valor é incrementado, o que nunca tornaria o teste falso.

Também poderíamos ao invés de usar `contador += 1` como a expressão de incremento, usar `++contador`, `contador++` e `contador = contador + 1`. O resultado seria o mesmo (neste caso, o uso de pós- e pré-incremento não faz diferença).

Se você quisesse incrementos de dois, você poderia escrever `contador += 2` (ou `contador = contador + 2`).

### 13.2.1 Diversas sentenças dentro de um laço

Como no comando `while`, o corpo da repetição pode ser definido por uma sentença simples ou composta. No caso de uma sentença composta (bloco), não se deve esquecer de usar as chaves ( { e } ) para delimitar o bloco da sentença composta.

Em um `for` também podemos ter mais de uma expressão de inicialização ou incremento. Nestes caso, elas devem ser separadas por vírgula ( ,) o que é ilustrado no exemplo abaixo:

**Exemplo 1:**

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

int main( ){

  int contador, total;

  for( contador = 0, total = 0; contador < 10; contador += 1 ){
    total += contador;
    cout << "contador = " << contador << ", total = " << total << endl;
  }
}
```

**Saída:**

```
contador = 0, total = 0
contador = 1, total = 1
contador = 2, total = 3
contador = 3, total = 6
contador = 4, total = 10
contador = 5, total = 15
contador = 6, total = 21
contador = 7, total = 28
contador = 8, total = 36
contador = 9, total = 45
```

No exemplo acima, `contador = 0, total = 0` é a inicialização, `contador < 10` é a expressão teste, e `contador += 1` é a expressão de incremento.

**Exemplo 2:** Um programa que imprime todos os números entre 30 e 5 (nesta ordem) divisíveis por 3, e no final imprime sua soma.

```
#include <iostream>
#include <iomanip>  // NecessÃ¡rio para uso da funÃ§Ã£o setw() em cout

using namespace std;

int main( ){

  int i, soma;

  soma = 0;
  for( i = 30; i >= 5; i -= 1 ){
    if( (i % 3) == 0 ){
      cout << "\t" << setw(2) << i << endl;
      soma += i;
    }
  }
  cout << "\t soma = " << soma << endl;
}
```

**Saída do programa:**

```
      30
      27
      24
      21
      18
      15
      12
       9
       6
      soma = 162
```

### 13.2.2 Usando while e for

Embora qualquer laço possa ser escrito usando while ou for, a escolha é baseada principalmente no estilo. Por exemplo, se o laço precisa de uma inicialização e um incremento (como no caso do tratamento de vetores), então o for geralmente é usado. No caso em que o número de repetições não é pré-determinado em geral usa-se o while.

Como o comando for:

```
for( inicializacao; teste; incremento )
  sentenca;
```

é equivalente a:

```
inicializacao;
while( teste ) {
   sentenca;
   incremento;
};
```

você pode escolher o que preferir, a princípio.

## 13.3 Definindo arrays e acessando seus elementos

A definição de arrays é muito parecida com a definição de variáveis. A única diferença é que em array é necessário especificar seu tamanho (quantos elementos ele tem).

Os colchetes `[` e `]` são usados na definição do tamanho, como mostra os exemplos a seguir:

```
int total[5];

float tamanho[42];
```

O primeiro exemplo é um array de `5` inteiros (o tipo `int`) com o nome `total`. Como a numeração de arrays começa com `0`, os elementos da array são numerados `0`, `1`, `2`, `3` e `4`.

O segundo exemplo é um array de `42` elementos do tipo `float` com índices de `0` a `41`.

Cada elemento do array `total` é do tipo inteiro e pode ser usado do mesmo jeito que qualquer variável inteira. Para nos referirmos a um elemento do array, usamos colchetes também (`[` e `]`). O valor dentro dos colchetes pode ser qualquer expressão do tipo **inteiro**. Quando um array é definido, armazenamento suficiente (bytes contínuos na memória) são alocados para conter todos os elementos do array.

Note na tabela de precedência abaixo que `[ ]` tem precedência maior que todos os demais operadores.

| Operador | Associatividade |
|---|---|
| `()  []` | esquerda para direita |
| `!  -  &` | direita para esquerda |
| `*  /  %` | esquerda para direita |
| `+  -` | esquerda para direita |
| `<  <=  >  >=` | esquerda para direita |
| `==  !=` | esquerda para direita |
| `&&` | esquerda para direita |
| `\|\|` | esquerda para direita |
| `=` | direita para esquerda |
| `,` | esquerda para direita |

Verifique se você entende as sentenças do programa abaixo.

```
int i, x, sala, total[5];
float area;
float tamanho[42];

x = total[3];

i = 4;

total[i] = total[i-1] + total[i-2];

total[4] = total[4] + 1;

tamanho[17] = 2.71828;

sala = 3;

area = tamanho[sala] * tamanho[sala];

cin >> tamanho[41];
```

Agora, podermos reescrever o programa que calcula a média de uma classe de 4 alunos:

```
int main()
{
   int indice, nota[4];
   float total;

   indice = 0;
   while (indice < 4)
   {
      cout << "Entre a nota do estudante " << indice << ": ";
      cin >> nota[indice];
      indice = indice + 1;
   }

   cout << "Notas:  ";

   total = 0;
   indice = 0;
   while (indice < 4)
   {
      cout << nota[indice] << " ";
      total = total + nota[indice];
      indice = indice + 1;
   }
   cout << endl << "Media: " << total / 4 << endl;
}
```

Exemplo de Saída:

```
Entre a nota do estudante 0: 93
Entre a nota do estudante 1: 85
Entre a nota do estudante 2: 74
Entre a nota do estudante 3: 100
Notas:  93 85 74 100
Media: 88
```

O código-fonte do programa é consideravelmente mais curto. Ele fica ainda mais elegante com o uso de operadores de atribuição aritmética (Seção 13.1) e da estrutura de repetição for (Seção 13.2)[4]:

```
int main()
{
   int indice, nota[4];
   float total;

   for(indice = 0; indice < 4; indice += 1)
   {
      cout << "Entre a nota do estudante " << indice << ": ";
      cin >> nota[indice];
   }

   cout << "Notas:  ";

   total = 0;
   for(indice = 0; indice < 4; indice += 1)
   {
      cout << nota[indice] << " ";
      total = total + nota[indice];
   }
   cout << endl << "Media: " << total / 4 << endl;
}
```

O único problema é que ainda não é fácil modificar o programa para cem alunos porque *4* está em vários pontos do programa. Nós podemos usar o `#define` para manter o tamanho do array como uma constante simbólica ao invés de utilizar uma constante numérica.

```
#define ESTUDANTES 4

int main()
{
   int indice, nota[ESTUDANTES];
   float total;

   for(indice = 0; indice < 4; indice += 1)
   {
      cout << "Entre a nota do estudante " << indice << ": ";
      cin >> nota[indice];
   }

   cout << "Notas:  ";
```

[4]Razão pela qual usaremos estas estruturas daqui por diante no texto

```
    total = 0;
    for(indice = 0; indice < 4; indice += 1)
    {
       cout << nota[indice] << " ";
       total = total + nota[indice];
    }
    cout << endl << "Media: " << total / ESTUDANTES << endl;
}
```

## 13.4 Inicialização de arrays

Os arrays podem ser inicializados quando são definidos. Se o array não for inicializado, então ele contem valores indefinidos (também conhecidos como *lixo*).

Para inicializar um array, um valor para cada elemento deve ser especificado. Estes valores devem estar entre chaves (`{` e `}`) e são separados por vírgula (`,`). Alguns exemplos:

```
int valor[4] = { 1, 42, -13, 273 };

// o tamanho do array pode ser omitido
int peso[] = { 153, 135, 170 };
```

No primeiro exemplo, `valor` é um array de 4 inteiros onde `valor[0]` e' `1`, `valor[1]` e' `42`, `valor[2]` e' `-13`, e `valor[3]` e' `273`.

Note que no segundo exemplo, o tamanho do array foi omitido. Neste caso, o compilador calcula o tamanho como sendo o número de elementos listados. Quando um array é definido, se ele não for inicializado, o tamanho do array deve ser especificado. Se o array for inicializado, o tamanho pode ser omitido. O segundo exemplo acima é equivalente a

```
int peso[3] = { 153, 135, 170 };
```

Se o tamanho não for omitido, o número de elementos presentes não deve exceder o tamanho. Se exceder, o compilador gerará uma mensagem de erro. Se houver menos elementos na lista de inicialização, então os elementos dados são usados para inicializar os primeiros elementos do array. Qualquer elemento não inicializado conterá lixo.

Note que este tipo de inicialização só é válido no contexto onde o array é definido. Uma sentença como a seguinte produzirá um erro do compilador, uma vez que arrays só podem ser inicializados quando definidos.

```
int erro[5];

erro = { 2, 4, 6, 8, 10 };      // ISTO ESTA' ERRADO
```

Há mais uma restrição na inicialização de um array. Os valores devem ser todos constantes – nenhuma variável ou expressão é permitida. O seguinte trecho de programa produz um erro porque um dos valores de inicialização é uma variável:

```
int x = 21;
int yy[3] = { 1, 2, x };   // ISTO ESTA' ERRADO
```

## 13.5 Verificação de Limite

Quando um array é definido, é alocado espaço em memória para conter todos os elementos do array (não mais). O tamanho do array é dado explicitamente escrevendo o tamanho, ou implicitamente, inicializando o

array. Embora arrays tenham tamanhos específicos, é possível que um programa tente acessar endereços de memória de elementos fictícios, ou seja, endereços de memória que não pertencem ao array. Isto acontece quando usamos um índice que não esteja entre *0* e *n-1* para um array de tamanho *n*. O compilador não gera nenhum aviso quando isto acontece. Quando executamos um acesso "fora dos limites" do array, o resultado pode ser desastroso. Isto siginifica que o programa pode não fazer nada, cancelar a execução, travar o computador, entrar em um loop infinito, etc.

Se você executar uma atribuição a um elemento do array fora do seu limite, você estará escrevendo em um endereço de memória que pode conter algo importante, destruindo-o. Em geral, erros como estes são difíceis de encontrar, já que o programa pode até executar, só que faz algo "estranho". Se você estiver usando o Code::Blocks , você poderá ver uma mensagem como "Esta aplicação violou a integridade do sistema devido a execução de uma instrução inválida e será cancelada.". Não entre em pânico !! Você provavelmente terá que reinicializar o seu computador e examinar o seu programa cuidadosamente para achar acessos a array fora do seu limite. É claro que ao reinicializar o seu computador, você perderá todo o seu trabalho se não tiver salvado antes. **MORAL:** depois que seu programa compilar com sucesso, salve o seu programa em disco antes de executá-lo.

Por exemplo, considere o seguinte programa:

```
#include <iostream>

using namespace std;

#define TAM 10

int main()
{
    int ops[TAM], i;

    // Acesso fora dos limites quando i == TAM
    for(i = 0; i <= TAM; i += 1)
    {
        ops[i] = 0;
    }
}
```

Este programa inicializa cada elemento do array com *0*. O problema ocorre quando *i* tem o valor *10*. Neste ponto, o programa coloca *0* em *ops[10]*. Isto pode produzir resultados indefinidos (e desastrosos) embora o compilador não gere nenhum erro. O problema está na condição do *while*, que não deveria permitir que o corpo da repetição fosse executado quando *i* assumisse o valor 10.

## 13.6 Arrays como argumentos de funções

Para passar um array como argumento (com todos os seus elementos) de uma função passamos o nome do array. Considere o exemplo abaixo:

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

#define TAMANHO 5

// função array_max(a)
//  ação:      acha o maior inteiro de um array de TAMANHO elementos
```

```
//  entrada:   array a de inteiros
//  saida:     o maior valor do array
//  suposições: a tem TAMANHO elementos
//  algoritmo:  inicializa max com o primeiro elemento do array; em
//              uma repeticao compara o max com todos os elementos
//              do array em ordem e muda o valor de max quando um
//              elemento do array for maior que o max ja' encontrado.
//
int array_max(int a[])
{
  int i, max;

  // Achar o maior valor do array
  max = a[0];

  i = 1;
  while (i < TAMANHO) {
    if (max < a[i])
      {
        max = a[i];
      }
    i = i + 1;
  }

  return max;
}


/* Programa principal */
int main()
{
  int i, valor[TAMANHO];

  i = 0;
  while (i < TAMANHO) {
    cout << "Entre um inteiro: ";
    cin >> valor[i];
    i = i + 1;
  }

  cout << "O maior eh " << array_max(valor) << endl;
}
```

Aqui está um exemplo de execução deste programa

```
 Entre um inteiro: 73
 Entre um inteiro: 85
 Entre um inteiro: 42
 Entre um inteiro: -103
 Entre um inteiro: 15
 O maior e' 85
```

Em `main()` a chamada para `array_max()` tem `valor` como seu argumento, que é copiado para o parâmetro formal `a`, que é um array de inteiros. Note que o tamanho não foi especificado, somente o nome do array, `a`. Porém é também correto incluir o tamanho (isto é uma questão de estilo – escolha o que você preferir):

```
int array_max(int a[TAMANHO])
{
  ...
}
```

A inclusão do tamanho de um array unidimensional na definição da função é somente por razões de legibilidade.

Até este ponto, parece que não há diferença entre passar uma variável simples e um array como argumento para uma função. Mas há uma diferença fundamental: **QUANDO DEFINIMOS UM ARRAY COMO ARGUMENTO FORMAL, ALTERAÇÕES NO ARRAY FEITAS DENTRO DA FUNÇÃO ALTERAM O CONTEÚDO DO ARRAY PASSADO COMO PARÂMETRO REAL NA CHAMDA DA FUNÇÃO. EM OUTRAS PALAVRAS, QUANDO SÃO PARÂMETROS DE FUNÇÕES, ARRAYS SÃO PASSADOS POR REFERÊNCIA (sem a necessidade do & na definição do parâmetro formal, como foi visto na Seção 11).**

Para ilustrar este conceito, considere o exemplo seguinte:

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

// Troca o valor de uma variavel
void troca( int a ){
   a = 20;
}

// Troca valores de elementos em um vetor
void troca_vet( int vet[] ){
   vet[0] = 60;
   vet[1] = 70;
   vet[2] = 80;
}

// Programa Principal
int main()
{
   int x, y;
   int v[3];

   x = 10;
   v[0] = 30;
   v[1] = 40;
   v[2] = 50;

   troca( x );
   cout << "x=" << x << endl;
   troca_vet( v );
   cout <<  "v[0]=" << v[0] << " v[1]=" << v[1] << " v[2]=" << v[2] ;
}
```

A saída deste programa é:

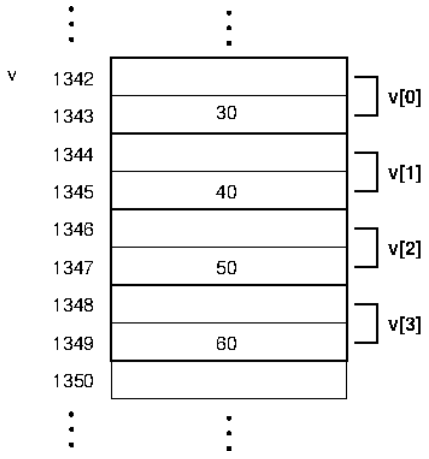
```
x=10
v[0]=60 v[1]=70 v[2]=80
```

O valor da variável `x` do programa principal não se altera porque como já vimos nas notas de aula 7, quando a função `troca` é chamada, o valor do argumento real `x` é avaliado, que é `10`, este valor é copiado para o parâmetro formal `a` da função `troca` e a função então é executada. O parâmetro `a` da função é tratada como variável local, portanto quando atribuímos `20` a `a`, estamos atribuindo `20` a uma variável local. Terminada a função, a execução retorna ao programa principal, que imprime o valor de `x`, que não foi alterado, ou seja, imprime `x=10`.

Quando a função `troca_vet` é chamada, o array `v` é passado como argumento e "copiado" para o parâmetro formal `vet`. A função é então executada, e os elementos do array são alterados para `60, 70, 80`. Como mencionado anteriormente, quando passamos um array como parâmetro, as alterações feitas no array dentro da função alteram o array passado como parâmetro. Portanto, quando a função termina e a execução continua no programa principal com a impressão dos valores dos elementos de `v`, será impresso `60, 70, 80`, os novos valores alterados de dentro da função `troca_vet`.

Vamos entender por que quando passamos só o nome do array como argumento as alterações afetam o array passado como parâmetro real. Como já mencionamos anteriormente, quando um array é definido, como `v` no programa principal acima, é alocado espaço suficiente na memória para conter todos os elementos do array. Na ilustração abaixo, são alocados 6 bytes de memória a partir do endereço `1342` para conter o array. O array como um todo não tem um valor, mas cada elemento do array tem (neste caso, foram inicializados com `30, 40, 50`). O nome do array, na verdade, contém o endereço onde começa o array, neste caso, o endereço `1342`.

Portanto, quando passamos o nome do array como argumento para uma função estamos na realidade passando como argumento o endereço de memória onde começa o array. No exemplo anterior, `1342` é passado como argumento para o parâmetro formal `vet` da função `troca_vet`. Portanto, da mesma forma que no caso da variável simples, o valor de `v`, que é o endereço `1342`, é copiado para o parâmetro `vet` de `troca_vet`. Então, quando a função `troca_vet` é executada, `vet` é um array de elementos do tipo `int` que começa no endereço `1342`. Quando atribuímos o valor `60` a `vet[0]`, estamos atribuindo `60` ao primeiro elemento do array que começa no endereço `1342`. Como este é o mesmo endereço onde começa o array `v` do programa principal, quando a função `troca_vet` termina, o array `v` "enxergará" o valor dos elementos do array que começa no endereço `1342`, que foram alterados pela função.




Quando passamos variáveis simples como argumento para uma função estamos passando somente o valor da variável, portanto, de dentro da função não é possível saber qual o endereço da variável para poder alterá-la.

Lembre-se que o endereço só é passado para a função quando passamos o array **COMO UM TODO** (ou seja, o nome do array, sem ser indexado por um elemento). Se passarmos como argumento apenas um elemento do array, o comportamento é o mesmo que se passássemos uma variável simples. Ou seja, o nome do array indexado por um valor entre colchetes refere-se ao valor do elemento do array, enquanto o nome do array sozinho refere-se ao endereço onde começa o array. Assim, no programa abaixo:

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

// Troca o valor de uma variavel
void troca( int a ){
   a = 20;
}

// Troca valores de elementos em um vetor
void troca_v( int vet[] ){
   vet[0] = 60;
   vet[1] = 70;
   vet[2] = 80;
}

// Programa Principal
int main(){
   int v[] = {30, 40, 50};

   troca( v[0] );
   cout << "v[0]=" << v[0] << endl;
   troca_v( v );
   cout <<  "v[0]=" << v[0] << " v[1]=" << v[1] << " v[2]=" << v[2] ;
}
```

A saída do programa é:

```
v[0]=30
v[0]=60 v[1]=70 v[2]=80
```

Outro exemplo: a função `inicializaArray` abaixo inicializa todos os elementos do array `valor` com um valor passado como argumento pelo programa principal.

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

#define TAMANHO 30

// função inicializaArray(a, k)
//  ação:         inicializa todos os elementos de a com k
//  entrada:      array de inteiros a, inteiro k
//  saida:        nenhum
//  suposições:   a tem TAMANHO elementos
//  algoritmo:    uma repeticao for, inicializando um elemento a
//                cada repeticao
```

```
//
void inicializaArray(int a[], int k)
{
  int i;

  i = 0;
  while (i < TAMANHO)
    {
      a[i] = k;
      i = i + 1;
    }
}

// Programa principal
int main()
{
  int valor[TAMANHO];

  // Inicializa todos os elementos do array com 42
  inicializaArray(valor, 42);

  // O resto do programa principal
  //  .
  //  .
  //  .

}
```

Como as alterações feitas por `inicializaArray` são vistas do programa principal, depois da função `inicializaArray` ser executada, no programa principal todos os elementos do array `valor` terão o valor `42`.

## 13.7 Exemplo: pesquisa linear de um array

Pesquisar (procurar) em um array um determinado valor (chamado de *chave*) é um problema muito comum em programação. Ele tem diversas aplicações. Por exemplo, podemos pesquisar um array de notas para verificar se algum aluno tirou `100` na prova. Há diversos algoritmos de pesquisa: cada um com suas vantagens e desvantagens. Nestas notas de aula, discutiremos um algoritmo simples, chamado de *pesquisa linear*. A pesquisa é feita usando uma repetição e examinando cada elemento do array a cada repetição e comparando o elemento com a chave que buscamos. A pesquisa termina quando um elemento do array que "casa" com a chave é encontrada, ou quando o array todo é percorrido e a chave procurada não é encontrada.

### 13.7.1 O Problema

Escreva uma função `pesquisa_linear` que tem como argumento de entrada: um array de inteiros a ser pesquisado, o tamanho do array, e uma chave (um valor inteiro) a ser procurado. A função retorna um inteiro: o índice do elemento do array (se a chave for achada) ou `-1` caso contrário.

1. Protótipo:

   ```
   int pesquisa_linear(int [], int, int);
   ```

2. Definição:

```
/* Procura uma chave em um array
 * entrada: array a ser pesquisado (arr ), tamanho do array (tam),
 *          chave a ser procurada (chave)
 * saida: o indice do elemento que e' igual a chave ou -1 caso nao ache
 * suposicao: nao assume que o array esteja ordenado
 */
int pesquisa_linear(int arr[], int tam, int chave)
{
  int i;

  for (i = tamanho - 1; i >= 0; i += 1)
  {
      if (arr[i] == chave)
      {
         return i;
      }
  }
  return -1;
}
```

## 13.8 Exemplo: somar os elementos de dois arrays

### 13.8.1 O Problema

Escrever uma função que some dois arrays de `floats`, do mesmo tamanho. Dar o resultado em um terceiro array. O tamanho dos arrays é também passado para a função.

1. Protótipo:

```
void soma_array( float [], float [], float [], int );
```

2. Definição de `soma_array()`:

```
void soma_array( float arr1[], float arr2[], float arr3[], int tam )
{
   int i;

   for(i = 0; i < tam; i += 1)
   {
       arr3[i] = arr1[i] + arr2[i];
   }
}
```

## 13.9 Exemplo: Ordenação de um vetor - Versão 1

Um outro programa muito popular com arrays é ordená-lo de acordo com algum critério. Por exemplo, um array de inteiros pode ser ordenado em ordem crescente ou decrescente. O apresentado a seguir é um algorítmo básico e nem um pouco eficiente, denominado *Select sort*.

Ele usa o fato simples de comparar cada elemento de um *array* com o restante deste. Quando se acha o menor, ocorre uma troca de valores entre o elemento sob análise e o outro elemento do *array* que é o menor.

Por exemplo, se começarmos com um array: `9 5 2 7 3 8 1 4 6`, (o primeiro elemento é `9` e o último elemento é `6`) isto é o que acontece com os elementos do array depois de cada passagem sobre ele (e consequente troca de valores):

```
passagem   conteudo do array depois da passagem
~~~~       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
0 -->       9  5  2  7  3  8  1  4  6

1 -->       1  5  2  7  3  8  9  4  6

2 -->       1  2  5  7  3  8  9  4  6

3 -->       1  2  3  7  5  8  9  4  6

4 -->       1  2  3  4  5  8  9  7  6

5 -->       1  2  3  4  5  8  9  7  6

6 -->       1  2  3  4  5  6  9  7  8

7 -->       1  2  3  4  5  6  7  9  8

8 -->       1  2  3  4  5  6  7  8  9
```

Note que mesmo que se começássemos com um array ordenado de 9 elementos, ainda assim o algoritmo dado faz 8 passagens sobre o array.

### 13.9.1 Protótipo da função e definição

1. Protótipo

```
void selectSort(int [], int);
```

2. Definicao

```
#include <iostream>

using namespace std;

#define TAM 9

void imprimePassos (int v[], int tam, int passo)
{
  int i;

  cout << "Passo " << passo << ": --> ";

  for(i = 0; i < tam; i += 1)
  {
    cout << v[i] << "  ";
```

```
  }
  cout << endl;
}

/* Uma funcao  que encontra o menor valor em um  array entre os indices
 * "a" e "b" (inclusive)
 */
int menor_indice(int v[], int a, int b)
{
  int i;
  int menor;

  menor = a;

  for (i = a+1; i <= b; i += 1)
  {
    if ( v[i] < v[menor] )
    {
      menor = i;
    }
  }

  return menor;
}


/* Uma funcao que troca os valores entre dois elementos de um array */
void troca_v( int vet[], int i, int j)
{
  int aux;

  aux = vet[i];
  vet[i] = vet[j];
  vet[j] = aux;
}

/* Uma funcao que ordena um array de inteiros usando o algoritmo de
 * Select sort.
 * Entrada:  array a ser ordenado -- lista[]
 *           tamanho do array -- tam
 */
void selectSort(int lista[], int tam)
{
  int i,j,
      idx_menor_elem;    /* indice do menor valor no array entre i e tam-1 *

  for(i = 0; i < tam; i += 1)
  {
    imprimePassos( lista, tam, i );
```

```
      idx_menor_elem = menor_indice ( lista, i, tam-1 );
      troca_v ( lista, i, idx_menor_elem );
    }
  }

  int main ()
  {
    int vetor[TAM], i;

    for(i=0; i < TAM; i += 1)
    {
      cin >> vetor[i];
    }

    selectSort(vetor, TAM);

  }
```

## 13.10 Exemplo: Ordenação de um vetor - Versão 2

O algoritmo abaixo é ligeiramente melhor que o anterior e é chamado *Bubble sort*. Ele é bastante simples, porém ainda não muito eficiente.

Basicamente, o algoritmo funciona da seguinte forma:

- na primeira passagem sobre o array: começando do último elemento do array até o segundo elemento, compare o valor de cada elemento com o valor do elemento anterior a ele. Se os elementos comparados estiverem fora de ordem, trocar os seus valores. Depois que esta primeira passada terminar, o que acontece é que o menor elemento do array torna-se o primeiro elemento do array.
- na segunda passagem pelo array: começando com o último elemento do array até o terceiro elemento, compare o valor de cada elemento com o valor do elemento anterior a ele. Se os dois elementos comparados estiverem fora de ordem, trocar os seus valores. Depois que esta passagem sobre o array terminar, o segundo menor elemento do array será o segundo elemento do array.
- repetir a passagem sobre o array de maneira similar até que a última passagem será simplesmente uma comparação dos valores do último elemento com o elemento anterior.

Por exemplo, se começarmos com um array: `9 5 2 7 3 8 1 4 6`, (o primeiro elemento é `9` e o último elemento é `1`) isto é o que acontece com os elementos do array depois de cada passagem sobre ele (e troca de valores adjacentes):

```
passagem   conteudo do array depois da passagem
~~~~       ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
0 -->       9  5  2  7  3  8  1  4  6

1 -->       1  9  5  2  7  3  8  4  6

2 -->       1  2  9  5  3  7  4  8  6

3 -->       1  2  3  9  5  4  7  6  8

4 -->       1  2  3  4  9  5  6  7  8
```

```
5 -->       1  2  3  4  5  9  6  7  8

6 -->       1  2  3  4  5  6  9  7  8

7 -->       1  2  3  4  5  6  7  9  8

8 -->       1  2  3  4  5  6  7  8  9
```

Note que, também aqui, mesmo que se começássemos com um array ordenado de 9 elementos, ainda assim o algoritmo dado faz 8 passagens sobre o array.

Isto pode ser melhorado da seguinte forma: Antes de começar cada passagem, inicializamos uma variável `ordenado` com 1. Se durante a passagem uma troca de valores ocorrer, trocamos o valor da variável para 0. Assim, se depois da passagem, o valor da variável continuar sendo 1, isso significa que nenhuma troca ocorreu e que o array está ordenado.

### 13.10.1 Algoritmo *Bubble Sort* otimizado

Enquanto o array nao estiver ordenado

1. inicializar `ordenado` com 1
2. comparar pares adjacentes do array
   troque seus valores se estiver fora de ordem
   `ordenado` = 0.

### 13.10.2 Protótipo da função e definição

1. Protótipo

```
void bubbleSort(int [], int);
```

2. Definicao

```
#include <iostream>

using namespace std;

#define TAM 9

void imprimePassos (int v[], int tam, int passo)
{
  int i;

  cout << "Passo " << passo << ": --> ";

  for(i = 0; i < tam; i += 1)
  {
    cout << v[i] << "  ";
  }
  cout << endl;
}
```

```
/* Uma funcao que troca os valores entre dois elementos de um array */
void troca_v( int vet[], int i, int j)
{
  int aux;

  aux = vet[i];
  vet[i] = vet[j];
  vet[j] = aux;
}

/* Uma funcao que ordena um array de inteiros usando o algoritmo de
 * Bubble sort.
 * Entrada:  array a ser ordenado -- lista[]
 *           tamanho do array -- tam
 */
void bubbleSort(int lista[], int tam)
{
  int ordenado,       /* se 1 depois da passagem o array esta' ordenado */
    elem_final = 1, /* em uma passagem, elementos do ultimo ate' elem_final
                       sao comparados com o elemento anterior */
    i,j,
    temp;

  /* enquanto o array nao estiver ordenado, fazer uma passagem sobre ele */
  ordenado = 0;
  while (ordenado == 0)
  {
    ordenado = 1;  /* assume que array esta' ordenado */

    /* Examina o array do ultimo elemento ate elem_final.  Compara
       cada elemento com o anteior e troca seus valores se estiver
       fora de ordem.  */

    imprimePassos(lista, tam, elem_final - 1);

    for(i = tam - 1; i >= elem_final; i -= 1)
    {
      if (lista[i] < lista[i - 1])  /* troca os elementos de i e i-1 */
      {
        troca_v (lista, i, i - 1);
        ordenado = 0;         /* marca como nao ordenado */
      }
    }

    elem_final = elem_final + 1;
  }
}

int main ()
```

```
    {
      int vetor[TAM], i;

      for(i=0; i < TAM; i += 1)
      {
        cin >> vetor[i];
      }

      bubbleSort(vetor, TAM);

    }
```

## 13.11 Comentários Finais

Neste curso, um dos únicos lugares que veremos o nome do array sem estar indexado é quando passamos o array (como um todo) para uma função. Para outras finalidades, veremos sempre o array indexado. Por exemplo, o seguinte trecho de programa está errado:

```
  int main(){
    int arr1[4] = {10, 20, 30, 40};
    int arr2[4];

    arr2 = arr1;        // ERRADO: NÃO copia arr1 em arr2
                        // tem que copiar elemento por elemento

    if( arr1 == arr2 ) // ERRADO: NÃO podemos comparar arrays inteiros
      cout << "X";     // tem que comparar elemento por elemento
}
```

# 14 Matrizes ou Arrays Multidimensionais

Nas notas de aula anteriores, apresentamos arrays unidimensionais. Em C++ , é possível também definir arrays com 2 ou mais dimensões. Eles são arrays de arrays. Um array de duas dimensões podem ser imaginado como uma matriz (ou uma tabela).

Como você deve ter imaginado, para definir e acessar arrays de dimensões maiores, usamos colchetes adicionais (`[` e `]`). Por exemplo:

```
int tabela[3][5];
```

Define um array bidimensional chamado `tabela` que é uma matriz 3 por 5 de valores do tipo `int` (15 valores no total). Os índices da primeira dimensão vão de 0 a 2, e os índices da segunda dimensão vão de 0 a 4.

Abaixo apresentamos um programa que imprime os elementos de um array bidimensional.

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

#define ALTURA 5
#define LARGURA 5

int main()
{
   int x;                              //  numero da coluna
   int y;                              //  numero da linha
   char matriz [ALTURA] [LARGURA];     //  array 2-D [num_lins, num_cols]


   //  preenche a matriz com zeros

   for(y = 0; y < ALTURA; y += 1)
   {
     for(x = 0; x < LARGURA; x += 1)
     {
        matriz[y][x] = 0;
     }
   }

   //  Imprime a matriz com zeros e a coordenada escolhida com 1

   cout << endl << "Entre coordenadas na forma \"y x\"." << endl;
   cout << "Use valores negativos para sair do programa." << endl;

   cout << "Coordenadas: ";
   cin >> y >> x;

   while (x >= 0 && y >= 0)
   {
      matriz[y][x] = 1;      //  coloca 1 no elemento escolhido

      for(y = 0; y < ALTURA; y += 1)  // imprime o array todo
```

```
      {
        for(x = 0; x < LARGURA; x += 1)
        {
             cout << matriz[y][x] << " ";
        }
        cout << endl << endl;
      }

      cout << endl;
      cout << "Coordenadas: ";
      cin >> y >> x;
   }
}
```

Neste exemplo, `matriz` é um array bidimensional. Ela tem número de elementos igual a `ALTURAxLARGURA`, sendo cada elemento do tipo `int`.

O exemplo abaixo preenche os elementos de um array bidimensional com os valores que representam a taboada e imprime a matriz.

```
//  Exemplo de array 2-D - taboada

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

#define LIN 10
#define COL 10

int main()
{
  int x;                          //  numero da coluna
  int y;                          //  numero da linha
  int tabela[LIN] [COL];          //  tabela de taboada

  //  preenche a tabela

  for (y=0; y < LIN; y+=1)
    for(x=0; x < COL; x+=1)
      tabela[y][x] = y*x;

  cout << "\n         Tabela de Multiplicacao\n";

  //  Imprime o numero das colunas

  cout << setw(6) << 0;
  for (x=1; x < COL; x+=1)
    cout << setw(3) << x;
  cout << endl;

  //  Imprime uma linha horizontal
  cout << "   ";
```

```
  for (x=0; x < 3*COL; x+=1)
    cout << "-";
  cout << endl;

  //  Imprime as linhas da tablea.
  //  Cada linha a precedida pelo indice de linha e uma barra vertical

  for (y=0; y < LIN; y+=1) {
    cout << setw(2) << y << "|";
    for(x=0; x < COL; x+=1)
      cout << setw(3) << tabela[y][x];
    cout << endl;
  }
}
```

A saída do programa é:

```
          Tabela de Multiplicacao
     0  1  2  3  4  5  6  7  8  9
   ------------------------------
 0|  0  0  0  0  0  0  0  0  0  0
 1|  0  1  2  3  4  5  6  7  8  9
 2|  0  2  4  6  8 10 12 14 16 18
 3|  0  3  6  9 12 15 18 21 24 27
 4|  0  4  8 12 16 20 24 28 32 36
 5|  0  5 10 15 20 25 30 35 40 45
 6|  0  6 12 18 24 30 36 42 48 54
 7|  0  7 14 21 28 35 42 49 56 63
 8|  0  8 16 24 32 40 48 56 64 72
 9|  0  9 18 27 36 45 54 63 72 81
```

## 14.1 Inicialização

Arrays multidimensionais podem ser inicializados usando listas aninhadas de elementos entre chaves. Por exemplo, um array bidimensional `tabela` com três linhas e duas colunas pode ser inicializado da seguinte forma:

```
double tabela[3][2] = { {1.0,  0.0},      //  linha 0
                        {-7.8, 1.3},      //  linha 1
                        {6.5,  0.0}       //  linha 2
                      };
```

Quando o array é inicializado, o tamanho da primeira dimensão pode ser omitido. A definição de array abaixo é equivalente a dada anteriormente.

```
double tabela[][2] = { {1.0,  0.0},      //  linha 0
                       {-7.8, 1.3},      //  linha 1
                       {6.5,  0.0}       //  linha 2
                     };
```

## 14.2 Arrays Multidimensionais – arrays de arrays

O formato da definição de um array de dimensão $k$, onde o número de elementos em cada dimensão é $n_0, n_1, \ldots, n_{k-1}$, respectivamente, é:

$$nome\_tipo\ nome\_array[n_0][n_1]...[n_{k-1}];$$

Isto define um array chamado $nome\_array$ consistindo de um total de $n_0 \times n_1 \times \ldots \times n_{k-1}$ elementos, sendo cada elemento do tipo $nome\_tipo$.

Arrays multidimensionais são armazenados de forma que o último subscrito varia mais rapidamente. Por exemplo, os elementos do array

```
int tabela[2][3];
```

são armazenados (em endereços consecutivos de memória) como

```
tabela[0][0], tabela[0][1], tabela[0][2], tabela[1][0], tabela[1][1], tabela[1][
```

Um array de dimensão k, onde o número de elementos em cada dimensão é $n_0, n_1, \ldots, \ldots, n_{k-1}$, respectivamente, pode ser imaginado como um array de dimensão $n_0$ cujos elementos são arrays de dimensão $k-1$.

Por exemplo, o array bidimensional `tabela`, com 20 elementos do tipo `int`

```
int tabela[4][5] = { {13, 15, 17, 19, 21},
                     {20, 22, 24, 26, 28},
                     {31, 33, 35, 37, 39},
                     {40, 42, 44, 46, 48} };
```

pode ser imaginado como um array unidimensional de 4 elementos do tipo `int[]`, ou seja, arrays de `int`; cada um dos 4 elementos é um array de 5 elementos do tipo `int`:

```
tabela[0]  ---> {13, 15, 17, 19, 21}
tabela[1]  ---> {20, 22, 24, 26, 28}
tabela[2]  ---> {31, 33, 35, 37, 39}
tabela[3]  ---> {40, 42, 44, 46, 48}
```

## 14.3 Arrays Multidimensionais como argumento para funções

Quando o parâmetro formal de uma função é um array multidimensional (um array com dimensão maior que um), todas as dimensões deste array, exceto a primeira, precisa ser explicitamente especificada no cabeçalho e protótipo da função.

$$tipo\_do\_resultado\ nome\_da\_funcao\,(\,nome\_do\_tipo\ nome\_do\_array[\,][n_1]...[n_{k-1}], \ldots\ldots\,)$$

Quando uma função com um parâmetro formal do tipo array é chamada, na chamada da função **somente** o nome do array é passado como parâmetro real. O tipo (e portanto a dimensão) do array passado como parâmetro real deve ser consistente com o tipo (e portanto a dimensão) do array que é o parâmetro formal. O programa abaixo mostra o exemplo da tabela de multiplicação escrita usando funções.

```
//  Exemplo de array 2-D - tabela de multiplicacao

#include <iostream>
#include <iomanip>
```

```
using namespace std;

#define LIN 10
#define COL 10

void inicializa_arr (int arr[][COL], int);
void imprime_arr (int arr[][COL], int);

int main()
{
  int  tabela[LIN] [COL];

  inicializa_arr(tabela, LIN);

  cout << "\n         Tabela de Multiplicacao\n";

  imprime_arr(tabela, LIN);
}

//  Inicializa o array com a tabela de multiplicacao

void inicializa_arr (int arr[][COL], int nlin)
{
  int x;                          //  numero da coluna
  int y;                          //  numero da linha

  //  preenche o array

  for (y=0; y < nlin; y++)
    for(x=0; x < COL; x++)
      arr[y][x] = y*x;
}

//  imprime um array LIN x COL

void imprime_arr(int arr[][COL], int nlin)
{
  int x;                          //  numero da coluna
  int y;                          //  numero da linha

  //  imprime o numero das colunas

  cout << setw(6) << 0;
  for (x=1; x < COL; x++)
    cout << setw(3) << x;
  cout << endl;


  //  imprime uma linha horizontal
  cout << "   ";
```

```
  for (x=0; x < 3*COL; x++)
    cout << "_";
  cout << endl;

  //  imprime as linhas do array.  cada linha e' precedida
  //  pelo numero da linha e uma barra vertical

  for (y=0; y < nlin; y++) {
    cout << setw(2) << y << "|";
    for(x=0; x < COL; x++)
      cout << setw(3) << arr[y][x];
    cout << endl;
  }
}
```

Outro exemplo com funçoes de manipulação de arrays bidimensionais:

```
//  funcoes com argumentos tipo array 2-D

#include <iostream>
using namespace std;

#define ALTURA 7
#define LARGURA 7

void seleciona_elem    (char [][LARGURA], int);
void pontos       (char [][LARGURA], int);
void imprime_matriz  (char [][LARGURA], int);
void marca_triang (char [][LARGURA], int);
void flip       (char [][LARGURA], int);
void espera_entrada(void);

// *** DEFINICAO DE FUNCOES *******

//  funcao que imprime um array 2-D nlin X LARGURA
void imprime_matriz(char matriz[][LARGURA], int nlin)
{
  int x,y;

  for(y=0; y<nlin; y+=1)
    {
      for(x=0; x<LARGURA; x+=1)
        cout <<  matriz[y][x] << " ";
      cout << endl << endl;
    }
  cout << endl;
}

//  funcao que preenche uma matriz nlin X LARGURA com pontos
void pontos( char matriz[][LARGURA], int nlin)
{
```

```
  int x,y;

  for(y=0; y<nlin; y+=1)
    for(x=0; x<LARGURA; x+=1)
      matriz[y][x] = '.';

}

/*  funcao que preenche os elementos selecionados da matriz com um
 * quadrado e imprime a matriz
 */
void seleciona_elem(char matriz[][LARGURA], int nlin)
{
  int x, y;

  cout << "\nEntre com as coordenadas na forma \"y x\"." << endl;
  cout << "Use numeros negativos para terminar." << endl;

  cout << "Coordenadas: ";
  cin >> y >> x;
  while (x >= 0 && y >= 0)
  {
    matriz[y][x]='@';        // preenche o elemento com quadrado
    imprime_matriz(matriz, nlin);   // imprime a matriz
    cout << "Coordenadas: ";
    cin >> y >> x;
  }
}

/*  funcao que marca todos os elementos abaixo da diagonal principal de
 * um array nlin X LARGURA com quadrados
 */
void marca_triang(char matriz[][LARGURA], int nlin)
{
  int x, y;

  cout << "Triangulo" << endl;
  pontos(matriz, nlin);
  for (y = 0; y < nlin; y+=1)
    for (x = 0; x <= y; x+=1)
      matriz[y][x] = '@';
}

//  funcao que rotaciona  ('flip') cada linha array tendo  o elemento da
//  diagonal principal como centro da rotaÃ§Ã£o
void flip(char matriz[][LARGURA], int nlin)
{
  int x, y;
  int temp;
```

```
  cout << "Flip ao longo da diagonal principal." << endl;
  for (y = 0; y < nlin; y+=1)
    for (x = 0; x <= y; x+=1){
      temp = matriz[y][x];
      matriz[y][x] = matriz[x][y];
      matriz[x][y] = temp;
    }
}

//  funcao que espera ate que uma tecla qualquer seja digitada
void pausar() {
  char c;
  cin.get(c);
}

// ********* MAIN ***********

//  alguns exemplos de chamadas de funcoes com argumentos array 2-D
int main()
{
  char matriz [ALTURA] [LARGURA];

  pontos(matriz, ALTURA);
  seleciona_elem(matriz, ALTURA);
  pausar();

  flip(matriz, ALTURA);
  imprime_matriz(matriz,ALTURA);
  pausar();

  marca_triang( matriz, ALTURA);
  imprime_matriz( matriz, ALTURA);
  pausar();

  flip( matriz, ALTURA);
  imprime_matriz(matriz, ALTURA);
  pausar();
}
```

# Parte II
# Tópicos Avançados

As seções seguintes apresentam temas mais avançados da linguagem C++ , não abordadas em sala de aula. Elas podem ser estudadas pelo aluno como atividade complementar, pois apresentam mecanismos bastantes úteis em programas mais complexos (tratamento de arquivos e textos, manipulação dinâmica de memória, etc.).

# 15 Operadores e Expressões Especiais

## 15.1 Operadores de Incremento e Decremento

Há alguns operadores em C++ que são equivalentes as seguintes expressões (que são bastante comuns em programas):

```
k = k + 1;

j = j - 1;
```

Estes operadores adicionais, que são `++` e `--`, podem ser usados para encurtar as operações acima:

```
k++;

j--;
```

Estes operadores também podem ser colocados depois do nome da variável:

```
++k;

--j;
```

O fato do operador de incremento ser colocado antes ou depois da variável não altera o efeito da operação – o valor da variável é incrementada ou decrementada de um. A diferença entre os dois casos é QUANDO a variável é incrementada. Na expressão `k++`, o valor de `k` é primeiro usado e então é incrementada – isto é chamado *pós-incremento*. Na expressão `++k`, `k` é incrementado primeiro, e então o valor (o novo valor) de `k` é usado – isso é chamado *pré-incremento*.

A diferença é ilustrada nos seguintes exemplos:

```
int main()
{
   int k = 5;

   cout << "k1 = " << k << endl;
   cout << "k2 = " << k++ << endl;
   cout << "k3 = " << k << endl;
}
```

O programa acima (que usa pós-incremento) imprimirá o seguinte:

```
k1 = 5
k2 = 5
k3 = 6
```

A segunda linha impressa com o valor de `k` é `5` porque o valor de `k++` era `5`, e `k` é `6` depois da impressão.

Para o programa:

```
int main()
{
   int k = 5;

   cout << "k1 = " << k << endl;
   cout << "k2 = " << ++k << endl;
   cout << "k3 = " << k << endl;
}
```

O programa, que usa pré-incremento, terá a seguinte saída:

```
k1 = 5
k2 = 6
k3 = 6
```

A segunda linha impressa é `6` porque o valor de `++k` é `6`.

Os operadores de atribuição não podem ser usados com expressões aritméticas. Por exemplo, as expressões

```
(ack + 2)++;

(nope + 3) += 5;
```

resultarão em erros de compilação.

Finalmente, quando usar o operador de incremento em um `cout`, tome cuidado para não fazer o seguinte:

```
cout << ++uhoh << uhoh * 2 << endl;
```

Embora isso seja perfeitamente legal em C++ , os resultados não são garantidados que sejam consistentes. A razão para isso é que não há garantia que os argumentos do `cout` sejam avaliados em uma determinada ordem. O resultado do `cout` será diferente dependendo se `++uhoh` é avaliado primeiro ou depois de `uhoh * 2`.

A solução para este problema é escrever o seguinte:

```
++uhoh;
cout << uhoh << uhoh * 2 << endl;
```

## 15.2 Expressões como Valor com Operadores de incremento e decremento

Já que incremento e decremento são formas de atribuição, o operando deve ser um *lvalue*. O valor de uma expressão de incremento ou decremento depende se o operador é usado na notação PRé ou PóS fixada (`x++, ++x, x--, --x`). Se for pré-fixada, o valor da expressão é o novo valor após o incremento ou decremento. Se for pós-fixada, o valor da expressão é o valor antigo (antes do incremento ou decremento). Por exemplo no caso de incremento, a expressão:

`x++` tem o valor de `x`
`++x` tem o valor de `x + 1`

Note que não importando a notação usada, o valor de `x` (o conteúdo do endereço de memória associada a `x`) será `x + 1`. A diferença está no valor das expressões `x++` e `++x`, não no valor de `x` (em ambos os casos o valor de `x` será incrementada de um).

## 15.3 Ambiguidade em certas expressões

Às vezes, problemas podem acontecer devido o fato que C++ não especifica a ordem de avaliação dos operadores em uma operação binária. Em outras palavras, em expressões como `a + b` ou `a < b`, não há maneira de saber se o valor de `a` será avaliado antes ou depois de `b` (pense em `a` e `b` como sendo qualquer expressão, não somente variáveis.) Qual deles será avaliado primeiro é particular de cada compilador, e diferentes compiladores em máquinas diferentes podem ter resultados diferentes. Portanto, se a avaliação de um dos operadores pode alterar o valor do outro, o resultado pode ser diferente dependendo da ordem de avaliação. Portanto, em expressões do tipo `x + x++`, o valor pode diferir dependendo do compilador utilizado. Isto porque não sabemos quando exatamente o incremento de `x` ocorre. Outros maus exemplos: `y = x + x--` e `x = x++`. De forma geral, para evitar este problema, não utilize senteças como estas.

### 15.3.1 Valor de expressões envolvendo atribuição aritmética

Como podemos ver, os operadores de incremento/decremento, e os operadores de atribuição aritmética vistos até aqui funcionam de forma similar ao comando de atribuição. O lado esquerdo da expressão de atribuição aritmética e o operando de incremento/decremento devem ser um *lvalue*. O valor da expressão da é igual ao valor da sentença de atribuição correspondente. Por exemplo:

`x += 3` é igual a `x = x + 3` e tem valor `x + 3`
`x *= y + 1` é igual a `x = x * (y + 1)` e tem valor `x * (y + 1)`
`y = i++` é igual a `y = i, i = i + 1;` e tem valor `i`
`y = ++i` é igual a `i = i + 1, y = i;` e tem valor `i + 1`

## 15.4 Exercício de fixação

O que é impresso pelos dois programas abaixo?

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

int main() {
  int n1, n2, n3;

  cout << "Entre com um numero inteiro: ";
  cin >> n1;
  n1 += n1 * 10;
  n2 = n1 / 5;
  n3 = n2 % 5 * 7;
  n2 *= n3-- % 4;
  cout << n2 << " " << n3 << " " << (n2 != n3 + 21) << endl;
}
```

## 15.5 Precedência de Operadores

A tabela de precedência abaixo mostra a posição dos operadores vistos aqui, incluindo a atribuição aritmética e decremento/incremento.

| Operador | Associatividade |
|---|---|
| `()  []  ->  .` | esquerda para direita |
| `!  -  ++  --  *  \&` | direita para esquerda |
| `*  /  %` | esquerda para direita |
| `+  -` | esquerda para direita |
| `<  <=  >  >=` | esquerda para direita |
| `==  !=` | esquerda para direita |
| `&&` | esquerda para direita |
| `\|\|` | esquerda para direita |
| `=  +=  -=  *=  /=  %=` | direita para esquerda |
| `,` | esquerda para direita |

# 16 Mais sobre tipos: conversão implícita e explícita

Expressões não tem somente um valor, mas também tem um tipo associado.

Se ambos os operandos de uma operação aritmética binária são do mesmo tipo, o resultado terá o mesmo tipo. Por exemplo:

`3 + 5` é `8`, e o tipo é `int`
`3.5 + 2.25` é `5.75`, e o tipo é `double`

O único comportamento não óbvio é a da divisão de inteiros:

`30 / 5` é `6`
`31 / 5` é `6`
`29 / 5` é `5`
`3 / 5` é `0`

Lembre-se de evitar escrever algo como `1 / 2 * x` significando $\frac{1}{2}x$. Você sempre obterá o valor `0` porque `1 / 2 * x` é `(1 / 2) * x` que é `0 * x` que é `0`. Para obter o resultado desejado, você poderia escrever `1.0 / 2.0 * x`.

## 16.1 Conversão de tipos

Valores podem ser convertidos de um tipo para outro implicitamente, da forma já comentada anteriormente.

Em expressões envolvendo operadores binários com operandos de tipos diferentes, os valores dos operandos são convertidos para o mesmo tipo antes da operação ser executada: tipos mais simples são "promovidos" para tipos mais complexos. Portanto, o resultado da avaliação de uma expressão com operandos de tipos diferentes será o tipo do operando mais complexo. Os tipos em C++ são (do mais simples para o mais complexo):

```
char < int < long < float < double
```

O sinal de < significa que o tipo da esquerda é promovido para o tipo da direita, e o resultado será do tipo mais a direita. Por exemplo:

`3.5 + 1` é `4.5`
`4 * 2.5` é `10.0`

Esta regra estende-se para expressões envolvendo múltiplos operadores, mas você deve se lembrar que a precedência e associatividade dos operadores pode influenciar no resultado. Vejamos o exemplo abaixo:

```
int main()
{
  int a, b;

  cout << "Entre uma fracao (numerador e denominador): ";
  cin >> a >> b;

  cout << "A fracao em decimal e  " << 1.0 * a / b << endl;
}
```

Multiplicando por `1.0` assegura que o resultado da multiplicação de `1.0` por `a` será do tipo real, e portanto, a regra de conversão automática evitará que o resultado da divisão seja truncado. Note que se tivéssemos primeiro feito a divisão `a/b` e depois multiplicado por `1.0`, embora o tipo da expressão `a/b*1.0` seja do tipo `double`, o valor da expressão seria diferente do valor de `1.0 * a/b`. Por que ?

Em atribuições, o valor da expressão do lado direito é convertido para o tipo da variável do lado esquerdo da atribuição. Isto pode causar promoção ou "rebaixamento" de tipo. O "rebaixamento" pode causar perda de precisão ou mesmo resultar em valores errados.

Em operações de atribuição, atribuir um `int` em um `float` causará a conversão apropriada, e atribuir um `float` em um `int` causará truncamento. Por exemplo:

```
float a = 3;     é equivalente a a = 3.0
int a = 3.1415; é equivalente a a = 3 (truncado)
```

Basicamente, se o valor da expressão do lado direito da atribuição é de um tipo que não cabe no tamanho do tipo da variável do lado esquerdo, resultados errados e não esperados podem ocorrer.

## 16.2 Modificadores de tipos

Os tipos de dados básicos em C++ podem estar acompanhados por modificadores na declaração de variáveis. Tais modificadores são: **long**, **short**, **signed** e **unsigned**. Os dois primeiros têm impacto no tamanho (número de bits) usados para representar um valor e os dois últimos indicam se o tipo será usado para representar valores negativos e positivos (**signed**) ou sem este modificador) ou apenas positivos (**unsigned**).

A Tabela 4 mostra uma lista completa de todos os tipos de dados em C++ , com e sem modificadores:

| Modificador | Tamanho em bits | Faixa de valores |
|---|---|---|
| char | 8 | -127 a 127 |
| unsigned char | 8 | 0 a 255 |
| int | 16 | -32767 a 32767 |
| unsigned int | 16 | 0 a 65535 |
| short int | 16 | -32767 a 32767 |
| unsigned short int | 16 | 0 a 65535 |
| long int | 32 | -2147483647 a 2147483647 |
| unsigned long int | 32 | 0 a 4294967295 |
| float | 32 | Mantissa de 6 dígitos |
| double | 64 | Mantissa de 10 dígitos |
| long double | 80 | Mantissa de 10 dígitos |

Tabela 4: Modificadores de Tipos de Dados

## 16.3 *Cast* de tipos

C++ tem um operador para alterar o tipo de um valor explicitamente. Este operador é chamado de *cast*. Executando um *cast* de tipos, o valor da expressão é forçado a ser de um tipo particular, não importando a regra de conversão de tipos.

O formato do *cast* de tipos é:

$$(nome\text{-}do\text{-}tipo)\ expressao$$

O parênteses NÃO é opcional na expressão acima.
Podemos usar o *cast* de tipos da seguinte forma:

```
int fahr = 5;
float cels;

cout << "Valor = " << (float) fahr << endl;
cels = (float)5 / 9 * (fahr - 32);
cout << "celsius = " << (int) cels << endl;
```

Agora que conhecemos o operador de *cast* de tipo podemos reescrever o programa que faz a conversão de fração para decimal.

```
int main()
{
  int a, b;

  cout << "Entre com uma fracao (numerador e denominador): ";
  cin >> a >> b;

  cout << "A fracao em decimal e  " << (float) a / b << endl;
}
```

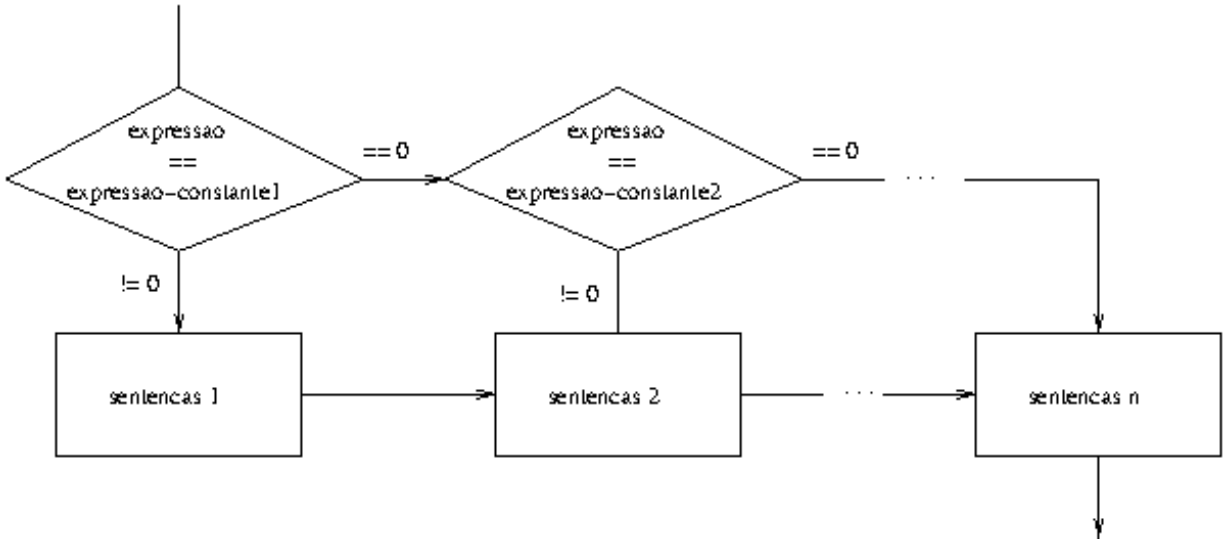
O *cast* de tipo tem a maior precedência possível, portanto podemos fazer o *cast* de `a` ou de `b` para ser do tipo `float`, e não há necessidade de parênteses extra. No exemplo acima, o *cast* causa o valor da variável `a` ser convertido para `float`, mas não causa mudança no tipo da variável `a`. O tipo das variáveis é definido uma vez na declaração e não pode ser alterado.

## 17 A sentença `switch`

A sentença `switch` é outra maneira de fazer decisões múltiplas. Ele pode ser usado para testar se uma dada expressão é igual a um valor constante e, dependendo do valor, tomar determinadas ações.

O formato da sentença `switch` é:

```
switch (expressao) {
        case expressao-constante 1:
                sentencas 1
        case expressao-constante 2:
                sentencas 2
                ⋮
        default:
                sentencas n
}
```




A sentença `switch` primeiro avalia a *expressão*. Se o valor da expressão for igual a uma das *expressões constantes*, as *sentenças* que seguem o `case` são executados. Se o valor da *expressão* não for igual a nenhuma das constantes, as sentenças que seguem `default` são executadas.

As *sentenças* que seguem o `case` são simplesmente uma lista de sentenças. Esta lista pode conter mais de uma sentença e não é necessário colocá-las entre chaves (`{` e `}`). A lista de sentenças também pode ser vazia, isto é, você pode não colocar nenhuma sentença seguindo o `case`.

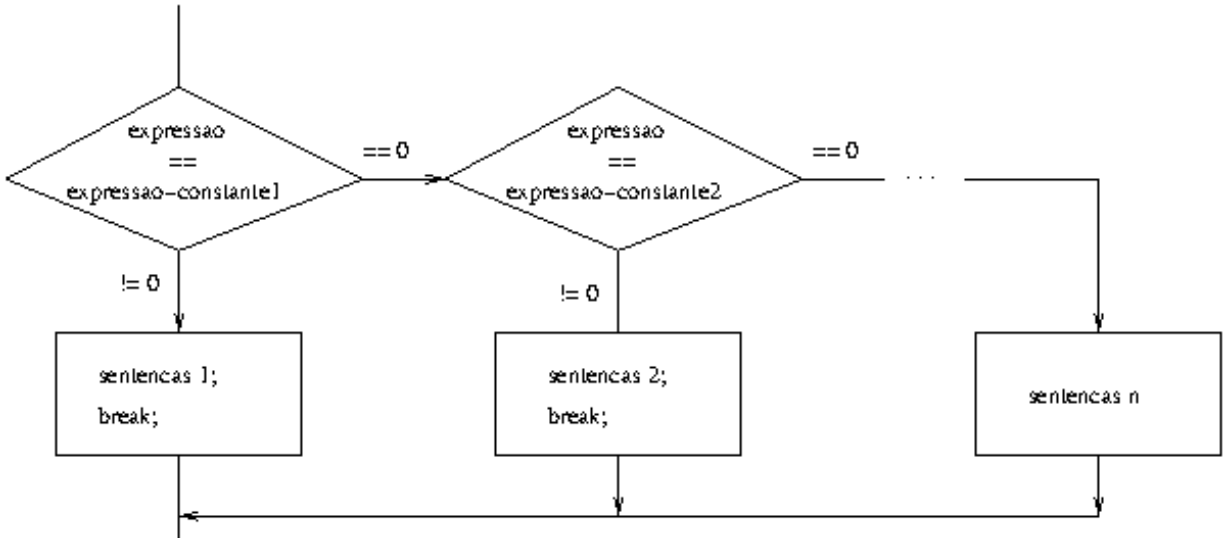
Também não é obrigatório colocar o `default`. Só o use quando for necessário.

Note no diagrama acima que TODAS as sentenças que seguem a constante com o valor igual ao da expressão serão executados. Para que se execute APENAS as sentenças que seguem o `case` que seja igual ao valor da expressão precisamos usar a sentença `break`, que veremos em seguida.

## 18 A sentença `break`

O `break` faz com que todas as sentenças que o seguem dentro da mesma sentença `switch` sejam ignorados. Ou seja, colocando a sentença `break` no final de uma sentença `case` faz com que as sentenças que seguem os `case`s subsequentes não sejam executadas. Em geral, é este o comportamento desejado quando se usa o `switch`, e `case`s sem o `break` no final são de pouca utilidade. Portanto, o uso de sentenças `case` sem o `break` devem ser evitados e quando utilizados devem ser comentados ao lado com algo como `/* continua proxima sentenca - sem break */`.

Com a sentença `break` o diagrama de fluxo fica:




Note a similaridade com o diagrama da sentença `else-if` e a diferença com o diagrama da sentença `switch` acima.

O próximo programa tem a mesma função de calculadora do programa anterior, porém utilizando a sentença `switch`.

**Exemplo 11:**

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

int main( ){

  float num1, num2;
  char op;

  // obtem uma expressao do usuario
  cout << "Entre com numero operador numero\n";
  cin >> num1 >> op >> num2;

  switch (op) {
  case '+':
    cout << " = " << setprecision(2) << num1 + num2;
    break;
  case '-':
    cout << " = " << setprecision(2) << num1 - num2;
    break;
  case '*':
    cout << " = " << setprecision(2) << num1 * num2;
    break;
  case '/':
    cout << " = " << setprecision(2) << num1 / num2;
    break;
  default:
    cout << " Operador invalido.";
    break;
```

```
  }
  cout << endl;
}
```

Como mencionado anteriormente, é possível não colocar nenhuma sentença seguindo um `case`. Isso é útil quando diversas sentenças `case` (diversas constantes) têm a mesma ação.

Por exemplo, podemos modificar o programa acima para aceitar `x` e `X` para multiplicação e `\` para divisão. O programa fica então:

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

int main( ){

  float num1, num2;
  char op;

  // obtem uma expressao do usuario
  cout << "Entre com numero operador numero\n";
  cin >> num1 >> op >> num2;

  switch (op) {
  case '+':
    cout << " = " << setprecision(2) << num1 + num2;
    break;
  case '-':
    cout << " = " << setprecision(2) << num1 - num2;
    break;
  case '*':
  case 'x':
  case 'X':
    cout << " = " << setprecision(2) << num1 * num2;
    break;
  case '/':
  case '\\':
    cout << " = " << setprecision(2) << num1 / num2;
    break;
  default:
    cout << " Operador invalido.";
    break;
  }
  cout << endl;
}
```

**Exercício 2:** Ler mes e ano e imprimir o numero de dias do mes no ano digitado.

# 19 Estruturas de Repetição

A linguagem C++ possui comandos para repetir uma sequência de instruções. Estas **estruturas de repetição**, também conhecidas como **laços** (do inglês ***loops***). As construções while e for foram vistas na Seção 9. Agora veremos do ... while e novas sentenças associadas a estruturas de repetições.

## 19.1 A Estrutura de Repetição do...while

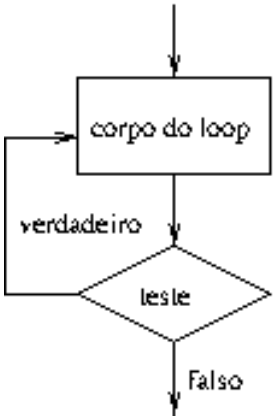
Há outro comando de repetição em linguagem C++ . O do...while é bastante parecido com while, com a diferença que a expressão de teste é avaliada DEPOIS que o corpo da repetição é executado.

O formato do do...while é:

do

*corpo da repetição*

while (*expressão teste* )




O exemplo abaixo usa o do...while:

```
int contador = 0;

do {
   cout <<  "contador = " << contador << endl;
   contador += 1;
} while( contador < 5 );

cout << "ACABOU !!!!\n";
```

A execução deste programa é idêntico ao primeiro exemplo mostrado para o comando while, com a expressão de teste mudada para o final.

Saída:

```
contador = 0
contador = 1
contador = 2
contador = 3
contador = 4
ACABOU !!!!
```

O do...while é usado quando o corpo da repetição deve ser executado pelo menos uma vez. Um exemplo comum disto é o processamento da entrada de um programa.

**Exemplo 3:** Neste exemplo, o teste do laço é baseado no valor digitado pelo usuário. O laço deve ser executado pelo uma vez antes que o teste sobre o valor seja executado.

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

int main( ){

  int num;

  cout << "Entre com um numero par:\n";

  do{
    cin >> num;
  } while( num % 2 != 0 );

  cout << "Obrigado.\n";
}
```

**Exemplo de execução:**

```
Entre com um numero par:
3
1
5
4
Obrigado.
```

Neste caso, o valor da variável num é digitado pelo usuário. Depois disso, o teste é executado para verificar se o número é par (o teste `num % 2 != 0` é falso se num é par já que o resto da divisão de qualquer número par por 2 é zero).

É possível escrever o programa acima usando while:

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

int main( ){

  int num;       // Atribui um numero impar a  num

  cout << "Entre com um numero par:\n";

  num = 1;
  while( num % 2 != 0 ){
    cin >> num;
  }
  cout << "Obrigado.\n";
}
```

O problema com este programa é que a variável num deve ser inicializada com um valor que torne o teste do laço verdadeiro. Neste exemplo, é simples encontrar tal valor. Para uma expressão teste mais complicada, isso pode não ser tão fácil.

## 20 A sentença `break`

Em uma estrutura de repetição, o `break` faz com que todas as sentenças que o seguem dentro sejam ignorados. Além disso, o `break` também TERMINA a repetição, fazendo com que o programa continue sua execução APÓS o bloco da repetição. A expressão da repetição e a reinicialização (no caso do `for`) não são avaliadas após a execução de um `break`.

## 21 A sentença `continue`

Em uma estrutura de repetição, o `continue` faz com que todas as sentenças que o seguem sejam ignorados. Além disso, esta sentença faz com que a reinicialização e a expressão da repetição sejam avaliadas e a execução do bloco de repetição é iniciado novamente.

Essencialmente, o `continue` significa "ignore o restante do bloco de repetição e parta para a próxima interação".

# 22 Tipo Enumerado

Em muitos programas, variáveis do tipo `int` são utilizadas não por suas propriedades numéricas, mas para representar uma escolha dentre um pequeno número de alternativas. Por exemplo:

```
int sexo;  // masculino = 1 , feminino = 2
int cor;   // vermelho = 1 , amarelo = 2 , verde = 3
```

A utilização de códigos para representar os valores que uma variável poderá assumir, certamente compromete a clareza da estrutura de dados do programa, tornando sua lógica obscura e inconsistente. Por exemplo:

```
cor = 3;
if( sexo == 2 ) ...
cor = cor + sexo;
for( cor = 1; cor < 10; cor ++ )...
```

Um tipo enumerado permite definir uma lista de valores que uma variável deste tipo poderá assumir. A definição de um tipo enumerado é feita da seguinte forma:

$$\texttt{enum}\ Nome\_do\_tipo\ \{\ valor_1, valor_2, \ldots, valor_n\ \};$$

Exemplos de definição de tipos enumerados:

```
enum TpCores {VERMELHO, AMARELO, VERDE};
enum TpDias {SEG, TER, QUA, QUI, SEX, SAB, DOM};
enum TpSexos {MASCULINO, FEMININO};
```

Variáveis destes tipos são definidas da seguinte forma:

```
enum TpCores var1, var2;
enum TpDias  var3;
```

Agora, é possível dar valores a estas variáveis, por exemplo:

```
var1 = AMARELO;
var3 = QUI;
```

é um erro usar valores não definidos na declaração do tipo. A expressão `var2 = AZUL;` causa erro de compilação.

Internamente, o compilador trata variáveis enumeradas como inteiros. Cada valor na lista de valores possíveis corresponde a um inteiro, começando com 0 (zero). Portanto, no exemplo `enum TpCores`, `VERMELHO` é armazenado como `0`, `AMARELO` é armazenado como `1`, e `VERDE` é armazenado como `2`.

**Utilização de tipos enumerados**

Variáveis de tipos enumerados são geralmente usados para clarificar a operação do programa. Considere o seguinte trecho de programa que codifica dias da semana como inteiros (sendo `sabado = 5` e `domingo = 6`) para verificar se o dia do pagamento cai no final de semana e altera a dia para a segunda-feira seguinte.

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

// prototipo da funcao que dada a data, retorna o dia da semana.
//   seg=0, ter=1, qua=2, qui=3, sex=4, sab=5, dom=6
```

```
int diaDaSemana( int dia, int mes, int ano );

int main(){
   int diaPgto, mesPgto, anoPgto;
   int diaSem;

   cout << "Entre com a data de pagamento (dd mm aa): ";
   cin >> diaPgto >> mesPgto >> anoPgto;
   diaSem = diaDaSemana( diaPgto, mesPgto, anoPgto );
   if( diaSem == 5 )
     diaPgto = diaPgto + 2;
   else if( diaSem == 6 )
     diaPgto++;
   cout << "Data do pagamento: " << diaPgto << "/" << mesPgto << "/"
        << anoPgto << endl;
}
```

Este programa ficaria mais legível se ao invés de codificar os dias da semana como inteiros e colocar a codificação como comentário, utilizar tipos enumerados. O programa ficaria então

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

enum TpSemana {SEG, TER, QUA, QUI, SEX, SAB, DOM};

// prototipo da funcao que dada a data, retorna o dia da semana
enum TpSemana  diaDaSemana( int dia, int mes, int ano );

int main(){
   int diaPgto, mesPgto, anoPgto;
   int diaSem;

   cout << "Entre com a data de pagamento (dd mm aa): ";
   cin >> diaPgto >> mesPgto >> anoPgto;
   diaSem = diaDaSemana( diaPgto, mesPgto, anoPgto );
   if( diaSem == SAB )
     diaPgto = diaPgto + 2;
   else if( diaSem == DOM )
     diaPgto++;
   cout << "Data do pagamento: " << diaPgto << "/" << mesPgto << "/"
        << anoPgto << endl;
}
```

Note que a função `diaDaSemana` agora retorna apenas um dos valores da lista `SEG, TER, QUA, QUI, SEX, SAB, DOM` e portanto, no programa principal ao invés de testar se o `diaSem == 5` podemos escrever `diaSem == SAB`, o que torna o programa muito mais legível.

# 23 Mais sobre funções

A ênfase aqui será em como funções funcionam. O que acontece quando uma função é chamada ? A que variável um nome está se referenciando?

O tratamento em tempo de execução de um nome de variável em C++ é simples: um nome de variável é ou uma variável local (a função) ou uma variável global (definida fora de qualquer função).

Em C++ , todas as funções tem que ser definidas. Para cada função deve ser definido um protótipo. O protótipo é escrito fora de qualquer função. Desta forma, nomes de funções são visíveis para todas as outras funções que podem então invocá-las. A função `main()` é especial: é onde a execução do programa começa, e o protótipo de `main()` pode ser omitido.

Uma definição de função consiste de quatro partes:

1. o nome da função;
2. a lista de parâmetros formais (argumentos) com seus nomes e tipos. Se não houver argumentos, a palavra `void` é escrita entre os parênteses.
3. o tipo do resultado que a função retorna através da sentença `return` ou `void` se a função não retorna nenhum valor. Lembre-se que somente um valor pode ser retornado por uma sentença `return`.
4. o corpo da função, que é uma sentença composta (começa e termina com chaves (`{ }`) contendo definição de variáveis e outras sentenças. Em C++ , não se pode definir uma função dentro de outra.

Para funções com argumentos: uma função é chamada dando o seu nome e uma lista de argumentos (expressões que são avaliadas e cujos valores são atribuídos para os correspondentes parâmetros formais da função).

Por exemplo, suponha que `triang_area()` e `circ_area()` sejam funções que calculam a área de triângulos e círculos, respectivamente. Seus protótipos são:

```
float triang_area(float , float);
float circ_area(float);
```

Estas funções podem chamadas de dentro de outras funções. Os argumentos reais com os quais elas são chamadas podem ser expressões constantes, or variáveis locais, ou qualquer expressão cuja avaliação resulte em valores do tipo `float` (inteiros são convertidos para `float` da mesma forma que ocorre com atribuição de inteiros para variáveis do tipo `float`). Alguns exemplos de chamadas:

```
float  area2, area3, area4, area5, base, altura, raio;

cout << "area do triangulo = " << triang_area(0.03, 1.25);
base = 0.03;
altura = 1.25;
area2 = triang_area(base, altura);
area3 = triang_area(1.6, altura);
area4 = triang_area( 0.03 + base, 2 * altura);
raio = base + altura;
area5 = triang_area(raio, circ_area(raio));
```

A última sentença do exemplo acima atribui a variável `area5` a área de um triângulo cuja base é igual ao valor da variável `raio` e a altura é igual a area de um círculo de raio igual ao valor da variável `raio`.

Quando um programa é executado, somente uma única função tem o controle em determinado momento. Falaremos mais sobre o que acontece quando uma função é chamada mais tarde nestas notas de aula.

**Variáveis Locais** **Variáveis que são definidas dentro de uma função são variáveis locais desta função. Parâmetros formais de uma função são variáveis locais da função.** Variáveis locais são privativas a função na qual são definidas. Somente esta função pode enxergá-las (ela conhece o endereço das variáveis e pode usar e modificar o seu conteúdo). Nenhuma outra função pode acessar variáveis locais de outra função sem permissão (uma função pode acessar variáveis locais de outra se esta passar o endereço da variável local como argumento – este assunto será tratado em notas de aula futuras). O fato de cada função manter variáveis locais "escondidas" do resto do mundo torna mais fácil a tarefa de escrever programas estruturados e modulares. Quando você está escrevendo uma função, você pode dar as suas variáveis locais o nome que quiser. Você também não precisa se preocupar se outra pessoa escrevendo outra função terá acesso ou altera variáveis locais a sua função.

**Variáveis locais que são definidas dentro da função devem ser inicializadas com algum valor antes de serem usadas.** Caso contrário, o seu valor é indefinido.

Já que parâmetros formais (argumentos) são variáveis locais da função, eles podem ser usados no corpo da função. Eles não devem ser definidos dentro da função (sua definição já está no cabeçalho da função). Os parâmetros formais não precisam ser inicializados. Seus valores são fornecidos pelo chamador da função através dos argumentos reais.

Considere o seguinte exemplo:

```
/*****************************************************************
 * Um programa que calcula a area de triangulos e circulos.
 * A base, altura e raio sao fornecidos pelo usuario.
 * A saida do programa e a area do triangulo e circulo.
 *****************************************************************/

#include <iostream>
using namespace std;

#define PI 3.1415

/*******************
    prototipos
*******************/
float triang_area(float, float);
float circ_area(float);

/*******************
definicao de funcoes
*******************/

main()
{
  /* definicao das variaveis locais */
  float   base, altura, raio;

  /* dialogo de entrada */
  cout << "\nEntre com a base e altura do triangulo: ";
  cin >> base >> altura;
  cout << "\nEntre com o raio do circulo: ";
  cin >> raio;

  /* chama as funcoes e imprime o resultado */
```

```
  cout << "Area do triagulo com base e altura " << base << " e "
       << altura << " = " << triang_area(base, altura) << endl;;
  cout << "Area do circulo com raio " << raio
       << " = " << circ_area(raio) << endl;
}

/*************************************************************
 * funcao: triang_area
 * calcula a area de um triangulo dada a base e altura
 * Entrada: base e altura do triangulo
 * Saida: area do triangulo
 *************************************************************/
float triang_area(float base, float alt)
{
  return 0.5*base*alt;
}

/*************************************************************
 * funcao: circ_area
 * calcula a area de um circulo dado o raio
 * Entrada: raio do circulo
 * Saida: area do circulo
 *************************************************************/
float circ_area(float r)
{
  return PI*r*r;
}
```

Este programa C++ consiste de três funções, `main()`, `triang_area()`, e `circ_area()`. `main()` tem variáveis locais chamadas `base, altura` e `raio`; `triang_area()` tem como variáveis locai seus parâmetros formais, `base` e `alt`; `circ_area()` tem como variável local seu parâmetro formal `r`.

Em geral, uma variável local só existe durante a execução da função na qual ela está definida. Portanto, variáveis locais existem desde o momento que a função é chamada até o momento em que a função é completada. Tais variáveis são chamadas de `automatic`. Em C++ , uma variável pode ser definida como sendo `static`. Neste caso, uma variável local não é visível de fora do corpo da função, mas ela não é destruída no final da função como variáveis automáticas são. Cada vez que a função é chamada, o valor das variáveis *static* é o valor final da variável da chamada anterior.

**Variáveis Globais**

Até este momento, todas as variáveis que vimos são definidas dentro de funções (no corpo da função ou como parâmetros formais). é possível também definir variáveis *fora* das funções. Tais variáveis são chamadas de *variáveis globais* ou *externas*. O formato da definição de variáveis globais é o mesmo da definição de variáveis locais. A única diferença é *onde* a variável é definida: variáveis globais são definidas fora de qualquer função. Ao contrário das variáveis locais, variáveis globais podem ser vistas por todas as funções definidas após a definição das variáveis globais.

Nós temos usado declarações "globais" este tempo todo – por exemplo, as declarações de protótipos de funções. Elas são declaradas fora de qualquer função e podem ser vistas por qualquer função que estão após sua declaração.

No exemplo seguinte, uma variável `saldo` que é atualizada por três funções diferentes é definida como uma variável global. As três funções que a atualizam não chamam uma a outra.

```
/*****************************************************************
 * Caixa eletronico simples
 * o saldo e o valor a ser alterado e entrado pelo usuario
 * a saida do programa e' o saldo atualizado, incluindo juros
 *****************************************************************/

#include <iostream>
using namespace std;

#define JUROS 0.07

/*******************
    prototipos
*******************/
void credito(float);
void debito(float);
void juros(void);

/*******************
      globais
 *******************/
float saldo;  /* saldo atual;
               * Alterada em: credito(), debito(), juros(), main()
               * Lida em:
               */


/************************
  definicao de funcoes
 ************************/

main()
{
  float valor;     // valor a ser  depositado/retirado

  cout << "Entre com o saldo atual: ";
  cin >> saldo;
  cout << "Deposito: ";
  cin >> valor;
  credito(valor);
  cout << "Retirada: ";
  cin >> valor;
  debito(valor);
  juros();
  cout << "Juros " << JUROS * 100 << "%.\n";
  cout << "Saldo = " << saldo << endl;
}

/*****************************************************************
 * Deposita um valor; atualiza a variavel global saldo
```

```
 * Entrada: valor a ser depositado
 * Saida: nenhum
 ****************************************************************/
void credito(float val)
{
   saldo += val;
}

/****************************************************************
 * Debita um valor; atualiza a variavel global saldo
 * Entrada: valor a ser debitado
 * Saida: nenhum
 ****************************************************************/
void debito(float val)
{
   saldo -= val;
}

/****************************************************************
 * Acumula juros; atualiza a variavel global saldo; juros: RATE
 * Entrada: nenhuma
 * Saida: nenhuma
 ****************************************************************/
void juros(void)
{
  saldo += (saldo * JUROS);
}
```

Um exemplo de execução do programa:

```
Entre com o saldo atual: 1000
Deposito: 200
Retirada: 80
Juros 7%.
Saldo =  1198.40
```

Variáveis globais devem ser usadas **SOMENTE** quando muitas funções usam muito as mesmas variáveis. No entanto, o uso de variáveis globais é perigoso (e não recomendado) porque a modularidade do programa pode ser afetada. Uma variável global pode ser alterada de dentro de uma função, e esta alteração pode influir no resultado de uma outra função, tornando-a incorreta (em um exemplo dado posteriormente nestas notas, duas chamadas a função `soma_y()` com o mesmo argumento (zero) produz resultados diferentes, `100` e `300`).

Quando variáveis globais são utilizadas, deve ser dado a elas nomes descritivos e um breve comentário qual a finalidade da variável e quais funções a acessam.

Neste curso, você utilizará variáveis globais SOMENTE QUANDO FOR DADO PERMISSãO PARA FAZÊ-LO. Caso contrário, não é permitido utilizá-las (ou seja, serão descontados pontos).

**Escopo de Variáveis**

Como já discutimos anteriormente, uma variável é uma abstração de dados que nós usamos em um programa. A variável representa um endereço de memória onde os valores são armazenados. Durante a execução do programa, valores diferentes poder ser armazenados neste endereço. Quando uma variável é definida, o nome da variável é "atrelada" a um endereço específico na memória. Até este momento, já discutimos o que é o nome de uma variável, seu endereço, tipo e valor. Outra característica que apresentaresmo agora é o `escopo`. O escopo de uma variável refere-se a parte do programa onde podemos utilizar a variável. Em outras, palavras, uma variável é "visível" dentro do seu escopo.

O escopo de uma variável local é a função na qual ela é definida. Os parâmetros formais de uma função também são tratados como variáveis locais.

O escopo de uma variável global é a porção do programa depois da definição da variável global (a partir do ponto onde ela é definida até o final do programa).

Se o nome de uma variável global é idêntico a uma variável local de uma função, então dentro desta função em particular, o nome refere-se a variável local. (Embora tais conflitos devem ser evitados para evitar confusão).

Por exemplo, considere o seguinte programa:

```
int valor = 3;     /* definicao da variavel global */

int main()
{
   /* definicao local de  valor */
   int valor = 4;
   cout << valor << endl;
}
```

A saída do programa acima será `4` já que `valor` refere-se a definição local.

Considere outro exemplo:

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

int soma_y(int);
int soma_yy(int);

int y = 100;       // variavel global

main()
{
  int z = 0;       // variavel local

  cout << soma_y(z) << endl;
  cout << soma_yy(z) << endl;
  cout << soma_y(z) << endl;
}

int soma_y(int x)
{
  return x + y;    // x e' variavel local, y e' global
}

int soma_yy(int x)
{
```

```
  y = 300;          // y e' variavel global
  return x + y;     // x e' variavel local
}
```

Vamos seguir a execução deste programa. Primeiro, a variável global `y` é criada e inicializada com 100. Então, a execução da função `main()` começa: é alocado espaço na memória para a variável local `z`. Esta variável é inicializada com `0`. Considere a primeira sentença `cout` :

```
cout << soma_y(z) << endl;
```

Esta é uma chamada para `cout` . Os parâmetros reais desta chamada é a expressão `soma_y(z)`. Ela consiste da chamada da função `soma_y()`. O valor desta expressão é o resultado retornado por `soma_y()`. Qual o resultado? A função `soma_y` é chamada com o parâmetro real `z`. Como `z = 0`, este é o valor que será passado para a função `soma_y`; o `0` é copiado para o parâmetro formal `x` da função `soma_y()`. Portanto, durante a excução da primeira chamada a função `soma_y()`, o valor da expressão `x + y` será `0 + 100`, que é 100. Portanto, o valor da primeira chamada `soma_ y(z)` é `100`, e este número será impresso com o primeiro `cout` em `main()`. Agora considere a segunda sentença:

```
cout << soma_yy(z) << endl;
```

Quando a função `soma_yy(z)` é chamada, o valor de `z` (a variável local `z`) ainda é `0`, portanto novamente `0` é copiado para o parâmetro formal `int x` da função `soma_yy`. Quando a execução de `soma_yy()` começa, ela primeiro troca o valor da variável global `y` para `300` e então retorna o valor de `x + y`, que neste caso é `0 + 300`. Portanto, o valor desta chamada a `soma_yy(z)` é `300`, e este número será impresso pelo segundo `cout` em `main()`.
Por último, considere a terceira sentença:

```
cout << soma_y(z) << endl;
```

Quando a função `soma_y(z)` é chamada, o valor de `z` ainda é `0`, portanto, `0` é copiada para o parâmetro formal `int x` da função `soma_y()`. Quando `soma_ y()` é executada pela segunda vez, a variável global `y` foi modificada para `300`, portanto o valor de `x + y` é `0 + 300`. Portanto, o valor da chamada `soma_yy(z)` é `300`, e este número será impresso pelo terceiro `cout` em `main()`.

Portanto, a saída da execução deste programa será

```
100
300
300
```

Neste exemplo, o escopo da variável global `y` é o programa todo.

O escopo da variável local `z`, definida dentro de `maio` é o corpo da função `main`. O escopo do parâmetro formal `x` da função `soma_y` é o corpo de `soma_y`. O escopo do parâmetro formal `x` da função `soma_yy` é o corpo de `soma_yy`.

## 23.1 Outro exemplo

Aqui apresentamos um exemplo de uma função mais complicada. Esta função calcula a "raiz quadrada inteira" de um número (o maior inteiro menor ou igual a raiz quadrada do número).

Este programa usa o algoritmo "divide e calcula média" (uma aplicação do método de Newton). Ele executa o seguinte:

Dado $x$, achar $\sqrt{x}$ computando sucessivamente

$$a_n = \begin{cases} 1 & \text{se } n = 0 \\ \frac{\frac{x}{a_{n-1}} + a_{n-1}}{2} & \text{caso contrário} \end{cases} \quad \text{para todo } n \,\epsilon\, \mathcal{N}$$

Os valores de $a_n$ convergem para $\sqrt{x}$ a medida que $n$ cresce.

Para achar a raiz quadrada inteira, este algoritmo é repetido até que

$$a_n^2 \leq x < (a_n + 1)^2$$

Por exemplo, para achar a raiz quadrada inteira de $42$ (usando divisão inteira que trunca a parte fracional do número)
$a_0 = 1$, $a_1 = (42/1 + 1)/2 = 21$, $a_2 = (42/21 + 21)/2 = 11$, $a_3 = (42/11 + 11)/2 = 7$, $a_4 = (42/7 + 7)/2 = 6$.
Uma vez que $a_4^2 = 6^2 = 36 \leq 42 < (a_4 + 1)^2 = 7^2 = 49$, o processo termina e a resposta é $6$.

(Não é necessário você entender por que este algoritmo funciona – portanto não se preocupe se não conseguir entendê-lo)

```
int raizInteira(int);          /* prototipo */

/*************************************************************
 * function: raizInteira(x)
 *  acao:        dado x, retorna a raiz quadrada inteira de x
 *  in:          inteiro positivo x
 *  out:         raiz quadrada inteira de x
 *  suposicoes:  x >= 0
 *  algoritmo:   metodo de dividr e calcular media:  comecando com
 *               um palpite de 1, o proximo palpite e' calculado como
 *               (x/palpite_ant + palpite_ant)/2.  Isso e' repetido
 *               ate' que palpite^2 <= x < (palpite+1)^2
 *************************************************************/
int raizInteira(int x)
{
    int palpite = 1;

    /* Continue ate' que o  palpite esteja correto */
    while (!(x >= palpite*palpite  &&  x < (palpite+1)*(palpite+1))) {
        /* Calcula proximo palpite */
        palpite = (x/palpite + palpite) / 2;
    }
    return palpite;
}
```

Note que usando a lei de DeMorgan, podemos re-escrever a expressão teste do `while` em uma forma equivalente:

```
x < palpite * palpite || x >= (palpite + 1) * (palpite + 1)
```

Deve estar claro neste ponto a diferenca entre ação e algoritmo. Uma pessoa que quer usar esta função precisa saber somente a ação, não o algoritmo. É também importante especificar os dados que são esperados pela função e retornados por ela para outras pessoas poderem usá-la. As suposições devem esclarecer as restrições da função sobre quando a função pode falhar ou produzir resultados errados. Neste caso, um número negativo produziria um erro, já que números negativos não possuem raiz quadrada.

Não há necessidade de ir em muitos detalhes em qualquer parte da documentação da função. Embora ela deva conter informação suficiente para que alguém (que não possa ver o código) saber utilizá-la. Detalhes sobre implementação e detalhes menores sobre o algoritmo devem ser colocados como comentários no próprio código.

## 24 Ativação de função

Uma função começa sua execução assim que for chamada. Cada execução da função é chamada de ativação da função. Como já mencionamos em notas de aula anteriores, variáveis locais são locais a ativação da função: cada ativação possui suas próprias variáveis locais. No começo da ativação, memória é alocada para as variáveis locais e no final da execução, elas são dealocadas.

Definições de funções em C++ não podem ser aninhadas, mas ativações de função podem: uma função, digamos A, pode chamar uma outra função, digamos B (dizemos que A chama B). Nos referimos a A como o "chamador" e B como a função "chamada".

O que acontece quando uma função chama outra (quando A chama B)? Um registro especial, chamado *registro de ativação* é criado. A informação neste registro é necessária para a ativação da função chamada e para a reativação do chamador depois que a execução da função chamada termina.

1. C++ usa chamada-por-valor, ou seja, o chamador avalia as expressões que são os parâmetros reais e passa seus valores para a função chamada.

2. A informação necessária para reiniciar a execução da função chamadora é guardada em um registro de ativação. Tal informação inclui o endereço da instrução do chamador que será executada depois que a função chamada termine.

3. A função chamada aloca espaço na memória para suas variáveis locais.

4. O corpo da função chamada é executado.

5. O valor retornado para a função chamadora através de um `return` é colocado em um lugar especial para que a função chamadora possa encontrá-lo. O controle retorna a função chamadora.

O fluxo de controle através de ativação de funções é da forma *último-que-entra-primeiro-que-sai*. Se A chama B e B chama C: A é ativado primeiro, então B é ativado (um registro de ativação para "A chama B" é criado e armazenado, A é temporariamente suspenso), então C é ativado (um registro de ativação de "B chama C" é criado e armazenado, A e B são suspensos); C é o último a iniciar execução, mas o primeiro a terminar (último-a-entrar-primeiro-a-sair). Depois que C termina, B é reativado. O registro de ativação "B chama C" foi criado por último, mas o primeiro a ser destruído (no momento que o controle é retornada para B). Depois que B termina, A é reativado. O registro de ativação correspondente a "A chama B" é destruído no momento em que o controle retorna para A.

## 25 Array de Caracteres

Nas notas de aula anteriores, enfatizamos arrays de números. Em geral, podemos ter arrays com elementos de qualquer um dos tipos vistos até agora (incluindo arrays – visto nas notas de aula 9). Nesta seção, apresentaremos arrays com elementos do tipo `char`, usados para representar textos em programas.

Abaixo, apresentamos um exemplo de programa que define e inicializa um array de caracteres, e depois imprime o array em ordem reversa.

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

int main()
{
   char arr1[] = {'c','i','2','0','8'};
   int i;

   for (i = 4; i >= 0; i -= 1)
      cout << arr1[i];
}
```

Arrays de caracteres são usados para armazenar texto, mas é muito inconveniente se tivermos que colocar cada caractere entre apóstrofes. A alternativa dada pela linguagem C++ é

```
char arr2[] = "ci208" ;
```

Neste caso, "`ci208`" é um *string* de caracteres ou uma constante do tipo *string*. Nós já usamos *strings* antes, com as funções `cout` (constantes do tipo *string* estão sempre entre aspas - "):

```
cout << "Entre com a nota para o estudante 2: ";
cin >> gr2;
```

## 26 Strings como array de caracteres

Strings são essencialmente arrays de caracteres (arrays com elementos do tipo `char`) que DEVEM terminar com `'\0'`. Normalmente é conveniente definir a constante NULO em seu programa para representar este terminador: `#define NULO '\0'`.

No exemplo acima, embora não tenhamos escrito explicitamente o caracter NULO, o compilador automaticamente o colocou como o último elemento do array `arr2[]`. Portanto, o tamanho de `arr2[]` é 6: 5 para os caracteres que indicamos (`ci208`) e 1 para o caractere NULO que o compilador introduziu automaticamente. As definições abaixo são equivalentes.

```
   char arr2[] =  {'c','i',' ', '2','0','8','\0'};

   char arr2[] =  {'c','i',' ', '2','0','8', NULO};
```

O caractere NULO marca o final de um string.

Outros exemplos:

```
// a maneira tediosa
char name1[] = { 'j','o','s','e',' ','s','i','l','v','a','\0' };

// e a maneira facil
char name2[] = "jose silva";
```

Embora o primeiro exemplo seja um string, o segundo exemplo mostra como strings são geralmente escritos (como constantes). Note que se você usar aspas quando escreve uma constante, você não precisa colocar `'\0'`, porque o compilador faz isso para você.

Quando você for criar um array de caracteres de um tamanho específico, lembre-se de adicionar 1 para o tamanho máximo de caracteres esperado para armazenar o caractere NULO. Por exemplo, para armazenar o string "`programar eh divertido`", você precisa de um array de tamanho 22 (21 caracteres + 1 para o NULO).

## 26.1 Imprimindo strings com `cout`

Strings podem ser impressos usando `cout` . Por exemplo:

```
int main()
{
    char mensagem[] = "tchau";

    cout << "ola" << endl << mensagem << endl;
}
```

A saída deste programa é:

```
ola
tchau
```

## 26.2 Lendo strings do teclado com `cin.getline()` e `cin`

A função `cin.getline()` lê uma linha de texto digitado no teclado e a armazena em um vetor indicado como primeiro argumento. O segundo argumento da função indica o número máximo de caracteres que será lido. Veja o exemplo abaixo:

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

int main()
{
  char nome[100];

  cout << "Entre seu nome: ";
  cin.getline (nome, 100);
  cout << "Oi, " << nome << "." << endl ;
}
```

Exemplo de execução

```
Entre seu nome: Jose Silva
Oi, Jose Silva.
```

Passando um nome de array para a função `cin.getline()`, como ilustrado no programa acima, coloca a linha inteira digitada pelo usuário no array `nome` (tudo ou máximo de 99 caracteres até que seja teclado enter). Note que se o usuário digitar caracteres demais (neste caso, mais de 99 caracteres), apenas os primeiros 99 caracteres digitados serão copiados para o array indicado no primeiro argumento da função.

A função `cin` pode ser usada de maneira similar. A única diferença é que `cin` lê somente a primeira palavra (tudo até que se digite um separador – um espaço em branco, tabulação, ou enter).

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

int main()
{
  char nome[100];

  cout << "Entre seu nome: ";
  cin >> nome;
  cout << "Oi, " << nome << "." << endl;
}
```

Exemplo de execução

```
Entre seu nome: Jose Silva
Oi, Jose.
```

Note que somente o primeiro nome é lido pelo `cin` porque a função pára no primeiro espaço em branco que encontra (enquanto `cin.getline()` pára quando encontra um enter).

## 26.3 Array de *Strings*

Em notas de aula anteriores, vimos alguns exemplos de arrays de arrays (matrizes ou tabelas). Como *strings* são também arrays, podemos definir arrays de *strings*. O programa abaixo inicializa um array de *strings* com nomes e os imprime.

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

#define NUM_NOMES 5        // define a quantidade de nomes no array
#define TAM       20       // define o tamanho maximo do nome

int main()
{
  char nomes[NUM_NOMES][TAM] = {"Jose Silva",
                                "Maria Silva",
                                "Antonio dos Santos",
                                "Pedro dos Santos",
                                "Joao da Silva"};
  int i;

  for(i = 0; i < 5; i++)
    cout << nomes[i] << endl;
}
```

A saída deste programa é:

```
Jose Silva
Maria Silva
Antonio dos Santos
Pedro dos Santos
Joao da Silva
```

## 26.4 Funções de *String*

Há funções para manipulação de string já definidas na biblioteca padrão C++ chamada `cstring`. Todas as funções que apresentaremos nesta seção são parte desta biblioteca. Portanto, se seu programa utilizar uma destas funções você deve incluir a linha `#include <cstring>` no início do seu programa.

O objetivo desta seção é mostrar como estas funções poderiam ser implementadas como exemplos de programas de manipulação de strings.

### 26.4.1 A função `strlen()`

A função `strlen()` tem como argumento um *string*. Ela retorna um inteiro que é o comprimento do string (o número de caracteres do string, não contando o caractere NULL). Por exemplo, o comprimento do string `"alo"` é `3`.

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

int main()
{
  char nome[100];
  int comprimento;

  cout << "Entre seu nome: ";
  cin.getline(nome, 100);
  comprimento = strlen(nome);

  cout << "Seu nome tem  " << comprimento << " caracteres." << endl;
}
```

Um exemplo de execução:

```
Entre seu nome: Dostoevsky
Seu nome tem 10 caracteres.
```

Abaixo, mostramos como a função `strlen()` poderia ser implementada.

```
int strlen( char str[] )
{
   int comprimento = 0;


   while ( str[comprimento] != NULL )
       ++comprimento;
   return comprimento;
 }
```

### 26.4.2 A função `strcmp()`

A função `strcmp()` é usada para comparar dois strings. Lembre que não podemos usar ==, como em `str1 == str2`, para comparar dois strings, uma vez que strings são arrays. Strings devem ser comparados caractere por caractere. A função `strcmp()` tem como argumento dois strings e retorna um inteiro.

Strings são ordenados de forma similar a maneira como palavras são ordenadas em um dicionário. Ordenamos palavras em um dicionário alfabeticamente, e ordenamos strings respeitando a ordem dos caracteres no conjunto de caracteres da máquina. A ordenação abaixo é válida em qualquer computador:

```
'0' < '1' < ... < '8' < '9'
'A' < 'B' < ... < 'Y' < 'Z'
'a' < 'b' < ... < 'y' < 'z'
```

A ordem relativa do três conjuntos (dígitos, letras maiúsculas e letras minúsculas) depende do computador utilizado.

Se `s1` e `s2` são strings, o resultado da chamada de função `strcmp(s1, s2)` é:

- se `s1` $=_s$ `s2`, `strcmp()` retorna `0`
- se `s1` $<_s$ `s2`, `strcmp()` retorna um número negativo (`< 0`)
- se `s1` $>_s$ `s2`, `strcmp()` retorna um inteiro positivo (`> 0`)

(onde $=_s$, $<_s$ e $>_s$ são $=$, $<$ e $>$ para strings)

$s1 <_s s2$ significa "$s1$ vem antes de $s2$ no dicionário". Exemplos: "`tudo`" é menor que "`xadrez`", "`calor`" é menor que "`calorao`", "`frio`" é menor que "`quente`", e é claro o string vazio , `NULL`, é menor que qualquer string.

Considere o exemplo abaixo que usa `strcmp()`:

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

int main()
{
  char palavra1[100], palavra2[100];
  int resultado;

  cout << "entre com uma palavra: ";
  cin >> palavra1;
  cout << "entre outra palavra: ";
  cin >> palavra2;

  resultado = strcmp(palavra1, palavra2);

  if (resultado == 0)
    cout << "igual" << endl;
  else if (resultado > 0)
    cout << "o primeiro e' maior" << endl;
  else
    cout << "o segundo e' maior" << endl;
}
```

Aqui está um exemplo de como a função `strcmp()` poderia ser implementada.

```
int strcmp( char s1[], char s2[] )
{
   int i = 0;

   while (1)
   {
```

```
      if (s1[i] == NULL && s2[i] == NULL)
         return 0;
      else if (s1[i] == NULL)
         return -1;
      else if (s2[i] == NULL)
         return 1;
      else if (s1[i] < s2[i])
         return -1;
      else if (s1[i] > s2[i])
         return 1;
      else
         ++i;
   }
}
```

Na biblioteca padrão, a função `strcmp()` faz distinção entre letras maiúsculas e minúsculas. Se você não quer que a função faça esta distinção, você pode modificar o seu string para ter apenas letras minúsculas (ou maiúsculas) antes de passá-lo como argumento para a função `strcmp()`. Para fazer isso, você pode usar a função da biblioteca padrão `tolower()`, que tem como argumento um caractere. Se o caractere passado é uma letra maiúscula, ele retorna esta letra minúscula; caso contrário, retorna o mesmo caractere. Por exemplo: `tolower('A')` é `'a'`, `tolower('1')` é `'1'`, `tolower('a')` é `'a'`.

### 26.4.3 A função `strcpy()`

A função `strcpy()` é usada para copiar o conteúdo de um string para outro. Ela tem dois argumentos: `strcpy(s1, s2)` copia o conteúdo do string `s2` para o string `s1`. A função `strcpy()` que apresentamos abaixo não retorna um valor. Seu protótipo é

```
void strcmp(char [], char []);
```

O exemplo abaixo mostra a utilização do `strcpy()`.

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

int main()
{
  char pal[100], palCopia[100];

  cout << "entre com uma palavra: ";
  cin >> pal;
  strcpy(palCopia, pal);

  cout << "entre outra palavra: ";
  cin >> pal;

  cout << "voce entrou primeiro: " << palCopia << endl;
}
```

Embora este programa pudesse ter sido escrito sem usar `strcpy()`, o objetivo é mostrar que se pode usar `strcpy()` para fazer "atribuição" de strings.

A função `strcpy()` poderia ter sido implementada da seguinte forma:

```
void strcpy( char s1[], char s2[] )
{
   int i = 0;

   while ( s2[i] != NULL )
   {
       s1[i] = s2[i];
       ++i;
   }
   s1[i] = s2[i];
}
```

## 27 O tipo `String`

Em C++ é possível uma forma alternativa de lidar com textos, usando a classe **String**.

# 28 Estruturas

A estrutura de dados ***array*** é usada para conter dados do mesmo tipo junto. Dados de tipos *diferentes* também podem ser agregados em tipos chamados de **estruturas** ou **registros** (tipo `struct` em linguagem C). Primeiro, o tipo estrutura é declarado (precisamos especificar que tipos de variáveis serão combinados na estrutura), e então variáveis deste novo tipo podem ser definidas (de maneira similar que usamos para definir variáveis do tipo `int` ou `char`).

## 28.1 Declaração de Estruturas

Uma declaração de estrutura *declara* um tipo *struct*. Cada tipo *struct* recebe um *nome* (ou *tag*). Refere-se àquele tipo pelo *nome* precedido pela palavra `struct`. Cada unidade de dados na estrutura é chamada *membro* e possui um *nome de membro*. Os membros de uma estrutura podem ser de *qualquer* tipo. Declarações de estrutura não são definições. Não é alocada memória, simplesmente é introduzida um novo tipo de estrutura.

Geralmente declarações de estruturas são globais. Elas são colocadas próximas ao topo do arquivo com o código fonte do programa, assim elas são visíveis por todas as funções (embora isto dependa de como a estrutura está sendo usada).

A forma padrão de declaração de uma estrutura é:

`struct` *nome-estrutura* `{`
    *declaração dos membros*
`}` *definição de variáveis (optional)*;

Abaixo se apresenta um exemplo de um tipo estrutura que contém um membro do tipo `int` e um outro membro do tipo `char`.

```
struct facil {
    int num;
    char ch;
};
```

Esta declaracao cria um novo tipo chamado `struct facil` que contém um inteiro chamado `num` e um caracter chamado `ch`.

## 28.2 Definição de variáveis de um tipo estrutura declarado

Como acontece com qualquer outro tipo de dados, variáveis de tipos de estruturas são definidas fornecendo o nome do tipo e o nome da variável. Considere a definição abaixo relativa a uma variável com o nome `fac1` que é do tipo `struct facil`:

```
struct facil fac1;
```

Tal definição está associada com a alocação de memória: memória suficiente será alocada para guardar um `int` e um `char` (nesta ordem). Como qualquer outra variável, `fac1` tem um nome, um tipo, e um endereço associados.

Variáveis de estruturas possuem também valores, e como outras variáveis locais, se elas não tem atribuídas um valor específico, seu valor é indefinido.

É possível definir variáveis durente a declaração do tipo estrutura:

```
struct facil {
    int num;
    char ch;
} fac1;
```

Note-se que sem conflito, nomes de membros (tais como `num` e `ch`) podem ser usados como nomes de outras variáveis independentes (fora do tipo estrutura definido) or como nomes de membros em outros tipos estrutura. No entanto, deve-se evitar situações que criem confusão.

## 28.3 Acesso a membros de uma estrutura: ponto ( . ), o operador membro de estrutura

Dada uma variável de estrutura, um membro específico é referenciado usando o nome da variável seguida de `.` (ponto) e pelo nome do membro da estrutura. Assim, as seguintes referências a membros de uma estrutura são válidas:

`fac1.num` se refere ao membro com nome `num` na estrutura `fac1`;
`fac1.ch` se refere ao membro com nome `ch` na estrutura `fac1`.

Membros de estrutura (como `fac1.num`) são variáveis, e podem ser usadas como valores (no lado direito de uma atribuição, em expressões como argumentos para funções), ou como *lvalues* (no lado esquerdo de atribuições, com operadores de incremento/decremento ou com o operador de endereço (&)). O exemplo abaixo mostra alguns exemplos do uso de membros de estrutura:

```
fac1.ch = 'G';
fac1.num = 42;

fac1.num++;

if (fac1.ch == 'H') {
   cout << fac1.num << endl;
}
```

Tentar acessar um nome de membro que não existe causa um erro de compilação.

## 28.4 Operadores usados com variáveis de estrutura: valores e *lvalues*

Uma variável de estrutura pode ser tratada como um objeto simples no todo, com um valor específico associado a ela (a estrutura `fac1` tem um valor que agrega valores de todos os seus membros). Note a diferença com *arrays*: se `arr[]` é um array de tamanho 2 definedo como `int arr[2] = {0,1};`, o nome `arr2` não se refere ao valor coletivo de todos os elementos do *array*. Na verdade, `arr2` é um apontador constante e se refere ao endereço de memória onde o *array* se inicia. Além disso `arr2` não é um *lvalue* e não pode ser mudado. Variáveis de estrutura são diferentes. Elas podem ser usadas como valores e *lvalues*, mas com certas limitações.

**Os únicos usos válidos de uma variável de estrutura são dos dois lados de um operador de atribuição (=), como operando do operador de endereço `&` (obtendo o endereço da estrutura), e referenciando seus membros**.

De todas as variações de atribuição (incluindo o incremento e decremento) atribuição de estruturas pode ser usada APENAS com =. O uso de outros operadores de atribuição ou de incremento causará um erro de compilação A atribuição de um valor de estrutura para outro copia *todos* os membros de uma estrutura para outra. Mesmo que um dos membros seja um *array* ou outra estrutura, ela é copiada integralmente. As duas estruturas envolvidas na atribuição devem ser do mesto tipo `struct`. Considere o seguinte exemplo:

```
struct facil {
    int num;
    char ch;
};

void main(void)
{
```

```
    struct facil fac1, fac2;

    fac1.num = 3;
    fac1.ch = 'C';

    /* Atribuindo fac1 a fac2 */
    fac2 = fac1;
}
```

Lembre-se que este tipo de atribuição é ilegal com *arrays*. Tentar fazer isto com dois *arrays* causa um erro de compilação (uma vez que nomes de *arrays* são apontadores constantes).

```
int a[5], b[5];

/* Está errado -- Não irá compilar */
a = b;
```

## 28.5 Inicialização de estruturas

Variáveis de estruturas não-inicializadas contém valores indefinidos em cada um de seus membros. Como em outras variáveis, variáveis de estruturas podem ser inicializadas ao serem declaradas. Esta inicialização é análoga ao que é feito no caso de *arrays*. O exemplo abaixo ilustra a inicialização de estruturas:

```
struct facil {
    int num;
    char ch;
};

void main(void)
{
    struct facil fac1 = { 3, 'C' }, fac2;

    fac2 = fac1;
}
```

Uma lista de valores separados por vírgula fica entre chaves (`{` e `}`). Os valores de inicialização devem estar na mesma ordem dos membros na declaração da estrutura.

## 28.6 Estruturas como argumentos de função e valores de retorno

Como qualquer outro valor do tipo `int` ou `float`, valores de estruturas podem ser passados como argumentos para funções, e podem ser retornados de funções. O exemplo abaixo ilustra tal prorpiedade:

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

#define LEN 50

struct endereco {
  char rua[LEN];
  char cidade_estado_cep[LEN];
};
```

```
struct endereco obtem_endereco(void);
void imprime_endereco(struct endereco);

struct endereco obtem_endereco(void)
{
  struct endereco ender;

  cout << "\t Entre rua: ";
  cin.getline(ender.rua, LEN);
  cout << "\t Entre cidade/estado/cep: ";
  cin.getline(ender.cidade_estado_cep, LEN);

  return ender;
}

void imprime_endereco(struct endereco ender)
{
  cout << "\t " << ender.rua << endl;
  cout << "\t " << ender.cidade_estado_cep << endl;
}

main()
{
  struct endereco residencia;

  cout << "Entre seu endereco residencial:\n";
  residencia = obtem_endereco();

  cout << "\nSeu endereco eh:\n";
  imprime_endereco(residencia);
}
```

No exemplo acima, a estrutura `struct endereco` contém dois *arrays* de tamanho 50. Dentro da função `obtem_endereco()`, a variável `ender` é declarada como sendo do tipo `struct endereco`. Após usar `cin.getline()` para o fornecimento da informação, o valor de `ender` é retornado para `main()`, de onde a função `obtem_endereco()` foi chamada. Este valor é então passado para a função `imprime_endereco()`, onde o valor de cada membro da estrutura é exibido na tela.

Este programa pode ser comparado ao programa abaixo, que usa valores do tipo `int` no lugar de valores do tipo `struct endereco` (claro que a informação lida e exibida é um simples valor numérico, e não um nome de rua, etc.):

```
int obtem_int(void);
void imprime_int(int);

int obtem_int(void)
{
     int i;

     cout << "Entre valor: ";
     cin >> i;
```

```
     return i;
}

void imprime_int(int i)
{
     cout << i << endl;
}

void main(void)
{
     int valor;

     valor = obtem_int();

     cout << "\nSeu valor:\n";
     imprime_int(valor);
}
```

## 28.7 Arrays de estruturas

*Arrays* de estruturas são como *arrays* de qualquer outro tipo. Eles são referenciados e definidos da mesma forma. O exemplo abaixo é análogo ao exemplo de endereço apresentado anteriormente, exceto que uma quantidade de `NUM` endereços é armazenada ao invés de apenas um.

```
#define LEN 50
#define NUM 10

struct endereco {
    char rua[LEN];
    char cidade_estado_cep[LEN];
};

void obtem_endereco(struct endereco [], int);
void imprime_endereco(struct endereco);

void obtem_endereco(struct endereco aender [], int index)
{
     cout << "Entre rua: ";
     cin.getline(aender[index].rua, LEN);
     cout << "Entre cidade/estado/cep: ";
     cin.getline(aender[index].cidade_estado_cep, LEN);
}

void imprime_endereco(struct endereco ender)
{
     cout << ender.rua << endl;
     cout << ender.cidade_estado_cep << endl;;
}

void main(void)
```

```
{
      struct endereco residencias[NUM];
      int i;

      for (i = 0; i < NUM; i++) {
          cout << "Entre o endereco da pessoa " << i << ":\n";
          obtem_endereco(residencias,i);
      }

      for (i = 0; i < NUM; i++) {
          cout << "endereco da pessoa " << i << ":\n";
          imprime_endereco(residencias[i]);
      }
}
```

Neste programa, o array `residencias` é passado para `obtem_endereco()`, juntamente com o indice onde deve ser guardado o novo endereço. Depois, cada elemento do array é passado para `imprime_endereco()` um por vez.

Observe-se ainda na função `obtem_endereco()` como os membros de cada elemento do array podem ser acessados. elements da estrutura em can be accessed as well. Por exemplo, para acessar a rua do elemento `residencias[0]` usa-se:

```
cout << residencias[0].rua << endl;
cout << residencias[0].cidade_estado_cep << endl;
```

## 28.8 Estruturas aninhadas

Como definido anteriormente, membros de estruturas podem ser de qualquer tipo. Isto inclui outras estruturas. Abaixo define-se duas estruturas, a segunda tendo membros que são também estruturas:

```
#define LEN 50

struct endereco {
    char rua[LEN];
    char cidade_estado_cep[LEN];
};

struct student {
    char id[10];
    int idade;
    struct endereco casa;
    struct endereco escola;
};

struct student pessoa;
```

Dadas estas definições, pode-se potencialmente acessar os seguintes campos de `pessoa`, uma variável do tipo `struct student`:

```
pessoa.id
pessoa.casa.rua
pessoa.casa.cidade_estado_cep
```

```
pessoa.escola.rua
pessoa.escola.cidade_estado_cep
```

Note o uso repetido de `.` quando se acessa membros dentro de membros.

# 29 Entrada e Saída Padrão

A forma com que um programa em C++ se comunica com o mundo externo é através de entrada e saída de dados: o usuário fornece dados via teclado e o programa imprime mensagens na tela. Todos os programas vistos até agora lêem suas entradas do teclado e produzem suas saídas na tela.

Em C++ toda entrada e saída é feita com fluxos (*streams*) de caracteres organizados em linhas. Cada linha consiste de zero ou mais caracteres e termina com o caracter de final de linha. Pode haver até 254 caracteres em uma linha (incluindo o caracter de final de linha). Quando um programa inicia, o sistema operacional automaticamente define quem é a **entrada padrão** (geralmente o teclado) e quem é a **saída padrão** (geralmente a tela).

As facilidades de entrada e saída não fazem parte da linguagem C++ . O que existe é uma biblioteca padrão de classes e funções (métodos) para manipular a transferência de dados entre programa e os dispositivos (*devices*) de saída e entrada padrão: `cin, cout, cin.get(), cin.put(), puts(), gets()`. Estas classes e funções são declaradas no arquivo `<iostream>`. Existem funções úteis para conversão e teste de caracteres declaradas no arquivo `<ctype.h>`.

As funções de entrada e saída operam sobre *streams* (fluxos) de caracteres. Toda vez que uma função se entrada é chamada (por exemplo, `cin, cin.get()`) ela verifica pela próxima entrada disponível na entrada padrão (por exemplo, texto digitado no teclado). Cada vez que uma função de saída é chamada, ela entrega o dado para a saída padrão (por exemplo, a tela).

As funções para leitura da entrada padrão e para escrita na saída padrão que têm sido usadas até agora são:

1. **Entrada e saída de caracteres:**

```
int cin.get( void );
int cin.put( int );
```

2. **Entrada e saída de *strings*:**

```
char *gets(char *);
int puts( char *);
```

3. **Entrada e saída formatada:**

```
int cin >> arg1 >> arg2 >> ...;
int cout << arg1 << arg2 << ...;
```

## 29.1 Comandos de entrada e saída: cin.get() e cin.put()

Vamos discutir algumas funções de entrada de dados (diferente do cin). A entrada de texto é considerada como um *fluxo* de caratecteres. Um *fluxo texto* é uma sequência de caracteres dividida em linhas; cada linha consiste de zero ou mais caracteres seguido do caractere de nova linha ( \n). Como programador, você não quer se preocupar em como as linhas são representadas fora do programa. Quem faz isso por você são funções de uma biblioteca padrão.

Suponha que você queira ler um único caractere, sem fazer qualquer tipo de conversão para um tipo de dados específico (o que é feito por cin). Isso pode ser feito usando a função cin.get().

A função cin.get() lê o caracter do teclado e mostra o que foi digitado na tela.

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

```
int main()
{
   char ch;

   cout << "Digite algum caracter: ";

   ch = cin.get();

   cout << "\n A tecla pressionada eh " << ch << endl;

}
```

O Resultado deste programa na tela é:

```
Digite algum caracter: A
 A tecla pressionada eh A.
```

A função cin.put() aceita um argumento de entrada, cujo valor será impresso como caracter:

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

int main()
{
   char ch;

   cout << "Digite algum caracter: ";

   ch = cin.get();

   cin.put(ch);

}
```

## 29.2 Considerações sobre Leitura de Dados pelo Teclado

### 29.2.1 Lendo o teclado usando cin.get()

cin.get() é uma função vinculada à primitiva principal de entrada cin. Cada vez que é chamada, esta função lê um caractere teclado; cin.get começa a ler depois que a tecla **RETURN** é digitada no final de uma sequência de caracteres (dizemos que a entrada para a função cin.get() está no *fluxo de entrada*). A função cin.get() retorna um valor, o caractere lido (mais precisamente, o código inteiro ASCII correspondente ao caractere).

Vejamos o que acontece quando um programa trivial é executado.

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

int main(){

    char ch;
```

```
    ch = cin.get();
}
```

cin.get() obtém sua entrada do teclado. Portanto, quando o programa acima é executado, o programa espera que o usuário digite alguma coisa. Cada caractere digitado é mostrado no monitor. O usuário pode digitar diversos caracteres na mesma linha, inclusive *backspace* para corrigir caracteres já digitados. No momento que ele teclar **RETURN** , o primeiro caractere da sequência digitada é o resultado da função cin.get(). Portanto, na instrução do programa acima o caractere (ou melhor, o seu código ASCII) é atribuído a variável ch. Note que o usuário pode ter digitado diversos caracteres antes de teclar **RETURN** , mas a função cin.get() só começará a ler o que foi digitado depois que for teclado **RETURN** . Além disso, com uma chamada da função cin.get() só o primeiro caractere da sequência digitada é lida.

Você deve saber que o caractere de nova linha, `\n`, que tem o código ASCII 10, é automaticamente adicionado na sequência de caracteres de entrada quando o **RETURN** é teclado. Isso não tem importância quando a função cin.get() é chamada uma única vez, mas isto pode causar problemas quando ele é usado dentro de um laço.

No inicício de qualquer programa que usa cin.get(), você deve incluir

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

Esta diretiva do pré-processador diz ao compilador para incluir informações sobre cin, cin.get() e EOF (mais sobre EOF adiante.).

Considere o seguinte programa:

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

int main(){

   char ch;

      cout <<  "Entre com uma letra: ";
      ch = cin.get();

      if( ch < 'A' || ch > 'z' )
         cout <<  "Voce nao teclou uma letra!";
      else
         cout <<  "Voce teclou " << ch
              << ", e seu codigo ASCII eh " << (int) ch << endl;
}
```

Um exemplo da execução do programa:

```
Entre com uma letra: A
Voce teclou A, e seu codigo ASCII eh 65.
```

No exemplo de execução acima o usuário teclou A e depois **RETURN** .

Outro exemplo de execução do programa:

```
Entre com uma letra: AbcD
Voce teclou A, e seu codigo  ASCII eh 65.
```

Neste caso o usuário digitou quatro caracteres e depois teclou **RETURN** . Embora quatro caracteres tenham sido digitados, somente uma chamada a função cin.get() foi feita pelo programa, portanto só um caractere foi lido. O valor atribuído ao argumento da função é o código ASCII do primeiro caractere lido.

O tipo do resultado da função cin.get() é int e não char. O valor retornado pela função é o código ASCII do caractere lido.

### 29.2.2 Marcando o final da entrada

Frequentemente quando você está digitando a entrada para o programa, você quer dizer ao programa que você terminou de digitar o que queria. Em ambiente Unix, digitando ^D (segure a tecla de Ctrl e pressione D) você diz ao programa que terminou a entrada do programa. Em ambiente MS-Windows, você faz isto digitando ^Z (segure a tecla de Ctrl e pressione Z).

Isto envia uma indicação para a função cin.get(). Quando isso ocorre, o valor de ch depois de executar ch = cin.get(); será um valor especial do tipo inteiro chamado EOF (que significa *end of file* – final do arquivo).

Considere o seguinte programa exemplo que conta o número de caracteres digitados (incluindo o caractere de "próxima linha"):

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

int main()
{

 int total = 0;
 char ch;

 // Le o proximo caractere em ch e pára quando encontrar
 // final do arquivo
 ch = cin.get();
 while( ! cin.eof() ) {
    total++;
    ch = cin.get();
 }
 cout << endl <<  total << " caracteres digitados" << endl;
}
```

Só para esclarecer: você deve teclar **RETURN** depois de entrar com o comando ^D (ou ^Z no MS-Windows).

### 29.2.3 Para evitar problemas com a entrada...

(Observação: nesta seção, espaços em branco são relevantes e são mostrados como ␣)

Quando você executa um programa, cada caractere que você digita é lido e considerado como parte do *fluxo de entrada*. Por exemplo, quando você usa cin.get(), você deve teclar **RETURN** no final. Como mencionado anteriormente, o primeiro caractere digitado é lido pelo cin.get(). Mas, o caractere de nova linha continua no fluxo de entrada (porque você teclou **RETURN** ).

De qualquer forma, se você executar um cin.get() depois de um cin ou de um cin.get() você lerá o caractere de nova linha deixado no fluxo de entrada.

Da mesma forma, quando você usa cin para ler informações, ele somente lê o que é necessário. Se voce usar cin para ler um número inteiro e digitar `42␣␣` (seguido de **RETURN**), o cin lê `42`, mas deixa `␣␣` (e o caractere de nova linha do **RETURN**) no fluxo de entrada.

Outro caso "problemático" é quando o cin é usado num laço. Se você digitar um valor do tipo errado, o cin lerá o valor errado e a execução do laço continuará na sentença após o cin. Na próxima iteração do laço o cin vai tentar ler novamente, mas o "lixo" deixado da iteração anterior ainda estará lá, e portanto a chamada corrente do cin também não dará certo. Este comportamento resultará num laço infinito (um laço que nunca termina), ou terminará e terá um resultado errado.

Há uma maneira simples de resolver este problema; toda vez que você usar cin.get() (para ler um caracter só) ou cin, você deve ler todo o "lixo" restante até o caractere de nova linha. Colocando as seguinte linhas após chamadas a cin.get() ou cin o problema é eliminado:

```
// Pula o restante da linha
while( cin.get(c) != '\n' );
```

Note que isso não é necessário após todas as chamadas a cin.get() ou cin. Só depois daquelas chamadas que precedem cin.get() (ou cin), especialmente em um laço.

## 29.3 Validação de entrada

Muitas vezes queremos em nossos programas que eles obtenham um certo tipo de dado do usuário (por exemplo, valores inteiros) mas queremos também tratar a condição em que o usuário digita um dado real ou texto no lugar, exigindo uma reentrada de dados.

O exemplo abaixo nos mostra como fazer isto, usando a sentença com `cin` como expressão de um `if` ou `while`.

**Exemplo 1:**

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <cstring>

using namespace std;

void cleanInput()
{
  char cc;

  if (cin.fail())
  {
    cin.clear();
    while (cin.get(cc) && cc != '\n');
  }
}

void printStatInput()
{
  cout << "\tGood: " << cin.good()
       << "     Fail: " << cin.fail()
       << "     Bad: " << cin.bad()
```

```
       << "     EOF: " << cin.eof() << endl << endl;
}


int main ()
{
  int b=1111, c=2222, d=3333;
  float x=1.111, y=2.222;
  void * cinerr;

  cout << "b c d = " << b << " " << c << " " << d << endl;
  cout << "x y = " << x << " " << y << endl;

  while (! (cinerr = cin >> b))
  {
    cleanInput();
    cout << "cin 1 = " << cinerr << " b = " << b << endl;
    printStatInput();
    cout << "Entrada 1: ";
  }
  cout << "** cin 1 = " << cinerr << " b = " << b << endl;

  while (! (cinerr = cin >> b >> c >> d))
  {
    cleanInput();
    cout << "cin 2 = " << cinerr << " b c d = " << b << " " << c << " " << d << e
    printStatInput();
    cout << "Entrada 2: ";
  }
  cout << "** cin 2 = " << cinerr << " b c d = " << b << " " << c << " " << d <<

  cinerr = cin >> x;
  cout << "cin 3 = " << cinerr << " x = " << x << endl;
  printStatInput();

  if (!cinerr)
  {
    /* Limpar condição de erro em cin, se houver.    */
    cout << "Limpando condições de erro de CIN" << endl;
    cin.clear();
    printStatInput();
  }

  cout << "Ultima entrada: ";
  cinerr = cin >> x >> y;
  cout << "cin 4 = " << cinerr << " x y = " << x << " " << y << endl;
  printStatInput();

  return 0;
```

```
}
```

**cin » ...** retorna um valor especial[5]. Se nulo, a entrada não foi aceita e o valor da variável em que houve erro fica inalterado. As variáveis posteriores na leitura em que houve erro também mantém seus valores inalterados. Execuções subsequentes de **cin** somente podem ser feitas após a execução de **cin.clear()**, para eliminar a condição de erro. Caso isto não seja feito, após o primeiro erro de leitura, todos os **cin » ...** subsequentes darão erro.

Também podem ocorrer problemas de conversão de tipos em **cin**. Observe o exemplo abaixo:

```
        int b;
float x;

cin >> b;
cin >> x;
```

Neste caso, se usuário digita um float (por exemplo, 3.7) no 1º cin, o 2o.º cin não solicitará dado: 'b' recebe 3, deixando os chars '.4' no buffer de leitura. o 2º cin, então, armazena direto 0.4 em 'x', não aguardando digitação do usuário. Note que neste caso, **cin » ...** não acusará erro (isto é, não retorna valor nulo).

## 29.4 Formatação de saída

Para formatar a saída de dados com `cout`, (por exemplo, restringir a quantidade de casas decimais mostradas, definir o tamanho máximo em caracteres, ou imprimir uma matriz de valores alinhados adequadamente pelas colunas) usam-se a função setw() e o método setf() (cout.setf(....)).

Os exemplos abaixo mostram como usar estes elementos.

**Exemplo 2:**

```
#include <iostream>
#include <iomanip>    // Necessário para usar os manipuladores setw()

  using namespace std;

  int main() {
    int n1, n2, n3;
    int soma;

    cout <<  "Entre com 3 numeros inteiros: ";
    cin >> n1 >> n2 >> n3;
    soma = n1 + n2 + n3;
    cout << "Soma = " << soma << endl;

    /* Define impressão de números em ponto fixo e
       mostra sempre o ponto decimal seguido por zeros
    */
    cout.setf (ios::fixed | ios::showpoint);

    /* Outros flags válidos para setf():
       ios::fixed
```

[5]Este valor é um ponteiro para um objeto.

```
       ios:scientific
       ios::showpoint
       ios::right
       ios::left
    */

    /* Define quantidade de algarismos a serem exibidos após ponto decimal.
       Em alguns  compiladores, pode indicar a  quantidade de algarismos
       SIGNIFICATIVOS.
    */
    cout.precision(2);

    // setw() fixa a largura do campo de saída.
    // Aplica-se apenas ao próximo item exibido na saída.
    // Necessário incluir <iomanip> (vide início do código)
    cout <<  "Media = " << setw(8) << soma / 3.0 << endl;
    cout <<  "Produto = " << n1 * n2 * n3 << endl;
  }
```

**Exemplo 3:**

```
//  Exemplo de array 2-D - taboada

#include <iostream>
#include <iomanip>    // NecessÃ¡rio para usar os manipuladores setw()

using namespace std;

#define LIN 10
#define COL 10

int main()
{
  int x;                //  numero da coluna
  int y;                //  numero da linha
  int taboada[LIN][COL]; // tabela de taboada

  //  preenche a taboada

  y=0;
  while(y < LIN)
  {
    x=0;
    while(x < COL)
    {
      taboada[y][x] = y*x;
      x = x+1;
    }
    y = y+1;
  }
```

```
  cout << "\n         Taboada de Multiplicacao\n";

  //  Imprime o numero das colunas
  cout << setw(6) << 0;
  x=1;
  while(x < COL)
  {
    cout << setw(3) << x;
    x = x+1;
  }
  cout << endl;

  //  Imprime uma "linha horizontal"
  cout << "   ";
  x=0;
  while (x < 3*COL)
  {
    cout << "-";
    x = x+1;
  }
  cout << endl;

  //  Imprime as linhas da tablea.
  //  Cada linha a precedida pelo indice de linha e uma barra vertical

  y=0;
  while (y < LIN)
  {
    cout << setw(2) << y << "|";

    x=0;
    while(x < COL)
    {
      cout << setw(3) << taboada[y][x];
      x = x+1;
    }

    cout << endl;
    y = y+1;
}
```

# 30 Arquivos

O armazenamento de dados em variáveis e *arrays* é temporário. Arquivos são usados para armazenamento permanente de grandes quantidades de dados (e programas) em dispositivos de armazenamento secundário, como discos.

Ás vezes não é suficiente para um programa usar somente a entrada e saída padrão. Há casos em que um programa deve acessar arquivos. Por exemplo, se nós guardamos uma base de dados com endereços de pessoas em um arquivo, e queremos escrever um programa que permita ao usuário interativamente buscar, imprimir e mudar dados nesta base, este programa deve ser capaz de ler dados do arquivo e também gravar dados no mesmo arquivo.

No restante desta seção discutiremos como arquivos de texto são manipulados em C++ . Como será visto, tudo ocorre de maneira análoga ao que acontece com entrada e saída padrão.

## 30.1 Acessando um arquivo: tipo `fstream`, funções open() e close()

C++ visualiza cada arquivo simplesmente como um fluxo (*stream*) seqüencial de bytes. Cada arquivo em C++ termina com um marcador de final de arquivo (*end-of-file*), definido pela constante `EOF`.

As regras para acessar um arquivo são simples. Antes que um arquivo seja lido ou gravado, ele é *aberto*. Um arquivo é aberto em um *modo* que descreve como o arquivo será usado (por exemplo, para leitura, gravação ou ambos). Um arquivo aberto pode ser processado por funções da biblioteca padrão em C++ . Estas funções são similares às funções de biblioteca que lêem e escrevem de/para entrada/saída padrão. Quando um arquivo não é mais necessário durante a execução do programa ele deve ser *fechado*. Ao final da execução de um programa todos os arquivos abertos são automaticamente fechados. Existe um número máximo de arquivos que podem ser simultaneamente abertos de forma que você deve fechar arquivos quando você não precisa mais deles em seu programa.

Quando um arquivo está *aberto*, um *stream* é associado ao arquivo. Este *stream* fornece um canal de comunicação entre um arquivo e o programa. Para definir tal *stream*, uma variável do tipo **ofstream**, **ifstream** ou **fstream** deve ser declarada. Esta variável irá ser usada para associar um arquivo em disco que tem um certo nome com as operações de leitura e gravação que veremos adiante. Três variáveis deste tipo e seus respectivos *streams* são abertos automaticamente quando um programa inicia sua execução: a entrada padrão (`cin`), a saída padrão (`cout`) e a saída padrão de erros (`cerr`).

Para abrir um arquivo, deve ser usada a função (método) open() associada à uma variável de um dos tipos **stream**. open() toma dois argumentos: o primeiro argumento é do tipo *string* e representa o nome do arquivo (por exemplo data.txt); o segundo argumento é a indicação do modo no qual o arquivo deve ser aberto. open() negocia com o sistema operacional e define o estado final do processo de abertura, que deve ser testado com o método fail()().

O programador/usuário não necessita saber detalhes de como a transferência de dados entre programa e arquivo é feita. As únicas sentenças necessárias no programa são a definição de uma variável do tipo **s**tream e a abertura do arquivo. As sentenças abaixo dão um exemplo de como abrir o arquivo data.txt para leitura.

```
ifstream infile;
infile.open("data.txt");
```

O protótipo do método open() é:

```
<stream>.open(char *name);
```

open() recebe um argumento: uma *string* que é um nome de um arquivo a ser aberto. o tipo do *stream* indica a operação que será feita no arquivo: **ofstream** para escrita apenas, **ifstream** para leitura apenas, e **fstream** para leitura e escrita. Para este último tipo, o método open() recebe um segundo argumento,

indicando o modo de abertura do arquivo: `ios::in` indica que o arquivo será aberto apenas para leitura; `ios::out`, para escrita apenas. Associado a estes valores podem ser usados os valores `ios::app`, `ios::ate`, `ios::trunc`, e `ios::binary`.

Se o arquivo não existe e é aberto para escrita, fopen() cria o arquivo. Se um arquivo já existente é aberto para escrita, o seu conteúdo é descartado. Há outros modos, incluindo anexação a um arquivo, leitura e escrita simultânea; para mais detalhes, veja a documentação da função nos livros-texto ou no manual on-line.

Para verificar se um arquivo é aberto com sucesso, usam-se os métodos fail() ou is_open(). Estas funções retornam o valor 1 (um) indicando respectivamente se houve falha na abertura ou se a abertura foi efetuada com sucesso. Alguns dos erros possíveis são: abrir um arquivo que não existe para leitura, abrir um arquivo para escrita quando não há mais espaço disponível em disco, ou abrir um arquivo para qualquer operação sendo que as permissões de acesso do arquivo não o permitem.

É recomendável que você teste se a abertura teve sucesso antes de continuar o programa. O trecho de programa abaixo ilustra como fazê-lo:

```
#include <iostream>
#include <fstream>

using namespace std;

....
fstream fp;
char fnome[13];

printf("Entre um nome de arquivo para abrir:");
cin >> fnome;

fp.open(fnome, ios::out);
if (fp.fail())
 {
   cout << "Erro na abertura de "<< fnome << " no modo escrita" << endl;
   return ;
 }
else
   cout << "Arquivo " << fnome << " aberto com sucesso no modo escrita" << endl;
...
```

No exemplo acima se o arquivo não puder ser aberto com sucesso, uma mensagem apropriada é exibida na saída padrão e o programa termina. Caso contrário uma mensagem indicando o sucesso na abertura do arquivo é exibida e o programa continua sua execução.

Cada arquivo aberto possui seu próprio *stream*. Por exemplo, se um programa vai manipular dois arquivos diferentes arq1.txt e arq2.txt simultaneamente (um para leitura e outro para escrita), dois *streams* devem ser declarados:

```
fstream fp1, fp2;

fp1.open("arq1.txt", ios::in);
fp2.open("arq2.txt", ios::out);
```

As variáveis do tipo *stream* estabelecem conexão entre o programa e o arquivo aberto. A partir do momento de abertura, o nome do arquivo é irrelevante para o programa. Todas as funções que operam sobre o arquivo usam o *stream* associado.

Terminada a manipulação do arquivo o programa deve fechar o arquivo. O método close() é usado com este propósito. Ela quebra a conexão entre o *stream* e o arquivo. O protótipo do método close() é:

```
<stream>.close();
```

Abaixo um exemplo de uso de fclose():

```
fstream fp1, fp2;

fp1.open("arq1.txt", ios::in);
fp2.open("arq2.txt", ios::out);

.........

fp1.close();
fp2.close();
```

## 30.2 Processando arquivos de texto

Arquivos podem guardar duas categorias básicas de dados: **texto** (caracteres codificados em ASCII ou UTF) ou **binário** (como dados armazenados em memória ou dados que representam uma imagem JPEG ou PNG).

Depois que um arquivo de texto é aberto, existem 3 formas diferentes de ler ou escrever sequencialmente os dados: (i) um caracter por vez, usando as funções da biblioteca padrão get() e put(); (ii) uma linha (*string*) por vez, usando gets() e puts(); e (iii) em um formato específico, usando os operadores <<() e >>(), como já fizemos com cout() e cin().

### 30.2.1 Entrada e saída de caracteres

As funções get() e put() operam sobre um arquivo aberto em modo de leitura.

Os protótipos de get() e put() are

```
<stream>.get();
<stream>.putc(char ch);
```

get() retorna o próximo caracter lido do arquivo representado pela *stream*, ou `EOF` se ocorrer final de arquivo ou um erro de leitura.

put() grava o caracter `ch` no arquivo representado pela *stream*. Esta funÃ§Ã£o retorna o caracter gravado ou `EOF`.

Abaixo segue um exemplo de programa que lê um arquivo caracter a caracter e imprime o que foi lido na saída padrão (a tela do computador):

```
/*************************************************************************
 * Lê um caracter por vez de um arquivo e
 * o imprime na saída padrão
 *************************************************************************/
#include <iostream> /* para funções padrão de E/S */
#include <fstream> /* para funções de E/S em arquivos */

using namespace std;

int main()
{
   ifstream fp;
```

```
   char fnome[13];
   char ch;

   /* dialogo com usuário */
   cout << "Entre um nome de arquivo: ";
   cin >> fnome;

   fp.open( fnome, ios::in ); /* abre arquivo*/
   if (fp.fail()) {
      printf("Erro ao abrir %s\n", fnome);
      return;
      }
   else {
      printf("Arquivo aberto com sucesso.\n");

      /* Lê o arquivo caracter a caracter e imprime em stdout (saída padrão) */
      while( (ch = fp.get()) != EOF )
          cout << ch;

      fp.close(); /* fecha arquivo */
   }
}
```

Observe o tipo da variável **ch**. Mude o tipo para **int** e execute novamente o programa (após compilação). Você pode exlicar a diferença nos resultados?

## Referências


[1] H. M. Deitel and P. J. Deitel. *C++: Como Programar*. Prentice-Hall, Inc., 5ª edition, 2006.

[2] Stanley B. Lippman. *C++ Primer*. Addison-Wesley Publishing Company, 1991. ISBN: 0-201-54848-8. Livro bastante didático sobre C++.

[3] Victorine Viviane Mizrahi. *Treinamento em Linguagem C++*. Prentice-Hall, Inc., 2ª edition, 2005.

[4] Walter Savitch. *C++ Absoluto*. Addison-Wesley Publishing Company, 2004. ISBN: 85-88639-09-2. Livro bastante didático sobre C++.

[5] Bjarne Stroustrup. *A Linguagem de ProgramaÃ§Ã£o C++*. Bookman, 2000.

**Créditos:** Documento produzido pelo Prof. Armando L.N. Delgado (DINF/ET/UFPR), baseado em revisão sobre material de Profª Carmem Hara e Prof. Wagner Zola (DINF/ET/UFPR).