Universidade Estadual Paulista "Júlio de Mesquita Filho"
Faculdade de Ciências Agrárias e Veterinárias
Departamento de Ciências Exatas

Linguagem de Programação C++

Alan Rodrigo Panosso

2019

Sumário

Apostila C++

1. Introdução

1.1 Histórico da linguagem C++

O *C++* foi inicialmente desenvolvido por Bjarne Stroustrup dos Bell Labs, durante a década de 1980 com o objetivo de programar uma versão distribuída do núcleo Unix. Como o Unix era escrito em *C*, dever-se-ia manter a compatibilidade, ainda que adicionando novos recursos. Stroustrup observou que a linguagem *Simula* possuía características bastante úteis para o desenvolvimento de software, mas que era muito lenta para uso prático. Por outro lado, a linguagem *BCPL* era rápida, mas possuía demasiado baixo nível, dificultando sua utilização no desenvolvimento de aplicações. A partir de sua experiência de doutorado, começou a acrescentar elementos do *Simula* no *C*.

A linguagem *C* foi escolhida como base para o desenvolvimento da nova linguagem, pois possuía uma proposta de uso genérico, era rápida e também portável para diversas plataformas. Algumas outras linguagens que também serviram de inspiração para o cientista da computação foram *ALGOL 68, Ada, CLU e ML*.

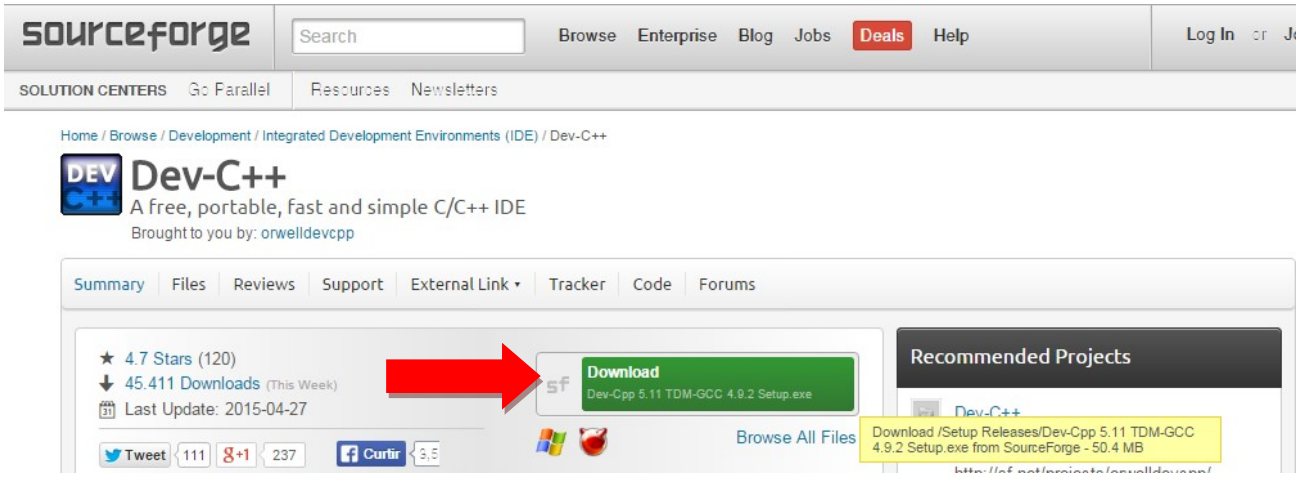
Ainda em 1983 o nome da linguagem foi alterado de *C with Classes* para *C++*. Antes, implementada usando um pré-processador, a linguagem passou a exigir um compilador próprio, escrito pelo próprio Stroustrup. Novas características foram adicionadas, como funções virtuais, sobrecarga de operadores e funções, melhorias na verificação de tipo de dado e estilo de comentário de código de uma linha (//). Em 1985 foi lançada a primeira edição do livro *"The C++ Programming Language"*, contendo referências para a utilização da linguagem, já que ainda não era uma norma oficial. A primeira versão comercial foi lançada em outubro do mesmo ano. Em 1989 a segunda versão foi lançada, contendo novas características como herança múltipla, classes abstratas, métodos estáticos, métodos constantes e membros protegidos, incrementando o suporte a orientação a objeto.

Assim como a linguagem, sua biblioteca padrão também sofreu melhorias ao longo do tempo. Sua primeira adição foi a biblioteca de *E/S*, e posteriormente a *Standard Template Library* (*STL*); ambas tornaram-se algumas das principais funcionalidades que distanciaram a linguagem em relação a *C*. Criada primordialmente na *HP* por Alexander Stepanov no início da década de 1990 para explorar os potenciais da programação genérica, a *STL* foi apresentada a um comitê unificado ANSI e ISO em 1993 à convite de Andrew Koenig. Após uma proposta formal na reunião do ano seguinte, a biblioteca recebeu o aval do comitê. Pode-se dizer que *C++* foi a única linguagem, entre tantas outras, que obteve sucesso como uma sucessora à linguagem *C*, inclusive servindo de inspiração para outras linguagens como *Java*, a *IDL* de *CORBA* e *C#*.

1.2 Compilador – Tutorial de Instalação

Dentre os vários compiladores disponíveis, escolhemos para adotar na disciplina o **Bloodshed Dev-C++**, ou simplesmente **Dev-C++** que é um ambiente de desenvolvimento integrado completo para as linguagens *C* e *C++*, que utiliza uma coleção de compiladores GNU. Este tutorial apresenta, de maneira simplificada, as principais etapas para a instalação do **Dev-C++**. Serão apresentadas a instalação no Sistema Operacional Windows 7.

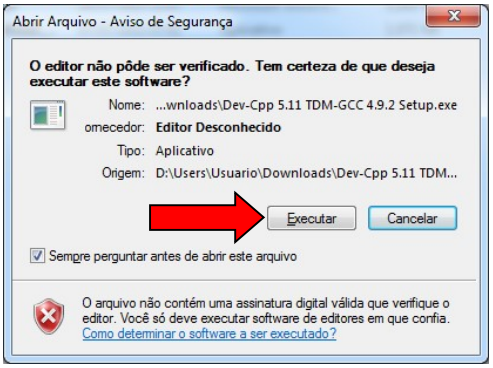
1. Inicialmente Acesse o endereço http://sourceforge.net/projects/orwelldevcpp/ e clique na opção "**Download**", como apresentado na imagem abaixo.



2. Na próxima página localize o download deve iniciar automaticamente, a versão mais atual do programa, "***Dev-Cpp 5.11 TDM-GCC 4.9.2 Setup***".

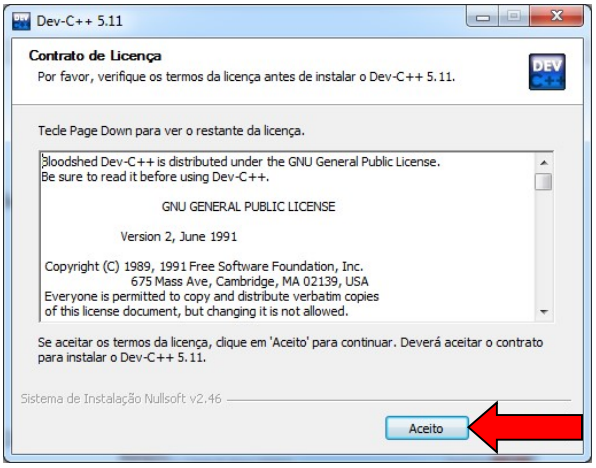
3. Após o download, localize o instalador do programa na pasta específica do seu computador e clique duas vezes para iniciar o processo de instalação.

4. Siga os passos abaixo para a instalação do programa:

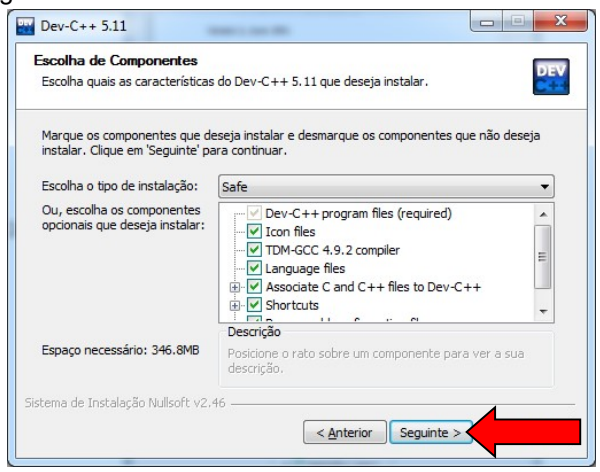


a. Selecione a Linguagem "Português".

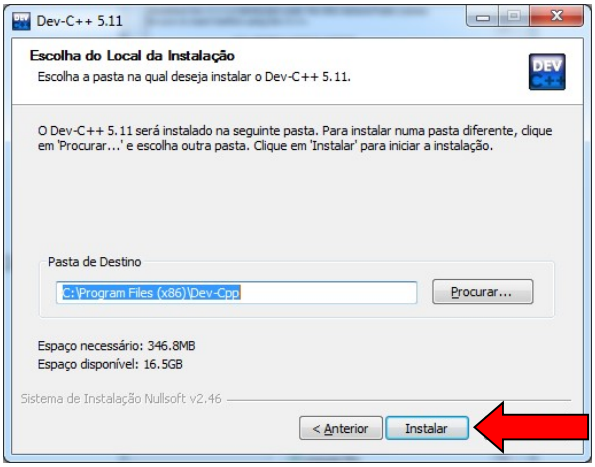
b. Clique em "Aceito".



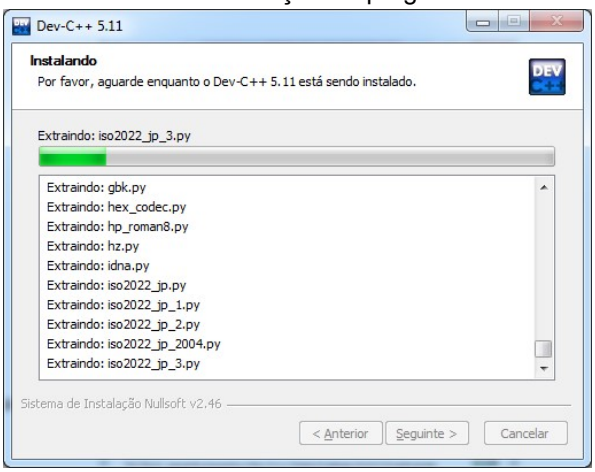
c. Clique em "Seguinte".



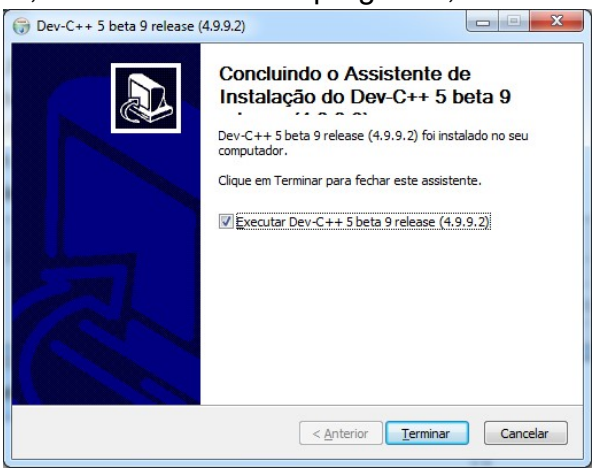
d. Selecione "Instalar".



e. Aguarde o assistente terminar a instalação do programa.

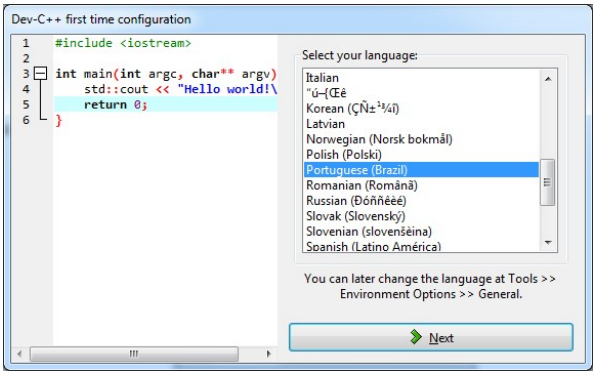


5. Ao final da instalação, deixe marcado a opção ***"Executar Dev-C++ ..."*** e clique em terminar. Ou, localize o ícone do programa, e inicialize-o.

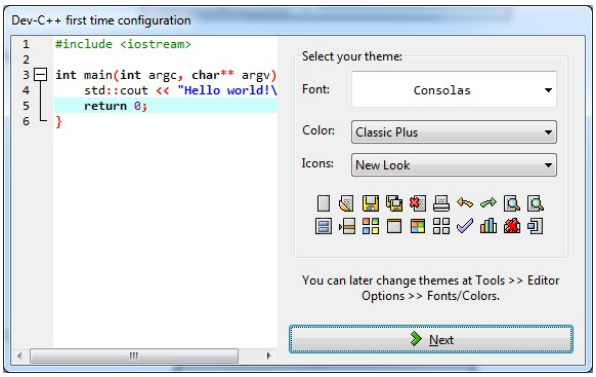


ou 

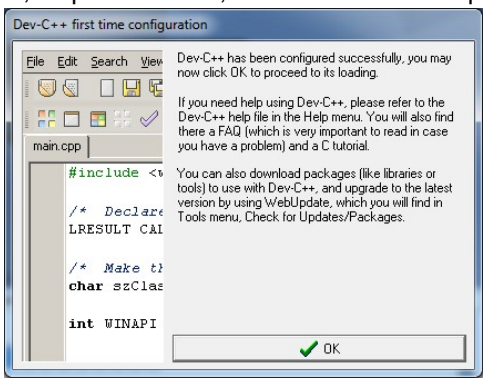
a. Selecione a linguagem "Portugues (Brazil)".



b. Ecolha a opções de De Fontes, Cores e Ícones.

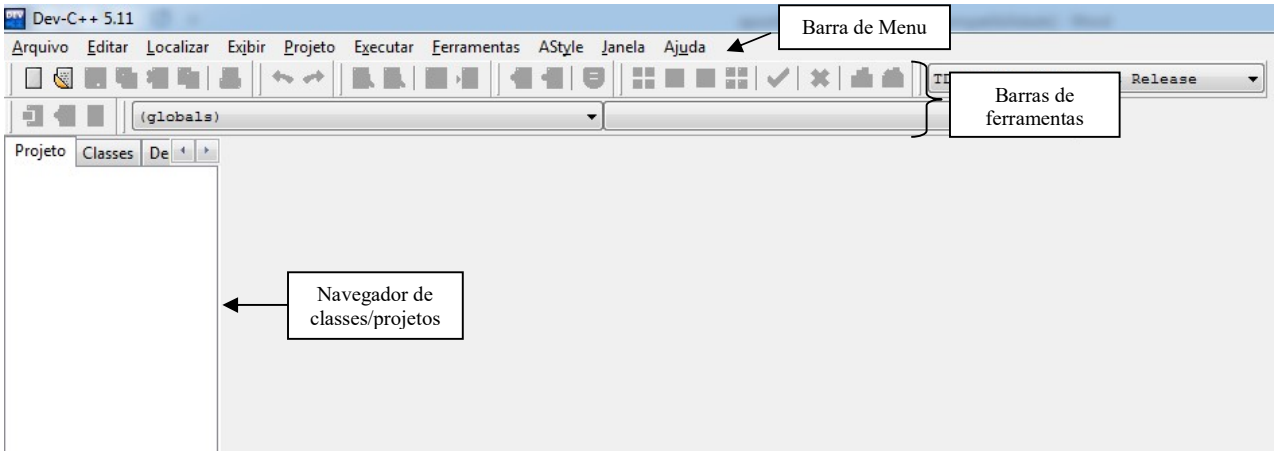


c. Ao final da aplicação, clique em "OK" , e o Dev-C++ estará pronto para ser utilizado.

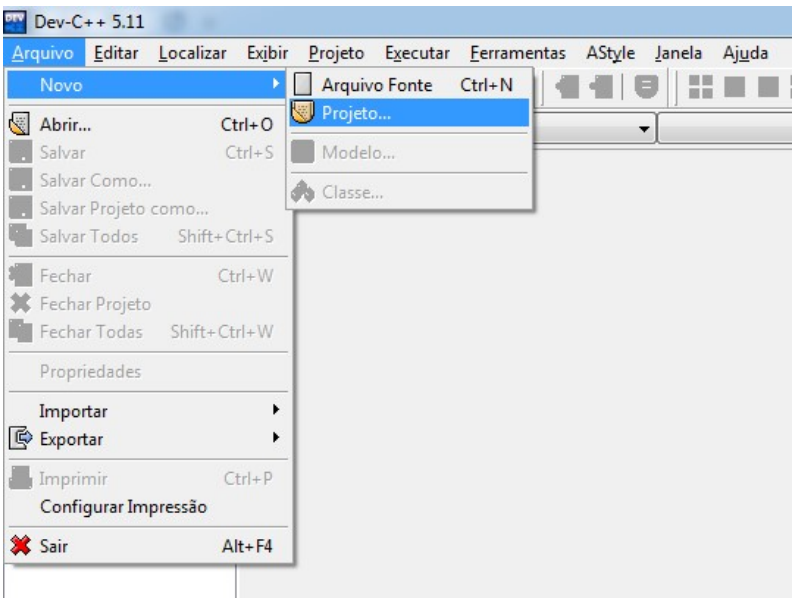


1.3 Apresentando o Dev-C++

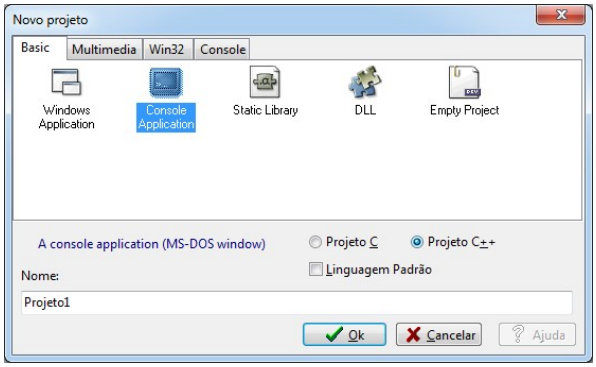
Após a realização das etapas anteriores, será apresentado ao usuário a seguinte tela, onde podemos distinguir os seguinte elementos:



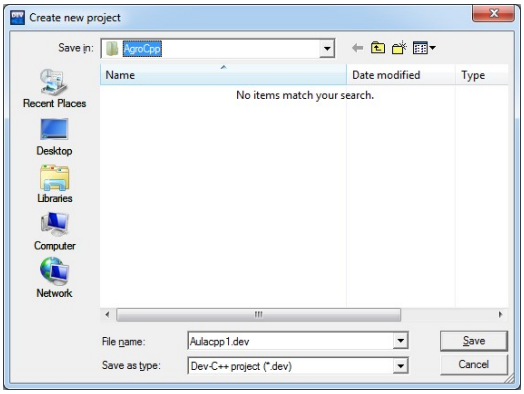
Na barra de ***Menu***, selecione a opção "***Arquivo/Novo/Projeto...***", como apresentado na imagem abaixo:



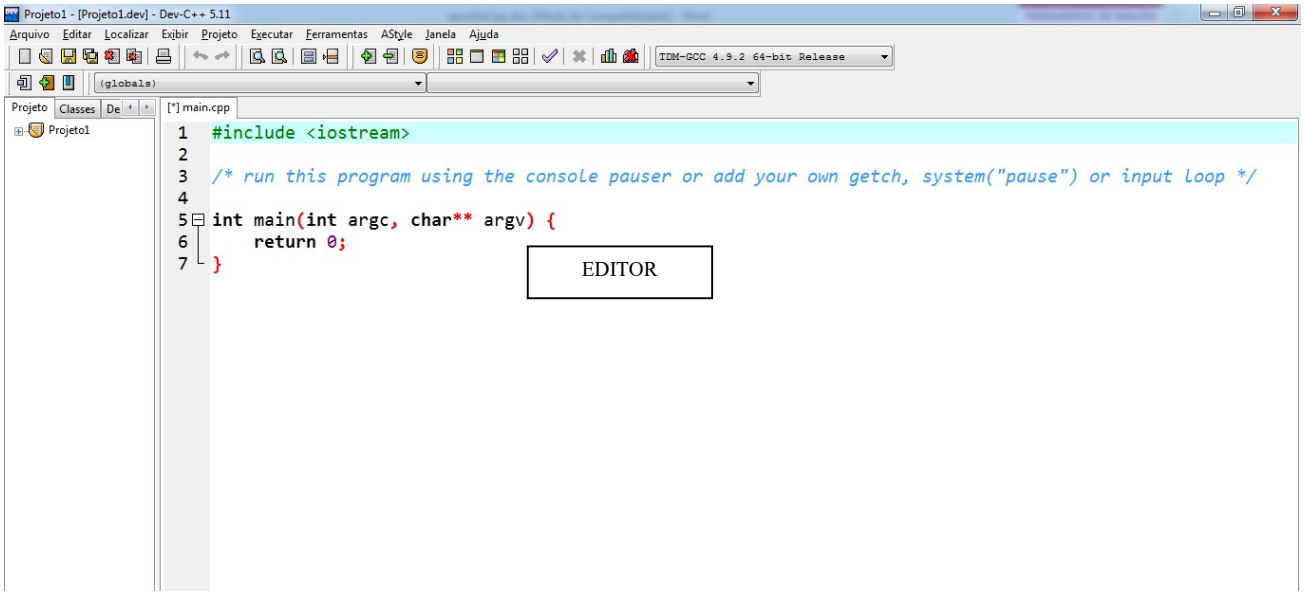
Na janela "**Novo Projeto**", escolha o ícone "**Console Application**" e em **"Opções de Projeto"** digite no campo **Nome**, o nome do projeto (sem espaços ou caracteres especiais) e em seguida marque a opção "***Projeto C++***".



Salve o projeto (**.dev**) em uma pasta pessoal destinada a armazenar seus projetos pessoais.



Em seguida será apresentado a Janela "**Editor"** com os comandos básicos para o início e desenvolvimento de um programa na linguagem *C++*.

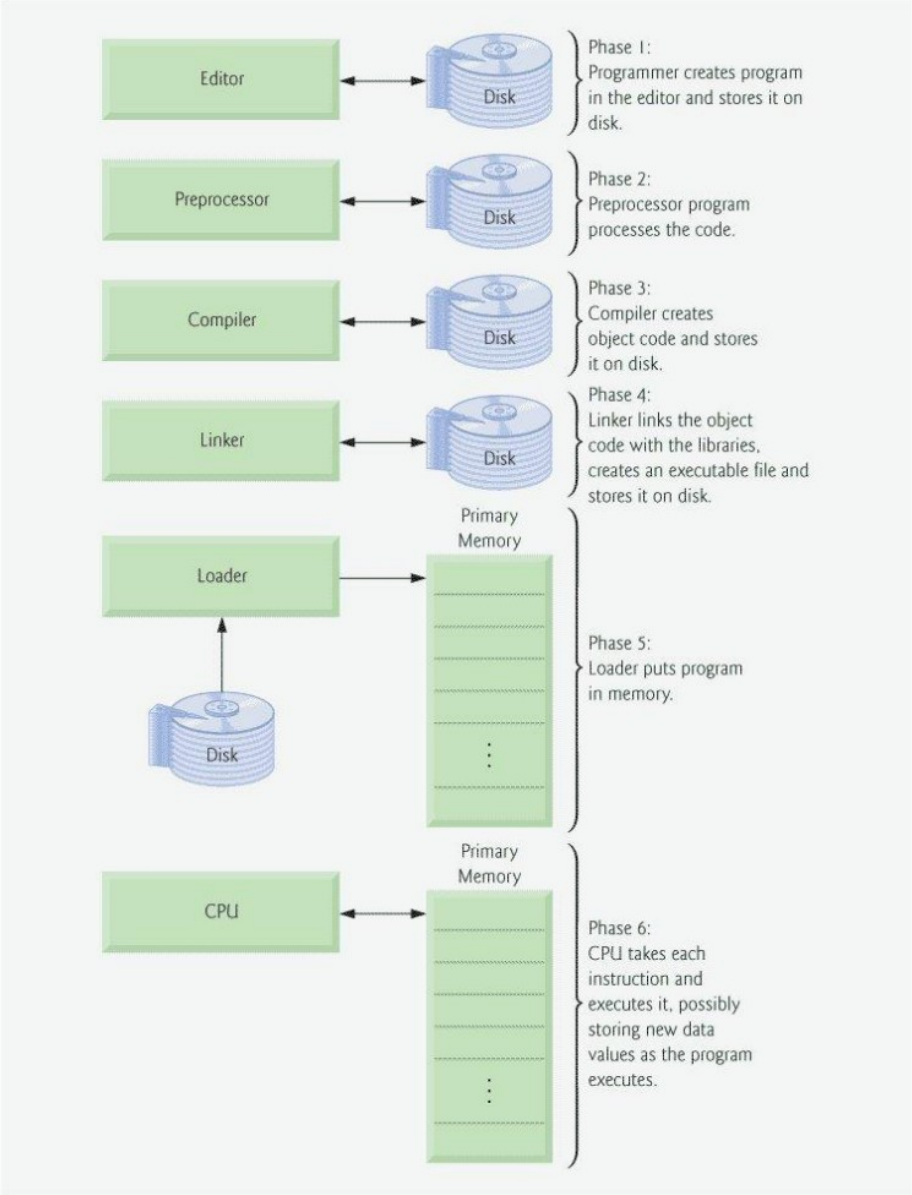


1.4 Compiladores

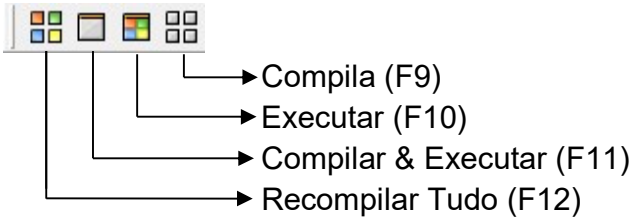
Um compilador é um programa de sistema que traduz um programa descrito em uma linguagem de alto nível para um programa equivalente em código de máquina (baixo nível) para um processador. Em geral, um compilador não produz diretamente o código de máquina, mas sim um programa em linguagem simbólica (*assembly*) semanticamente equivalente ao programa em linguagem de alto nível.

O programa em linguagem simbólica é então traduzido para o programa em linguagem de máquina por meio de montadores. Para desempenhar suas tarefas, um compilador deve executar dois tipos de atividade. A primeira atividade é a análise do código fonte, onde a estrutura e significado do programa de alto nível são reconhecidos.

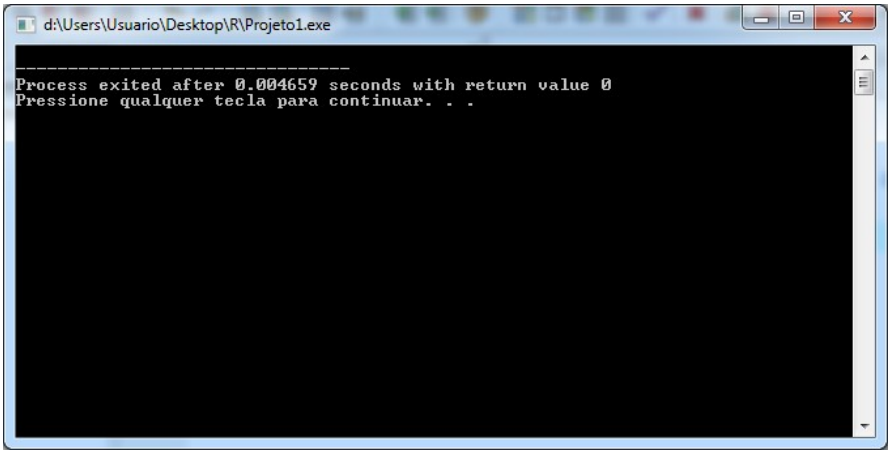
A segunda atividade é a síntese do programa equivalente em linguagem simbólica. Embora conceitualmente seja possível executar toda a análise e apenas então iniciar a síntese, em geral estas duas atividades ocorrem praticamente em paralelo. A figura abaixo apresenta as operações executas em um ambiente típico C++.



No ***Dev-C++***, as opções para compliação e execussão são encontradas na barra de ferramentas **"Compilar e Executar"**.



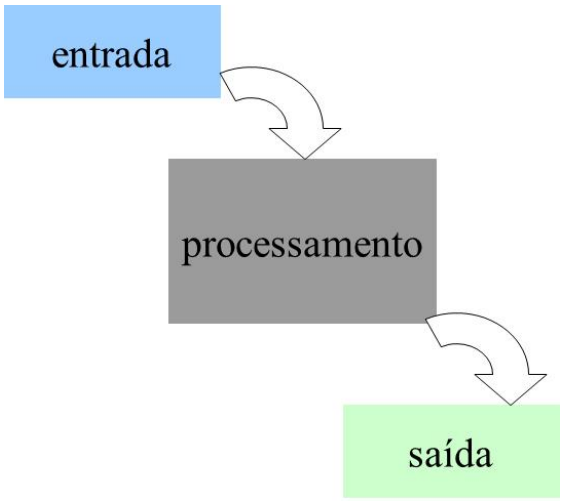
Selecione o botão  "**Compilar & Executar**" ou pressione a tecla **"F9"**. A seguinte tela deve aparecer, indicando que todos os passos anteriormente apresentados foram executados com sucesso pelo compilador.



Pressione qualquer tecla para continuar, ou feche a janela do console para voltar ao editor do programa.

2. Estrutura de um programa

Todo programa consiste em uma ou mais funções, tendo como particularidade desse fato a possibilidade de construir programas modulares e estruturados.



Assim, a estrutura de um programa em C++ é:

```
// Estrutura do C++
[<definições de pré-processamento – cabeçalhos>]
[<declaração das variáveis globais>]
[<tipo>] main([<parâmetros>])
{
        /* Este trecho é reservado para o corpo da função, com a declaração de
        suas variáveis locais, seus comandos e funções de trabalho*/
        return [<valor>];
}

[<tipo>] funçãonome([<parâmetros>])
{....
        [<declaração de parâmetros>]
        [return; ou return(); ou return(valor);]
}
]
```

2.1 Controle dos Fluxos de Entrada e de Saída

É por meio dos fluxos de entrada e de saída que os operadores de acesso ao teclado e vídeo podem ser efetuados. Assim sendo será possível efetuar entrada (`cin` – console input) e saída de dados (`cout` – console output) para todas as operações realizadas por um programa.

`cin` – trabalha com o teclado.

`cout` – trabalha com o monitor.

Para fazer uso básico dos fluxos `cin` (fluxo de entrada) e `cout` (fluxo de saída), é necessário inserir no início do programa a chamada ao arquivo de cabeçalho correspondente denominado `iostream`.

Chamada do cabeçalho:

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

2.2 Primeiro Programa

A melhor maneira de iniciarmos o aprendizado de uma nova linguagem de programação é escrevendo nossos próprios programas. Neste material, será adotada a seguinte estrutura, o primeiro painel apresentará os códigos fonte para os programas, e o segundo painel apresentará o resultado do programa uma vez compilado e executado.

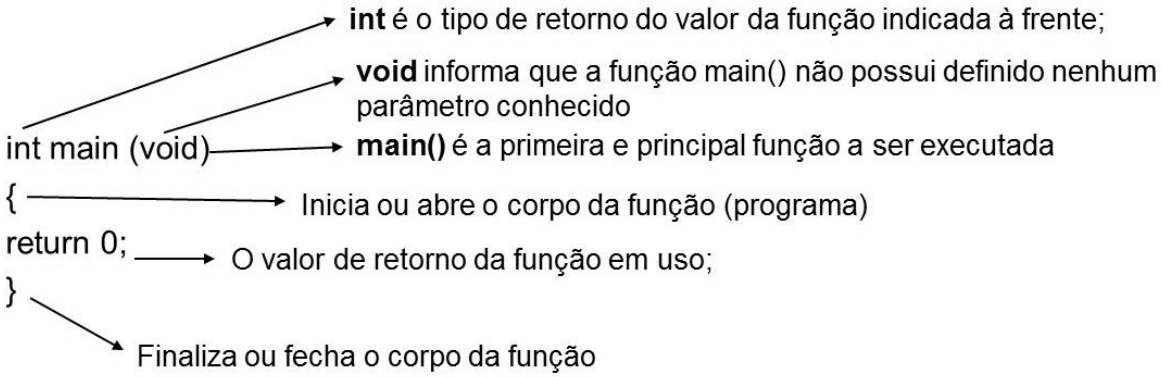
Janela Código Fonte

```
// Programa em C++
#include <iostream>
using namespace std;
int main(void)
{
    cout << "Oi mundo" << endl;
    system ("Pause");
}
```

Resultado (após compilação e execução)

```
Oi mundo
Press any key to continue...
```

O programa principal escrito em linguagem C++ é uma função, por natureza própria.



Vamos entender as linhas do primeiro programa:

`// Programa em C++`

Está é uma linha de comentário. Todas as linhas começando com duas barras (//) não tem qualquer efeito no comportamento de um programa. Recomendamos que, no processo de aprendizagem, várias linhas de comentários sejam adicionadas, para explicações ou observações a respeito do código.

C++ apresenta duas maneiras de fazer comentários

```
// Linha de comentário
/* Bloco de comentário
   que pode abranger
   um grande número de linhas*/
```

A primeira forma de comentário é conhecida como linha de comentário, descartando tudo a partir das duas barras (`//`) até o final da mesma linha. O segundo tipo é conhecido como bloco de comentário, descartando tudo entre a instrução `/*` e a primeira ocorrência da instrução `*/`, com a possibilidade da inclusão mais de uma linha.

```
#include <iostream>
```

Linhas iniciando com o sinal (`#`) são diretrizes para o pré-processador (pré-compilação), ou seja, instruções para o compilador. Neste caso, estamos dizendo ao pré-processador para incluir os arquivos padrões da biblioteca `iostream`. Estes arquivos específicos incluem as declarações da biblioteca básica de controle do fluxo de entrada (`cin`) e saída de dados (`cout`).

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

Definição inicial da função principal. A função principal é o ponto no qual todo o programa em *C++* inicia a sua execução. Independentemente de sua localização dentro do código fonte, ou seja, não importa se existem outras funções definidas (com outros nomes) antes ou depois dela, as instruções contidas dentro das chaves dessa função (`{ }`) sempre serão as primeiras a serem executadas. É, portanto, o programa principal, a estrutura básica de execução.

```
cout << "Oi mundo" << endl;
```

Esta é uma linha de instrução em C++. Uma instrução, ou declaração, são expressões simples, ou composta, que realmente produzem algum efeito no programa. De fato, esta instrução realiza uma única ação, gerar um efeito visível no primeiro programa.

`cout` representa o fluxo padrão de saída em *C++*, nessa declaração estamos inserindo a sequência de caracteres `"Oi mundo"` na saída de dados, no caso a tela do computador, por meio do console.

`endl` esse comando insere uma nova linha na saída da tela, para evitar que o programa apresente a saída abaixo:

```
Oi mundoPress any key to continue...
```

Observe que deve-se utilizar o operador de fluxo << (fluxo de saída).

Observe também que toda a declaração termina com o caractere "ponto & vírgula" (;). Representa o final de uma declaração, e precisa ser adicionado no final de todas as linhas de declaração nos programas em *C++*.

```
using namespace std;
```

Todos os elementos da biblioteca padrão de *C++* são declaradas dentro do que chamamos `namespace`, com o nome `std`. Linha muito frequente em programas em C++. Evita que, por exemplo, quanto utilizarmos `cout <<`, seja necessário escrevermos `std::cout <<`.

```
system("Pause");
```

Esta declaração pausa a janela de saída do programa, permitindo, ao usuário, a sua inspeção, nas versões mais atuais do Dev-C++, pode-se omitir essa instrução.

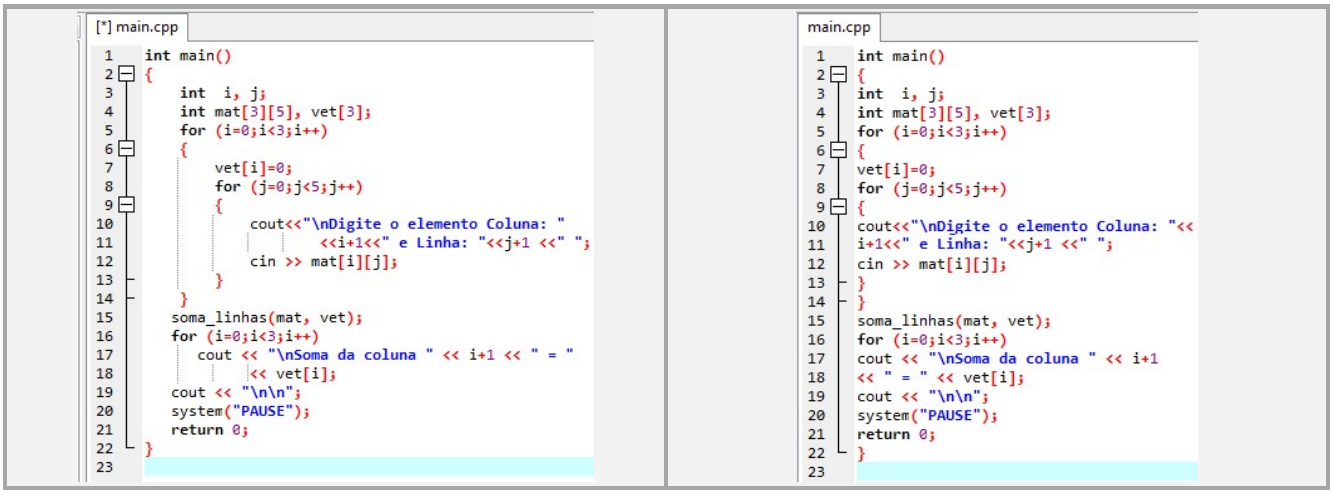
Observe que a função principal do nosso primeiro programa poderia ser escrita da seguinte maneira:

```
int main () {cout << "Oi mundo" << endl; system ("Pause");}
```

Sem comprometer o funcionamento do programa. Entretanto, um código estruturado em diferentes linhas é mais legível para os seres humanos, tornando muito mais simples a sua inspeção e compreensão, esse processo chama-se **Indentação**.

Em ciência da computação, **indentação** significa recuo, é um termo aplicado ao código fonte de um programa para ressaltar ou definir a estrutura do algoritmo. É empregada com o objetivo de ressaltar a estrutura do algoritmo, aumentando assim a legibilidade do código. Mesmo para uma única linguagem de programação, podem existir diversos estilos de indentação. Todos eles têm em comum, entretanto, o conceito de que blocos de código dependentes de um comando, declaração ou definição, devem ser identificados por um aumento no nível de indentação. Isto é, ***o espaçamento que antecede o código de cada linha*** deve ser aumentado com relação ao comando, declaração ou definição que o antecede.

Código **com indentação**	Código **sem indentação**



[*] main.cpp

```
int main()
{
    int  i, j;
    int mat[3][5], vet[3];
    for (i=0;i<3;i++)
    {
        vet[i]=0;
        for (j=0;j<5;j++)
        {
            cout<<"\nDigite o elemento Coluna: "
                <<i+1<<" e Linha: "<<j+1 <<" ";
            cin >> mat[i][j];
        }
    }
    soma_linhas(mat, vet);
    for (i=0;i<3;i++)
       cout << "\nSoma da coluna " << i+1 << " = "
            << vet[i];
    cout << "\n\n";
    system("PAUSE");
    return 0;
}
```

main.cpp

```
int main()
{
int  i, j;
int mat[3][5], vet[3];
for (i=0;i<3;i++)
{
vet[i]=0;
for (j=0;j<5;j++)
{
cout<<"\nDigite o elemento Coluna: "<<
i+1<<" e Linha: "<<j+1 <<" ";
cin >> mat[i][j];
}
}
soma_linhas(mat, vet);
for (i=0;i<3;i++)
cout << "\nSoma da coluna " << i+1
<< " = " << vet[i];
cout << "\n\n";
system("PAUSE");
return 0;
}
```

2.3 Segundo Programa

Vamos estudar alguns comandos básicos.

```
#include <iostream>
using namespace std;
int main()
{
  int a, b, soma;
  cout << "Digite o valor de a: ";
  cin >> a;
  b=2;
  soma = a + b;
  cout << soma << '\n';
  system ("Pause");
}
```

```
Digite o valor de a: 5
7
Press any key to continue...
```

Para o compilador, este segmento nada mais é do que uma sequência de caracteres em um arquivo texto. O primeiro passo da análise é reconhecer que agrupamentos de caracteres têm significado para o programa, por exemplo, saber que `int` é uma palavra-chave da linguagem e que `a` e `b` serão elementos individuais neste programa. Posteriormente, o compilador deve reconhecer que a sequência `int a`, corresponde à declaração de uma variável inteira cujo identificador recebeu o nome `a`.

As regras de formação de elementos e frases válidas de uma linguagem são expressas na gramática da linguagem. O processo de reconhecer os comandos de uma gramática é conhecido como reconhecimento de sentenças.

`cin >> variável;`

Fluxo de entrada de dados para transferência do conteúdo do teclado para a variável. O trabalho com a entrada padrão no *C++* é feito aplicando-se o operador de extração (`>>`) no comando `cin`. Isso precisa ser seguido pela variável que irá guardar o dado que será lido.

Declarando a variável como desejada, então o programa fica na espera por uma entrada do `cin` (teclado) para que possa armazena-la em um espaço reservado da memória. O comando `cin` só pode processar a entrada do teclado depois que a tecla ENTER for pressionada. No nosso exemplo, digitamos o número inteiro `5`, e esse valor foi guardado dentro da variável `a` por meio da sentença: `cin >> a;`

Sendo assim, mesmo que você peça um único caractere, o `cin` não irá processar a entrada até que o usuário pressione ENTER depois que o caractere tenha sido digitado. Deve-se atentar ao tipo da variável que estamos utilizando para guardar o valor extraído pelo `cin`. Se você pedir um inteiro, você receberá um inteiro, se você pedir um caractere, você receberá um caractere, e se você pedir uma `string` de caracteres, você receberá uma `string` de caracteres.

```
cout << soma << '\n';
```

Observe o uso de execução do programa com o uso do caractere `'\n'` é semelhante ao uso do manipulador de saída `endl`.

Caractere Especial	Finalidade
\"	Apresenta o símbolo de aspas no ponto em que é indicado.
\?	Apresenta o símbolo de ponto de interrogação no ponto em que é indicado.
\\	Apresenta o símbolo de barra no ponto em que é indicado.
\'	Apresenta o símbolo de apóstrofo no ponto em que é indicado.
\0	Gera um caractere nulo (zero)
\a	Gera um sinal de áudio (beep) no alto-falante.
\b	Executa um retrocesso de espaço do ponto que é indicado.
\f	Gera um salto de página de formulário (uso de impressora).
\n	Gera uma nova linha a a partir do ponto que é indicado.
\r	Gera um retorno de carro sem avanço de linha.
\t	Gera um espaço de tabulação do ponto que é indicado.
\v	Gera a execução da tabulação vertical.

2.4 Terceiro Programa

Vamos estudar alguns novos comandos básicos.

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

```
Informe seu nome: Jose
Informe seu sobrenome: Silva
Ola
```

```
{
  char NOME[50], SOBRENOME[20];
  cout << "Informe seu nome: ";
  cin.getline(NOME, sizeof(NOME));
  cout << "Informe seu sobrenome: ";
  cin >> SOBRENOME;
  cout << "Ola, \n " << NOME;
  cout << " " << SOBRENOME << endl;
  system ("pause");
  return 0;
}
```

```
 Jose Silva
Press any key to continue...
```

`char NOME[50], SOBRENOME[20];`

Nesta instrução estamos declarando duas variáveis do tipo caractere, esses conceitos serão abordados posteriormente de maneira mais detalhada. Assim, as variáveis `NOME` e `SOBRENOME` apresentam, respectivamente, os tamanhos de 50 e 20, os quais representam o número de caracteres que poderão ser armazenados nas respectivas variáveis. Se tivéssemos declarado `char NOME`, essa variável seria capaz de armazenar apenas um caractere.

`cin.getline(NOME, sizeof(NOME));`

Observe o uso da função `cin.getline()` pertencente ao fluxo de entrada `cin`. Efetua a leitura de caracteres até que seja pressionado a tecla ENTER ou até atingir o valor máximo de caracteres permitidos. Já, a função `sizeof()` retorna o tamanho da variável `NOME`, que no caso é 50, para poder ser utilizado como parâmetro (argumento) da função `cin.getline()`.

3. Variáveis e Tipos Primários de Dados

3.1 Identificadores

Variável é, no sentido de programação, uma **região de memória** de um computador, previamente identificada, que tem por finalidade **armazenar as informações** (dados) de um programa temporariamente. Uma variável armazena **apenas um valor por vez**, sendo considerado **valor** qualquer conteúdo armazenado em uma variável. Um valor está relacionado **ao tipo de dado** de uma variável, podendo ser numérico (inteiro ou real), lógico ou caractere, ou seja, a forma pela qual o computador manipula essas informações (dados).

3.2 Identificadores (nome das variáveis)

Um identificador (nomes das variáveis, ou constantes) válido é uma sequência de uma ou mais letras, dígitos ou underline (_). Nomes de variável podem ser atribuídos com um ou mais caracteres. Caracteres como espaços em branco, pontuações e símbolos especiais **não podem** ser utilizados para formar os identificadores. Identificadores de variáveis **devem sempre ser inicializados por letras**, ou mesmo por um underline (_). Os identificadores **não devem ser iniciados por dígitos** (números).

Outra regra que devemos considerar é que os identificadores não podem coincidir com palavras-chaves da linguagem *C++*, ou palavras específicas do compilador, as quais são reservadas para o seu uso. As palavras-chaves reservadas são:

```
asm, auto, bool, break, case, catch, char, class, const, const_cast,
continue, default, delete, do, double, dynamic_cast, else, enum, explicit, export,
extern, false, float, for, friend, goto, if, inline, int, long, mutable, namespace,
new, operator, private, protected, public, register, reinterpret_cast, return,
short, signed, sizeof, static, static_cast, struct, switch, template, this, throw,
true, try, typedef, typeid, typename, union, unsigned, using, virtual, void,
volatile, wchar_t, while, and, and_eq, bitand, bitor, compl, not, not_eq, or,or_eq,
xor, xor_eq.
```

Outro aspecto bastante importante é que a linguagem *C++* é sensível ao case (*case sensitive*). Isso significa que um identificador escrito em letra maiúscula não é equivalente a outro identificador com o mesmo nome, mas escrito em letras minúsculas. Por exemplo, a variável `RESULTADO`, não é o mesmo que a variável `resultado`, ou `Resultado`. Eles são três diferentes identificadores de variáveis.

Na execução de um programa, as variáveis são armazenadas na memória do computador, entretanto, o computador precisa saber que tipo de dado queremos armazenar nelas, uma vez que não se vai ocupar a mesma quantidade de memória para armazenar um simples número, uma única letra ou um número muito grande, e eles não serão interpretados da mesma maneira.

A memória dos computadores é organizada em bytes. Um byte é a menor quantidade de memória que pode ser manipulada em *C++*. Um byte pode armazenar uma pequena quantidade de dados: uma única letra ou um número inteiro pequeno (entre 0 e 255). Além

disso, o computador também é capaz de manipular tipos de dados complexos, oriundos do agrupamento de vários bytes como números `long` ou não inteiros.

Os tipos fundamentais de dados em *C++*, com o tamanho em bytes são:

Nome	Descrição	Tamanho*	Extensão*
`char`	Caractere ou inteiro pequeno	1 byte	signed: -128 to 127 unsigned: 0 to 255
`short int (short)`	Inteiro curto	2 bytes	signed: -32768 to 32767 unsigned: 0 to 65535
`int`	Inteiro	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`long int (long)`	Inteiro longo	4 bytes	signed: -2147483648 to 2147483647 unsigned: 0 to 4294967295
`bool`	Valor booleano. Pode assumir um de dois valores: verdadeiro ou falso	1 bytes	true ou false
`float`	Número real (ponto flutuante)	4 bytes	± 3.4e±38 (~7 dígitos)
`double`	Número real de dupla precisão	8 bytes	± 1.7e ± 308 (~15 dígitos)
`long double`	Número real de dupla precisão longa	8 bytes	± 1.7e ± 308 (~15 dígitos)

Os valores das colunas Tamanho e Extensão dependem do sistema em que o programa é compilado. Os valores apresentados foram encontrados na maioria dos sistemas de 32 bits. As variáveis numéricas podem ser declaradas como `signed` ou `unsigned` dependendo da extensão numérica necessária a ser armazenada. Variáveis declaradas como `signed` podem representar ambos valores positivos e negativos, enquanto `unsigned` pode representar apenas valores positivos.

3.3 Declaração de variáveis

Apesar de a linguagem C++ permitir a declaração de variáveis em qualquer parte do programa, o bom hábito de programar nos diz que todo dado a ser armazenado na memória de um computador por meio de uma variável deve ser previamente **declarado no início do programa**. Primeiro é necessário saber o seu tipo para depois fazer o armazenamento. Armazenando o dado ele pode ser utilizado e manipulado a qualquer momento durante a execução do programa.

Quando definimos uma variável em *C++*, precisamos informar ao compilador o tipo da variável: um número inteiro, um número de ponto flutuante (real), um caractere, e assim por diante. Essa informação diz ao compilador quanto espaço deve ser reservado na memória para a variável, e o tipo de valor que será armazenado nela. Os tipos fundamentais de variáveis utilizadas são:

int

O tipo de dado `int` (inteiro) serve para armazenar valores numéricos inteiros positivos, ou negativos, e zero (0) excluindo-se qualquer valor fracionário. Existem vários tipos de inteiros, com tamanhos diferentes (dependendo do sistema operacional e/ou arquitetura do processador): `int`, pode possuir 16 bits, 32 bits ou 64 bits `short int`, deve possuir tamanho de no mínimo 16 bits e não pode ser maior que `int`. `long int`, deve possuir tamanho mínimo de 32 bits. `long long int`, deve possuir tamanho mínimo de 64 bits.

```
#include <iostream>
using namespace std;
int main()
{
    int idade;
    cout<< "Digite a sua idade: ";
    cin>> idade;
    cout<< "Sua idade daqui 10 anos sera "
        << idade+10 << " anos" << endl;
    system("Pause");
    return 0;
}
```

```
Digite a sua idade: 20
Sua idade daqui 10 anos sera 30 anos
Press any key to continue...
```

OBS.: No programa abaixo declararemos a variável fim como inteiro, e em seguida, vamos imprimir na tela o seu valor.

```
#include <iostream>
using namespace std;
int main()
{
    int fim;
    cout << fim <<endl;
    system("Pause");
    return 0;
}
```

```
4215734
Press any key to continue...
```

Portanto, quanto não se atribuí valores à variável, no momento de sua declaração, será atribuído a ela a informação que estiver na memória do computador. Dependendo do compilador utilizado, o valor zero (0) pode ser atribuído, entretanto, a forma de execução é diferente para cada compilador, dever-se-á, portanto, inicializar a variável:

```
#include <iostream>
using namespace std;
int main()
{
    int fim=0;
    cout << fim <<endl;
    system("Pause");
    return 0;
}
```

```
0
Press any key to continue...
```

float

O tipo de dado float serve para armazenar números reais (de ponto flutuante), ou seja, com casas decimais, fracionários. São tipos reais de dados numéricos positivos, negativos e

números fracionários e inteiros. O float é usado para definir número de ponto flutuante com seis dígitos decimais de precisão absoluta (32 bits).

```
#include <iostream>
using namespace std;
int main()
{
    float a, b, d;
    a=5;
    b=3;
    d=a/b;
    cout << d <<endl;
    system("Pause");
    return 0;
}
```

```
1.66667
Press any key to continue...
```

OBS.: Nesse momento, devemos chamar a atenção para a conversão de tipos de variáveis, observe o exemplo, onde declararemos as variáveis `a`, `b`, e `d` como inteiros:

```
#include <iostream>
using namespace std;
int main()
{
    int a, b, d;
    a=10;
    b=3;
    d=a/b;
    cout << d <<endl;
    float c;
    c=a/b;
    cout << d << endl;
    system("Pause");
    return 0;
}
```

```
3
3
Press any key to continue...
```

Como `d` é inteiro, o resultado é arredondado para o valor `3`. Seguindo o fluxo de execução do programa, declaramos a variável `c` como ponto flutuante, atribuímos a ela o resultado da expressão `a/b` e imprimimos o seu resultado na tela do computador. Observe que procedimento também não resolveu o nosso problema de conversão, o valor continua sendo `3`. Para resolvermos esse problema, devemos declarar `c` como `float` e no momento da atribuição, utilizarmos o comando `(float)`, como apresentado abaixo:

```
#include <iostream>
using namespace std;
int main()
{
    int a, b;
    float c;
```

```
3.33333
Press any key to continue...
```

```
    a=10;
    b=3;
    c=(float)a/b;
    cout << c << endl;
    system("Pause");
    return 0;
}
```

double

O tipo de dado **double** serve para armazenar números de ponto flutuante de dupla precisão, normalmente tem o dobro do tamanho e capacidade do **float**.

```
#include <iostream>
using namespace std;
int main()
{
    double a, b, d;
    a=5;
    b=3;
    d=a/b;
    cout << d << endl;
    system("Pause");
    return 0;
}
```

```
1.66667
Press any key to continue...
```

Utilizando a variável `double`, teremos um número mais significativo de saída. O **double** é utilizado quando necessitamos de uma aproximação maior e mais exata do resultado. O **double** consome 64 bits de memória para o seu armazenamento. Esse consumo é explicado pela alta precisão de seu número (cerca de 15 decimais de precisão absoluta). O **long double** é usado para definir número de ponto flutuante (números fracionários com quinze dígitos decimais de precisão absoluta) de 80 bits.

char

O tipo char ocupa 1 byte, e serve para armazenar caracteres ou inteiros. Isso significa que o programa reserva um espaço de 8 bits na memória RAM ou em registradores do processador para armazenar um valor (**char** de tamanho maior que 8 bits é permitido pela linguagem, mas os casos são raros).

```
#include <iostream>
using namespace std;
int main()
{
    char nome[8];
    cout << "Digite seu nome: ";cin >> nome;
    cout << "Seu nome e: " << nome << endl;
    system("Pause");
    return 0;
}
```

```
Digite seu nome: Marcos
Seu nome e: Marcos
Press any key to continue...
```

No nosso exemplo, pode-se guardar uma variável do tipo caractere com até 20 caracteres de comprimento, caso exceda o número máximo de caracteres, ocasionará erro no compilador. Caso fosse declarado apenas "`char nome`", a variável `nome` armazenaria apenas uma letra, por exemplo, se o usuário digitasse um nome com 7 letras, a variável

apenas armazenaria a primeira letra digitada. A linguagem de programação C++ utiliza de duas formas: Utilizando apenas um caractere delimitado por apóstrofos 'e'; Ou uma sequência de caracteres delimitador por aspas "ee";

string

São variáveis tipo caracteres que podem armazenar valores maiores que um caractere simples. A biblioteca `<string>` fornece esse suporte. Não é tipo fundamental de variável, mas se comporta como um tipo fundamental. Para declararmos variáveis como `string` precisamos incluir uma linha adicional no cabeçalho do programa, o arquivo `#include <string>`.

```
#include <iostream>
#include <string>
using namespace std;
int main()
{
    string nome;
    nome = "Marcos da Silva";
    cout << nome << endl;
    nome = "Podemos mudar uma string";
    cout << nome << endl;
    system("Pause");
}
```

```
Marcos da Silva
Podemos mudar uma string
Press any key to continue...
```

bool

Os tipos de dados **lógicos** (tipo booleano) possuem apenas um de dois valores: **true** (verdadeiro = 1) e false (falso = 0) e são referenciados pela palavra-chave **bool**. Eles são utilizados para verificar o resultado de uma notificação ou expressão lógica.

```
#include <iostream>
using namespace std;
int main()
{
    bool t1, t2, t3;
    t1 = (4>=6);//falso
    t2 = true;//verdadeiro
    t3 = (true || 5==25/5);//verdadeiro
    cout << t1 << endl;
    cout << t2 << endl;
    cout << t3 << endl;
    system("Pause");
}
```

```
0
1
1
Press any key to continue...
```

4. Constantes

Constante é tudo aquilo que é fixo ou estável. No processo de programação existirá vários momentos em que este conceito deverá ser utilizado. Por exemplo, o valor `1.23` da fórmula seguinte é uma constante explícita:

```
RESULTADO = ENTRADA * 1.23
```

É necessário para programar, ter conhecimento de constantes e seus respectivos significados, praticamente todos os programas fazem uso delas. São indispensáveis para a criação de programas de forma rápida eficiente. Assim, as constantes são expressões com valor fixo. Em *C++* o modificador chamado `const` tem comportamento variado dependendo do local onde está sendo declarado. Sua função, basicamente, é estabelecer um vínculo entre declaração e obrigatoriedade da coerência no uso do símbolo declarado. A princípio, quando declaramos uma constante fazemos com que seja obrigatório o uso do símbolo de forma que o mesmo não possa ter seu valor alterado. Assim, se fizermos:

```
const int x = 4;
```

O inteiro `x` não poderá deixar de ter valor igual a `4`. Qualquer tentativa de modificar o valor da constante ao longo do programa será reportada como erro pelo compilador.

4.1 Definindo Constantes

Podemos definir os nomes de nossas constantes sem o consumo de memória, simplesmente utilizando o comando `#define` do pré-processador.

```
#define identificador value
```

```
#include <iostream>
#define PI 3.14159
#define NL '\n'
using namespace std;
int main()
{
    double raio=5, circulo;
    circulo=2*PI*raio;
    cout << circulo; cout << NL;cout << NL;
    system("Pause");
}
```

```
31.4159

Press any key to continue...
```

4.2 Classificação de Constantes e Variáveis

Tanto variáveis como constantes podem ser locais, ou globais. Uma variável global é aquela declarada no corpo principal do código fonte do programa, fora de todas as funções, enquanto uma variável local é aquela declarada dentro do corpo de uma função ou de um bloco. Assim, temos que:

Locais
Podem ser usadas apenas na função onde foi declarada, sendo inutilizáveis para as demais funções.

Globais

Podem ser usadas **em qualquer lugar do programa.**

Observe o exemplo:

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

int Inteiro;
char aNome;
char string[20];
unsigned int nFilhos;

int main()
{
    unsigned short idade;
    float aNumero, outraVar;
    cout << "Entre com a sua idade: ";
    cin >> idade;
    ...
}
```

Variáveis Globais

Variáveis Locais

Instruções

5. Expressões

Tanto variáveis como constantes podem ser utilizadas na elaboração de **cálculos matemáticos** com a utilização de operadores aritméticos. É muito comum a necessidade de trabalhar com expressões aritméticas, uma vez que, na sua maioria, todo trabalho computacional está relacionado e envolve a utilização de cálculos matemáticos. Essas expressões são definidas pelo relacionamento existente entre variáveis e constantes numéricas com a utilização dos operadores aritméticos.

Considere a fórmula:

$$AREA = \pi * RAIO^2$$

para o cálculo da área de uma circunferência, onde:
AREA e RAIO são variáveis.
π é uma constante (pi = 3.14159)
Multiplicação e a potência são os operadores aritméticos.
Neste caso, a expressão a ser escrita para a execução em um computador é:

```
AREA = 3.14159 * RAIO * RAIO
```

ou

```
AREA = 3.14159 * pow(RAIO,2)
```

Assim, temos que as expressões abaixo

$$AREA = \frac{BASE \times ALTURA}{2}$$

deverá ser escrita como `AREA=(BASE*ALTURA)/2`

Outro exemplo:

$$X = 0.8 + \left\{ 43 \times \left[\frac{55}{(30+20)} \right] \right\}$$

```
X=0.8+(43*(55/(30+2)))
```

5.1 Instruções

As instruções são representadas pelo conjunto de palavras-chave (vocabulário) de uma linguagem de programação que tem por finalidade comandar, em um computador, o seu funcionamento (fluxo do programa) e a forma como os dados armazenados devem ser tratados. As instruções da linguagem de programação C++ caracterizam-se por serem na

sua maioria funções internas, e devem ser escritas com letras minúsculas, e terminadas com ponto e vírgula " ; ".

5.2 Atribuição

Um comando de atribuição permite-nos fornecer um valor a uma variável, em que o tipo de dado deve ser compatível com o tipo da variável, isto é, somente podemos atribuir um valor lógico a uma variável capaz de comportá-lo. Em *C++* o sinal de igual "=" é o símbolo de atribuição, por meio dele armazenamos valores ou resultados de expressões nas variáveis anteriormente declaradas com o mesmo tipo primário do dado que estamos atribuindo a ela.

Exemplo:

```
A = 5;
```

Esta instrução atribuí o valor inteiro `5` à variável `A`. A parte a esquerda do sinal de igual (=) deve sempre ser uma variável, enquanto a parte à direita do sinal de atribuição pode ser uma constante, uma variável, um resultado de uma operação, ou qualquer combinação destes. A regra mais importante é que a atribuição é realizada da direita-para-a-esquerda, e nunca no outro sentido.

```
5 = A;
```

Não faz sentido essa atribuição, então, o programa retorna um erro.

5.3 Expressões Aritméticas

Expressões aritméticas são aquelas cujos operadores são ***aritméticos*** e cujos operandos são constantes, ou variáveis do tipo numérico (inteiro ou real). Chamamos de operadores ***aritméticos*** o conjunto de símbolos que representa as operações básicas da matemática, em *C++* os principais operadores são:

Operator	Função	Exemplos
+	Adição	2 + 3, X + Y
-	Subtração	4 - 2, N - M
*	Multiplicação	3 * 4, A * B
/	Divisão	10/2, X1/X2
++	Incremento	i++ equivale a i=i+1
--	Decremento	i-- equivale a i=i-1

Demais operações estão disponíveis em C++, por exemplo, as operações de radiciação e potenciação, observe a tabela:

Operator	Função	Exemplos
pow(x,y)	x elevado a y = x^y	pow(2,3)
sqrt(x)	Raiz quadrada de x	sqrt(9)

Tais função podem ser utilizadas desde que adicionado ao cabeçalho do programa a biblioteca `<math.h>`, a qual apresenta várias funções matemáticas e trigonométricas, dentre elas temos também:

`sin()`: Retorna o valor do seno. Recebe como parâmetro o valor dos graus em `double`.
`cos()`: Retorna o valor do cosseno. Recebe como parâmetro o valor dos graus em `double`.
`tan()`: Retorna o valor da tangente. Recebe como parâmetro o valor dos graus em `double`.
`log()`: Retorna o valor do logaritmo na base 2. Exige um parâmetro do tipo `double`.
`log10()`: Retorna o valor do logaritmo na base 10. Exige um parâmetro do tipo `double`.

Usaremos outras operações matemáticas não-convencionais, porém, muito úteis na construção de programas, que são o resto e o quociente de divisão inteira:

Operator	Função	Exemplos
`%`	`Resto da divisão`	`9 % 4 resulta em 1` `27 % 5 resulta em 2`
`int(x/y)`	`Quociente da divisão`	`int(9/4) resulta em 2` `int(27/5) resulta em 5`

No caso do quociente da divisão, se as variáveis `x` e `y` forem reais (`float` ou `double`) será necessário utilizar a função `int(x/y)` para obtenção do quociente da divisão, caso contrário, se as variáveis forem inteiros (`int` ou `long`), o resultado da expressão `x/y` será o quociente da divisão.

Na resolução das expressões aritméticas, as operações guardam uma hierarquia entre si:

Prioridade	Operadores
1ª	Parênteses mais internos
2ª	Potenciação, radiciação e funções trigonométricas
3ª	`*  /  int()  %`
4ª	`+  -`

Em caso de empate (operadores de mesma prioridade), a resolução ocorre da esquerda para a direita, conforme a sequência existente na expressão trigonométrica. Para alterar a prioridade, utilizamos os parênteses mais internos.

5.4 Exemplo 1

Desenvolva um programa que receba a massa e o volume de uma amostra qualquer e calcule sua densidade.

```
#include <iostream>
using namespace std;
int main()
{
      float massa, vol, den;
```

```
        cout << "Digite a massa do material: ";
        cin >> massa;
        cout << "Digite o volume do material: ";
        cin >> vol;
        den = massa/vol;
        cout << "A densidade do material sera " << den << endl;
        system("Pause");
}
```

5.5 Exemplo 2

Construa um programa que calcule a média aritmética entre quatro notas bimestrais fornecidas por um aluno (usuário):

```
#include <iostream>
using namespace std;
int main()
{
    float N1, N2, N3, N4, media;
    cout << "Nota do Primeiro Bimestre: "; cin >> N1;
    cout << "Nota do Segundo Bimestre: "; cin >> N2;
    cout << "Nota do Terceiro Bimestre: "; cin >> N3;
    cout << "Nota do Quarto Bimestre: "; cin >> N4;
    media = (N1+N2+N3+N4)/4;
    cout << "Media final= " << media << endl;
    system("Pause");
}
```

5.6 Exemplo 3

Construa um algoritmo que, tendo como dados de entrada as coordenadas de dois pontos quaisquer do plano P(x1, y1) e Q(x2, y2), imprima a distância (d) entre eles.

Lembre-se que:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

```
#include <iostream>
#include <math.h>
using namespace std;
int main()
{
    float x1, y1, x2, y2,d;
    cout << "Digite X do ponto P: "; cin >> x1;
    cout << "Digite Y do ponto P: "; cin >> y1;
    cout << "Digite X do ponto Q: "; cin >> x2;
    cout << "Digite Y do ponto Q: "; cin >> y2;
    d=sqrt(pow((x2-x1),2)+pow((y2-y1),2));
    cout << "A ditancia entre pontos = " << d << endl;
    system("Pause");
}
```

5.7 Expressões Lógicas

Denominamos expressão lógica aquela cujos operadores são ***lógicos*** ou ***relacionais*** e cujos operandos são relações ou variáveis do tipo lógico.

Utilizamos os operadores ***relacionais*** para realizar comparações entre dois valores de mesmo tipo primitivo. Tais valores são representados por constantes, variáveis, ou expressões aritméticas. Os operadores ***relacionais*** são comuns para construirmos as expressões. Em *C++* são eles:

Operator Relacional	Função	Exemplos
>	Maior que	5 > 4, x > y
>=	Maior ou igual a	5 >= 3, X >= y
<	Menor que	3 < 6, x < y
<=	Menor ou igual a	3 <= 5, x <= y
==	Igual a	3==3, x == y
!=	Não igual (diferente)	8 != 9, x != y

O resultado obtido de uma relação sempre é um valor ***lógico***. Por exemplo, analisando a relação numérica `A + B == C`, o resultado será verdadeiro (`1`) ou falso (`0`) à medida que o valor da expressão aritmética `A + B` seja igual ou diferente do conteúdo da vaiável `C`, respectivamente.

Utilizamos três operadores ***lógicos*** básicos para a formação de novas proposições lógicas compostas a partir de outras proposições lógicas simples. Os operadores ***lógicos*** em C++ são apresentados na tabela abaixo.

Operator Lógico	Função	Exemplos
!	Não (negação)	! (3 > 5)
&&	E (conjunção)	3 > 5 && 4 < 8
||	Ou (disjunção)	3 > 5 || 4 < 8

5.8 Tabela-verdade

Tabela-verdade é o conjunto de todas as possibilidades combinatórias entre os valores de diversas variáveis lógicas, as quais se encontram em apenas duas situações Verdadeiro (**`true`** `equivale a 1`) ou Falso (**`false`** `equivale a 0`), e um conjunto de operadores lógicos. Observe o programa abaixo:

```
#include <iostream>
using namespace std;
int main()
{
    cout << !(5 == 5) << endl;
    cout << !(6 <= 5) << endl;
    cout << !true << endl;
    cout << !false << endl;
    system("pause");
}
```

```
0
1
0
1
Press any key to continue...
```

```
cout << !(5 == 5) << endl; // retorna falso (0), pois 5 == 5 é verdadeiro
cout << !(6 <= 5) << endl; //retorna verdadeiro (1), pois 6<= 5 é falso
cout << !true << endl; //retorna falso, ou seja, 0
```

```
cout << !false << endl;//retorna verdadeiro, ou seja, 1
```

Tabela verdade para a operação de conjunção (E), dado duas variáveis lógicas A e B:

A	B	Exemplos
true	true	true
true	false	false
false	true	false
false	false	false

Operação de disjunção ou não-exclusão (OU), dado duas variáveis lógicas A e B:

A	B	Exemplos
true	true	true
true	false	true
false	true	true
false	false	false

6. Formatando as saídas dos programas

O comando `cout` permite estabelecer o tamanho de um campo para a impressão. Isso significa que podemos definir o número de colunas que será ocupado por um valor ou texto a ser impresso. Em geral, a definição de tamanho de campo é usada para alinhamento e estética de um relatório.

6.1 Exemplo 4

Faça um programa que calcule o salário líquido de um profissional que trabalhe por hora. Para tanto, é necessário possuir alguns dados básicos, tais como valor da hora de trabalho, número de horas trabalhadas no mês e o percentual de desconto do INSS, para que seja possível apresentar os resultados do valor do salário bruto, valor de desconto e o valor do salário líquido.

Após o programa escrito, Entre com: `10.33`, `12.5` e `6.5` e a saída deve ser: `129.125`, `8.39312` e `120.732`.

```
# include <iostream>
using namespace std;
int main(void)
{
    float HT, VH, PD, TD, SB, SL;
    cout << "Entre com a quantidade de horas trabalhadas.:  "; cin >> HT;
    cout << "Entre com o valor da hora de trabalho.......:  "; cin >> VH;
    cout << "Entre com o valor do percentual de desconto..:  "; cin >> PD;
    SB = HT * VH;
    TD = (PD / 100) * SB;
    SL = SB - TD;
    cout << "O Salario Bruto....: " << SB << endl;
```

```
    cout << "Desconto.................: " << TD << endl;
    cout << "O Salario Liquido.: " << SL << endl;
    system ("pause");
    return 0;
}
```

Saída do programa:

```
Entre com a quantidade de horas trabalhadas.:  10.33
Entre com o valor da hora de trabalho.............:  12.5
Entre com o valor do percentual de desconto..:  6.5
O Salario Bruto....: 129.125
Desconto.............: 8.39312
O Salario Liquido.: 120.732
Press any key to continue . . .
```

Apesar dos resultados do programa estarem corretos, eles estão sem nenhum formato. Para utilizar esses recursos, é necessário fazer a inclusão do arquivo de cabeçalho `<iomanip>`.

Para definir a quantidade de casas decimais a ser apresentada em um fluxo de saída após o ponto decimal, deve-se utilizar o manipulador `setprecision()`, o qual possui como parâmetro o número de casas decimais a ser definido.

Para fixar a apresentação do ponto decimal (valor decimal), mesmo quando o resultado for um valor de formato inteiro, deve-se fazer o uso do manipulador `setiosflag()`, o qual deve-se utilizar como parâmetro o sinalizador de formato (`flag`) `ios:fixed`, que direciona a saída para apresentar o ponto decimal.

Parâmetros do manipulador `setiosflag()`:

Sinalizador	Finalidade
ios::dec	Utilizado para formatar saída de valor em formato decimal
ios::fixed	Utilizado para apresentar um valor numérico em formato decimal
ios::hex	Utilizado para formatar saída de valor em formato hexadecimal
ios::left	Utilizado para apresentar uma saída justificada à esquerda.
ios::oct	Utilizado para formatar saída de valor em formato octal
ios::right	Utilizado para apresentar uma saída justificada à direita
ios::scientific	Utilizado para apresentar uma saída de valor numérico em formato científico.
ios::showbase	Utilizado para formatar a apresentação de valores octais e hexadecimais.
ios::showpoint	Utilizado para forçar a apresentação de um ponto decimal em um valor real.
ios::showpos	Utilizado para apresentar o sinal unário de um valor numérico (positivo/negativo)
ios::uppercase	Utilizado para apresentar caractere em formato maiúsculo.

`setiosflags` – acionar diversos recursos de sinalização de formatos.
`resetiosflags` – desativar o recurso.

Assim, os manipuladores de tamanho são:

`setw:` Seleciona o tamanho de próximo campo a ser impresso.
`setprecision:` Define o número de casas decimais a ser impressas para um número em ponto flutuante.
`setfill:` Seleciona o caractere que deverá preencher as colunas em branco iniciais de um campo.
`setiosflags:` Seleciona o modo de apresentação de um número (com ponto decimal, notação científica etc.).

6.2 Exemplo 5

Vamos reescrever o programa anterior definindo o formato da saído do programa para duas casas decimais. Entre com : 10.33, 12.5 e 6.5 e a saída deve ser: 129.13, 8.39 e 120.73.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(void)
{
 float HT, VH, PD, TD, SB, SL;
 cout << setprecision(2); //Fixa em duas casas decimais
 cout << setiosflags(ios::right); //Tabulação a partir da direita
 cout << setiosflags(ios::fixed); //Estabelece a apresentação do ponto decimal
 cout << "Entre com a quantidade de horas trabalhadas.:  "; cin >> HT;
 cout << "Entre com o valor da hora de trabalho.............:  "; cin >> VH;
 cout << "Entre com o valor do percentual de desconto..:  "; cin >> PD;
 SB = HT * VH;
 TD = (PD / 100) * SB;
 SL = SB - TD;
 cout << "O Salario Bruto....: " << setw(8) << SB << endl;
 cout << "Desconto.............: " << setw(8) << TD << endl;
 cout << "O Salario Liquido.: " << setw(8) << SL << endl;
 system ("pause");
 return 0;
}
```

Saída do programa:

```
Entre com a quantidade de horas trabalhadas.:  10.33
Entre com o valor da hora de trabalho.............:  12.5
Entre com o valor do percentual de desconto..:  6.5
O Salario Bruto....:   129.13
Desconto.............:     8.39
O Salario Liquido.:   120.73
Press any key to continue . . .
```

Compare a saída atual com a saída do programa escrito no exemplo anterior. A definição da largura do campo é realizada pela função `setw(8)`. Observe o programa abaixo que exemplifica a função `setw()`.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(void)
{
 int lapis=45, borrachas=2354, cadernos=8;
 cout << "Lapis      " << lapis << endl;
 cout << "Borrachas  " << borrachas << endl;
 cout << "Cadernos   " << cadernos << endl;
 system ("pause");
 return 0;
}
```

```
Lapis       45
Borrachas   2354
Cadernos    8
Press any key to continue...
```

Observer agora a ação da função `setw()`.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(void)
{
 int lapis=45, borrachas=2354, cadernos=8;
 cout << "Lapis      " << setw(12)
                          << lapis << endl;
 cout << "Borrachas  " << setw(12)
                       << borrachas << endl;
 cout << "Cadernos   " << setw(12)
                        << cadernos << endl;
 system ("pause");
 return 0;
}
```

```
Lapis                 45
Borrachas           2354
Cadernos               8
Press any key to continue...
```

Observer agora a ação da função `setfill()`.

```
#include <iostream>
#include <iomanip>
using namespace std;
int main(void)
{
 int lapis=45, borrachas=2354, cadernos=8;
 cout << setfill('.');
 cout << "Lapis      " << setw(12)
                          << lapis << endl;
 cout << "Borrachas  " << setw(12)
                       << borrachas << endl;
 cout << "Cadernos   " << setw(12)
                        << cadernos << endl;
 system ("pause");
 return 0;
}
```

```
Lapis      ..........45
Borrachas  ........2354
Cadernos   ...........8
Press any key to continue...
```

6.3 Precisão Numérica

Diferença entre os tipos de dados com ponto flutuante. **float** utiliza seis dígitos de precisão absoluta e os tipos **double** e **long double** utilizam quinze.

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

```
2.33333
2.333333333333333
2.333333333333333
```

```
int main(void)
{
    float P1 = 7.0/3.0;
    double P2 = 7.0/3.0;
    long double  P3 = 7.0/3.0;
    float R1 = 10.0/9.0;
    double R2 = 10.0/9.0;
    long double R3 = 10.0/9.0;
    cout << setprecision(6);
    cout << P1 << endl;
    cout << setprecision(15);
    cout << P2 << endl;
    cout << P3 << endl;
    cout << endl;
    cout << setprecision(6);
    cout << R1 << endl;
    cout << setprecision(15);
    cout << R2 << endl;
    cout << R3 << endl;
    system ("pause");
    return 0;
}
```

```
1.11111
1.111111111111111
1.111111111111111
Press any key to continue...
```

6.4 Imprimindo Caracteres Gráficos

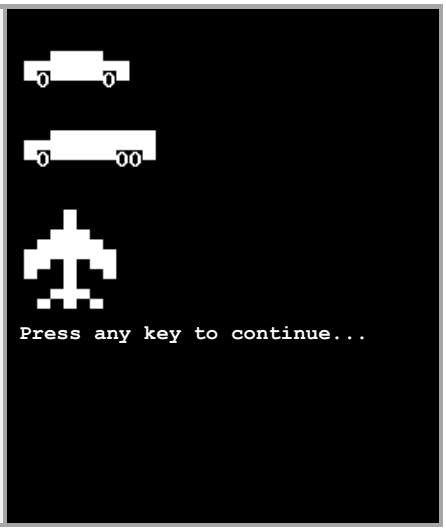
Todo o caractere (letra, dígito, pontuação etc.) é representado no computador por um número, o código ASCII dispões de números de 0 a 127 abrangendo letras, dígitos entre 0 a 9, caracteres de pontuação e caracteres de controle como salto de linha, tabulação etc. Os computadores usam 129 caracteres adicionais, com os códigos ASCII de 128 a 255, que consistem em símbolos de línguas estrangeiras e caracteres gráficos.

```
\xdd
```

Onde `dd` representa o código do caractere na base hexadecimal.

Observe o programa a seguir:

```
#include <iostream>
using namespace std;
int main()
{
 cout << "\n\n";
 //impressão do carro
 cout << "\n\t\xDC\xDC\xDB\xDB\xDB\xDB\xDC\xDC";
 cout << "\n\t\xDFO\xDF\xDF\xDF\xDFO\xDF";
 cout <<"\n\n";
 //impressão da caminhonete
 cout << "\n\t\xDC\xDC\xDB\xDB\xDB\xDB\xDB\xDB\xDB\xDB";
 cout << "\n\t\xDFO\xDF\xDF\xDF\xDF\xDFOO\xDF";
 cout <<"\n\n";
//impressão do avião
 cout << "\n\t"<<"   "<<"\xDB";
 cout << "\n\t"<<" "<<"\xDC\xDB\xDB\xDB\xDC";
 cout << "\n\t\xDB\xDB\xDF\xDB\xDF\xDB\xDB";
 cout << "\n\t\xDF"<<"  "<<"\xDB"<<"  "<<"\xDF";
 cout << "\n\t"<<" "<<"\xDC\xDF\xDB\xDF\xDC";
```

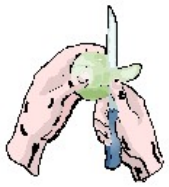


```
 cout <<"\n\n";
 system("Pause");
 return 0;
}
```

7. Controle de Fluxo de Execução - Estrutura Condicional SE

Também conhecidos como ***comando de desvio***, a estrutura condicional SE permite que o programa decida autonomamente entre dois caminhos possíveis, qual irá executar.

```
Traga a cesta com as batatas
Se a roupa é clara então
   coloque avental
Fim se
Descasque as batatas
```

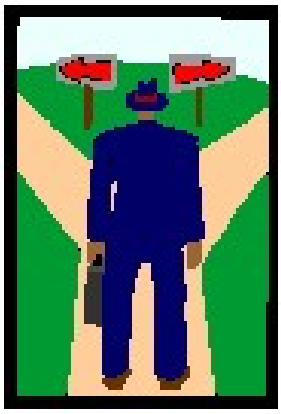


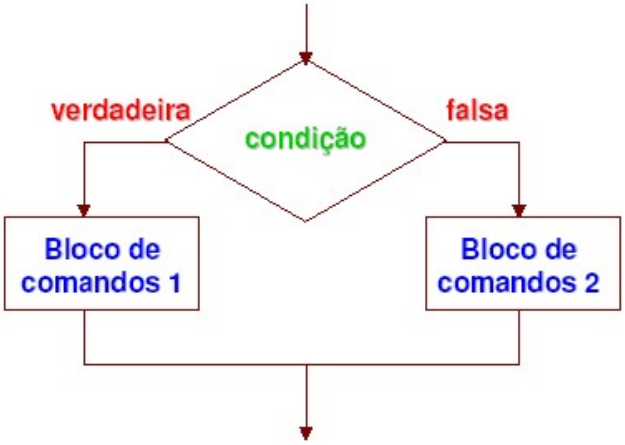
Os métodos de tomada de decisão no C++, estão presentes para as tarefas mais corriqueiras que o programa deve executar. Além desta forma de controle de decisões, C++ provê certas funcionalidades relacionadas a objetos que modificam a forma como o código é estruturado e, por consequência, decidem como o programa deve se comportar em determinadas situações. Examinemos os métodos básicos e analisemos de forma simples as estruturas de decisão, para entender como isso poderá nos ajudar a tornar o código mais bem construído.

Neste contexto, precisamos definir os chamados ***Blocos de Comandos***, os comandos na nossa linguagem são escritos um por linhas, já um ***bloco de comandos*** é uma série de comandos, portanto, em um ***bloco*** ou todos os comandos são executados ou nenhum é executado. Em C++ os blocos de comandos são definidos entre chaves (`{ }`). Esse conceito é fundamental para o desenvolvimento de programas em C++, uma vez que o fluxo de execução de um programa muitas vezes se divide onde somente um dos blocos de comando é executado.

7.1 Estrutura if/else

Uma ação muito importante que o processador de qualquer computador executa, e que o torna diferente de qualquer outra máquina, é a tomada de decisão definindo o que é verdadeiro e o que é falso. Se quisermos fazer um bom programa, esse programa deve ser capaz de definir caminhos diferentes de acordo com decisões que o próprio programa toma. Para isso, precisamos de uma estrutura seletiva da qual o único valor possível é o bit 1 ou 0, resumindo: retornar o valor VERDADEIRO (`true`) ou FALSO (`false`).





Forma Geral do comando SE em pseudocódigos pode ser definida como:

```
se <expressão booleana> então
    bloco de comandos 1
senão
    bloco de comandos 2
fim se
```

Caso o bloco de comandos depois do senão seja vazio, esta parte pode ser omitida. A forma geral simplificada é, portanto:

```
se <expressão booleana> então
   bloco de comandos
fim se
```

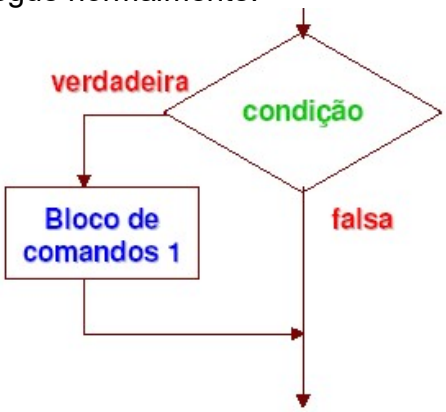
Em muitas linguagens de programação, quem faz isso é o "`if`", ou seja, se uma dada condição for verdadeira, faça tal ação. O "`else`" é o que chamamos de caso contrário, ou seja, se for falso execute o que está no ELSE.

A Ideia é selecionaremos apenas uma ação ou um único conjunto de ações (bloco de comandos), não podendo realizar 2 condições diferentes. Assim, a estrutura de seleção permite a escolha de um grupo de ações (bloco) a ser executado quando determinadas condições, representadas por expressões lógicas e/ou relacionais, são satisfeitas ou não. Os tipos de seleção são: Simples, Composta, Encadeada e Múltipla Escolha.

Seleção Simples: Quando precisamos testar 01 condição antes de executar uma ação. Ocorre somente se a condição for VERDADEIRA.

```
if (condição)
{
Declaração;
}
```

Neste caso o bloco de comandos é executado somente se a condição for verdadeira, caso contrário o algoritmo prossegue normalmente.



Seleção Composta: Utilizadas em situações em que duas alternativas dependem de uma mesma condição, uma da condição VERDADEIRA, e outra da condição FALSA.

```
if (condição)
{
  declaração 1;
}
  else
  {
  declaração 2;
  }
```

Exemplo de Comando SE.

```
início
   acorde
   se estiver fazendo sol então
      vai à praia
   senão
      lê jornal
      dorme
      acorda
   fim se
   almoça
fim
```

Sequência de ações:

- Se estiver fazendo sol
  - acorda
  - vai à praia
  - almoça

- Senão
  - acorda
  - lê jornal
  - dorme
  - acorda
  - almoça

Exemplo de Algoritmos

```
início
  imprima 'Primeiro número?'
  leia num1
  imprima 'Segundo número?'
  leia num2
  se num1 > num2 então
    imprima 'O maior é ', num1
  senão
    imprima 'O maior é ', num2
  fim se
fim
```

Comandos Aninhados: As duas formas do comando SE podem aparecer dentro de outros comandos SE. Diz-se que o comando SE interno está ***aninhado*** no comando SE externo.

```
if(condição)
{
  if(condição)
  {
  declaração 1;
  }
}
```

```
se estiver sol então
  se eu tiver dinheiro então
      Vou à praia
  fim se
fim se
```

O fim se estará relacionado com o SE mais próximo.

Observe as seguintes construções:

```
se estiver sol então
     se eu tiver dinheiro então
          vou à praia
     fim se
fim se
```

ou

```
se estiver sol e
          eu tiver dinheiro
então
     vou à praia
fim se
```

As duas construções são equivalentes. Então, quando usar ***comandos aninhados***?

A resposta a essa pergunta é simples, pois utilizamos os ***comandos aninhados*** quando tivermos que executar blocos de comandos diferentes para a cláusula ***senão*** (`else`) das duas condições, observe o exemplo:

```
se estiver sol então
   se eu tiver dinheiro  então
        Vou à Fazenda Felicidade
   senão
        Vou à praia
   fim se
senão
   Vou dormir
fim se
```

Como construir sem aninhamento

Exemplo anterior

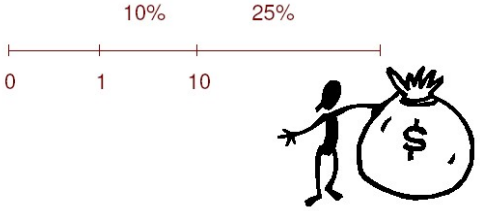
```
se estiver sol então
   se eu tiver dinheiro  então
        Vou à Fazenda Felicidade
   senão
        Vou à praia
   fim se
senão
   Vou dormir
fim se
```

Sem aninhamento

```
se estiver sol e tiver dinheiro então
  Vou à Fazenda Felicidade
fim se

se estiver sol e não tiver dinheiro
  então Vou à praia
fim se

se não estiver sol então
     Vou dormir
fim se
```

7.2 Exemplo 6

Uma empresa vai dar um abono aos empregados que tenham mais de 1 ano de casa:10% para os que tenham menos de 10 anos e 25% para os demais. Calcule o abono de um dado funcionário, dado o seu salário e o tempo de casa.



```
início
  imprima 'Entre com o salario: '
  leia salario
  imprima 'Entre com o tempo de servico:'
  leia tempo
  se tempo > 1 então
     se tempo < 10 então
        salario ← 1.1*salario
     senão
        salario ← 1.25*salario
     fim se
  fim se
  imprima 'Salário com abono: ', salario
fim
```

O algoritmo acima em C++ será:

```
#include <iostream>
using namespace std;
int main(void)
{
 float tempo, salario;
 cout << "Entre com o salario: ";
 cin >> salario;
 cout << "Entre com o tempo de servico: ";
 cin >> tempo;
 if(tempo>1)
    if(tempo <10)
      salario=1.1*salario;
    else
      salario=1.25*salario;
 cout << " Salario com abono = "<< salario << endl;
 system ("pause");
 return 0;
}
```

Seleção Composta: É o agrupamento de várias seleções, ocorre quanto uma determinada ação, ou bloco deve ser executado se um grande conjunto de possibilidades ou combinações de situações for satisfeito.

```
if(condição 1)
{
  declaração1;
}
  else if(condição 2)
  {
  declaração 2;
  }
...
  else if(condição (N-1))
  {
    declaração (N-1)
  }
else
{
declaração N;
}
```

Observe que podemos criar um programa com quantas condições desejarmos, restringindo a cada condição, uma ação ou um conjunto de ações. Um dos erros mais comuns é criar condições redundantes, como mostra o seguinte exemplo:

Elabore um programa que diga qual ação, ou ações foram escolhidas:
a) Ação 1: caso o número seja maior ou igual a 2.
b) Ação 2: caso o número seja maior que 1.
c) Ação 3: caso não seja satisfeita nenhuma condição.

```
cin>>a;
if (a>=2)
{
cout<<"Ação 1 escolhida"<<endl;
}
else if (a>1)
{
cout<<" Ação 2 escolhida"<<endl;
}
else
{
cout<<" Ação 3 escolhida"<<endl;
}
```

Observe que o erro encontra-se no uso do "`else if`", com ele, excluímos possibilidades possíveis de respostas. Por exemplo, se a variável `a=3`, o compilador nos dará como saída apenas a primeira condição ("`Ação 1 escolhida`"), onde na verdade temos duas respostas, pois satisfaz a ação 1 e 2 simultaneamente.

Se substituirmos o `if` no lugar do `else if`, o compilador nos dará as 2 respostas possíveis ("`Ação 1 escolhida`" e "`Ação 2 escolhida`"), com isso, corrigiríamos o problema da redundância do nosso exemplo. Com o uso apenas do `if` e do `else` é possível que compilador execute várias condições que ocorram simultaneamente.

7.3 Estrutura switch/ Escolha (caso selecione)

Textualmente conhecida como ***alternativa de múltiplas escolhas***, é uma alternativa para os SES aninhados, deixando o algoritmo com uma estrutura melhor.

Sintaxe:

```
escolha (expressão)
{
caso <rótulo 1> : comando 1;
                  comando 2;
                  pare;
caso <rótulo 2> : comando 1;
                  comando 2;
                  pare;
senão comando;
}
```

A expressão é avaliada e o valor será comparado com um dos rótulos.

O rótulo será aqui definido como uma constante caracter (de um caracter) ou uma constante numérica inteira.

A opção *senão* é opcional.

A estrutura é muito utilizada em algoritmos com menus, tornando-os mais claros do que quando usamos ses aninhados.

Quando um conjunto de valores discretos precisa ser testado e ações diferentes são associadas a esses valores, pode-se utilizar a seleção de múltipla escolha (**switch**). O objetivo é testar vários valores constantes possíveis para uma expressão, semelhante à seleção composta. Ou seja, é quase que um `if` com várias possibilidades, mas com algumas diferenças importantes:

Primeira diferença: Os `cases` não aceitam operadores lógicos. Portanto, não é possível fazer uma comparação. Isso limita o case a apenas valores definidos.

Segunda diferença: O `switch` executa seu bloco em cascata. Ou seja, se a variável indicar para o primeiro `case` e dentro do `switch` tiver `5 cases`, o `switch` executará todos os outros `4 cases` a não ser que utilizemos o comando para sair do `switch` (ao **`break`**).

A estrutura avalia a `expressão` e verifica se ela é equivalente à `constante1`, se for, ala executa o grupo de instruções do caso 1, até encontrar a instrução `break`. Quando se encontra com a instrução `break`, o programa salta para o final da estrutura `switch`.

```
switch (expressão)
{
   case constante1:
     grupo de instruções 1;
     break;
   case constante2:
     grupo de instruções 2;
     break;
   ...
   default:
   grupo de instruções default
}
```

A instrução `default`, do inglês padrão, é o case que é ativado caso não tenha achado nenhum case definido. Ou seja, é o que aconteceria em último caso. Vamos imaginar o seguinte cenário: Seu programa pede para que o usuário digite apenas duas opções (S ou

N) para reiniciar o programa. Mas, propositalmente ou por engano, o usuário digita uma opção totalmente diferente. E agora? O que seu programa deve fazer? É aqui que o default entra. Geralmente o default é quando é previsto um erro, uma entrada de dado incorreta ou não de acordo com o contexto. O default tem seu papel parecido com o `else`, da estrutura `if/else`, caso nenhuma condição for feita, fará os comandos definidos no `default`.

A maior utiliza dessa estrutura encontra-se na construção de MENUS, fornece uma visão esclarecedora das opções do programa, mostra efetivamente a estrutura do menu onde cada ramificação pode ser uma opção do menu.

7.4 Exemplo 7

Construa um programa que dado o tipo genético (de 1 a 8) de determinada espécie animal, seja capaz de classificar a sua região de procedência de acordo com a tabela abaixo.

Tipo genético	Procedência
1	Sul
2	Norte
3	Leste
4 a 7	Oeste
Caso contrário	Exótica

```
#include <iostream>
using namespace std;
int main()
{
int gene;
    cout << "Digite o tipo Genetico: ";
    cin >> gene;
    switch (gene)
    {
    case 1: cout<< "Regiao Sul\n";
         break;
    case 2: cout<< "Regiao Norte\n";
         break;
    case 3: cout<< "Regiao Leste\n";
         break;
    case 4:
    case 5:
    case 6:
    case 7: cout<< "Regiao Oeste\n";
         break;
    default: cout << "Exotica\n";
    }
    system("Pause");
}
```

7.5 Exemplo 8

Dados dois números reais quaisquer, desenvolva um programa que diga se eles são iguais ou diferentes.

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

```
{
    float a,b;
    cout << "digite o primeiro numero: "; cin >> a;
    cout << "digite o segundo numero: "; cin >> b;
    if (b==a)
       cout<<"Os números digitados sao iguais"<<endl;
    else
       cout<<"Os numeros digitados sao diferentes"<<endl;
    system("Pause");
}
```

7.6 Exemplo 9

Dado um número qualquer, determinar se este é neutro, positivo ou negativo.

```
#include <iostream>
using namespace std;
int main()
{
    float a;
    cout << "digite o numero: ";
    cin >> a;
    if (a==0)
       cout << "O número digitado e nulo" << endl;
       else if (a>0)
          cout << "O número digitado e positivo" << endl;
       else
          cout << "O número digitado e negativo" << endl;
    system("Pause");
}
```

7.7 Exemplo 10

Elabore um programa que identifique se um número inteiro digitado é par, impar ou nulo.

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cout << "digite o numero: ";
    cin >> n;
    if (n%2==0)
    {
      if(n==0)
      {
          cout << "nulo" << endl;
      }
      else
      {
           cout << "par" << endl;
       }
     }
    else
    {
       cout<<"impar"<<endl;
    }
```

```
    system("Pause");
}
```

8. Laços

Laços são comandos usados sempre que uma ou mais instruções devam ser repetidas enquanto uma certa condição estiver sendo satisfeita.

Em C++ existem três comandos de laços:

`for`
`while`
`do-while`

8.1 O laço for (para)

O laço `for` é em gral usando quando queremos repetir algo um número fixo de vezes. Isso significa que utilizamos um laço `for` quando sabemos de antemão o numero de vezes a repetir. O exemplo seguinte imprime uma linha com 20 asteriscos (*) utilizando um laço `for` na sua forma mais simples.

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    for(i=0; i < 20; i++ )
              cout << '*';
     cout << endl;
    system ("PAUSE");
    return 0;
}
```

```
********************
Press any key to continue...
```

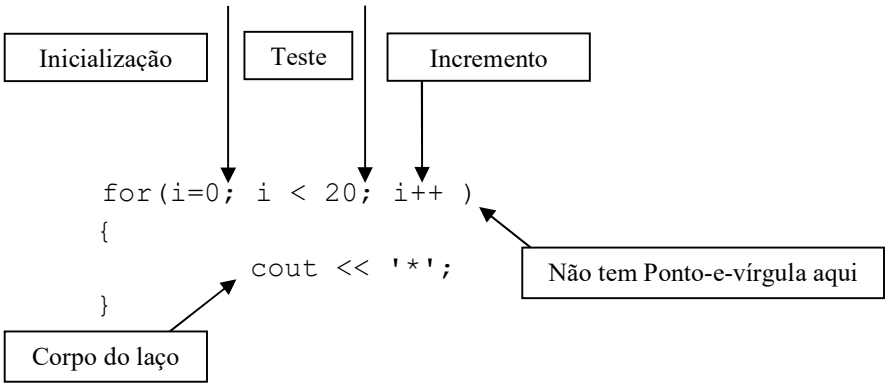
A sintaxe consiste na palavra-chave `for` seguida de parênteses que contêm três expressões separadas por ponto-e-vírgula. Chamaremos a primeira dessas expressões de *inicialização*, a segunda de *teste* e a terceira de *incremento*. Qualquer uma das três pode conter qualquer instruções em C++.

Em sua forma mais simples, a inicialização é uma instrução de atribuição (`i = 0`) e é sempre executada uma única vez, antes que o laço seja iniciado.

O teste é uma condição avaliada como verdadeira ou falsa e controla o laço (`i < 20`). Essa expressão é avaliada toda vez que o laço é iniciado ou reiniciado. Se verdadeira (diferente de zero), a instrução do corpo do laço é executada (`cout << '*'`). Quando o teste torna-se falso (igual a zero), o laço é encerrado e o controle passa para a instrução seguinte ao laço.

O incremento geralmente define a maneira pela qual a variável de controle será alterada cada vez que o laço for repetido (`i++` o mesmo que `i=i+1`). Essa expressão é executada sempre, logo após a execução do corpo do laço.

Ponto-e-vírgula



No nosso exemplo, o laço `for` é executado 20 vezes. Na primeira vez, a inicialização assegura que `i` vale zero. Na última vez, `i` vale 19; a informação é dada no teste (`i<20`). Em seguida, `i` passa a valer 20 e o laço termina. O corpo do laço não é executado quando `i` vale 20.

O programa abaixo escreve os números e 1 a 5.

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    for (i = 1; i <= 5; i++)
      cout << i;
    cout << endl;
    system ("PAUSE");
    return 0;
}
```

```
12345
Press any key to continue...
```

O programa abaixo a mesma operação que o anterior, entretanto, observe o incremento.

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    for (i = 1; i <= 5; i=i+1)
      cout << i;
    cout << endl;
    system ("PAUSE");
    return 0;
}
```

```
12345
Press any key to continue...
```

O mesmo raciocínio pode ser empregado nas instruções abaixo.

```
for (i = 10; i >= 5; i--)
    cout << i;
```

ou

```
for (i = 10; i >= 5; i=i-1)
    cout << i;
```

Nos trechos de programa acima, que expressam a mesma coisa, o comando cout << i; será executado dez vezes, ou seja, para i valendo 10, 9, 8, 7, 6, 5, 4, 3, 2 e 1.

O incremento/decremento poderá ser realizado em mais de uma unidade, observe os trechos de código abaixo:

```
for (i = 0; i <=10; i=i+2)
    cout << i;
```

No trecho de programa acima, o comando cout << i; será executado seis vezes, ou seja, para i valendo 0, 2, 4, 6, 8 e 10.

```
for (i = 100; i >= 0; i=i-20)
    cout << i;
```

No trecho de programa acima, o comando cout << i; será executado seis vezes, ou seja, para i valendo 100, 80, 40, 20 e 0.

Observe e analise o programa que imprime a tabuada do 6.

```
#include <iostream>
using namespace std;
int main()
{
    int i;
    for(i=1; i<10; i++)
       cout << '\n' << i << "x6= " << (i*6);
    cout << endl;
    system ("PAUSE");
    return 0;
}
```

```
1x6=6
2x6=12
3x6=18
4x6=24
5x6=30
6x6=36
7x6=42
8x6=48
9x6=54

Press any key to continue...
```

Vamos modificar o programa anterior para que a tabuada do 6 seja impressa de forma invertida.

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

```
9x6=54
```

```
{
    for(int i=9; i>0; i--)
       cout << '\n' << i << "x6= " << (i*6);
    cout << endl;
    system ("PAUSE");
    return 0;
}
```

```
8x6=48
7x6=42
6x6=36
5x6=30
4x6=24
3x6=18
2x6=12
1x6=6

Press any key to continue...
```

Observe que a variável `i` foi declarada no interior da expressão da inicialização do laço `for` do programa anterior. Essa construção é bastante comum em C++, pois, geralmente, declaramos as variáveis tão próximo quando possível do pondo onde elas serão usadas.

```
#include <iostream>
using namespace std;
int main()
{
    for(int i=9; i>0; i--)
       cout << '\n' << i << "x6= " << (i*6);
    cout << endl;
    system ("PAUSE");
    return 0;
}
```

```
9x6=54
8x6=48
7x6=42
6x6=36
5x6=30
4x6=24
3x6=18
2x6=12
1x6=6

Press any key to continue...
```

Se um laço `for` deve executar várias instruções a cada iteração, elas devem estar entre chaves:

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

Como dito anteriormente, em C++ um ***bloco de código*** é uma série de instruções entre chaves e é tratado como um grupo de instruções em uma única unidade. Isso significa que um bloco de código é tratado como uma única instrução.

```
#include <iostream>
using namespace std;
int main() // Início;
{
    float soma=0.0;
    const int max=10;
    for(int i =0; i< max; i++)
    {
            cout << "\nDigite a nota " << (i+1) <<" : ";
            float nota;
```

```
            cin >> nota;
            soma+=nota;//soma=soma+nota;
    }
    cout << "\nMedia = " << (soma/max) << endl;
    system("Pause");
    return(0);
}
```

Um aspecto importante dos blocos de código é o de que uma variável declarada dentro de um bloco não é visível fora dele. Ou seja, no programa anterior, declaramos a variável `nota` dentro do bloco do código do laço `for`. Essa variável só poderá se acessada pelas instruções desse mesmo bloco e que estão após a sua declaração.

8.2 Exemplo 11

Faça um programa que leia o número de termos, determine e mostre os valores de acordo com a série a seguir:

Série = 2, 7, 3, 4, 21, 12, 8, 63, 48, 16, 189, 192, 32, 567, 768 ...

Dica:

Série = 2, 7, 3, 4, 21, 12, 8, 63, 48, 16, 189, 192, 32, 567, 768 ...

```
#include <iostream>
using namespace std;
int main()
{
 int i, num_termos=0, num1=2, num2=7, num3=3;
 cout << "Entre com o numero de termos da serie: ";
 cin >> num_termos;
 cout << num1 << endl << num2 << endl << num3 <<endl;
 for (i=4; i<=num_termos; i++)
  {
      num1 = num1*2;
      cout << num1 <<endl;
      i=i+1;
      if (i!=num_termos)
      {
        num2 = num2*3;
        cout << num2 <<endl;
        i = i+1;
         if (i!=num_termos)
         {
         num3 = num3*4;
         cout << num3 <<endl;
         i = i+1;
         }
      }
  }
    system("PAUSE");
```

```
        return 0;
    }
```

8.3 O laço while (enquanto)

O segundo comando de laço em C++ é o `while` (do inglês que significa enquanto). À primeira vista, o laço parece simples se comparado ao laço `for`, utiliza os mesmo elementos, mas eles são distribuídos de maneira diferentes no programa. Utilizamos o laço `while` quando o laço pode ser terminado inesperadamente, por condição desenvolvida dentro do corpo do laço.

Trata-se de uma estrutura de repetição que pode ser utilizada quando o número de repetições necessárias não é fixo.

Assim, os comandos serão repetidos até a condição assumir o valor falso. Nesse tipo de estrutura, o teste condicional ocorre no início. Isto significa que existe a possibilidade da repetição não ser executada quando a condição assumir valor falso logo na primeira verificação.

O comando `while` consiste na palavra-chave `while` seguida de uma expressão de teste entre parênteses. Se a expressão de teste for verdadeira, o corpo do laço `while` é executado uma vez, e a expressão de teste é válida novamente. Esse ciclo de teste e execução é repetido até que a expressão de teste se torne falsa, então, o laço termina e o controle do programa passa para a linha seguinte ao laço.

O corpo de um `while` pode ter uma única instrução terminada por um ponto-e-vírgula, várias instruções entre chaves ou ainda uma nenhuma instrução mantendo o ponto-e-vírgula.

Em geral, um laço `while` pode substituir um laço `for` da seguinte maneira em um programa:

```
Inicialização;
while( Teste)
{
     .
     .
     Incremento;
     .
}
```

Em geral, um laço `while` pode substituir um laço `for` da seguinte maneira em um programa:

```
#include <iostream>
using namespace std;
int main()
{
  int i=0; //Incicialização
  while(i < 20) //Teste
  {
     cout << '*';
     i++; //Incremento
   }
   cout << endl;
    system ("PAUSE");
    return 0;
```

```
**********
Press any key to continue...
```

```
}
```

Em situação em que o número de iterações é conhecido o laço for é a escolha mais natural. Enquanto a condição for verdadeira, os comandos que estão dentro das chaves serão executados, observe o programa abaixo:

```
#include <iostream>
using namespace std;
int main ()
{
  int X = 0;
  while (X != 5)
  {
    cout << "Valor de X = " << X << endl;
    X = X+1;
  }
  cout<<"X depois de sair do while = " << X;
  cout << endl;
  system("Pause");
  return 0;
}
```

```
Valor de X = 0
Valor de X = 1
Valor de X = 2
Valor de X = 3
Valor de X = 4
X depois de sair do while = 5
Press any key to continue...
```

Neste programa, os comandos `cout << "Valor de X = " << X;` e `X = X+1;` serão executados 5 vezes. O teste condicional avaliará `X` valendo 0,1, 2, 3, 4 e 5.

TELA	X	
Mensagem	**0**	**VALOR INICIAL**
Valor de X = 0	1	Valores obtidos dentro da estrutura de repetição
Valor de X = 1	2	
Valor de X = 2	3	
Valor de X = 3	4	
Valor de X = 4	5	Valor obtido dentro da estrutura de repetição, que torna a condição falsa e interrompe a repetição
Valor de X = depois de sair da estrutura = 5		

Observe o programa abaixo:

```
#include <iostream>
using namespace std;
int main ()
{
  int X = 1, Y=10;
  while (Y > X)
  {
    cout << "Valor de Y = " << Y << endl;
    Y = Y-2;
  }
  cout<<"Y depois de sair do while = " << Y;
  cout << endl;
  system("Pause");
  return 0;
}
```

```
Valor de Y = 10
Valor de Y = 8
Valor de Y = 6
Valor de Y = 4
Valor de Y = 2
X depois de sair do while = 0
Press any key to continue...
```

Neste programa, os comandos `cout << "Valor de Y = " << Y;` e `Y = Y-2;` serão executados 5 vezes. O teste condicional avaliará `Y` valendo 10, 8, 6, 4, 2 e 0.

TELA	X	Y	
Mensagem	1	10	**VALOR INICIAL**
Valor de Y = 10	1	8	Valores obtidos dentro da estrutura de repetição
Valor de Y = 8	1	6	
Valor de Y = 6	1	4	
Valor de Y = 4	1	2	
Valor de Y = 2	1	0	Valor obtido dentro da estrutura de repetição, que torna a condição falsa e interrompe a repetição
Valor de Y = depois de sair da estrutura = 0			

Analise a estrutura abaixo:

```
#include <iostream>
using namespace std;
int main ()
{
  int n;
  cout << "Digite o numero inicial: ";
  cin >> n;
  while(n>0)
  {
   cout << n << ", ";
   n--;
  }
  cout << "FOGO!\n";
  system("Pause");
  return 0;
}
```

```
"Digite o numero inicial: 7
7, 6, 5, 4, 3, 2, 1, FOGO!
Press any key to continue...
```

Quando criamos um laço do tipo `while`, devemos sempre considerar que ele deve terminar em algum momento. Entretanto, devemos fornecer dentro do bloco algum método para forçar a condição se tornar falsa em algum ponto. Por outro lado, o loop continuará para sempre. No Programa anterior, incluímos a instrução `n--;` que decrementa a variável que é avaliada no teste em um (`n--` equivale a `n=n-1`) e torna a condição (`n>0`) falsa depois de um número de iterações (loops). Assim, quando `n` torna-se `0`, é quando o nosso `while` e a nossa contagem termine.

Analise o exemplo:

```
#include <iostream>
using namespace std;
int main ()
{
  int X = 1, Y=1;
  while (X < Y)
  {
    cout << "Valor de X = " << X << endl;
    X = X+1;
  }
  cout<<"Acabou o while";
  cout << endl;
  system("Pause");
  return 0;
```

```
Acabou o while
Press any key to continue...
```

```
}
```

No trecho de programa acima, os comandos `cout << "Valor de X = " << X;` e `X=X+1;` não serão executados, pois, para os valores iniciais de X e Y a condição é falsa, logo, não ocorrerá a entrada na estrutura de repetição para execução de seus comandos.

Agora, observe um exemplo de laço infinito:

```
#include <iostream>
using namespace std;
int main ()
{
  int X = 1, Y=1;
  while (X <= Y)
  {
    cout << "Valor de X = " << X << endl;
  }
  cout<<"Acabou o while";
  cout << endl;
  system("Pause");
  return 0;
}
```

```
Valor de X = 1
Valor de X = 1
Valor de X = 1
Valor de X = 1
Valor de X = 1
Valor de X = 1
Valor de X = 1
Valor de X = 1
Valor de X = 1
Valor de X = 1
Valor de X = 1
```

No trecho de programa acima, o comando `cout << "Valor de X = " << X;` e executados infinitamente, pois o valor de X ou Y não é modificado em nenhum momento do meu programa, não contrariando a condição do laço, tornando-o um laço infinito.

8.4 O laço do-while (repetir)

Este é o terceiro e último comando de laço em C++. Este é bastante similar ao laço `while` e é utilizado em situações em que é necessário executar o corpo do laço uma primeira vez e depois avaliar a expressão de teste e criar um ciclo repetido. Trata-se de uma estrutura de repetição que pode ser utilizada quando o número de repetições necessárias não é fixo. Os comandos serão repetidos até a condição assumir o valor falso.

A sintaxe do laço `do-while` é:

```
Inicialização;
do
{
     .
     .
     Incremento;
     .
}while(Teste);
```

O comando `do-while` consiste na palavra-chave `do` seguida de um bloco de uma ou mais instruções entre chaves e termina pela palavra-chave `while` seguida de uma expressão de teste entre parênteses terminada por um ponto-e-vírgula.

Primeiro, o bloco de código é executado; em seguida, a expressão de teste entre parênteses é avaliada; se verdadeira, o corpo do laço é mais uma vez executado e a expressão de teste é avaliada novamente. O ciclo de execução do bloco e teste é repetido até que a expressão se torne falsa (igual a zero), então o laço termina e o controle do programa passa para a linha seguinte ao laço.

Observe o exemplo abaixo:

```
#include <iostream>
using namespace std;
int main ()
{
  int X = 0;
  do
  {
    cout << "\nValor de X = " << X;
       X = X+1;
  }while (X != 5);
  cout << "\nX depois de sair do repetir" << X;
  cout << endl;
  system("Pause");
  return 0;
}
```

```
Valor de X = 0
Valor de X = 1
Valor de X = 2
Valor de X = 3
Valor de X = 4
X depois de sair do repetir = 5
Press any key to continue...
```

No trecho de programa acima, o comando `cout << "Valor de X = " << X;` e `X=X+1;` serão executados 5 vezes. O teste condicional avaliará X valendo 0,1, 2, 3, 4 e 5.

TELA	X	
Mensagem	**0**	**VALOR INICIAL**
Valor de X = 0	1	Valores obtidos **dentro** da estrutura de repetição
Valor de X = 1	2	
Valor de X = 2	3	
Valor de X = 3	4	
Valor de X = 4	5	Valor obtido dentro da estrutura de repetição, que torna a **condição falsa** e **interrompe** a repetição
Valor de X = depois de sair da estrutura = 5		

Observe o exemplo abaixo:

```
#include <iostream>
using namespace std;
int main ()
{
  int X = 1, Y=10;
  do
  {
    cout << "\nValor de Y = " << Y;
       Y = Y-1;
  }while (Y > X);
  cout << "\nY depois que sair do repetir = " << Y;
  cout << endl;
  system("Pause");
  return 0;
}
```

```
Valor de Y = 10
Valor de Y = 8
Valor de Y = 6
Valor de Y = 4
Valor de Y = 2
X depois de sair do while = 0
Press any key to continue...
```

No trecho de programa acima, os `comandos cout << "Valor de Y = " << Y;` e `Y = Y-2;` serão executados 5 vezes. O teste condicional avaliará Y valendo 10, 8, 6, 4, 2 e 0.

TELA	X	Y	
Mensagem	1	10	**VALOR INICIAL**
Valor de Y = 10	1	8	Valores obtidos **dentro** da estrutura de repetição
Valor de Y = 8	1	6	
Valor de Y = 6	1	4	
Valor de Y = 4	1	2	
Valor de Y = 2	1	0	Valor obtido dentro da estrutura de repetição, que torna a **condição falsa** e **interrompe** a repetição
Valor de Y = depois de sair da estrutura = 0			

Analise o exemplo:

```
#include <iostream>
using namespace std;
int main ()
{
  int X = 1, Y=1;
  do
  {
    cout << "Valor de X = " << X << endl;
    X = X+1;
  } while (X < Y);
  cout<<"Acabou o repetir";
  cout << endl;
  system("Pause");
  return 0;
}
```

```
Valor de X = 1
Acabou o repetir
Press any key to continue...
```

No trecho de programa acima, os comandos `cout << "Valor de X = " << X;` e `X=X+1;` serão executados 1 vez independente da condição ser falsa, pois os valores iniciais de X e Y serão verificados no final da estrutura.

Agora, observe um exemplo de laço infinito:

```
#include <iostream>
using namespace std;
int main ()
{
  int X = 1, Y=1;
  do
  {
    cout << "Valor de X = " << X << endl;
  } while (X <= Y);
  cout<<"Acabou o repetir";
  cout << endl;
  system("Pause");
  return 0;
}
```

```
Valor de X = 1
Valor de X = 1
Valor de X = 1
Valor de X = 1
Valor de X = 1
Valor de X = 1
Valor de X = 1
Valor de X = 1
Valor de X = 1
Valor de X = 1
Valor de X = 1
```

No trecho de programa acima, o comando `cout << "Valor de X = " << X;` e executados infinitamente, pois os valores de X ou Y não são modificados em nenhum

momento do laço de repetição, não contrariando a condição do laço, tornando-o um laço infinito.

8.5 Exemplo 12

Elabore um programa que calcule o fatorial de um número dado.

```
#include <iostream>
using namespace std;
int main()
{
    int n;
    cout << "digite o numero: ";
    cin >> n;
    int fat=1;
    for(int i=1; i<=n; i++)
        fat=fat*i;
    cout << fat << endl;
    system("Pause");
    return 0;
}
```

8.6 Exemplo 13

Elabore um programa que imprima os termos da serie abaixo, o programa deve parar o processamento quando for impresso um termo negativo.

15, 30, 60, 12, 24, 48, 9, 18, 36, 6, 12, 24, ...

```
#include <iostream>
using namespace std;
int main()
{
    int i, j, n;
    for(j=15; j>0; j-=3)
    {
       i=0;
       n=j;
       while(i<3)
       {
           cout<<n<<"\t";
           n=n+n;
           i++;
       }
    }
    cout<<endl;
    system("Pause");
    return 0;
}
```

8.7 Exemplo 14

Faça um programa que analise se o número k, fornecido pelo usuário é maior que o numero n, também fornecido pelo usuário, se isso for verdadeiro, e imprima todos os valores de k a n, se k for igual a n calcule o fatorial de k, e caso k for menor que n, apenas informe ao usuário. Ao final do programa, faça uma pergunta se o usuário quer informar novamente os valores de k e n, para o programa executar mais uma vez.

```
#include <iostream>
using namespace std;
int main()
{
    int k, n;
    char r;

    do
    {
    cout<<"Digite o valor de k: ";
    cin>>k;
    cout<<"Digite o valor de n: ";
    cin>>n;
    if(k>n)
           for(int i=k; i>=n; i--)
                  cout << i << endl;
           else if(k==n)
           {
                  int fat=1;
                  for(int i=1; i<=k; i++)
                         fat=fat*i;
                  cout << "Fatorial de " << k;
                  cout << " = " << fat << endl;
           }
           else
           cout << "k e menor do que n" << endl;
    cout<<"Deseja digidar dois numeros novamente (s/n): ";
    cin>>r;
    }while(r=='s');
    system("Pause");
    return 0;
}
```

9. Estruturas Composta de Dados (Vetores)

Geralmente, os algoritmos são elaborados para **manipulação de dados**. Quando estes dados estão **organizados** (dispostos) de forma coerente, caracterizam **uma forma**, uma ***estrutura de dados***.

A organização dos dados é chamada **de estrutura composta de dados** que se divide em duas formas fundamentais:

- **homogêneas** (vetores e matrizes)
- **heterogêneas** (registros).

9.1 Estrutura de Dados Composta Homogênea

As estruturas de dados homogêneas possibilitam o **armazenamento** de grupos de valores em **uma única variável** que será armazenada na memória do computador. Essas estruturas são ditas "**homogêneas**" porque os valores que serão armazenados **são de um mesmo tipo de dado**.

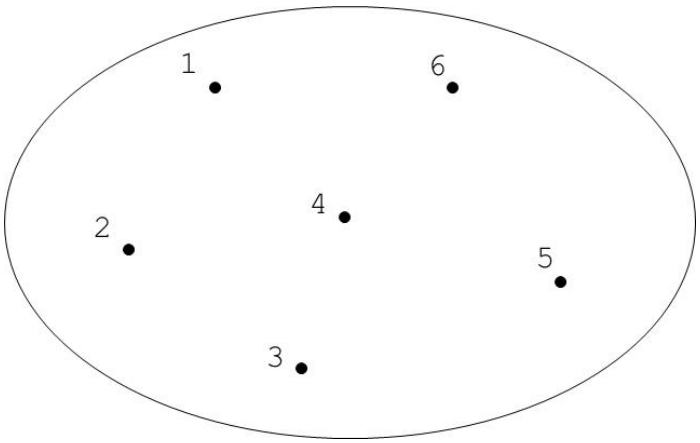
Estas estruturas homogêneas são divididas em ***unidimensionais*** e ***multidimensionais***. Normalmente, as estruturas ***unidimensionais*** são chamadas de ***vetores*** e as ***multidimensionais*** são chamadas de ***matrizes***. De fato, um vetor também é uma matriz, porém varia em somente uma dimensão.

```
algoritmo "alturas de atletas"
// Síntese
//    Objetivo: armazenar 5 alturas de atletas de basquete
//    Entrada: 5 alturas
//    Saída: -
// Declarações
    var
       altura1,altura2,altura3,altura4,altura5 : real
  inicio
    escreva("Informe a altura do 1º atleta: ");
    leia (altura1)
    escreva("Informe a altura do 2º atleta: ")
    leia (altura2)
    escreva("Informe a altura do 3º atleta: ")
    leia (altura3)
    escreva("Informe a altura do 4º atleta: ")
    leia (altura4)
    escreva("Informe a altura do 5º atleta: ")
    leia (altura5)
fimalgoritmo
```

O algoritmo acima demostra a utilização de 5 variáveis para armazenar as alturas dos atletas, no caso de estruturas de dados homogêneas, todas as alturas seriam armazenadas em uma única variável.

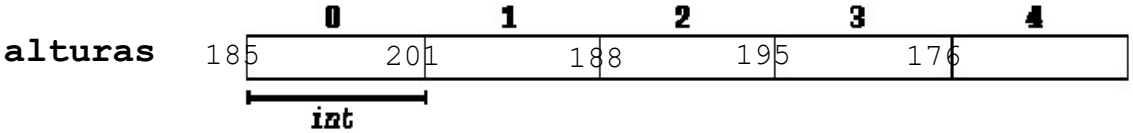
9.2 Vetores

Um ***vetor*** é uma série de um mesmo tipo de dados alocados de maneira contígua na memória do computador que podem ser individualmente referenciados por um número. Portanto, é uma ***coleção de elementos de um mesmo tipo***. Cada um dos elementos é unicamente identificado ***por um número inteiro.***



Isso significa que poderíamos armazenar a altura dos 5 atletas de basquete, do exemplo anterior, em um vetor sem termos que declarar 5 diferentes variáveis, cada uma com um identificador (nome) diferente. Ao invés disso, utilizando um vetor podemos armazenar os 5 valores de altura de um mesmo tipo de dado, `int` por exemplo, em um único identificador.

Por exemplo, um vetor contendo 5 valores de alturas chamado `alturas` pode ser representado por:

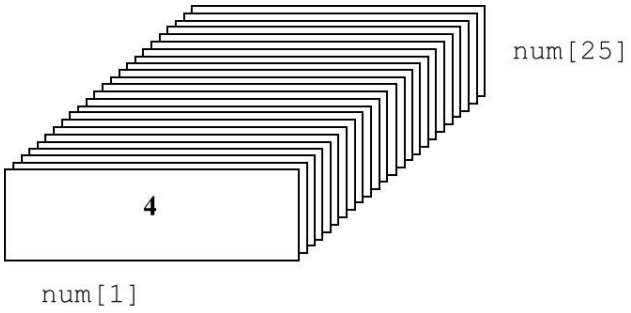


nesse vetor cada espaço em branco representa um elemento do vetor, que nesse caso são valores inteiros das alturas dos atletas em centímetros (185, 201, 188, 195 e 176). Estes elementos são numerados de 0 a 4, observe que em C++ os vetores o primeiro índice será sempre o Zero, independentemente do seu comprimento (número total de elementos). O valor do **índice não** deve ser confundido com o conteúdo da **posição do vetor**. O índice identifica o elemento dentro do conjunto. O índice tem de ser obrigatoriamente **inteiro**. O elemento do vetor pode ser um número inteiro, um número real, uma variável booleana, um caractere, uma string, etc. O índice de um vetor corresponde à numeração das casas numa rua.

O número de uma casa nada tem a ver com o seu conteúdo.

Representação Gráfica de um Vetor:

`num[1] ← 4`



Assim como uma variável comum, o vetor deve ser declarado antes de ser utilizado. Uma declaração típica de um vetor em C++ é:

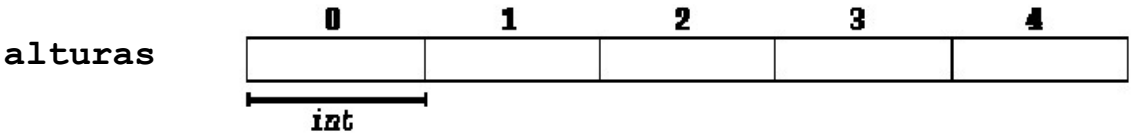
```
tipo nome [elementos];
```

Onde o `tipo` é qualquer tipo de dado válido, visto até agora (`int`, `float`...), `nome` é um identificador válido e o `elementos`, que deve ser sempre definido dentro de colchetes (`[ ]`) especifica a quantidade de elementos que o vetor deve possuir.

Portanto, para declararmos o vetor chamado `alturas` como aquele apresentado no diagrama anterior, simplesmente utilizamos:

```
int alturas [5];
```

E o vetor será criado como:



9.3 Inicialização de Vetores

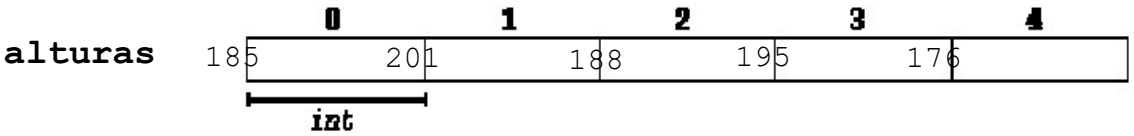
Quando declaramos um vetor em um escopo local (dentro de uma função, por exemplo), se não especificado, seus elementos não serão inicializados como algum valor default, portanto, seu conteúdo será indeterminado até o momento em que armazenaremos

algum valore nele. Os elementos de vetores globais ou estáticos, por outro lado, serão automaticamente inicializados com valores default, o que para todos os tipos de dados fundamentais, significa que serão preenchido com zero (0).

Em ambos os casos, globais e locais, quando declaramos um vetor, devemos possibilitar a atribuição de valores iniciais para cada um dos seus elementos pela encerrando os valores em chaves (`{ }`).

```
int alturas [5] = {185, 201, 188, 195, 176};
```

Dessa maneira, nosso vetor altura apresentará, finalmente, a seguinte conformação.



A quantidade de elementos entre chaves (`{ }`) deve ser tão grande quanto o tamanho de elementos que declaramos entre colchetes (`[ ]`) . Por exemplo, quando declaramos o vetor:

```
int alturas [5];
```

especificamos que ele tem 5 elementos e na lista de alturas:

```
{185, 201, 188, 195, 176}
```

temos 5 elementos.

Em C++ podemos iniciar um vetor da seguinte forma:

```
int alturas [ ] = {185, 201, 188, 195, 176};
```

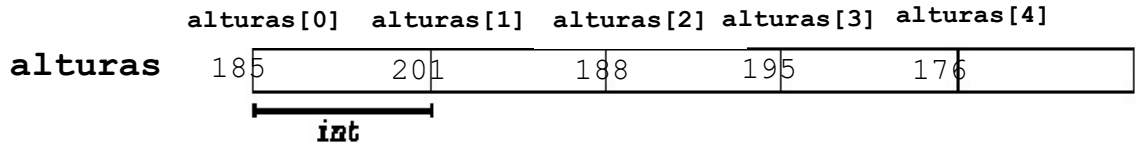
Observe que deixamos vazio os colchetes (`[ ]`), nesse caso, o compilador assumirá que o tamanho para o vetor será o número de elementos incluídos entre chaves (`{ }`). Após a declaração, portanto, o nosso vetor alturas terá o tamanho de 5 uma vez que fornecemos 5 elementos dentro das chaves.

9.4 Acessando os elementos dos vetores

Em qualquer ponto de um programa, poderemos acessar individualmente qualquer um dos elementos de um vetor, como se este elemento fosse uma variável normal, assim, somos capazes de ler e modificar esses valor. O formato para acessar um elemento é simples:

```
nome [índice];
```

Seguindo o exemplo anterior, nosso vetor alturas apresenta 5 elementos, e cada um desses elementos são do tipo inteiro (int), os nomes que poderemos utilizar para nos referirmos a cada um dos seus elementos são:



por exemplo, para armazenarmos o valor `190` no terceiro elemento do vetor `alturas`, nós poderemos escrever a seguinte declaração:

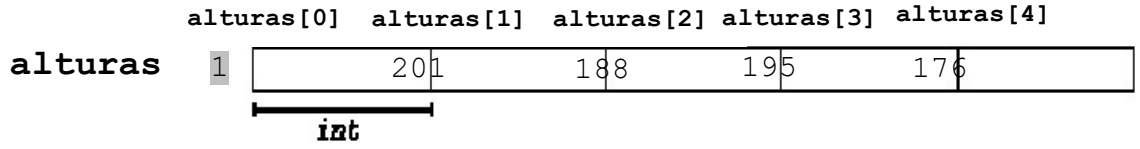
```
alturas [2] = 190;
```

e, se quisermos passar o valor do terceiro elemento do vetor alturas para a variável a, poderes utilizar a instrução:

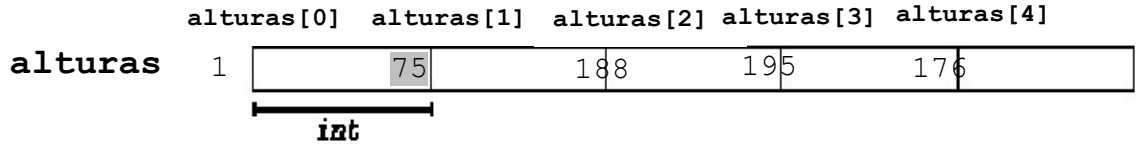
```
a = alturas [2];
```

Abaixo serão apresentadas mais algumas operações válidas para vetores:

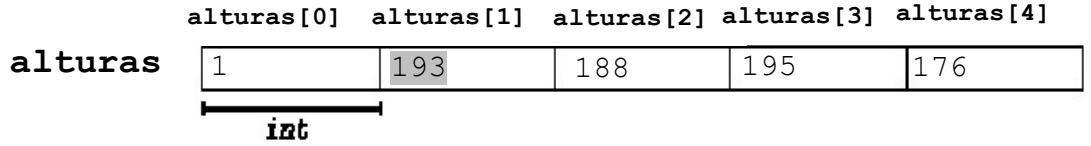
```
a = 1;
alturas [0] = a;
```




```
alturas [a] = 75;
```




```
b = alturas[a+2]; //portanto b = 195
alturas[alturas [0]] = alturas[2] + 5;
```



Seguindo o exemplo anterior, nosso vetor alturas apresenta 5 elementos, e cada um desses elementos são do tipo inteiro (int), os nomes que poderemos utilizar para nos referirmos a cada um dos seus elementos são:

O programa abaixo calcula a soma dos valores de alturas de jogadores de basquete presentes no vetor `alturas`.

```
#include <iostream>
using namespace std;
int alturas [5] = {185, 201, 188, 195, 176};
```

```
Soma das alturas = 945
```

```
int i, soma=0;
int main ()
{
  for (i=0; i <=5; i++)
      soma += alturas[i]; // soma = soma + alturas[i]
  cout << "Soma das alturas = " << soma;
  cout << "\n\n";
  system("Pause");
  return 0;
}
```

```
Press any key to continue...
```

9.5 Exemplo 15

Elabore um programa que calcule a média ($\bar{x}$), a variância (s^2), o desvio padrão (s), o mínimo, o máximo e o coeficiente de variação (cv) das alturas dos atletas:

$$\bar{x} = \frac{\sum_{i=1}^{n} x_i}{n} \qquad s^2 = \frac{\sum_{i=1}^{n} x_i^2 - \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n}}{n-1}$$

$$s = \sqrt{s^2} = \sqrt{\frac{\sum_{i=1}^{n} x_i^2 - \frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n}}{n-1}} \qquad cv = \frac{100 \times s}{\bar{x}}$$

Onde n é o número de elementos do vetor e x é o vetor de alturas. Para auxiliar, programe as funções para calcular:

$$soma = \sum_{i=1}^{n} x_i \quad \text{e} \quad soma\ ao\ quadrado = \sum_{i=1}^{n} x_i^2$$

```
#include <iostream>
#include <math.h>
using namespace std;

//vairáveis globais
float dados[10] = {21.5, 25.5, 14.5, 15.8, 17.6, 8.9, 9.6, 18.6, 17, 12};
int n=10;

//função principal
int main()
{

    //calculando a soma dos elementos de um vetor
    float soma=0.0;
    for(int i=0; i<=9; i++)
        soma+=dados[i]; //soma=soma+dados[i]

    //calculando a soma de quadrados do vetor
    float somaquad=0.0;
    for(int i=0; i <=9; i++)
        somaquad+=dados[i]*dados[i];

    //calculando a média aretimética do vetor
    float media=0.0;
    for(int i=0; i<=9; i++)
        media+=dados[i]; // media = media + dados[i]
```

```
    media/=10; // media = media/10

    // calculando a variancia amostral dos dados
    float variancia;
    variancia=(somaquad - soma*soma/10)/(10-1);

    //calculando o desvio padrao amostral dos dados
    float desvpad;
    desvpad=sqrt(variancia);

    //encontrando o menor valor dos dados
    float minimo=dados[1];
    for(int i =0; i <=n-1; i++)
        if(minimo>dados[i]) minimo = dados[i];

    //encontrando o maior valor dos dados
    float maximo=dados[1];
    for(int i =0; i <=n-1; i++)
        if(maximo<dados[i]) maximo = dados[i];


    //função que calcula o coeficiente de variação
    float cv;
    cv=100*desvpad/media;

    cout << "Soma               : " << soma<< endl;
    cout << "Soma de Quadrados  : " << somaquad << endl;
    cout << "Media              : " << media << endl;
    cout << "Variancia Amostral: " << variancia << endl;
    cout << "Desvio Padrao      : " << desvpad << endl;
    cout << "Minimo             : " << minimo << endl;
    cout << "Maximo             : " << maximo << endl;
    cout << "CV                 : " << cv << endl;
    system("Pause");
    return 0;
}
```

```
Soma               : 161
Soma de Quadrados  : 2832.48
Media              : 16.1
Variancia Amostral: 26.7089
Desvio Padrao      : 5.16806
Minimo             : 8.9
Maximo             : 25.5
CV                 : 32.0998
Press any key to continue . . .
```

9.6 Métodos de ordenação de vetores

ORDENAÇÃO é uma das tarefas básicas em processamento de dados. A ordenação de um vetor significa classificar os seus elementos, ou seja, colocar os elementos do vetor em ordem de acordo com algum critério (ordem crescente, ou ordem decrescente). Existem vários algoritmos de ordenação que se diferenciam pela velocidade da ordenação e complexidade de implementação:

Métodos de ordenação Simples:

Método da Seleção
Método de Bolha
Método da Inserção

Métodos de ordenação Sofisticados:
Método Shell Sort
Método Quick Sort
Método Heap Sort

Vamos colocar em ordem crescente os valores de alturas dos atletas. Para isso, inicialmente, nosso programa deve:

Método Seleção

- Selecionar o elemento do vetor (10 posições) que apresenta o menor valor;
- Trocar esse elemento pelo primeiro;
- Repetir essas operações, envolvendo agora somente os 9 elementos restantes (selecionando o de menor valor com a segunda posição, depois o 8 elementos (trocando o de menor valor com a terceira posição), depois os 7, os 6 e assim por diante, até restar um único elemento, o maior deles.

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

//vairáveis globais
float dados[10] = {21.5, 25.5, 14.5, 15.8, 17.6,
                   8.9, 9.6, 18.6, 17, 12};
int n=10;

//função principal
int main()
{
    cout<< "Vetor dados original: \n";
    cout<<"Indice\tElemento\n";
    for(int i=0; i<= n-1; i++)
        cout << "["<<i<<"]\t"<<dados[i] << endl;
    cout<< "\n\nVetor dados ordenado: \n";

    //ordenação do vetor
    float aux;
    int i, j, k;
    for(i=0; i<=n-2;i++)
    {
        k=i;
        aux=dados[i];
        for(j=i+1; j<=n-1;j++)
            if (dados[j]<aux)
            {
                k=j;
                aux=dados[k];
            }
        dados[k]=dados[i];
        dados[i]=aux;
    }
    for(int i=0; i<= n-1; i++)
        cout << "["<<i<<"]\t"<<dados[i] << endl;
    system("Pause");
    return 0;
}
```

```
Vetor dados original:
Indice  Elemento
[0]     21.5
[1]     25.5
[2]     14.5
[3]     15.8
[4]     17.6
[5]     8.9
[6]     9.6
[7]     18.6
[8]     17
[9]     12


Vetor dados ordenado:
[0]     8.9
[1]     9.6
[2]     12
[3]     14.5
[4]     15.8
[5]     17
[6]     17.6
[7]     18.6
[8]     21.5
[9]     25.5
```

Método da Bolha

Neste método de ordenação, o programa utiliza a seguinte estratégia para ordenar o vetor:

- Comparar os pares de elementos adjacentes, permutando-os quando estiverem foram de ordem, até que todos estejam ordenados. O objetivo da função é colocar o menor item da lista nessa variável na primeira posição do vetor a ser ordenado `v[0]`. A função percorre um por um dos elementos seguintes a fim de encontrar o menor deles. Sempre que encontra um elemento menor, eles são trocados. Terminado essa operação, é tomado o próximo item, `v[1]`; ele deverá conter o próximo menor valor. Novamente, são realizadas as comparações e trocas. O processo continua até que a lista toda seja ordenada.

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

//vairáveis globais
float dados[10] = {21.5, 25.5, 14.5, 15.8, 17.6,
                   8.9, 9.6, 18.6, 17, 12};
int n=10;

//função principal
int main()
{
    cout<< "Vetor dados original: \n";
    cout<<"Indice\tElemento\n";
    for(int i=0; i<= n-1; i++)
        cout << "["<<i<<"]\t"<<dados[i] << endl;
    cout<< "\n\nVetor dados ordenado: \n";

    //ordenação do vetor
    int i, j;float aux;
    for(i=0;i<=n-1;i++)
        for(j=i+1;j<=n-1;j++)
            if(dados[i] >dados[j])
            {
                aux=dados[j];
                dados[j]=dados[i];
                dados[i]=aux;
            }

    for(int i=0; i<= n-1; i++)
        cout << "["<<i<<"]\t"<<dados[i] << endl;
    //system("Pause");
    return 0;
}
```

```
Vetor dados original:
Indice  Elemento
[0]     21.5
[1]     25.5
[2]     14.5
[3]     15.8
[4]     17.6
[5]     8.9
[6]     9.6
[7]     18.6
[8]     17
[9]     12


Vetor dados ordenado:
[0]     8.9
[1]     9.6
[2]     12
[3]     14.5
[4]     15.8
[5]     17
[6]     17.6
[7]     18.6
[8]     21.5
[9]     25.5
```

- O processo de ordenação é feito por meio de dois laços. O laço mais externo determina qual elemento do vetor será usado como base de comparação. O laço mais interno compara cada item seguinte com o de base e, quando encontra um elemento menor que o de base, faz a troca, realizada no corpo do `if` por meio de uma variável auxiliar (`aux`).

9.7 Exemplo 16

Elabore um programa, que calcule a mediana das alturas dos atletas. A mediana é o valor que ocupa a posição central do vetor ordenado dos dados, o valor é precedido e seguido pelo mesmo número de observações e nem sempre pertence ao conjunto de dados, uma vez que:

$$Md = \begin{cases} x_{\left(\frac{n+1}{2}\right)}, & se\ n\ é\ ímpar; \\ \dfrac{x_{\left(\frac{n}{2}\right)} + x_{\left(\frac{n}{2}+1\right)}}{2}, & se\ n\ é\ par. \end{cases}$$

Onde x é o vetor de alturas e n o número de elementos do vetor.

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

//vairáveis globais
float dados[10] = {21.5, 25.5, 14.5, 15.8, 17.6,
                   8.9, 9.6, 18.6, 17, 12};
int n=10;
int main()
{
    //calculando a mediana
    float mediana;
    int i, j;
    float aux;

    // ordenando o vetor
    for(i=1;i<=n-1;i++)
       for(j=n-1;j>=0;j--)
          if(dados[j-1] >dados[j])
          {
             aux=dados[j-1];
             dados[j-1]=dados[j];
             dados[j]=aux;
          }

    //aplicando a fórmula da mediana
    if (n%2 ==1)
       mediana= dados[(n-1)/2];
    else
        mediana=(dados[(n/2)-1]+dados[n/2])/2;

    cout << "Mediana: " << mediana << endl;
    system("Pause");
    return 0;
}
```

9.8 Número desconhecidos de elementos

Nos exemplos anteriores, utilizamos um número fixo de elementos do vetor. Como faríamos se conhecemos de antemão quantos itens entrariam no vetor? Para exemplificar, vamos escrever um programa que aceita até 100 valores de altura, que pode ser modificado para aceitar qualquer valor de altura.

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

const int N = 100;

int main()
{
```

```
	float alturas[N], media=0.0;
	int i=0;
	cout << setprecision(4);
	do
	{
		cout << "Digite o valor de altura " << i+1 << " : ";
		cin >> alturas[i];
	}	while(alturas[i++]>=0.0);
	i--;

	for(int j=0; j<i;j++)
		media+=alturas[j] ;// media=media+alturas[j]
	cout << "Medias das alturas: " <<(media/i)<<endl;
	system("Pause");
	return 0;
}
```

O laço `for` foi substituído pelo laço `do-while`. Esse laço repete a solicitação da altura ao usuário, armazenando-a no vetor `alturas[]`, até que seja fornecido um valor menor que zero. Quando o último elemento for digitado, a vaiável `i` terá alcançado um valor acima do total de elementos. Isso é verdadeiro, pois é contado o número negativo de finalização da entrada. Assim devemos subtrair `1` do valor contido em `i` para encontrar o número correto de elementos.

Entretanto, a linguagem C++ não realiza verificação de limites em vetores, por isso, se o usuário decidir inserir mais de 100 valores de alturas, devemos expulsar os valores excedentes. Para providenciar a verificação do limite de uma matriz é de responsabilidade do programa. Portanto, não podemos permitir que o usuário digite dados acima do limite, assim, temos:

```
	do
	{
	  if( i>= N)
	  {
		cout << "BUFFER LOTADO" << endl;
		i++;
		break; // Saida do laço do-while
	  }
		cout << "Digite o valor de altura " << i+1 << " : ";
		cin >> alturas[i];
	}	while(alturas[i++]>=0.0);
```

Assim, quando `i` atingir `100`, que é `1` acima do último índice do vetor, a mensagem `" BUFFER LOTADO "` será impressa e o comando `break` fará que o controle saia do laço `do-while` e passe para a segunda parte do programa, ou seja, o cálculo da média.

9.9 Exemplo 17

Elabore um programa que dado um vetor de 10 valores inteiros, crie e imprima outro vetor, armazenando os elementos do primeiro a partir do centro para as bordas, de modo alternado (vetor inteiro com 10 posições também).

Nesta primeira forma de resolução, utilizaremos a variável inteira `j` para receber os índices do vetor resposta (`vr`), inicializando-a com o valor do índice do elemento central, ou seja, a posição 4 (`10/2-1`). Observe que dentro do laço `for` utilizamos um `if` com a seguinte condição, se o resto da divisão de `i` (índice do vetor original) por `2` for igual a zero, `j` será incrementado em `i+1` (lembre que o primeiro valor de `i` é zero), caso contrário `j` será decrementado em `i+1`.

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

int main()
{
    int vetor[10]={11,22,33,44,55,66,77,88,99,1010};
    int vr[10]={};
    int j=10/2-1;
    //preenchimento de vr
    for(int i=0; i <=9;i++)
    {
        if(i%2==0)
            {
                vr[ j ]=vetor[i];
                j=j+(i+1);
            }
        else
            {
                vr[ j ]=vetor[i];
                j=j-(i+1);
            }
    }
    //Impressão dos vetores
    cout<<"vetor\tvr"<<endl;
    for(int i=0;i<=9;i++)
        cout<<vetor[i]<<"\t"<<vr[i]<<endl;

    system("pause");
    return 0;

}
```

```
vetor   vr
11      99
22      77
33      55
44      33
55      11
66      22
77      44
88      66
99      88
1010    1010
Press any key to continue . . .
```

Na segunda forma de resolução desse problema, vamos utilizar um segundo laço `for`, ao invés do `if`. Para isso vamos precisar de mais duas variáveis (`l` e `k`). O laço for realizará sucessivas multiplicações de `k` (iniciado com o valor de -`1`) `l` vezes, com `l` variando de `0` até o valor de `i`, fazendo `k` variar entre `1` e `-1` a cada contagem do primeiro laço. Portanto, quando `k=1`, `j` será incrementado em `i+1` e, quando `k=-1` ,`j` será decrementado em `i+1`.

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

int main()
{
    int vetor[10]={11,22,33,44,55,66,77,88,99,1010};
    int vr[10]={};
    int j=10/2-1;
    //preenchimento de vr
    for(int i=0; i <=9;i++)
```



```
vetor   vr
11      99
22      77
33      55
44      33
55      11
66      22
77      44
88      66
99      88
1010    1010
Press any key to continue . . .
```

```
    {
        vr[j]=vetor[i];
        int k=-1;
            for(int l=0;l<=i;l++) k=k*(-1);
        j=j+(i+1)*k;
    }
    //Impressão dos vetores
    cout<<"vetor\tvr"<<endl;
    for(int i=0;i<=9;i++)
        cout<<vetor[i]<<"\t"<<vr[i]<<endl;

    system("pause");
    return 0;
}
```

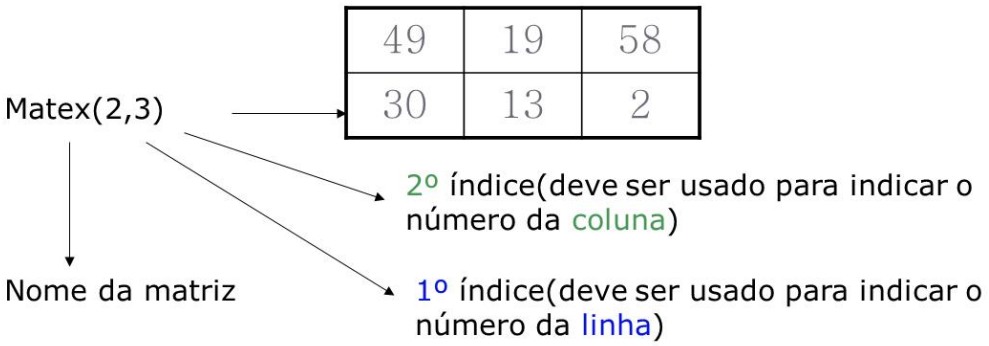
Em outra abstração, utilizaremos a função potência (`pow()`) da biblioteca `math.h`, para substituir o segundo `for` do programa anterior, implementando a expressão $j = j + (-1)^i \times (i+1)$.

```
#include <iostream>
#include <math.h>
using namespace std;

int main()
{
    int vetor[10]={11,22,33,44,55,66,77,88,99,1010};
    int vr[10]={};
    int j=10/2-1;
    //preenchimento de vr
    for(int i=0; i <=9;i++)
    {
        vr[j]=vetor[i];
        j=j+int(pow(-1,i))*(i+1);
    }
    //Impressão dos vetores
    cout<<"vetor\tvr"<<endl;
    for(int i=0;i<=9;i++)
        cout<<vetor[i]<<"\t"<<vr[i]<<endl;

    system("pause");
    return 0;
}
```

```
vetor   vr
11      99
22      77
33      55
44      33
55      11
66      22
77      44
88      66
99      88
1010    1010
Press any key to continue . . .
```

10. Estruturas Composta de Dados (Matrizes)

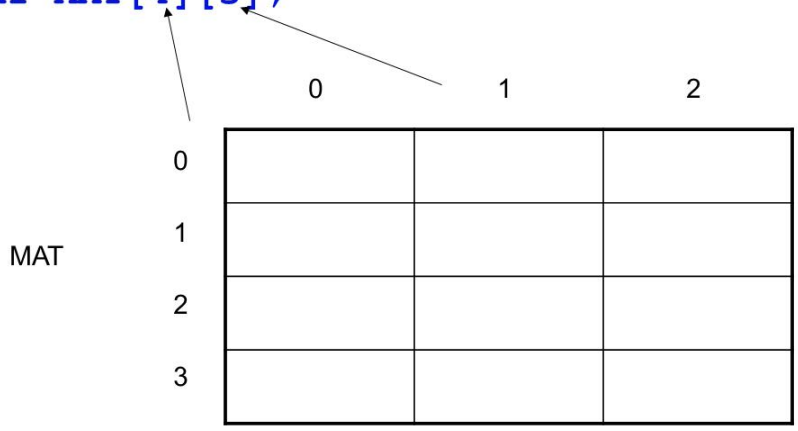
As estruturas ***multidimensionais*** são chamadas de ***matrizes***, podem ter duas ou mais dimensões e, assim como em vetores, apresenta o mesmo tipo de dados (estrutura homogênea), referenciados por n variáveis inteiras (índices) iguais ao número de dimensões da matriz, ou seja, cada uma referenciará uma dimensão da matriz.



10.1 Declaração

Para declararmos uma matriz de duas dimensões, por exemplo, utiliza-se dois pares de colchetes. Cada par de colchetes adicionais obtemos matrizes com uma dimensão a mais. A lista de valores usada para inicializar a matriz é uma lista de constantes separadas por vírgulas e envolta por chaves.

```
char MAT[4][3];
```




```
flot Y[2][4][3];
```

A declaração criou uma matriz chamada Y contendo:

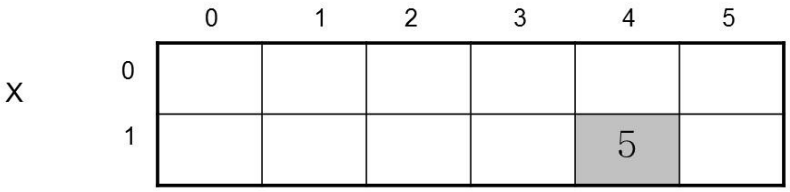
2 linha (0 a 1)

4 colunas (0 a 3)

3 profundidade (0 a 2)

Atribuindo valor a uma matriz:

`X[1][4]=5;` Atribui o valor 5 à posição identificada pelos índices 1 (2ª linha) e 4 (5ª coluna).



Exemplo:

```
int soma[3][3];
float nota[2][3]={{9,8,7},{6.5,4,2}};
```

Da mesma maneira como, ocorre com os vetores, os índices começam sempre em 0 (zero).

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

int main()
{
    float soma[3][3]={};
    float nota[2][3]={{9,8,7},{6.5,4,2}};

    //impressao da matriz soma
    cout<<"Matriz soma: "<<endl;
    for(int i=0; i<=(3-1);i++)
    {
        for(int j=0; j<=(3-1); j++)
            cout<< soma[i][j] << "\t";
        cout << endl;
    }
    cout <<endl;

    //impressão da matriz nota
    cout<<"Matriz nota: "<<endl;
    for(int i=0; i<=(2-1);i++)
    {
        for(int j=0; j<=(3-1); j++)
            cout<< nota[i][j] << "\t";
        cout << endl;
    }
    cout <<endl;
    system("pause");
    return 0;
}
```

```
Matriz soma:
0       0       0
0       0       0
0       0       0

Matriz nota:
9       8       7
6.5     4       2

Press any key to continue...
```

Observe o próximo exemplo, onde vamos declarar, atribuir valores e imprimir uma matriz com 3 dimensões:

```
#include <iostream>
using namespace std;
int main()
{
    int m1[3][4][5]={
        {
            {3,2,3,2,4},
            {6,5,3,6,4},
            {6,5,4,8,9},
            {6,6,6,6,6}
        },
        {
            {6,6,6,6,6},
            {4,5,3,7,6},
            {6,7,5,1,6},
            {9,6,5,6,4}
        },
        {
            {1,1,1,1,1},
            {2,2,2,2,2},
            {3,4,3,2,6},
            {7,8,6,7,8}
        }
    };
```

```
[0]
   3   2   3   2   4
   6   5   3   6   4
   6   5   4   8   9
   6   6   6   6   6

[1]
   6   6   6   6   6
   4   5   3   7   6
   6   7   5   1   6
   9   6   5   6   4

[2]
   1   1   1   1   1
   2   2   2   2   2
   3   4   3   2   6
   7   8   6   7   8

Press any key to continue . . .
```

```
    //imprimindo a matriz de 3 dimensoes
    for(int k = 0; k<=3-1;k++)
    {
        cout <<"["<< k << "]"<< endl;
        for(int i=0; i<=4-1;i++)
        {
            for(int j=0;j<=5-1;j++)
                cout<<"   "<<m1[k][i][j];
            cout<<endl;
        }
    }
    cout<<endl;
    system("pause");
    return 0;
}
```

10.2 Exemplo 18

Construa um programa, que efetue a soma e a impressão do resultado entre a multiplicação de duas matrizes de números inteiros com 9 elementos:

```
#include <iostream>

using namespace std;

int main()
{
    int MAT1[3][3]={
        {2,4,6},
        {1,3,5},
        {7,8,9}
    };
    int MAT2[3][3]={
        {1,3,5},
        {7,8,9},
        {2,4,6}
    };

    //somando as matrizes
    int R[3][3];
    for(int i=0; i<=(3-1);i++)
        for(int j=0; j<=(3-1); j++)
            R[i][j]=MAT1[i][j]+MAT2[i][j];
    for(int i=0; i<=(3-1);i++)
    {
        for(int j=0; j<=(3-1); j++)
            cout<< R[i][j] << "\t";
        cout << endl;
    }
    cout <<endl;

    system("pause");
    return 0;
}
```

10.3 Exemplo 19

Construa um programa que multiplique as Matrizes A e B do tipo (3 x 3) e guardando e imprimindo o resultado em uma matriz R, ou seja R = A*B. Lembre-se:

$$\begin{pmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{pmatrix} + \begin{pmatrix} B_{11} & B_{12} & B_{13} \\ B_{21} & B_{22} & B_{23} \\ B_{31} & B_{32} & B_{33} \end{pmatrix}$$

O método de multiplicação de matrizes deve ser:

$$R_{11} = A_{11} * B_{11} + A_{12} * B_{21} + A_{13} * B_{31}$$
$$R_{12} = A_{11} * B_{12} + A_{12} * B_{22} + A_{13} * B_{32}$$
$$R_{13} = A_{11} * B_{13} + A_{12} * B_{23} + A_{13} * B_{33}$$
$$R_{21} = A_{21} * B_{11} + A_{22} * B_{21} + A_{23} * B_{31}$$
$$R_{22} = A_{21} * B_{12} + A_{22} * B_{22} + A_{23} * B_{32}$$
$$R_{23} = A_{21} * B_{13} + A_{22} * B_{23} + A_{23} * B_{33}$$
$$R_{31} = A_{31} * B_{11} + A_{32} * B_{21} + A_{33} * B_{31}$$
$$R_{32} = A_{31} * B_{12} + A_{32} * B_{22} + A_{33} * B_{32}$$
$$R_{33} = A_{31} * B_{13} + A_{32} * B_{23} + A_{33} * B_{33}$$

Assim, o programa ficará:

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

int main()
{
    int MAT1[3][3]={{2,4,6},{1,3,5},{7,8,9}};
    int MAT2[3][3]={{1,3,5},{7,8,9},{2,4,6}};
    int R[3][3];
    for(int i=0; i<=(3-1);i++)
        for(int j=0; j<=(3-1); j++)
        {
            R[i][j]=0;
            for(int k=0; k<=(3-1);k++)
                R[i][j]+=MAT1[i][k]*MAT2[k][j];
        }
    //Imprimindo a matriz resposta
    for(int i=0; i<=(3-1);i++)
    {
        for(int j=0; j<=(3-1); j++)
            cout<< R[i][j] << "\t";
        cout << endl;
    }
    cout <<endl;

    //system("pause");
    return 0;
}
```

```
42      62      82
32      47      62
81      121     161
```

10.4 Exemplo 20

O tempo em horas que um determinado avião dispensa para percorre o trecho entre duas localidades distintas esta disponível na seguinte tabela:

	1	2	3	4	5	6	7
1		2	11	6	15	11	1
2	2		7	12	4	2	15
3	11	7		11	8	3	13
4	6	12	11		10	2	1
5	15	4	8	10		5	13
6	11	2	3	2	5		14
7	1	15	13	1	13	14	

Construa um programa em C++ que, a partir da tabela anterior informe ao usuário o tempo necessário para percorrer duas cidades escolhidas em um menu, até o momento em que ele fornecer duas cidades iguais (origem destino).

Menu de cidades:

Cidade	Código
Metropolis	1
Pasargada	2
Gotham	3
Springfield	4
Arraial dos Tucanos	5
Tangamandapio	6
Sunset Valley	7

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

int main()
{
    int o=1, d=2;

    int tabela[7][7] =
    {
        {0,  2,    11,   6,    15,   11,   1},
        {2,  0,    7,    12,   4,    2,    15},
        {11, 7,    0,    11,   8,    3,    13},
        {6,  12,   11,   0,    10,   2,    1},
        {15, 4,    8,    10,   0,    5,    13},
        {11, 2,    3,    2,    5,    0,    14},
        {1,  15,   13,   1,    13,   14,   0}
    };

    for(int i=1; i<=27; i++) cout << "*";cout<<endl;
    cout << "*** Aerolineas Tabajara ***"<<endl;
    for(int i=1; i<=27; i++) cout << "*";cout<<endl<<endl;

    do
    {
        for(int i=1; i<=27; i++) cout << "=";cout<<endl;
        cout << "Cidade               Codigo"<<endl;
        for(int i=1; i<=27; i++) cout << "-";cout<<endl;
        cout << "Metropolis.............1"<<endl;
        cout << "Pasargada..............2"<<endl;
        cout << "Gotham.................3"<<endl;
        cout << "Springfield............4"<<endl;
        cout << "Arraial dos Tucanos....5"<<endl;
```

```
        cout << "Tangamandapio..........6"<<endl;
        cout << "Sunset Valley..........7"<<endl;
        for(int i=1; i<=27; i++) cout << "=";cout<<endl;
        cout << "Entre com o codigo da cidade de origem: "; cin >> o;
        cout << "Entre com o codigo da cidade de destino: "; cin >> d;
        if(o!=d && o>0 && o<8 && d>0 && d<8)
            cout << "Tempo de viagem igual a " << tabela[o-1][d-1]<<" horas\n\n";

    }while(o!=d && o>0 && o<8 && d>0 && d<8);
    system("pause");
    return 0;
}
```

Desenvolva um programa que permita ao usuário informar várias cidade, até inserir uma cidade 0 e que imprima o tempo total para cumprir todo o percurso específico entre as cidades fornecidas.

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

int main()
{
    int o=1, d=2, cont=0, soma=0;
    int tabela[7][7] =
    {
        {0,  2,    11,   6,    15,   11,   1},
        {2,  0,    7,    12,   4,    2,    15},
        {11, 7,    0,    11,   8,    3,    13},
        {6,  12,   11,   0,    10,   2,    1},
        {15, 4,    8,    10,   0,    5,    13},
        {11, 2,    3,    2,    5,    0,    14},
        {1,  15,   13,   1,    13,   14,   0}
    };

    for(int i=1; i<=27; i++) cout << "*";cout<<endl;
    cout << "*** Aerolineas Tabajara ***"<<endl;
    for(int i=1; i<=27; i++) cout << "*";cout<<endl<<endl;

    for(int i=1; i<=27; i++) cout << "=";cout<<endl;
    cout << "Cidade               Codigo"<<endl;
    for(int i=1; i<=27; i++) cout << "-";cout<<endl;
    cout << "Metropolis.............1"<<endl;
    cout << "Pasargada..............2"<<endl;
    cout << "Gotham.................3"<<endl;
    cout << "Springfield............4"<<endl;
    cout << "Arraial dos Tucanos....5"<<endl;
    cout << "Tangamandapio..........6"<<endl;
    cout << "Sunset Valley..........7"<<endl;
    for(int i=1; i<=27; i++) cout << "=";cout<<endl;
    cout << "Entre com o codigo da cidade "<< cont+1<<": "; cin >> o;

    do
    {
        cont++;
        cout << "Entre com o codigo da cidade  "<<cont+1<<": "; cin >> d;
        if(o!=0 && d!=0)
            soma+= tabela[o-1][d-1];
        o=d;
    }while(o!=0 && d!=0);
     cout << "Tempo total de viagem com " << cont-2<< " paradas sera de "<< soma <<"
horas\n\n";
    system("pause");
    return 0;
}
```

Escreva um programa que auxilie um usuário a escolher um roteiro de férias, sendo que o usuário fornece quatro cidades, a primeira é a sua origem, a última o seu destino obrigatório e as outras duas caracterizam as cidades alternativas de descanso (no meio da viagem). Por isso, o algoritmo deve fornecer ao usuário qual das duas é a melhor opção, ou seja, qual fará com que a duração das duas viagens (origem para descanso, descanso para destino) seja a menor possível.

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

int main()
{
    int o=0, d=0, a1=0,a2=0, soma1=0, soma2=0;
    int tabela[7][7] =
    {
        {0,  2,    11,   6,    15,   11,   1},
        {2,  0,    7,    12,   4,    2,    15},
        {11, 7,    0,    11,   8,    3,    13},
        {6,  12,   11,   0,    10,   2,    1},
        {15, 4,    8,    10,   0,    5,    13},
        {11, 2,    3,    2,    5,    0,    14},
        {1,  15,   13,   1,    13,   14,   0}
    };

    for(int i=1; i<=27; i++) cout << "*";cout<<endl;
    cout << "*** Aerolineas Tabajara ***"<<endl;
    for(int i=1; i<=27; i++) cout << "*";cout<<endl<<endl;

    for(int i=1; i<=27; i++) cout << "=";cout<<endl;
    cout << "Cidade               Codigo"<<endl;
    for(int i=1; i<=27; i++) cout << "-";cout<<endl;
    cout << "Metropolis.............1"<<endl;
    cout << "Pasargada..............2"<<endl;
    cout << "Gotham.................3"<<endl;
    cout << "Springfield............4"<<endl;
    cout << "Arraial dos Tucanos....5"<<endl;
    cout << "Tangamandapio..........6"<<endl;
    cout << "Sunset Valley..........7"<<endl;
    for(int i=1; i<=27; i++) cout << "=";cout<<endl;
    cout << "Entre com o codigo da cidade de origem: "; cin >> o;
    cout << "Entre com o codigo da cidade alternativa 1: "; cin >> a1;
    cout << "Entre com o codigo da cidade alternativa 2: "; cin >> a2;
    cout << "Entre com o codigo da cidade de destino: "; cin >> d;

    soma1= tabela[o-1][a1-1]+tabela[a1-1][d-1];
    soma2= tabela[o-1][a2-1]+tabela[a2-1][d-1];
    if(soma1<soma2) cout << "Melhor rota: "<<o<<"-"<<a1<<"-"<< d <<" = "<< soma1 << "
horas\n\n";
    else cout << "Melhor rota: "<<o<<"-"<<a2<<"-"<< d <<" = "<< soma2 << " horas\n\n";
    system("pause");
    return 0;
}
```

11. Modularização

11.1 Introdução e Histórico

No fim da década de 60, alguns problemas no desenvolvimento de sistemas de programação levaram os países desenvolvidos a um evento chamado "crise de software".

Os custos das atividades de programação mostravam a cada ano uma clara tendência a se elevarem muito em relação aos custos dos equipamentos, e isto era devido ao avanço tecnológico na fabricação dos equipamentos de computação e a lenta evolução de técnicas aplicadas ao desenvolvimento de software. A ausência de uma metodologia para a construção de programas conduzia a programas geralmente cheios de erros e com altos custos de desenvolvimento que, consequentemente, exigiam custos elevados para a sua correção e manutenção futuras.

A programação estruturada foi o resultado de uma serie de estudos e propostas de metodologias para desenvolvimento de software. Uma das técnicas aplicadas na programação estruturada, a **modularização** de programas e uma ferramenta para a elaboração de programas visando, os aspectos de confiabilidade, legibilidade, manutenibilidade e flexibilidade, dentre outros.

A **modularização** e um processo que aborda os aspectos da decomposição de algoritmos em módulos. **Módulo** e um grupo de comandos**,** constituindo um trecho do algoritmo, com uma função bem definida e o mais independente possível em relação ao resto do algoritmo.

11.2 Modularização em pseudocódigo

Seja um algoritmo para calcular o salário liquido de um empregado, com as seguintes etapas:

início
  Leia os dados do empregado
  Determine o salário
  Escreva o salário
fim.

Onde "Determine o salário" pode ser refinado como:

- **Calcule as vantagens**
- **Calcule as deduções**

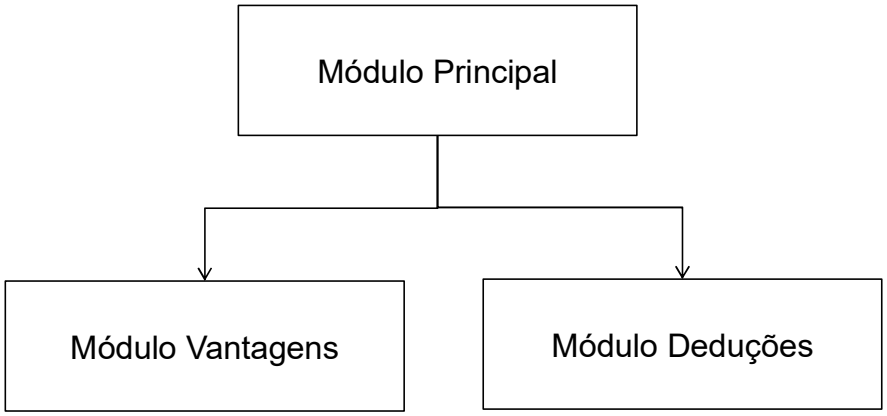
SALARIOLIQ ← VANTAGENS – DEDUCOES

No refinamento **sem modularização** não houve preocupação de como o processo de cálculo das vantagens e deduções seria efetuado. Essas ações constituem funções bem definidas e que serão executadas por módulos específicos, neste caso, o algoritmo anterior ficaria:

início
  Leia os dados do empregado
   Ative o **módulo** "Calculo das vantagens"
   Ative o **módulo** "Calculo das deduções"

SALARIOLIQ ← VANTAGENS – DEDUCOES
Escreva o salário
fim.

Exemplo da descrição estrutural da modularização:



A maneira mais intuitiva de proceder a modularização de problemas e feita definindo um ***módulo principal de controle*** e ***módulos específicos para as funções do algoritmo***.

Recomenda-se que os módulos de um programa tenham um ***tamanho limitado,*** pois módulos muito grandes são difíceis de ser compreendidos e, em geral, são multifuncionais.

LEMA :

"Dividir para conquistar"
Nunca divida de menos e nunca divida de mais

As linguagens de programação dispõem de recursos que facilitam a construção e manipulação de módulos, permitindo não só a modularização dos comandos do programa, como também dos dados utilizados. ***Cada módulo*** pode definir as próprias ***estruturas de dados***, suficientes e necessárias apenas para atingir o ***objetivo final do módulo***. Todo módulo e constituído por uma sequência de comandos que operam sobre um conjunto de ***objetos***, que podem ser ***globais*** ou ***locais***.

- **Objetos globais** são entidades que podem ser usadas em módulos internos a outro modulo do algoritmo onde foram declaradas.
- **Objetos locais** são entidades que só podem ser usadas no módulo do algoritmo onde foram declaradas. Estes objetos não possuem nenhum significado fora deste módulo.
- São exemplos de objetos globais ou locais: variáveis, arquivos, outros módulos, etc.
- A comunicação entre módulos devera ser feita através de vínculos, utilizando-se objetos globais ou transferência de parâmetros.

Os benefícios da modularização são que a independência do módulo permite uma manutenção mais simples e evita efeitos colaterais no restante do algoritmo.

- A elaboração do modulo pode ser feita independentemente e em épocas diferentes do restante do algoritmo;
- Testes e correções dos módulos podem ser feitos separados;

- Um módulo pode ser utilizado em outros algoritmos que requeiram o mesmo processamento por ele executado.

11.3 Ferramentas

Sub-rotinas e **funções** são módulos que servem aos objetivos:

- ***Evitar que em certa sequência de comandos*** necessária em vários locais de um algoritmo tenha que ser escrita repetidamente nesses locais;
- ***Dividir*** e ***estruturar*** um algoritmo em partes fechadas e logicamente coerentes;
- Aumentar a ***legibilidade*** de um algoritmo.

Sub-rotinas e **funções** são módulos hierarquicamente subordinados a um algoritmo, comumente chamado de módulo principal. Da mesma forma uma sub-rotina ou uma função pode conter outras sub-rotinas e funções aninhadas.

A **sub-rotina** e a **função** são criadas através das suas declarações em um algoritmo e para serem executadas, necessitam de ativação por um comando de chamada. A declaração de uma sub-rotina ou função e constituída de um ***cabeçalho*****,** que a identifica e contem seu nome e uma lista de parâmetros formais, e de um ***corpo*** que contem declarações locais e os comandos.

Sub-rotinas
Criação de sub-rotina:

```
subrotina NOME (lista-de-parâmetros-formais)
           declarações dos objetos locais a sub-rotina
           comandos da sub-rotina
fim subrotina;
```

Chamada da sub-rotina:

```
       NOME (lista-de-parâmetros-atuais);
```

Funções

As funções têm a característica de ***retornar*** ao algoritmo que as chamou ***um valor*** associado ao nome da função.

- Criação de função:

```
função tipo NOME (lista-de-parâmetros-formais)
     declarações dos objetos locais a função
     comandos da função
     retorno valor
fim função;
```

- Chamada da função:

```
     NOME (lista-de-parâmetros-atuais);
```

Como esta função ira retornar (**retorno)** um valor, este pode ser atribuído a alguma variável, contanto que esta seja de tipo compatível.

```
A ← NOME (lista-de-parâmetros-atuais);
```

Ao terminar a execução dos comandos de uma sub-rotina ou função, o fluxo de controle retorna ao comando seguinte àquele que provocou a chamada.

11.4 Transferência de Parâmetros

Os parâmetros de uma sub-rotina ou função classificam-se em:

- **de entrada –** são aqueles que têm ***seus valores estabelecidos fora da sub-rotina*** ou função e ***não podem*** ser modificados dentro dela.
- **de saída –** são aqueles que têm seus ***valores estabelecidos dentro*** da sub-rotina ou função.
- **de entrada-saída** – são aqueles que têm seus ***valores estabelecidos fora*** da sub-rotina ou *função*, mas ***podem ter seus valores alterados dentro*** dela.

A ***vinculação*** entre módulos pode ser feita através da ***transferência*** ou ***passagem de parâmetros***, que associam parâmetros atuais (argumentos) com parâmetros formais. Dentre os modos de transferência de parâmetros, pode se destacar: a ***passagem por valor***, a ***passagem por resultado*** e a ***passagem por referência***.

Na ***passagem de parâmetros por valor***, as alterações feitas nos parâmetros formais, dentro da sub-rotina ou função, não se refletem nos parâmetros atuais. O valor do parâmetro atual e copiado no parâmetro formal, na chamada da sub-rotina ou função. Assim, quando a passagem e por valor significa que o parâmetro e **de entrada**.

Na ***passagem de parâmetros por resultado***, as alterações feitas nos parâmetros formais, na sub-rotina ou função, refletem-se nos parâmetros atuais. O valor do parâmetro formal é copiado no parâmetro atual, ao retornar da sub-rotina ou função. Assim, quando a passagem e por resultado significa que o ***parâmetro e de saída***.

Na ***passagem de parâmetros por referencia***, a toda alteração feita num parâmetro formal corresponde a mesma alteração feita no seu parâmetro atual associado. Neste caso, quando a passagem e por valor significa que o parâmetro e de ***entrada-saída***.

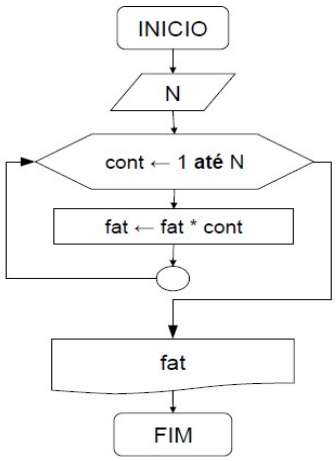
Exemplo: Leia um número inteiro e escreva o seu fatorial. Lembrando que o fatorial de um número e o produto de todos os números anteriores e inclusive ele.

5! = 1*2*3*4*5 = 120

0! = 1 (por definição)

Pseudocódigo sem modularização:

Inicio
{declaração de variáveis}
inteiro N, fat = 1, cont
{comandos de entrada de dados}
leia (N)
{processo e saída de dados*}*
para cont ← 1 até N faça
fat ← fat * cont

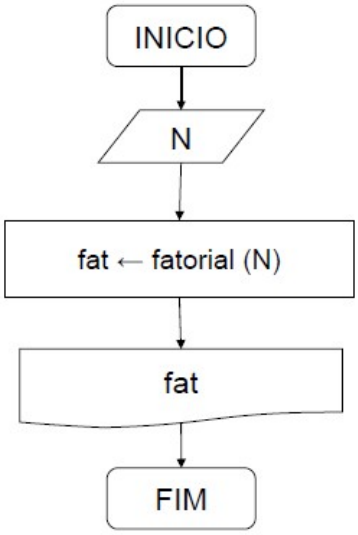


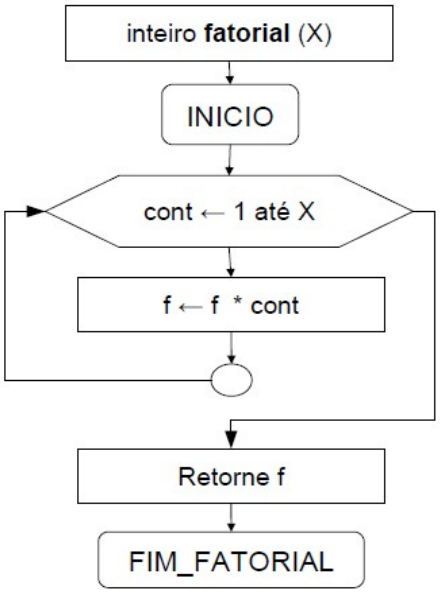
```
        fim-para
        {comandos de saída de dados}
        escreva(fat)
fim.
```

Pseudocódigo com modularização:

```
inicio
{declaração de variáveis}
    inteiro N, fat
{comandos de entrada de dados}
    leia (N)
{processo e saída de dados}
    fat ← fatorial(N)
{comandos de saída de dados}
    escreva (fat)
fim
```

```
inteiro fatorial (X)
inicio
{declaração de variáveis locais}
    inteiro cont, f
      para cont ← 1 até X faça
              f ← f * cont
      fim_para
      retorne f
fim_fatorial
```





11.5 Modularização em Sub-Rotinas

Sub-rotinas, também chamada de subprogramas, são ***blocos*** de instruções que realizam tarefas especificas. O ***código*** de uma sub-rotina é carregado ***uma vez*** e pode ser executado ***quantas vezes forem necessárias***. As principais vantagens da utilização se sub-

Rotinas são que como o problema pode ser subdividido em pequenas tarefas, os programas tendem a ficar ***menores*** e ***mais organizados***.

Os programas em geral são executados linearmente, uma linha após a outra, até o fim. Entretanto, quando são utilizadas sub-rotinas, é possível a realização de desvios na execução dos programas.

Sintaxe:

função ***nome*** ()
entradas: nono, nonono[], nono, ...
saídas: nono, nonono, nono[], ...)
inicio
{ comandos }
nome ← ...
fim

O exemplo abaixo mostra a utilização de uma **sub-rotina** que recebe um parâmetro (o valor atual do salário) e que, ao final, **retorna** um valor (**aumento que será dado ao salário**) para quem a chamou. Porém, as sub-rotinas podem não receber parâmetros nem retornar valor.

```
ALGORITMO
  DECLARE sal NUMERICO
  LEIA sal
  aum ← calculo(sal)
  novosal ← sal + aum
  ESCREVA "Novo salário é", novosal
FIM_ALGORITMO
```

Variáveis Globais

Programa Principal

```
SUB-ROTINA calculo (sal NUMERICO)
  DECLARE perc, valor NUMERICO
  LEIA perc
  Valor ← sal * perc / 100
  RETORNE valor
FIM_SUB_ROTINA
```

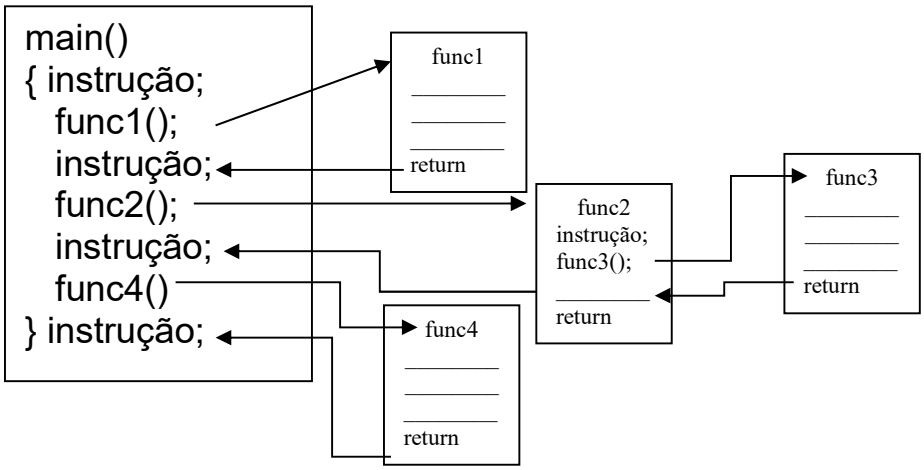
Variáveis Locais

Sub-Rotina

11.6 Modularização em C++ - Funções

Uma ***função*** é, então, um subprograma que pode atuar sobre dados e retornar um valor. Todo programa em C++ tem pelo menos uma função, `main()`. Quando seu programa inicia, `main()` é automaticamente chamada. `main()` pode chama outras funções algumas das quais podem, ainda, chamar outras.

Cada função tem o seu próprio nome, e quando o nome é encontrado, a execução do programa é desviada para o corpo dessa função. Quanto a função retorna, a execução recomeça na próxima linha da função que tiver feito a chamada, como ilustrado na figura abaixo.



Quando um programa chama uma função, a execução para para a função e, depois, recomeça na linhas localizada após a chamada da à função.

Como dito anteriormente, um programa em C/C++ tem no mínimo uma função de chamada `main`, por onde a execução começa. Existem as funções pré-definidas pelo compilador, que são incluídas pela diretiva `#include`.

Uma função é um conjunto de instruções desenhadas para cumprir determinado trabalho e agrupadas em uma unidade com um nome para referi-la. Vamos começar mostrando uma função que converte a temperatura de graus Fahrenheit para Celsius.

```
#include <iostream>
using namespace std;
int celsius(int fahr); // protótipo
int main()
{
   int c, f;
   cout << "Digite a temperatura em graus Fahrenheit: ";
   cin >> f;

   c = celsius(f); // chamada à função

   cout << "Celsius = " << c <<  endl;

   system("PAUSE");
   return 0;
}

int celsius(int fahr) // definição da função
{
    int c;
    c=(fahr-32)*5/9;
    return c;
}
```

Como podemos identificar, a estrutura de uma função C++ é semelhante à da função `main`(). A diferença é que a `main`() possui um nome especial. Os componentes necessários para adicionar uma função a um programa são: o protótipo da função, a chamada da função

e a sua definição. Uma função não pode ser chamada antes de ter sido declarada, sua declaração é dita **protótipo** da função, uma instrução colocada no início do programa (vide exemplo anterior), permitindo que o compilador verifique a sintaxe de sua chamada. Assim, a linha de código:

```
int celsius(int fahr); // protótipo
```

informa que a função de nome **celsius()** é do tipo **int** e recebe como argumento um valor **int** que será armazenado na variável **fahr**.

Em C++ as funções poderão ser escritas antes de sua instrução de chamada, nesse caso, seu **protótipo não será obrigatório** assim, nosso exemplo anterior poderia ser escrito da seguinte maneira, sem a necessidade de definirmos o protótipo da função no início do programa:

```
#include <iostream>
using namespace std;
int celsius(int fahr)
{
    int c;
    c=(fahr-32)*5/9;
    return c;
}

int main()
{
   int c, f;
   cout << "Digite a temperatura em graus Fahrenheit: ";
   cin >> f;

   c = celsius(f); // chamada à função

   cout << "Celsius = " << c <<  endl;

   system("PAUSE");
   return 0;
}
```

Entretanto, bons programadores escrevem os protótipos de todas as suas funções, pois no futuro elas poderão ser armazenadas em arquivos de bibliotecas e necessitarão dos protótipos.

12. Tipos de Funções

As que iremos trabalhar são funções definidas pelo programador, do tipo:

Sem passagem de parâmetro e **sem** retorno.
Com passagem de parâmetro e **sem** retorno.
Sem passagem de parâmetro e **com** retorno.
Com passagem de parâmetro e **com** retorno.

12.1 Sem passagem de parâmetro e sem retorno

É o tipo mais simples (**não recebe** e **não repassa** valor).

```
#include <cstdlib>
#include <iostream>
using namespace std;
void soma()
{
     int a,b,s;
     cout<<"Digite o primeiro numero ";     cin>>a;
     cout<<"\nDigite o segundo numero ";    cin>>b;
     s=a+b;
     cout<<"\nA soma e "<< s <<"\n";
     getchar();
}

int main()
{
    soma();
    system("PAUSE");
}
```

Obs.: no momento em que a função **soma** foi chamada no programa principal (`main`), nenhum valor foi colocado entre parênteses, indicando que não houve passagem de parâmetros. Além disso, não foi utilizado o comando **`return`**, sinalizando que ela não retornou valor para quem a chamou. Por está razão, seu tipo é **`void`**.

12.2 Com passagem de parâmetro e sem retorno

É representado por aquelas que **recebem** valores no momento em que são chamadas (parâmetros), mas que no final **não retornam** valor para quem as chamou (retorno).

```
#include <cstdlib>
#include <iostream>
using namespace std;
void calcula_media( float numero1, float numero2)
{
     float media;
     media = (numero1+numero2)/2;
     cout << "\nA media e = " << media << endl;
     getchar();
}

int main(int argc, char *argv[])
{
     float n1, n2;
     cout<<"Digite o primeito numero: ";
     cin>>n1;
     cout<<"\nDigite o segundo numero: ";
     cin>>n2;
     calcula_media(n1, n2);
     system("PAUSE");
     return 0;
}
```

Obs.: no momento em que a função `calcula_media` foi chamada, duas variáveis foram colocadas entre parênteses, indicando que houve passagem de parâmetros.

Os valores são copiados para as variáveis `numero1` e `numero2`, descritas no cabeçalho da função. Além disso, não foi utilizado o comando `return`, sinalizando que ela não retornou valor para quem a chamou.

Por está razão, seu tipo é `void`.

12.3 Sem passagem de parâmetro e com retorno

É representado por aquelas que **não recebem** valores no momento em que são chamadas (parâmetros), mas que, no final, **retornam** valor para quem as chamou (retorno).

```
#include <cstdlib>
#include <iostream>
using namespace std;
float multiplicacao()
{
     float n1, n2, produto;
     cout<<"Digite o primeiro numero ";
     cin>>n1;
     cout<<"\nDigite o segundo numero ";
     cin>>n2;
     produto = n1*n2;
     return produto;
}

int main()
{
    float resposta;
    resposta = multiplicacao();
    cout<<"\nO resultado e : "<< resposta <<endl;
    system("PAUSE");
    return 0;
}
```

Obs.: no momento em que a função `multiplicacao` foi chamada, nenhum valor foi colocado entre parênteses, indicando que não houve passagem de parâmetros.

Chegando no final da função o comando `return` é encontrado. Isto indica que a execução da função chegou ao fim e que o conteúdo da variável `produto` será devolvido para quem a chamou.

Retornando ao programa principal, o valor é armazenado na variável `resposta`. Por está razão, seu tipo é `float`, exatamente igual ao tipo de valor retornado.

12.4 Com passagem de parâmetro e com retorno

É representado por aquelas que **recebem valores** no momento em que são chamadas (parâmetros), e no final, **retornam valor** para quem as chamou (retorno).

```
#include <cstdlib>
#include <iostream>
using namespace std;
int divisao(int d1, int d2)
{
     int q;
     q = d1/d2;
     return q;
}

int main(int argc, char *argv[])
{
    int  n1, n2;
    int resposta;
    cout<<"Digite o primeiro numero ";
```

```
    cin>>n1;
    cout<<"\nDigite o segundo numero ";
    cin>>n2;
    resposta = divisao(n1, n2);
    cout<<"\nO resultado e "<< resposta<<endl;
    system("PAUSE");
    return 0;
}
```

Obs.: no momento em que a função `divisao` foi chamada, duas variáveis foram colocadas entre parênteses, indicando que houve passagem de parâmetros.

Os valores são copiados para as variáveis `d1` e `d2`, descritas no cabeçalho da função.

Chegando no final da função, o comando `return` é encontrado. Isto indica que a execução da função chegou ao fim e que o conteúdo da variável `q` será devolvido para quem a chamou.

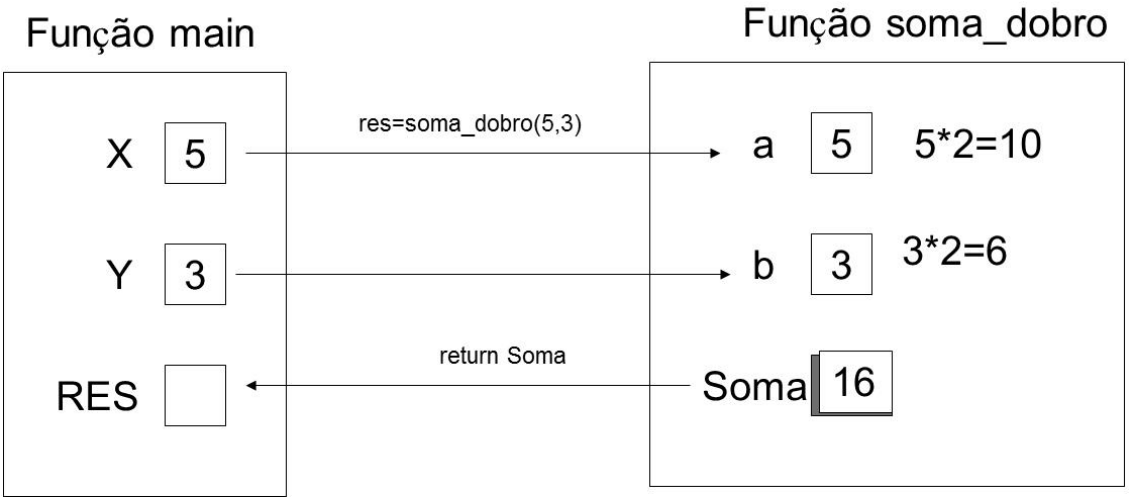
Retornando ao programa principal, o valor é armazenado na variável `resposta`.

Por está razão, seu tipo é `int`, exatamente igual ao tipo de valor retornado.

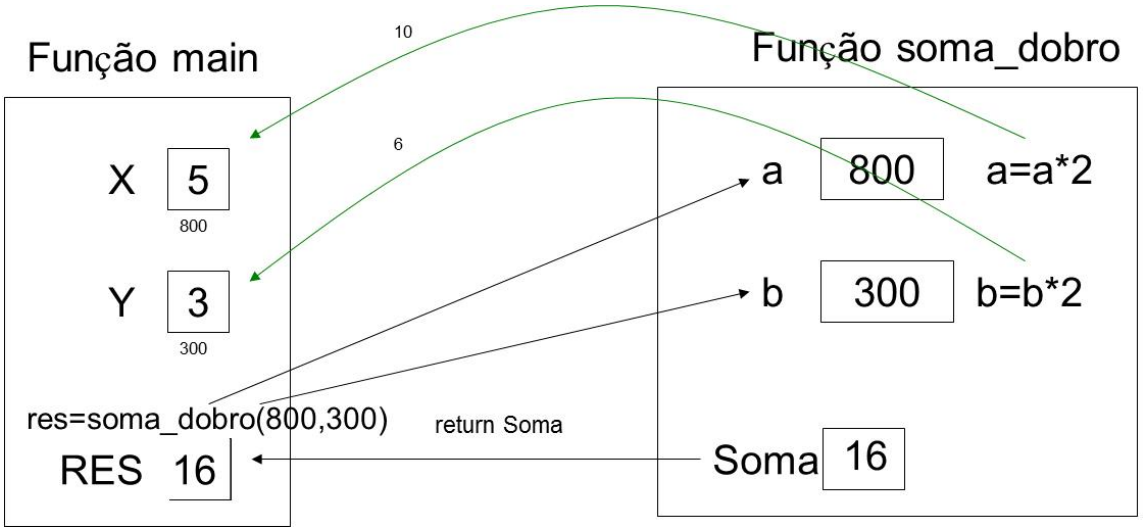
12.5 Tipos de passagem de parâmetro

Os tipos de passagem de parâmetros são por **valor** e por **referência**.

Valor: São todos os exemplo vistos até aqui. A figura abaixo ilustra a passagem de parâmetro por **Valor**:



Referência: Significa que os parâmetros são passados para uma função correspondente a **endereços de memória** ocupados por variáveis. Dessa maneira, toda vez que for necessário acessar determinado valor, isso será feito por meio de **referência**, ou seja, **apontando o seu endereço**. A figura abaixo ilustra a passagem de parâmetro por referência.



Quando argumentos são passados por **valor**, a função chamada cria variáveis e copia nelas o valor dos argumentos passados. Dessa forma, a função não tem acesso às variáveis originais da função que chamou, portanto, ela não poderá efetuar qualquer modificação nelas. Na passagem por referência, a função pode acessar as variáveis da função que as chamou. Esse mecanismo possibilita que uma função retorne mais de um valor para a função que chama. Os valores a serem retornados são colocados na referência de variáveis da função chamada.

O Operador Unário de Referência &: Com esse operador podemos montar novos tipos de dados, chamados de *referências*. Referência é um outro nome para uma variável já existente. As instruções:

```
Int n;
Int& A = n;
```

Informa que `A` é um outro nome para `n`. Toda a operação em qualquer um dos nomes tem o mesmo resultado. A referência não é uma cópia da variável que se refere, é a mesma variável sobre nomes diferentes.

```
#include <iostream>
using namespace std;
int main ()
{
  int n;
  int& A =n;
  n=5;
  cout << "O valor de A e = " << A << endl;
  A = 8;
  cout << "O valor de n e = " << n << endl;
  system("Pause");
  return 0;
}
```

```
O valor de A e = 5
O valor de n e = 8
Press any key to continue...
```

Obs.: Toda a referência deve ser obrigatoriamente inicializada

```
int n;
```

```
int& A; // Errado
int& A =n; // Certo
```

O próximo e exemplo mostra o uso simples de referência como argumento de uma função.

```
#include <iostream>
using namespace std;
void reajusta20(float& p, float& r);
int main ()
{
  float preco, val_reaj;
  do
  {
   cout << "Insira o preco atual ";
   cout << "ou 0 para terminar: ";
   cin >> preco;
   reajusta20(preco,val_reaj);
   cout << "Preco novo = " << preco;
   cout << endl;
   cout << "Aumento     = " << val_reaj;
   cout << "\n\n";
  }while(preco!=0.0);
  system("Pause");
  return 0;
}
void reajusta20(float& p, float& r)
{
     r=p*0.2;
     p=p*1.2;
}
```

```
Insira o preco atual ou 0 para terminar: 50
Preco novo = 60
Aumento     = 10

Insira o preco atual ou 0 para terminar: 0
Preco novo = 0
Aumento     = 0

Press any key to continue...
```

No exemplo a seguir, criaremos uma função que troca o conteúdo de duas variáveis, por meio da função **troca()** para ordenar uma lista de 3 números fornecidos pelo usuário. Utilize os números 50, 13 e 28 respectivamente para as variáveis `n1`, `n2` e `n3`.

```
#include <iostream>
using namespace std;
void troca(float& n, float& m);
int main ()
{
  float n1, n2, n3;
  cout << "Digite os tres numeros: "<<endl;
  cin >> n1 >> n2 >> n3;
  if (n1>n2) troca(n1,n2);
  if (n1>n3) troca(n1,n3);
  if (n2>n3) troca(n2,n3);
  cout << "n1= "<<n1<<endl;
  cout << "n2= "<<n2<<endl;
  cout << "n3= "<<n3<<endl;
  system("Pause");
  return 0;
}
void troca(float& n, float& m)
{
     float aux;
     aux=n;
     n=m;
     m=aux;
}
```

```
Digite os tres numeros:
50
13
28
n1= 13
n2= 28
n3= 50
Press any key to continue...
```

12.6 Valores default de parâmetros de funções

Quando declaramos uma função, podemos especificar o valor default de cada um de seus parâmetros. Este valor será usado se o argumento correspondente a esse parâmetro for deixado em branco no momento em que a função for chamada. Pra fazer isso, simplesmente utilizamos o operador de atribuição (=) e o valor para o argumento no protótipo da função. Assim, se o valor para o parâmetro não for passado quando a funções for chamada, o default será usado, entretanto, se um valor for especificado, este default será ignorado e o valor passado será utilizado.

```
#include <iostream>
using namespace std;
int divide(int a=2, int b=1);
int main()
{
    cout << divide() << endl;
    cout << divide(12) << endl;
    cout << divide(12,4)<< endl;
    system("pause");
    return 0;
}

int divide(int a, int b)
{
    int r;
    r=a/b;
    return (r);
}
```

```
2
12
3
Press any key to continue...
```

Como podemos ver, existem três chamadas à função `divide`(), na primeira:

```
divide()
```

não foram especificados argumentos, mas função permite a utilização de dois argumentos. Então, a função assume como valores de seus parâmetros: 2 para o primeiro e 1 para o segundo, como definido no protótipo da função. Portanto o resultado é 2 (2/1)

```
int divide(int a=2, int b=1);
```

Na segunda chamada à função, foi especificado somente o primeiro argumento da função:

```
divide(12)
```

Nesse caso, a função assume que o valor do segundo parâmetro é 1, uma vez que esse parâmetro não foi passado. Portanto o resultado retornado é 12 (12/1).

Na terceira chamada à função:

```
divide(12,4)
```

Os dois parâmetros foram fornecido, assim, os valores default para `a` (`int a=2`) e `b` (`int b=1`) foram ignorados, assim o retorno da função é 3 (12/4).

Obs.: Podemos escrever protótipos de funções que tenham parâmetros com um valor default e parâmetros sem valor default, mas, após a primeira especificação do valor default, todos os parâmetros seguintes devem ser especificados com valores defaults. Observe também que, nas chamadas às funções, se o primeiro argumento for omitido, todos os subsequentes deverão sê-lo. Se o segundo argumento for omitido, todos os subsequentes deverão sê-lo e assim por diante, portanto, a seguinte instrução é ilegal e retornará um erro:

```
divide(,4) // ERRADO
```

12.7 Sobrecarga de Funções

Sobrecarregar funções significa criar uma família de funções com o mesmo nome, mas com a lista de argumentos diferentes. Funções sobrecarregadas devem ter a lista de argumentos diferentes ou em número ou em tipo. Pois, é por meio dos argumentos que o sistema reconhece qual função deverá ser chamada.

```
#include <iostream>
using namespace std;
int opere (int a, int b)
{
  return(a*b);
}
float opere (float a, float b)
{
  return(a/b);
}
int main ()
{
  int x=5,y=2;
  float n=5.0, m=2.0;
  cout << opere (x,y);
  cout << "\n";
  cout << opere (n,m);
  cout << "\n";
  system("pause");
  return 0;
}
```

```
10
2.5
Press any key to continue...
```

No exemplo acima definimos duas funções com o mesmo nome, `opere`() mas uma delas aceita dois argumentos do tipo `int` e a outra aceita os argumentos do tipo `float`. O compilador sabe qual deve chamar em cada caso pela inspeção do tipo de valor passado como argumento quanto a função é chamada. Se ela for chamada com dois inteiros como seus argumentos será utilizada a função que possui dois inteiros como argumentos no protótipo. O mesmo raciocínio segue para o caso dos argumentos utilizados forem `floats`.

Observe que o sistema considera somente a lista de argumentos para escolher a função apropriada a cada chamada, e não o valor de retorno.

12.8 Recursividade de Funções

Recursividade é a propriedade que uma função tem de ser chamada por ela mesma, ou seja, uma função é dita ***recursiva*** se definida em termos dela mesma. Isto é útil em

muitas tarefas, como em ordenação ou cálculos de fatoriais de números. Por exemplo, para obtermos o fatorial de um número (n!) a fórmula matemática deve ser:

```
n! = n * (n-1) * (n-2) * (n-3) ... * 1
```

assim, observe o exemplo do fatorial de 5:

```
5! = 5 * 4 * 3 * 2 * 1 = 120
```

e a função recursiva para calcular o fatorial em C++ poderia ser escrita como:

```
#include <iostream>
using namespace std;
long fatorial(int a)
{
     if(a>1)
       return (a*fatorial(a-1));
     else
       return (1);
 }

int main ()
{
  long n;
  cout << "Digite um numero: ";
  cin >> n;
  cout << n <<"! = "<< fatorial(n) << endl;
  system("Pause");
  return 0;
}
```

```
Digite um numero: 5
5! = 120
Press any key to continue...
```

Note que na função `fatorial` nós incluímos uma chamada a ela mesma, mas somente se o argumento passado for maior que 1, caso contrário a função realizaria um loop recursivo infinito, pois, uma vez atingido o zero (`0`) a multiplicação continuaria por todos os números negativos o que provocaria um estouro de memória ou um erro no tempo de execução.

O código gerado por uma função recursiva exige a utilização de mais memória, o que torna a execução mais lenta. Ao escrevermos uma função recursiva, três pontos devem ser lembrados:

1. Definir o problema em termos recursivos, ou seja, definir o problema em termos dele mesmo.
2. Encontrar uma ***condição básica*** de término para evitarmos problemas de estouro de memória e erro de tempos de execução.
3. Cada vez que uma função é chamada recursivamente, deve estar mais próxima de satisfazer a condição básica, isso garante que o programa não girará em uma sequência infindável de chamadas.

Obs.: A linguagem C/C++ **não permite** que vetores e matriz sejam passados na **íntegra** como parâmetro para uma função. Para resolver esse problema, deve-se passar apenas o endereço da **posição inicial** do vetor ou da matriz. Esse endereço é obtido utilizando-se o nome do vetor (ou da matriz) **sem o índice** entre colchetes.

```
#include <iostream>
using namespace std;
void soma_linhas(int m[ ][5], int v[ ])
```

```
{
     int i, j;
     for (i=0;i<3;i++)
      for (j=0;j<5;j++)
        v[i]=v[i]+m[i][j];

}

int main()
{
    int  i, j;
    int mat[3][5], vet[3];
    for (i=0;i<3;i++)
    { vet[i]=0;
    system("cls");
    for (j=0;j<5;j++)
     {
      cout<<"\nDigite o elemento Coluna: "<<i+1<<" e Linha: "<<j+1 <<" ";
      cin >> mat[i][j];
     }
    }
    soma_linhas(mat, vet);
   for (i=0;i<3;i++)
      cout << "\nSoma da coluna " << i+1 << " = " << vet[i];
   cout << "\n\n";
   system("PAUSE");
   return 0;
}
```

12.9 Exemplo 21

Elabore um programa, utilizando modularização, que calcule a média ($\bar{x}$), a variância (s^2), o desvio padrão (s), o mínimo, o máximo e o coeficiente de variação (cv) das alturas dos atletas do exemplo 15, lembrando:

$$\bar{x}=\frac{\sum_{i=1}^{n} x_i}{n} \qquad s^2=\frac{\sum_{i=1}^{n} x_i^2-\frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n}}{n-1}$$

$$s=\sqrt{s^2}=\sqrt{\frac{\sum_{i=1}^{n} x_i^2-\frac{\left(\sum_{i=1}^{n} x_i\right)^2}{n}}{n-1}} \qquad cv=\frac{100\times s}{\bar{x}}$$

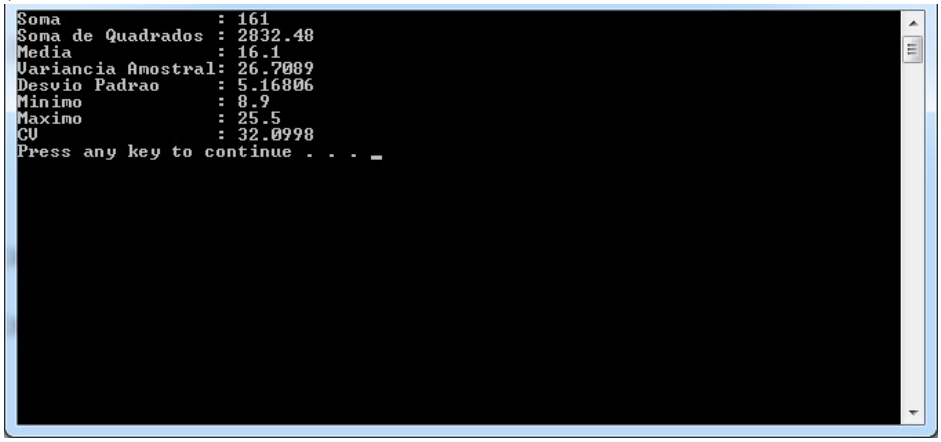
Onde n é o número de elementos do vetor e x é o vetor de alturas. Para auxiliar, programe as funções para calcular:

$$soma=\sum_{i=1}^{n} x_i \quad \text{e} \quad soma\ ao\ quadrado=\sum_{i=1}^{n} x_i^2$$

```
#include <iostream>
#include <math.h>
using namespace std;

//protótipos
```

```
float soma(float v[]);
float somaquad(float v[]);
float media(float v[]);
float variancia(float v[]);
float desvpad(float v[]);
float minimo(float v[]);
float maximo(float v[]);
float cv(float v[]);

//vairáveis globais
float dados[10] = {21.5, 25.5, 14.5, 15.8, 17.6, 8.9, 9.6, 18.6, 17, 12};
int n=10;

//função principal
int main()
{
    cout << "Soma              : " << soma(dados) << endl;
    cout << "Soma de Quadrados : " << somaquad(dados) << endl;
    cout << "Media             : " << media(dados) << endl;
    cout << "Variancia Amostral: " << variancia(dados) << endl;
    cout << "Desvio Padrao     : " << desvpad(dados) << endl;
    cout << "Minimo            : " << minimo(dados) << endl;
    cout << "Maximo            : " << maximo(dados) << endl;
    cout << "CV                : " << cv(dados) << endl;
    system("Pause");
    return 0;
}

//função que calcula a soma dos elementos de um vetor
float soma(float v[])
{
      float soma=0.0;
      for(int i=0; i<=9; i++)
              soma+=v[i]; // media = mdia + v[i]
      return soma;
}

//função que calcula a soma dos quadrados do vetor
float somaquad(float v[])
{
      float somaquad=0.0;
      for(int i=0; i <=9; i++)
              somaquad+=v[i]*v[i];
      return somaquad;
}
//função que calcula a média aretimética do vetor
float media(float v[])
{
      float media=0.0;
      for(int i=0; i<=9; i++)
              media+=v[i]; // media = mdia + v[i]
      media/=10; // media = media/5
      return media;
}

//função que calcula a variancia amostral dos dados
float variancia(float v[])
{
      float variancia;
      variancia=(somaquad(v) - soma(v)*soma(v)/10)/(10-1);
      return variancia;
}

//função que calcula o desvio padrao amostral dos dados
float desvpad(float v[])
{
      float desvpad;
      desvpad=sqrt(variancia(v));
      return desvpad;
}

//função que retorna o menor valor dos dados
```

```
float minimo(float v[])
{
	float minimo=v[1];
	for(int i =0; i <=n-1; i++)
		if(minimo>v[i]) minimo = v[i];
	return minimo;
}

//função que retorna o maior valor dos dados
float maximo(float v[])
{
	float maximo=v[1];
	for(int i =0; i <=n-1; i++)
		if(maximo<v[i]) maximo = v[i];
	return maximo;
}

//função que calcula o coeficiente de variação
float cv(float v[])
{
	float cv;
	cv=100*desvpad(v)/media(v);
	return cv;
}
```



12.10 Exemplo 22

Elabore um programa, utilizando modularização que calcule a moda de um vetor de elementos numéricos, lembre que a moda é o valor que ocorre com maior frequência no vetor. Portanto, podemos ter vetores, com uma (unimodal) ou mais moda (plurimodal), ou mesmo sem valor de moda (amodal). Teste o programa com os vetores:

V1={1,6,8,9,9,5,9,7,9,9,5,6,4}
V2={5,5,8,6,5,8,5,7,8,4,8}
V3={1,8,4,5,9,7,6,3,2,0}

```
#include <iostream>
using namespace std;
void moda(int v[], int n);

int v1[13]={1,6,8,9,9,5,9,7,9,9,5,6,4};
int v2[11]={5,5,8,6,5,8,5,7,8,4,8};
int v3[10]={1,8,4,5,9,7,6,3,2,0};

int main()
{
	cout << "\nModa do vetor 1" << endl;
	moda(v1,13);
	cout << "\nModa do vetor 2" << endl;
	moda(v2,11);
```

```
        cout << "\nModa do vetor 3" << endl;
        moda(v3,10);
        system("Pause");
        return 0;
}
// função para o cálculo da moda
void moda(int v[],int n)
{
     // ordenando o vetor
     float aux;
     for(int i=1;i<=n-1;i++)
        for(int j=n-1;j>=0;j--)
           if(v[j-1] >v[j])
           {
              aux=v[j-1];
              v[j-1]=v[j];
              v[j]=aux;
           }
        //contar quantas classes de elementos
        int cont=0;
        for(int i=0;i<=n-2;i++)
               if (v[i]!=v[i+1]) cont++;
        cont++; //somando 1 ao numero de classes
        //criando o vetor com o numero de elementos
        int vc[cont], hist[cont];
               //preenchendo o vetor contagem com os elementos se n for diferente de cont
        if(n==cont)
               cout << "Vetor amodal"<<endl;
        else
        {
               int j=0;
               for(int i=0;i<=n-2;i++)
                      if (v[i]!=v[i+1])
                      {
                             vc[j]=v[i];
                             j++;
                      }
               vc[j]=v[n-1];
               //contar os elementos das classes
               for(int j=0;j<=cont-1;j++)
               {
               int k=0;
               for(int i=0;i<=n-1;i++)
                      if(vc[j]==v[i]) k++;
               hist[j]=k;
               }
               // buscando a posição do maior elemento em hist
               int pm, max;
               max=hist[0];
               for(int j=0;j<=cont-1;j++)
                      if(max < hist[j]) max=hist[j] ;
               //retornando a moda, os elementos de vc cujo respectivo hist é igual ao max
               cout<< "Moda: \n";
               for(int j=0;j<=cont-1;j++)
                      if(hist[j]==max) cout<<vc[j]<<endl;
        }
}
```

13. Tópicos Relacionados

Alguns tópicos serão abordados com o objetivo de introduzir ao estudantes, alguns novos conceitos de programação.

13.1 Gerando números pseudo-aleatórios

O gerador de números pseudo-aleatórios é inicializado usando o argumento passado como semente na função `srand()` (`cstdlib`). Para cada valor diferente da semente utilizada em uma chamada, o gerador de números pseudo-aleatórios gerará uma sucessão diferente de resultados nas chamadas subsequentes para a função `rand()`. Duas inicializações diferentes com a mesma semente irá gerar a mesma sucessão de resultados em chamadas subsequentes para a função `rand()`. Para exemplificar, vamos preencher dois vetores A e B, ambos com 10 posições com números aleatórios tendo como semente `1235`, após isso, vamos imprimir lado a lado os elementos dos vetores.

```
#include <iostream>
#include <cstdlib>

using namespace std;

int main()
{
    int A[10], B[10];
    srand(1235);//semente aleatória
    //preenchendo A
    for(int i=0; i< 10; i++)
        A[i]=rand();

    //preenchendo B
    srand(1235); //semente aleatória
    for(int i=0; i< 10; i++)
        B[i]=rand();

    // imprissão dos vetores lado a lado
    cout<<"A\tB"<<endl;
    for(int i=0; i<10; i++)
        cout <<A[i]<<"\t"<<B[i]<<endl;
    system("pause");
    return 0;
}
```

```
A       B
4071    4071
10961   10961
30626   30626
53      53
602     602
11644   11644
18386   18386
21581   21581
20568   20568
7156    7156
```

A fim de gerar números aleatórios propriamente ditos, `srand` normalmente é inicializada com algum valor em tempo de execução distinto do computador, como o valor retornado pela função `time(NULL)` (biblioteca `ctime`). Este procedimento pode ser considerado suficiente para as necessidades de randomização mais triviais. Assim, execute várias vezes o código abaixo e observe as diferentes saídas.

```
#include <iostream>
#include <cstdlib>
#include <ctime>

using namespace std;
```

```
int main()
{
    int A[10];

    srand(time(NULL));//semente aleatória
    for(int i=0; i< 10; i++)
        A[i]=rand();
    cout<<"A"<<endl;
    for(int i=0; i<10; i++)
        cout <<A[i]<<endl;
    cout<<endl;
    system("pause");
    return 0;
}
```

A função `rand()` retorna um número integral pseudo-aleatório entre 0 e `RAND_MAX`, que, por sua vez, é uma constante definida em `cstblib`. Portanto, o range de valores para `rand()` é de 0 a 32767. Portanto, para gerar números pseudo-aleatórios, entre quaisquer outros valores recomendamos a utilização de:

```
(int)(((float)rand()/RAND_MAX)*(LS-LI+1) + LI)
```

Onde LS é o valor do limite superior e LI é o valor do limite inferior, por exemplo, vamos gerar 10 números aleatórios entre 100 e 150.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main()
{
        srand(1235);
        for(int i=0; i<10;i++)

        cout<<(int)(((float)rand()/RAND_MAX)*(150-
100+1) + 100)<<endl;
        system("pause");
        return 0;
}
```

```
106
117
147
100
100
118
128
133
132
111
Press any key to continue . . .
```

13.2 Salvando dados em um arquivo

Uma vez gerados os números pseudo-aleatórios, vamos guarda-los em um arquivo denominado "aleatorio.txt". Para isso, precisamos dos recursos da biblioteca `fstream` para entrada e saída de dados em arquivos (files). Inicialmente devemos incluir a biblioteca no cabeçalho do programa e, posteriormente, definir um objeto dessa classe, que denominaremos arquivo. Esse objeto será responsável por construir, abrir e fechar o arquivo "aleatorio.txt". O arquivo será salvo, por padrão, na mesma pasta do projeto atual. Observe o programa abaixo, onde guardaremos os números aleatórios de 100 a 150.

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

int main()
{
        ofstream arquivo; //criando um objeto da classe ofstream
```

```
	arquivo.open("aleatorio.txt"); // construindo/abrindo o arquivo aleatorio.txt
	srand(1235); //definindo a semente aleatória
	for(int i=0; i<10;i++)
		arquivo << (int)(((float) rand()/RAND_MAX)*(150-100+1)+100) << endl;
	arquivo.close();//fechando o arquivo aleatorio
	system("pause");
	return 0;
}
```

Acesse a pasta do seu projeto, e verifique se o arquivo foi criado e se os dados foram salvos, como apresentado abaixo.