# Cartela de C++

## Um programa C++ (baskara)

```
#include<iostream>
#include<cmath>
using namespace std;
int main(){
  float a,b,c,delta,x1,x2;
  cout<<"Informe coeficientes ";
  cin>>a>>b>>c;
  delta=(b*b)-4*a*c;
  if (delta>=0) {
    x1=((-b)+sqrt(delta))/(2*a);
    x2=((-b)+sqrt(delta))/(2*a);
    cout<<" x1="<<x1<<;
    cout<<"     x2="<<x2<<endl;
  }
  else {
    cout<<"imaginarias "<<endl;
  }
}
```

## Processo

1. Escrever o código fonte (texto plano) usando um editor de texto. Sugestão: editor Codeblocks. Seu nome: aaaa.cpp
2. Compilar o código fonte.
3. Se houver erros de síntaxe, o compilador avisará. Corrigir e retornar.
4. Se a compilação funcionar, o compilador gerará o módulo aaaa.exe
5. Executar o aaaa.exe. CRITICAR o resultado. Havendo erros, voltar.
6. sucesso !

## Comandos de pré-compilador

Todos começam com o caracter `#` e aparecem na cor verde.

`#include<iostream>` Carrega o módulo que contém os comandos `cin` e `cout`.

`#include<cmath>` Carrega o módulo matemático.

`#define` Permite definir constantes no programa. Por exemplo

```
#define PI 3.1415
```

Note que o comando `#define` cria uma equivalência que é integralmente substituída ANTES de começar a compilação. Assim, se você definir `#define EULER 2.7182`, cada vez que mencionar `EULER` no código, esta menção será substituída pelo valor `2.7182` antes de começar a compilação. A palavra EULER vai desaparecer do código compilado sem deixar rastro.

## Cabeçalho

```
#include{iostream}
using namespace std;
```

a declaração using namespace std significa que os nomes de `cin` e `cout` não serão compostos. É uma maneira de deixar o código "limpo".

## Comentário

Textos livres colocados dentro do código para serem vistos por olhos humanos.

```
\\ autor P. Kantek jul/87
/* programa com características
que fogem da trivialidade   */
int main() {  // função principal
```

Note que na última linha acima, há uma parte que não é comentário (`int main() {`) e depois há um comentário (`// função principal`).

## Entrada

O comando `cin` faz diversas coisas:

1. interrompe o programa
2. espera o operador digitar algum conteúdo (se encerra por enter ou por um espaço em branco)
3. transfere o que foi digitado para a variável citada ao lado do `cin`
4. prossegue o programa na próxima instrução após o `cin`.

Note que a operação será conduzida na entrada padrão do programa, por definição o teclado do computador onde o programa está rodando.

Veja-se o exemplo

```
int a,b;
cin>>a>>b;
```

As variáveis `a` e `b` serão preenchidas com o que for digitado (nessa ordem).

## Leitura de caracteres

Para ler uma frase (mais de uma palavra), você precisa

```
#include<iostream>
#include<string>
... string frase;
// cin>>frase;
// para no primeiro espaço
  getline(cin, frase);
  fin=frase.length()-1;
```

## Saída

A saída ocorre quando se usa o comando `cout`. Note a inversão das flechas em relação ao `cin`. Aqui não há interrupção do programa. Note que se for a última instrução do programa, deve-se providenciar uma interrupção proposital a fim de que o operador do programa possa ler o que foi impresso. Para isso inclua um `system ("Pause");` antes do return 0;. Providencie um `#include <cstdlib>` no início do código.

A saída é feita na saida padrão que usualmente é o monitor de vídeo principal do computador onde o programa está rodando. Veja-se um exemplo

```
int x,y=0;
x=99;
cout<<"O valor eh "<<x;
```

Durante a operação de `cout`, o cursor fica parado ao final do conteúdo exibido. Para comandar um salto de linha, deve-se associar ao conteúdo a constante `endl`. Veja no exemplo:

```
cout<<a<<" "<<b;
cout<<a<<" "<<b<<endl;
```

No primeiro caso, o cursor para ao lado de `b`. No segundo, na linha de baixo. Quando duas variáveis são impressas (como em `cout<<a<<b`) seus dois valores são apresentados sem nenhuma separação entre eles. Então, no exemplo se vale 23 e b vale 7, a apresentação será 237. Para evitar este problema acostume-se a imprimir espaços entre as variáveis, como em `cout<<a<<" "<<b`.

## Definição de variáveis

Toda variável precisa ser definida ANTES de ser mencionada pelo programa. Por essa razão costuma-se começar um programa pelas definições de variáveis que serão usadas. Toda variável tem um nome, um tipo e eventualmente um tamanho (que muitas vezes é pré-determinado).

### Regras de nomes

1. única palavra (espaços em branco não são permitidos)
2. letras, números e o caracter sublinha (_)
3. começa obrigatoriamente por uma letra
4. a caixa faz diferença, assim Oba ≠ OBA ≠ oba ≠ oBa ...
5. não se devem usar caracteres acentuados
6. o tamanho do nome fica a critério do programador

### Tipos

Alguns tipos possíveis:

**int** variáveis inteiras, com sinal e com a característica da contabilidade.

**float** variáveis reais. São armazenadas na forma $\pm mantissa \times 2^{\pm expoente}$. A mantissa sempre descreve um número entre 0 e 1, enquanto o expoente desloca livremente o ponto decimal por qualquer parte do número. Deve-se notar que nas proximidades de 0, os saltos que a variável admite são pequenos, enquanto que nos limites superior e inferior do expoente, o salto é muito maior.

**char** um único caracter cujo valor deve ser escrito entre aspas simples

**string** um conjunto de caracteres cujo valor deve estar escrito entre aspas duplas

**bool** um valor booleano. Na prática é tratado como uma variável inteira e significa FALSO quando ela vale zero e VERDADEIRO quando vale qualquer valor diferente de zero.

| tipo | tam. | faixa |
|---|---|---|
| int | 4 | ±2.147.483.647 |
| float | 4 | ±1.5E-45 a 3.4E38 |
| double | 8 | ±5E-324 a 1.7E308 |
| char | 1 | -128 a 127 |

## long e double

São modificadores do tamanho para variáveis numéricas... Acompanhe alguns exemplos

```
int a, alfa, residuo;
float x1, raiz;
char a = 'v';
string nome = "Curitiba / Paraná";
```

Uma variável PRECISA ser inicializada ANTES de ser usada. Você não pode supor que uma variável numérica recém definida conterá, por exemplo, zero. Ela pode conter qualquer coisa. Por essa razão, costuma-se inicializar todas as variáveis de uma de duas maneiras: pela sua leitura ou pela colocação do elemento neutro na operação fundamental que a variável vai sofrer, via operação de atribuição. Então um contador recebe zero, um produto recebe 1, um avaliador de maior recebe $-\infty$, um avaliador de menor recebe $+\infty$ e assim por diante.

## Operações aritméticas

`+` adição. Por exemplo `a=b+c;`

`-` subtração entre duas variáveis, p. ex: `a=b-c;`. Quando usado junto a uma única variável, este símbolo significa oposto, comandando a troca do sinal do operando. P. ex: `x=-y;`

`*` multiplicação, por exemplo `x=x*y;`.

`/` divisão inteira quando os dois operandos e o resultado são definidos como inteiros.

`/` divisão real quando o resultado é definido como float. Se for uma expressão solta, o resultado dependerá do tipo dos operandos envolvidos.

`%` resto da divisão, disponível quando todos os envolvidos forem inteiros. Se não forem ocorre um erro.

Exemplos:

```
2/3;  // resultado é 0
2.0/3;  // resultado é 0.666666
int a=2, b=3;
a/b;   //  vale 0
float a=2, b=3;
a/b;   //  vale 0.666666
int f=10, g=7;
f%g;   // vale 3
float j=10, k=7;
j%k;   // erro sintático
```

Note que a linguagem C++ (assim como todas as demais linguagens de programação) tem uma tabela ou uma regra de prioridade na interpretação de operações aritméticas. Por exemplo `2*1+3` em C++ é 5, mas em APL é 8. Em resumo, a tal precedência em C++ é: 1. parênteses, 2. oposto (menos unário), 3. multiplicação, divisão e resto e 4. adição e subtração. Note que a maior precedência é dada pelos parênteses (em TODAS as linguagens), então uma regra bem simples, é: na dúvida use e abuse dos parênteses.

## Conversões

Quando tipos diferentes estão envolvidos em uma expressão o compilador tenta adaptar o resultado antes de efetuar a atribuição, acompanhe

| | | |
|---|---|---|
| int = | float | o resultado é truncado no ponto decimal |
| float = | int | o resultado recebe .0 |
| qualquer = | mistura de tipos | o tipo do resultado é o de maior "prioridade": double → float → int → char |

Eventualmente, para poder realizar operações aritméticas como `sqrt(x)`, `sin(x)`, `log(x)`, `exp(x)` `log10(x)`, `round(x)`, `pow(x,y)`, `abs(x)` entre muitas outras será necessário incluir a biblioteca `cmath`, acompanhe

```
#include<cmath>
```

## Operadores relacionais

`==` Igualdade. Se `a` e `b` tiverem o mesmo valor, `a==b` devolve 1 (verdadeiro).

`!=` Desigualdade. Se `a` e `b` tiverem o mesmo valor, `a!=b` devolve 0 (falso).

`>` Maior. Se `a=2` e `b=3`, `b>a` retorna 1 (verdadeiro)

`<` Menor. Se `a=2` e `b=3`, `a<b` retorna 1 (verdadeiro)

`>=` Maior ou igual.

`<=` Menor ou igual.

## Conectivos lógicos

Conectam expressões relacionais. São eles

`&&` Conhecido como AND, e na matemática com o símbolo ∧ conecta duas expressões relacionais, e devolve verdadeiro se e somente se as duas expressões envolvidas forem verdadeiras.

`||` Conhecido como OR, e na matemática com o símbolo ∨ conecta duas expressões relacionais e devolve verdadeiro se qualquer uma das expressões (ou as duas) forem verdadeiras.

*not* Único conectivo unário, só se aplica a uma expressão lógica negando seu valor.

Veja alguns exemplos

Suponha a condição de aprovação nesta disciplina: Você estará aprovado ao final do semestre se tiver nota maior ou igual a 5 e tiver frequencia maior que 75%. Um programa testaria isso assim

```
float nota, freq;
if ((nota>=5) && (freq>75)) { ...
```

Neste caso, se houver a necessidade de negar a condição de APROVADO para REPROVADO o comando seria

```
if ((nota<5) || (freq<=75)) { ...
```

Note a negação de cada condição, e a mudança do conectivo de AND (`&&`) para OR (`||`). Isto se deve ao teorema de De Morgan.

## Condicional

É o comando que permite estabelecer caminhos alternativos a depender das condições lógicas. O comando é

```
if (condição) {
   ... bloco1 ...
}
[else {
   ... bloco2 ...
}]
```

A condição (simples ou composta, tanto faz) é avaliada. Note que obrigatoriamente ela é escrita dentro de parênteses. Se a condição for verdadeira, o bloco imediatamente a seguir (no exemplo, o bloco1) é executado. A condição negada (a razão da palavra `else`) não é obrigatória, razão pela qual está escrita entre colchetes. Mas, se presente

o bloco2 só é executado se a condição original for falsa.

Em resumo no exemplo

```
if (a>7) {
   x=x+1;
}
else {
   z=z+1;
}
```

Se $a > 7$ o comando $x = x + 1$ será executado e $z = z + 1$ será ignorado. Se $a \leq 7$, $x = x + 1$ não será executado e $z = z + 1$ será executado.

**Condicionais compostas** Nada impede que dentro de uma condicional haja outra condicional. Veja

```
if (a<7) {
   if (b>5) {
      h++;   // se a<7 && b>5 }
   j++;  // se a<7 }
else {
   k++;   // se a>=7 }
```

Perceba a importância da indentação e sobretudo da correta disposição das chaves dos blocos.

**repetição** Este comando serve para repetir blocos de processamento. Seu formato

```
while (condição) {
   ... bloco1 ...
}
```

Se a condição for verdadeira o bloco1 é executado. Até aqui é igual ao comando `if (condição)`. A diferença vem agora. Quando o bloco1 acabar, ocorre um desvio até a palavra `while` e a condição é reavaliada. Veja o exemplo

```
int a=0;
while (a<10) {
   cout<<a;
   a=a+3;
}
cout<<"acabou";
```

O bloco será executado 4 vezes (serão impressos os valores 0, 3, 6 e 9) e depois a mensagem "acabou"será impressa.

Um especial cuidado deve ser tomado de maneira a garantir que dentro do bloco haja situações em que a condição deixa de ser verdadeira. Caso contrário pode-se ter um laço infinito e o programa nunca deixará de executar o bloco. Veja um contraexemplo

```
int b = 8;
while (b<20) {
   b=b-2;  // note que NUNCA b>20...
}
```

**break** Um comando especial para sair incondicionalmente do bloco onde ele é emitido. Veja o exemplo

```
while (1==1){
   a=a+1;
   if (a>10) {
      break;
 }  }
```

Se dependesse apenas da condição `1==1` o bloco nunca deixaria de ser executado. Mas perceba que dentro do bloco há uma condição (`a>10`) e quando ela for verdadeira o bloco deixará de ser executado via o comando `break`.

Perceba que se a condição original do `while` for falsa já na entrada do bloco ele não será executado NENHUMA vez.

**do while** Nos casos em que o bloco deve ser executado pelo menos uma vez, usa-se um comando parecido, mas ligeiramente diferente

```
do {
   ... bloco1 ...
} while (condição)
```

Agora ao entrar no comando `do` o bloco1 é executado. Ao final do bloco, a condição é avaliada. Se ela for verdadeira há um retorno ao início do bloco. Se falsa, ocorre a saída.

**for** Embora tudo o que for necessário repetir poderá ser feito com `while`, há um outro comando muito usado, chamado `for`. Seu formato

```
for(inicialização;saída;alteração){
   ... bloco1 ...
}
```

Antes de entrar no bloco1, a inicialização é executada. A seguir a condição de saída é verificada. Se ela for verdadeira, o bloco é executado. (Se a condição for falsa, o comando todo é abandonado). Ao final do bloco a condição alteração é executada e há um desvio para o início do bloco1 (antes a condição de saída é verificada novamente). Veja o exemplo:

```
int a;
for (a=0;a<8;a=a+3){
  cout<<a<<" ";  // impressos 0,3,6
}
```

Note-se que qualquer uma das 3 especificações do for pode ser omitida sem que ocorra erro sintático. Quanto ao erro semântico fica por conta do programador...

**Funções** Um programa fonte em C++ é um conjunto de uma ou mais funções. A única obrigatória é a função de nome `main()` que recebe o controle quando o programa que a contém é chamado. A função é reconhecida em um programa C++ pela presença do abre e fecha parêntesis ao lado do nome.

O formato da função main é

```
int main() {
... função principal...
   return [valor]
}
```

O comando `return` indica o final da função e o seu parâmetro, se presente sinaliza qual o valor será retornado a quem chamou este programa.

**return** Por convenção, aqui, se tem: retorno de 0 (zero) significa um final bem sucedido, enquanto qualquer retorno diferente de zero indicará algum tipo de problema e o valor retornado pode ser usado para sinalizar qual o tipo de problema que ocorreu.

Você pode encerrar um programa (ou a sua função principal) simplesmente emitindo o comando `return` sem nenhum parâmetro. Neste caso, o cabeçalho da função main deve ser

```
void main() {...
```

sendo que aqui, a palavra `void` sinaliza nehum retorno.

Outra maneira de encerrar uma função é simplesmente esgotar os comandos que devem ser executados. Quando a } do bloco principal for encontrado, a função se encerra.

**outras funções** Além da principal, o autor pode definir e criar tantas funções quantas desejar. A única regra aqui é que a função deve ser definida antes de ser executada, para que o compilador conheça os parâmetros e o retorno dela.

Acompanhe a definição de uma função chamada `dobro` que retorna o dobro do valor passado a ela.

```
int dobro(int x){
   return x*2;
}
```

Quando a função for chamada com 10, ela retornará 20. Quando chamada com -1, retornará -2, e assim por diante.

A partir do momento em que a função foi definida, ela pode ser usada em qualquer ponto do programa, por exemplo, dentro de expressões aritméticas.

No exemplo acima, a variável `x` não existe exatamente, fora da função. Ela apenas sinaliza o que fazer com o parâmetro da função quando ela for chamada. Assim por exemplo

```
int a=8;
cout<<dobro(a);
```

Quando a função `dobro` é chamada, o `x` do seu cabeçalho tem o valor de `a` (neste caso 8) atribuído a ele e a devolução é de `x*2` ou no caso `8*2` que é 16. Este é o valor impresso.

**passagem por valor** Quando uma função é chamada por valor (como exemplificado acima) é uma cópia do valor do parâmetro que é passada à função. Assim, não importa o que a função fizer com esse parâmetro, ele manterá seu valor intocado na função chamada. Acompanhe

```
int triplo(int x){
   x=x*3;
   return x;
}
int a=10;
cout<<triplo(a);
cout<<a;
}``
```

perceba que dentro da função `triplo` o parâmetro `x` tem seu valor alterado. Mas, independente disso, na primeira impressão obtem-se 30 (o triplo de 10), mas na segunda impressão obtem-se 10 e não 30, pois a variável `a` foi preservada.

**passagem por referência** Pode-se alterar a passagem de parâmetro, indicando a passagem por referência. Neste caso não é uma cópia e sim a própria variável que é passada para a função. Acompanhe

```
int triplo(int & x){
   x=x*3;
   return x;
}
int a=10;
cout<<triplo(a);
cout<<a;
}``
```

Agora, as duas impressões são do valor 30. Pois, dentro da função triplo, é a própria variável `a` que está sendo manuseada, embora com o nome de `x`.

A passagem por referência é indicada pela presença de um & na definição dos parâmetros.

**Vetor** Um arranjo homogêneo de variáveis múltiplas que serão acessadas pelo mesmo nome e por um índice de referência.

Suponha precisar as taxas de inflação dos últimos 10 anos. Você poderá fazer

```
float infl[10];
```

Perceba um contador de ocorrências escrito ao lado do nome da variável, sempre entre colchetes.

O acesso pode ser feito com valores numéricos dentro do índice (dentro dos colchetes). Assim `infl[5]` corresponderá ao SEXTO valor do vetor `infl`. Um exemplo

```
int x[8];
int i;
for (i=0;i<8;i++){
   x[i]=i;  // valores 0,1,2,3,4,5,6,7
}
```

**Matrizes** É a extensão do conceito de vetor (que tem 1 dimensão) para um arranjo contendo duas dimensões. Então, se um vetor tem um comprimento, uma matriz tem largura e altura. Veja-se um exemplo

```
#include<iostream>
#include<iomanip>
using namespace std;
int main(){
   int mm[5][7];
   int i,j;
   for (i=0;i<5;i++){
      for(j=0;j<7;j++){
         mm[i][j]=(i*7)+j;
      }
   }
   for (i=0;i<5;i++){
      for(j=0;j<7;j++){
         cout<<setw(4)<<mm[i][j];
      }
      cout<<endl;
   }
}
```

O resultado aqui foi

```
   0   1   2   3   4   5   6
   7   8   9  10  11  12  13
...
```

Perceba que na impressão da matriz há que se estabelecer um tamanho para as colunas a fim que os números sejam apresentados em linhas verticais independente de seus comprimentos individuais. Isso foi feito com

```
#include<iomanip>
...
cout<<setw(4)<<mm...
```

Quando um vetor ou matriz (ou array) é passado como parâmetro para uma função, ele sempre é passado por REFERÊNCIA, e neste caso NÃO é colocado o caractere & na chamada. Assim, guarde esta regra e lembre-se dela ao criar este tipo de parâmetro em funções.

**Protótipos** Há uma regra já enunciada que diz que antes de usar uma função ela tem que ser definida. Mas, agora vamos supor um caso possivelmente real: Suponha a função `a(...)` que usa dentro dela a função `b(...)`. E, mais ainda, suponha que a função `b()` usa a função `a()` dentro dela. Está criado o impasse (aqui chamado de *deadlock*. A maneira de fugir deste dilema, é pela utilização de protótipos. Eles são uma descrição formal de como uma função será chamada e quais resultados devolverá. Ao usar um protótipo, você ganha o direito de "adiar" a definição da função até isto ser mais conveniente a você.

O fomato do protótipo é exatamente igual ao cabeçalho da função, só que terminado por ;

```
int funcaoa(int a, float n);
void funcaob();
```

**C++** O nome C++ vem da abreviatura muito comum em C de trocar o comando `C=C+1` que é muito comum, pela expressão `C++` que tem exatamente o mesmo significado.

## Exemplos

```
int primo(int x){
  int ate,diva,qtd;
  ate=1+sqrt(x);
  diva=2;
  if (x==2){return 1;}
  while(diva<=ate){
      if ((x%diva)==0){
        return 0; }
      diva++;  }
  return 1;
}
```

**Seja a multiplicação matricial** $C[i][m] = \sum_{x=1}^{j} A[i][x] \times B[x][m]$

```
int A[10][20]; int B[20][50]; int C[10][50];
   //... obtém A e B...
for (i=0;i<10;i++){
   for (m=0;m<50;m++){
      som=0;
      for (x=0;x<20;x++){
         som=som+A[i][x]*B[x][m];  }
      C[i][m]=som; }
}        // ...C está calculado
```