Conceitos básicos

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Até o momento nós estudamos

- Sistemas computacionais
- Algoritmos
- Problemas
- Linguagens

Agora vamos focar no nosso objetivo principal

- Resolver problemas projetando programas que sejam bem escritos e funcionem corretamente.

Vamos começar com um problema!

O André viaja muito. Sempre antes de fazer uma viagem ele calcula quanto irá gastar com combustível. Ele determina a distância que ele irá percorrer na viagem, o preço do litro do combustível e consulta as suas anotações para ver o consumo do carro, isto é, a quantidade de quilômetros que o carro anda com um litro de combustível e então faz o cálculo do custo. O André acha um pouco chato fazer os cálculos na mão, então ele pediu para você escrever um programa que faça os cálculos para ele.

Como projetar um programa que atenda a necessidade do André?

Antes de começarmos a projetar programas, vamos estudar algumas construções básicas.

Função `main`

Todo programa começa a execução a partir de um ponto. Em C++ esse ponto é a função `main`.

O programa mais simples que podemos escrever é

Arquivo `basico.cpp`

```
int main( )
{
}
```

Compilação (Linux)

```
g++ basico.cpp -o basico
```

Execução (Linux)

```
./basico
```

O que será exibido na tela após a execução desse programa?

Nada! O programa inicia e termina logo em seguida, sem fazer “nada”.

Saída

Agora vamos escrever algumas instruções dentro da função `main`. Os primeiros computadores foram criados para fazerem cálculos matemáticos, então vamos começar com isso.

Podemos escrever expressões matemáticas de forma similar ao que estamos acostumados

```
int main()
{
    2 + 4 * 3;
}
```

O que será exibido na tela após a execução desse programa?

Nada.

A sentença que escrevemos instrui o computador a fazer `4 * 3` e somar o resultado com `2`, apenas isso. Precisamos usar uma instrução específica para exibir o resultado na tela.

Algumas instruções, como a soma e a multiplicação, estão disponíveis diretamente para o programador, outras instruções, como as de entrada e saída, estão em bibliotecas que precisam ser requisitadas explicitamente.

Para usarmos a instrução de saída, precisamos incluir o arquivo de cabeçalho `iostream` (`io` é de *input*/*output* – entrada/saída em inglês) usando a diretiva `#include`.

```cpp
#include <iostream>

int main()
{
    std::cout << "Olá mundo!" << std::endl;
}
```

A instrução `std::cout` (***c****onsole* ***out****put*) é usado para exibir informações no terminal (console).

O símbolo `<<` é usado para indicar à instrução `std::cout` um item que deve ser exibido. Os projetistas escolheram o símbolo `<<` porque dá a ideia de que a informação à direita do símbolo está sendo "transmitida" para o console (`std::cout`). O símbolo `std::endl` (*end line* – fim de linha em inglês) indica que a linha atual deve ser encerrada e o cursor posicionado no início da próxima linha.

As aspas (`"`) são utilizadas para delimitar uma sequência de caracteres (texto), que deve ser exibida na tela pelo `std::cout`. O ponto e vírgula (`;`) é usado para indicar o fim da instrução.

Saída

```cpp
#include <iostream>

int main()
{
    std::cout << "Olá mundo!" << std::endl;
}
```

O que ocorre após a execução do programa?

A mensagem “Olá mundo!” é exibida na tela e o cursor vai para o início da próxima linha.

Podemos usar a instrução de saída para exibir resultados de expressões

```
#include <iostream>

int main()
{
    std::cout << 2 + 4 * 3 << std::endl;
}
```

O que será exibido na tela após a execução desse programa?

O valor 14.

Podemos exibir várias “coisas” com várias instruções de saída

```
#include <iostream>

int main()
{
    std::cout << "O resultado da expressão é ";
    std::cout << 2 + 4 * 3;
    std::cout << std::endl;
}
```

O que será exibido na tela após a execução desse programa?

“O resultado da expressão é 14”.

Também podemos exibir várias coisas com uma única instrução de saída

```
#include <iostream>

int main()
{
    std::cout << "O resultado da expressão é "
              << 2 + 4 * 3
              << std::endl;
}
```

Todas as construções que estão na biblioteca padrão (*standard library*) do C++ tem o prefixo `std::`, que é um “espaço de nomes”. A ideia de espaço de nomes é evitar colisões de nomes de bibliotecas diferentes, que é importante em programas que utilizam bibliotecas de diversas fontes distintas.

Como nossos programas serão pequenos e vamos utilizar apenas uma biblioteca externa, então a colisão de nomes não será uma preocupação. Por isso, vamos usar uma diretiva que permite omitir o prefixo do espaço de nomes.

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Não precisamos mais do prefixo std:: em cout e endl!" << endl;
}
```

Programa completo

Já vimos a instrução de saída e expressões matemáticas, o que está faltando para fazermos um programa “completo”? Instrução de entrada.

Antes de vermos a instrução de entrada, precisamos aprender sobre variáveis.

Uma **variável** é um nome para uma região da memória que é utilizada para armazenar valores.

Cada variável tem um tipo, que determina o conjunto de valores que podem ser armazenados na memória associada com ela.

Uma variável é primeiro declarada para depois poder ser usada. Na declaração a variável também pode ser inicializada.

A forma geral para declaração de variável é

```
Tipo nome [= valor inicial];
```

Alguns exemplos a seguir omitem a função `main` para não ficar repetitivo, mas na hora de testar o código, ele deve ser colocado dentro da função `main`.

```
int a = 10;
int b = 2 * a;
```

As variáveis `a` e `b` foram declaradas como inteiras (`int`), o que significa que apenas valores numéricos inteiros (no intervalo de -2.147.483.648 a 2.147.483.647) podem ser armazenados nessas variáveis.

Além de números inteiros, também temos números de ponto flutuante (`double`), que são utilizadas para armazenar valores aproximados de números reais (15 dígitos significativos).

```
double c = 40.1;
```

Agora podemos ver a instrução de entrada!

A instrução de entrada, assim como a instrução de saída, está na biblioteca `iostream`. Para ler um número de entrada fazemos

```
#include <iostream>

using namespace std;

int main()
{
    int a;
    cin >> a;
    cout << a << endl;
}
```

O que veremos na tela após a execução desse programa?

O mesmo número repetido duas vezes, na primeira vez é a digitação do usuário e a segunda vez é pelo uso do `cout`.

O que tem de errado com esse programa?

O usuário não é informado sobre o que digitar e nem o que a saída significa.

Entrada (arquivo `echo.cpp`)

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Digite um número: ";
    int a;
    cin >> a;
    cout << "O número que você digitou foi " << a << "." <<endl;
}
```

Exercício

Agora já podemos fazer um programa completo!

Escreva um programa que calcule a área de um retângulo.

Exercício (arquivo `area-retangulo.cpp`)

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Digite a medida da base: ";
    double base;
    cin >> base;

    cout << "Digite a medida da altura: ";
    double altura;
    cin >> altura;

    double area = base * altura;
    cout << "A area do retangulo e "
         << area
         << "."
         << endl;
}
```

Erros

Vamos parar um pouco e pensar sobre erros.

Durante a compilação de um programa, podem ocorrer dois tipos de erros:

- Sintáticos
- Semânticos

Um erro sintático ocorre quando não seguimos as regras sintáticas da linguagem e o compilador não consegue "entender" a estrutura do programa.

```
int x
cin >> x;
```

Qual é o erro nesse código?

Falta `;` no final da sentença.

Erro gerado pelo `g++`

```
x.cpp: In function ‘int main()’:
x.cpp:8:5: error: expected initializer before ‘cin’
    8 |     cin >> x;
      |     ^~~
```

Erro gerado pelo `clang++`

```
x.cpp:7:10: error: expected ';' at end of declaration
    int x
         ^
         ;
1 error generated.
```

Erros sintáticos

```
int x = (10 + 4;
```

Qual é o erro nesse código?

Faltou fechar o parêntese.

Erro gerado pelo `g++`

```
x.cpp: In function ‘int main()’:
x.cpp:7:20: error: expected ‘)’ before ‘;’ token
    7 |     int x = (10 + 4;
      |              ~      ^
      |                     )
```

Erro gerado pelo `clang++`

```
x.cpp:7:20: error: expected ')'
    int x = (10 + 4;
                   ^
x.cpp:7:13: note: to match this '('
    int x = (10 + 4;
            ^
1 error generated.
```

Erros sintáticos

```
int nome com espaco = 10;
double namespace = 20.3;
```

Quais os erros nesse código?

Usar nomes inválidos para variáveis.

Um nome não pode ter espaços e nem ser uma palavra chave (como `namespace`).

Erro gerado pelo `g++`

```
x.cpp: In function ‘int main()’:
x.cpp:7:14: error: expected initializer before ‘com’
    7 |     int nome com espaco = 10;
      |              ^~~
x.cpp:8:12: error: expected unqualified-id before ‘namespace’
    8 |     double namespace = 20.3;
      |            ^~~~~~~~
```

Erro gerado pelo `clang++`

```
x.cpp:7:13: error: expected ';' at end of declaration
    int nome com espaco = 10;
            ^
            ;
x.cpp:8:12: error: expected unqualified-id
    double namespace = 20.3;
           ^
```

Um erro semântico ocorre quando o compilador não “consegue” atribuir um significado para uma construção.

```
int a = 10 + "3";
```

Qual é o erro nesse código? Usar operandos de tipos inválidos para uma operação.

Erro gerado pelo `g++`

```
x.cpp: In function ‘int main()’:
x.cpp:7:16: error: invalid operands of types ‘int’ and ‘const char [2]’ to binary ‘operator*’
    7 |     int a = 10 * "3";
      |             ~~ ^ ~~~
      |             |    |
      |             int  const char [2]
```

```c
int a = 10.6;
```

Qual o erro nesse código? Por padrão, esta construção é válida e não gera erro!

Nesse caso, apesar de ir contra a nossa intuição, o compilador atribui um significado para a construção, que é armazenar apenas a parte inteira de `10.6` em `a`.

Às vezes o comportamento da linguagem não está de acordo com a nossa intuição, por isso precisamos conhecer com precisão a semântica da linguagem!

Algumas construções que podem ser propensas a erros são aceitas por padrão pelos compiladores do C++. Como programadores iniciantes, é bom termos um compilador mais "exigente", que nos ajude a identificar essas construções.

Então, para compilarmos os nossos programas, vamos utilizar as opções `-Wall -Wextra -Wconversion -Werror`, que faz o compilador apontar como erro mais construções que não são muito claras.

```
int main()
{
    int a = 10.6;
}
```

Compilando o programa acima com o comando

```
g++ -Wall -Wextra -Wconversion -Werror arquivo.cpp
```

Produz a seguinte mensagem de erro

```
x.cpp: In function ‘int main()’:
x.cpp:7:13: error: conversion from ‘double’ to ‘int’ changes value from ‘1.06e+1’ to ‘10’
    7 |     int a = 10.6;
      |             ^~~~
```

Erros de execução

Se um programa foi compilado corretamente, isto é, não tem erros de sintaxe ou semântica, significa que ele não tem erros? Não! Erros podem ocorrer durante a execução do programa.

Um erro de execução pode fazer o programa

- Ser interrompido e exibir uma mensagem de erro (crashar)
- Entrar em um laço infinito e nunca terminar (travar)
- Continuar a execução e produzir a resposta errada

No programa que calcula a área do retângulo, o que acontece se o usuário digitar um número muito grande ou digitar algo que não é um número? O programa executa até o final mas produz uma resposta incorreta.

Como garantir que um programa não terá erros em tempo de execução? Veremos isso ao longo da disciplina.

Vamos voltar para o programa que calcula a área de um retângulo.

```
#include <iostream>
using namespace std;
int main() {
    cout << "Digite a medida da base: ";
    double base;
    cin >> base;

    cout << "Digite a medida da altura: ";
    double altura;
    cin >> altura;

    double area = base * altura;
    cout << "A area do retangulo e "
         << area  << "." << endl;
}
```

Quais construções da linguagem usamos nesse programa?

- Uso de biblioteca e namespace;
- Comandos de entrada e saída;
- Tipo de dado e variáveis numéricos;
- Operações com números.

O que precisamos para escrever programas mais interessantes?

- Outras operações com números;
- Outros tipos de dados e operações;
- Maneira de criar e nomear novas operações;

Outras operações com números

As quatro operações aritméticas podem ser usadas com número inteiros ou de ponto flutuante. Se os dois operandos são `int`, então o resultado é `int`. Se pelo menos um dos operandos é `double`, então o resultado é `double`.

Outras operações com números

```
// Soma e subtração
10 + 3 - 20.5; // -7.5
// Multiplicação
13 * 3; // 39
// Divisão "real"
21 / 3.0; // 7.0
21 / 8.0; // 2.625
// Divisão inteira
21 / 3; // 7;
21 / 8; // 2;
// Menos e mais unário
int a = 10;
-a; // -10;
+a; // +10;
```

Para números inteiros, também temos a operação de módulo (resto da divisão)

```
21 % 3; // 0
21 % 8; // 5
```

Outras operações com números

Outras operações estão disponíveis na biblioteca `cmath`. Alguns exemplos

```
// Função piso - maior inteiro que não é maior que o número
floor(1.0); // 1.0
floor(1.3); // 1.0
floor(1.5); // 1.0
floor(1.7); // 1.0
// Função teto - menor inteiro que não é menor que o número
ceil(1.3); // 2.0
ceil(1.5); // 2.0
ceil(1.7); // 2.0
ceil(2.0); // 2.0
// Função de aredondadmento - inteiro mais próximo do número
round(1.3); // 1.0
round(1.5); // 2.0
round(1.7); // 2.0
```

```
// Módulo (resto da divisão) para double
fmod(5.5, 2.0); // 1.5
fmod(15.0, 5.0); // 0.0
// Seno (o argumento é em radianos)
sin(3.14); // 0.00159265
// Raiz quadrada
sqrt(2.0); // 1.41421
// Exponencial
pow(2, 4); // 16
```

Podemos fazer muitas coisas com valores numéricos! Mas também temos outros tipos de valores.

Informações textuais podem ser representas por cadeias de caracteres.

Usamos o tipo **string**, definido na biblioteca **string**, para armazenar cadeia de caracteres.

Valores literais de strings são delimitados por ".

```
#include <string>

using namespace std;

int main()
{
    string nome = "Joao da Silva";
}
```

Assim como podemos fazer operações com números, também podemos fazer operações com strings.

```
string nome = "Joao da Silva";

// Concatenação de strings
string junior = nome + " Filho"; // "Joao da Silva Filho"

// Extração de substring
// A partir da primeira letra, que está no índice 0,
// pegue 4 bytes (nesse caso, caracteres)
string primeiro_nome = nome.substr(0, 4); // "Joao"
```

```
string nome = "Joao da Silva";
```

Como extrair a substring `"Silva"` de `nome`? `nome.substr(8, 5)`.

Note que o segundo argumento para `substr` representa a quantidade de bytes (nesse caso, caracteres) a serem extraídos.

O que acontece se especificarmos uma quantidade de bytes para extrair maior do que o "disponível", como por exemplo, `nome.substr(8, 7)`? O método retorna apenas o que está disponível, nesse caso, `"Silva"`.

Escreva um programa que leia um nome de uma pessoa e escreva uma mensagem de boas vindas para a pessoa.

Para ler uma string é necessário usar a função `getline` da biblioteca `string` da seguinte forma:

```
string s;
getline(cin, s);
```

```cpp
#include <iostream>
#include <string>

using namespace std;

int main()
{
    cout << "Nome: ";
    string nome;
    getline(cin, nome);

    cout << "Ola " << nome <<  ", seja bem vindo!" << endl;
}
```

Qual resposta você daria para as seguintes expressões?

`5 < 10` , Verdadeiro.

`9 + 2 > 7 * 2` , Falso.

O tipo `bool` (booleano) tem dois valores, verdadeiro (**`true`**) e falso (**`false`**). Assim como números e strings, os valores do tipo booleano podem ser armazenados e manipulados.

```
bool a = true;
bool b = false;

string nome = "Jose";
// O operador == verifica se dois valores são iguais
bool chama_jose = nome == "Jose"; // true

// Operador >= (maior ou igual)
int idade = 17;
bool maior_de_idade = idade >= 18; // false
// Também podemos usar >, < e <=
```

Os operadores de == (igual) e != (diferente) também podem ser usados para números e outros tipos de dados.

```
bool a = true;
bool b = false;
```

Qual o resultado de `a == b`? O resultado é **`false`** pois o valor armazenado em **a** não é igual ao valor armazenado em **b**.

Qual o resultado de `a != b`? O resultado é **`true`** pois o valor armazenado em **a** é diferente do valor armazenado em **b**.

Três operações são comuns com valores booleanos:

- Negação
- E lógico (conjunção)
- Ou lógico (disjunção)

Em C++ o símbolo da operação de negação é `!`.

Qual o valor da expressão `4 < 5`? **`true`**. E da expressão `!(4 < 5)`? **`false`**.

Qual o valor da expressão `2 == 1 + 2`? **`false`**. E da expressão `!(2 == 1 + 2)`? **`true`**.

Tabela verdade

a	!a
`false`	`true`
`true`	`false`

Em C++ o símbolo da operação de conjunção é &&

Tabela verdade

a	b	a && b
false	false	false
false	true	false
true	false	false
true	true	true

Em C++ o símbolo da operação de disjunção é ||

Tabela verdade

a	b	a && b
false	false	false
false	true	true
true	false	true
true	true	true

Em uma determinada cidade o transporte público é gratuito para crianças menores de 10 anos, adultos a partir de 60 anos e professores a partir de 50 anos. Escreva um programa que leia a idade de uma pessoa e se ela é professor e indique se essa pessoa pode usar o transporte público de forma gratuita.

Em C++ na entrada e saída de valores booleanos `0` é usado para indicar **`false`** e `1` é usado para indicar **`true`**.

Booleanos

```cpp
#include <iostream>

using namespace std;

int main()
{
    cout << "Idade: ";
    int idade;
    cin >> idade;

    cout << "E professor? ";
    bool eh_professor;
    cin >> eh_professor;

    // Note que && tem priodade sobre ||
    bool livre = idade < 10 || idade >= 60 || idade >= 50 && eh_professor;

    cout << "Pode usar o transporte publico de forma gratuita? " << livre << endl;
}
```

Combinado dados de tipo diferente

Podemos usar tipos diferentes na mesma expressão.

```
string texto = "1023"

// texto.length() produz a quantidade de bytes em texto
bool tem_4_caracteres = texto.length() == 4; // true

// stoi converte uma string que representa um número inteiro
// em um número inteiro
int x = stoi(texto) + 10; // 1033

// to_string converte um número para uma string
string r = texto + to_string(x); // "10231033"
```

Novos tipos de expressões

Inicialmente as expressões (cálculos) dos nossos programas usavam apenas operadores matemáticos.

- `30 * 2 + a`

Nos últimos exemplos vimos que as expressões podem conter outros tipos de construções: as chamadas de funções e métodos.

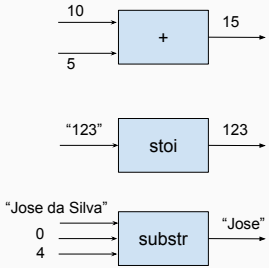
Chamada de funções

- `sin(3.14)`
- `stoi("123")`

Chamada de métodos

- `texto.length()`
- `nome.substr(0, 4)`

Embora a forma de utilizar operadores, funções e métodos seja diferente, o propósito dessas construções é o mesmo: computar valores de saída a partir de valores de entrada.



Se o propósito é o mesmo, por que não usar a mesma forma?

Por conveniência! Por exemplo, se não tivéssemos a forma de operadores e apenas a forma de chamada de funções, então deveríamos escrever `+(*(30, 2), a)` ao invés de `30 * 2 + a`, o que seria inconveniente.

Se não tivéssemos a forma de chamada de método, então deveríamos escrever `substr(nome, 0, 5)` ao invés de `nome.substr(0, 5)`, o que parece ok! Mas a questão é que chamadas de métodos são mais flexíveis do que chamada de funções, mas essa flexibilidade não será importante nessa disciplina e por isso não vamos discutir esse aspecto. Para nós, basta sabermos que algumas operações são chamadas como métodos.

Criar e nomear novas operações

Vimos que para escrever programas mais interessantes vamos precisar de:

- Outras operações com números;
- Outros tipos de dados e operações;
- Maneira de criar e nomear novas operações; Ou seja, nós vamos criar as nossas próprias funções!

A forma geral para definição de funções é

```
TipoSaida nome_da_funcao(TipoEntrada1 entrada1, TipoEntrada2 entrada2, ...)
{
    return saida_da_funcao;
}
```

Baseado no programa que calcula a área de um retângulo, vamos criar uma função para calcular a área de um retângulo. (Solução desenvolvida em sala)

```c
// Calcula a área do retângulo com a base b e a altura a.
// 10.0 5.0 -> 50.0
// 2.0 3.0 -> 6.0
double area_retangulo(double b, double a) {
    return b * a;
}

int main()
{
    // entrada omitida para economizar espaço no slide
    double area = area_retangulo(base, altura);
    // saída omitida para economizar espaço no slide
}
```

Exercício

Projete um programa que a partir de um tempo especificado em número de horas, minutos e segundos, calcule o total de segundos do tempo.

Projeto feito em sala.

```c
// As h, m, s serão representados por inteiros positivos

// Calcula o total de segundos do tempo com h horas,
// m minutos e s segundos.
// Exemplos
// h=0 m=0 s=4 -> 4 + 0 * 60 + 0 * 3600 -> 4
// h=0 m=10 s=5 -> 5 + 10 * 60 + 0 * 3600 -> 605
// h=3 m=2 s=25 -> 25 + 2 * 60 + 3 * 3600 -> 10945
int total_segundos(int h, int m, int s)
{
    return s + m * 60 + h * 3600;
}
```

Referências

- Tutorial C++ - W3 Schools