Computação para Engenharia (CPE)
Profs.: Caio César de Melo e Silva e Claudia Barenco Abbas

Notas de aula 1: Introdução

1 Linguagens compiladas e C++

1.1 Quais os motivos para se utilizar C++?

Na sua essência, um computador não passa de um processador com uma memória, capaz de executar pequenas instruções como *"**guarde** o valor 5 no **endereço de memória** 56743"*. Sendo assim, por qual razão escrevemos linhas de código em uma linguagem de programação, em vez de informar instruções diretamente para o processador?

Vantagens:

1. **Concisão:** linguagens de programação nos permite escrever sequências comuns de comandos de maneira mais concisa. C++ fornece alguns atalhos muito interessantes.

2. **Manutenibilidade:** alterar um código se torna mais fácil quando se precisa alterar apenas algumas linhas de texto, ao invés de alterar centenas de instruções para o processador. A linguagem C++ é uma linguagem que suporta a **orientação a objetos**, tal paradigma de programação favorece a manutenibilidade.

3. **Portabilidade:** diferentes processadores possuem diferentes conjuntos de instruções. Programas escritos como um texto podem ser traduzidos para vários conjuntos de instruções de diferentes processadores. Programas escritos em C++ funcionam praticamente em qualquer processador.

C++ é uma linguagem de *alto nível*, isto é, quando se escreve um programa nessa linguagem, seus comandos são expressivos o suficiente para que não haja preocupações com os detalhes do processador. Porém, diferentemente da maioria das linguagens populares, C++ permite que sejam manipuladas algumas estruturas de baixo nível como, por exemplo, endereços de memória.

1.2 O processo de compilação

Um programa de computador é traduzido de um texto (mais especificamente, de arquivos fonte) para comandos do processador seguindo o fluxo apresentado na figura 1.

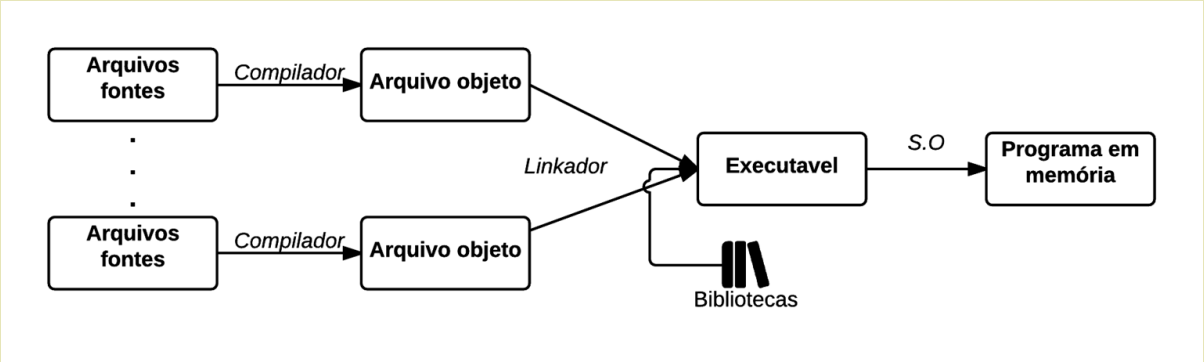


Figura 1. Processo de compilação

Arquivos objeto são intermediários que representam uma cópia incompleta do programa: cada arquivo fonte representa apenas um pedaço do programa, que quando compilados conjuntamente, geram arquivos objeto com marcadores para suas dependências no programa.

O *linkador* utiliza esses arquivos objeto e também as bibliotecas referenciadas no código para resolver todas as dependências do programa e gerar o código executável, que pode ser "rodado" pelo sistema operacional.

O compilador e o *linkador* são também programas de computador. O passo no processo de compilação no qual o compilador realiza a leitura do arquivo é denominado *parse*.

No C++, todos esses passos são realizados antes do programa ser executado. Em outras linguagens, podem ser realizados durante a execução do programa, o que leva tempo. Esse é uma das razões que justificam porque programas em C++ tendem a ter um melhor desempenho do que o mesmo código escrito em muitas outras linguagens.

Na realidade, C++ possui um passo adicional no processo de compilação, o código é primeiro avaliado por um *preprocessador*, que realiza algumas modificações no código, antes de enviar ao compilador. A figura 2 apresenta o novo diagrama.

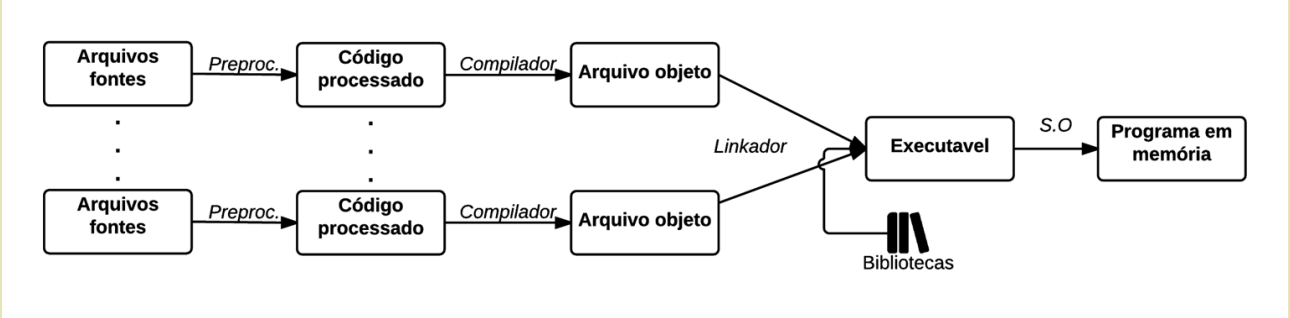


Figura 2. Processo de compilação C++

Nota randômica: C++ é *case sensitive*, a palavra tomate é diferente de Tomate.

2 Primeiro programa

Tradicionalmente em programação, é utilizado um programa "Olá, Mundo!" para se entender as funcionalidades básicas da linguagem. Segue o código:

2.1 O código

```
1  // programa olá mundo
2  #include <iostream>
3
4  int main()
5  {
6     std::cout << "Olá mundo!\n";
7     return 0;
8  }
```

2.2 *Tokens*

Tokens são as menores partes de um programa que possuem significado para o compilador – os menores símbolos, com significado, da linguagem. O código acima apresenta os 6 diferentes tipos de *tokens*, apesar de não apresentar a utilização mais comum dos operadores.

Tipo do *token*	Descrição	Exemplos
Palavras-chave ou *keywords*	Palavra com um significado especial para o compilador	int, double, for, auto, if
Identificadores ou *identifiers*	Nomes de "coisas" que não estão inclusas na linguagem	cout, std, x, myFunction, MyClass
Literais ou *literals*	Valores constantes cujo o valor é especificado diretamente no código	"Olá mundo!\n", 24.3, 0, 'a'

Operadores ou *operators*	Operações lógicas ou matemáticas	+, -, /, <<, &&, %
Separadores ou *separators*	Definem a estrutura do programa	{}, (), ;
Espaços ou *whitespaces*	Qualquer tipo de espaçamento; são ignorados pelo compilador	espaço, tab, nova linha, comentários

2.3 Explicação linha a linha

Na linha:

1. O símbolo **//** indica que qualquer coisa que o procede até o final da linha será um comentário, e é ignorado pelo compilador. Outra forma de escrever um comentário consiste em colocá-lo entre **/* */**, por exemplo, (x = 3 + /* comentário está aqui */ 2;). Dessa forma é possível escrever um comentário em múltiplas linhas. Comentários existem para explicar coisas não obvias no código.

Utilize-os: documente bem o seu código!

2. Linhas que começam com # são comandos para o preprocessador, que normalmente altera o código que será compilado. **#include** informa ao preprocessador que deve ser adicionado, no arquivo atual, o conteúdo de um outro arquivo, nesse caso o arquivo ***iostream***, que possui funções para entrada e saída de dados.

3. int main(){...} define o código que deve ser executado quando o programa iniciar. As chaves indicam um agrupamento de comandos chamado de **bloco**. A palavra *int* indica que a função denominada *main* retorna um valor inteiro. Mais detalhes de sintaxe da linguagem na próxima nota de aula.

4.

- **cout <<**: Essa é a sintaxe para imprimir alguma coisa na tela.

- **Namespaces**: Em C++, identificadores podem ser definidos em um contexto - uma espécie de diretório de nomes, chamado *namespace*. Quando se deseja acessar um identificador definido em um *namespace*, é necessário informar ao compilador em qual *namespace* ele se encontra utilizando o *operador de resolução de escopo*(::). No nosso exemplo, informamos ao compilador para procurar por *cout* no namespace *std*, cujo o qual possui a maioria dos identificadores padrões da linguagem C++.

Uma forma mais elegante seria adicionar na linha 3:

```
using namespace std;
```

Assim, o compilador é informado que qualquer identificador desconhecido deve ser procurado no *namespace* std, e std:: pode ser omitido do código. Essa é uma prática recomendada.

- **String**: Uma sequência de caracteres, como "Olá mundo!", é conhecida como *string*. Uma *string* definida explicitamente no programa, isto é entre aspas duplas (ex.: "aqui vai a string"), é conhecida como uma ***string literal***.

- **Indicadores na *string***: O **'\n'** é um indicador da criação de uma nova linha. Existem alguns símbolos utilizados para representar caracteres especiais dentro de textos literais. A tabela abaixo mostra alguns dos indicadores de string em C++:

Indicador	Carácter que representa
\a	Som do sistema (um beep)
\b	Move o cursor para trás
\f	Quebra de página
\n	Nova linha
\r	Retorna o cursor para o início da linha
\t	Tab
\\	Barra invertida
\'	Aspas simples
\"	Aspas duplas

5. **return 0** indica que o programa deve informar ao sistema operacional que foi finalizado com sucesso. A sintaxe será explicada nas próximas notas. Por enquanto considere que está é a última linha do bloco main.

Nota randômica: Todo comando deve terminar com ';', exceto comandos para o preprocessador e na criação de blocos {}.

3 Funcionalidades básicas

Até agora o programa apresentado não faz muitas coisas. Vamos adicionar algumas funcionalidades.

3.1 Valores e afirmativas

Primeiramente algumas definições:

- Uma afirmativa é uma unidade de código que realiza algum processamento.
- Uma expressão é uma afirmativa que possui valor – por exemplo, um número, uma *string*, a soma de dois números, etc... **4 + 2**, **x – 1**, e **"Olá mundo"** são todas expressões.

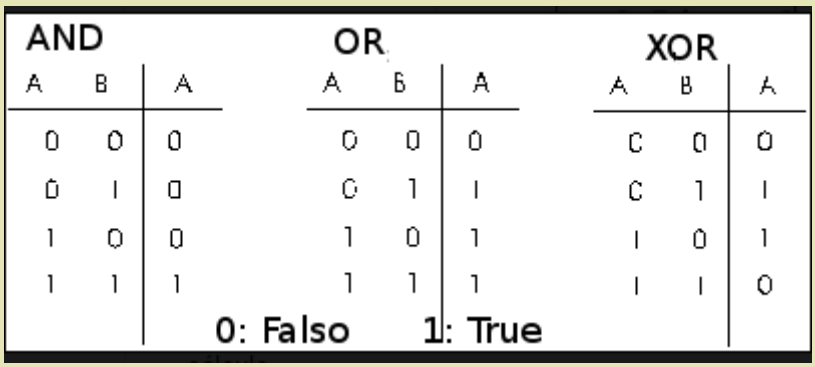
Nem toda afirmativa é uma expressão. Não faz muito sentido encontrar o valor da afirmativa #include<iostream>, por exemplo.

3.2 Operadores

Cálculos aritméticos podem ser realizados através dos operadores matemáticos. Operadores atuam nas expressões para gerar novas expressões. Por exemplo, trocando a expressão "Olá mundo" por (5 + 7) / 3, faria com que o programa imprimisse o número 4. Nesse caso, o operador '+' atuou na expressão 5 e 7.

Tipos de operadores:

- **Matemáticos**: +,-,*,/, e parênteses possuem o significado usual da matemática. % (operador de modulo) retorna o resto da divisão entre dois valores. 9 % 8 retorna 1.
- **Lógicos**: E (&&), OU (||), NOT (!), operadores da lógica booleana, onde as afirmações podem assumir o valor verdadeiro (true) ou falso (false). Veremos exemplos nas próximas notas de aula.



- **Bit a Bit:** Usado para manipular valores binários. Esse curso não será focado nesse tipo de operação.

3.3 Tipos de dados

Cada expressão possui um tipo – uma descrição formal do dado que indica o valor de uma expressão. Por exemplo, 0 é inteiro, 5.65 é ponto flutuante (float), e "Olá mundo" é uma string. Dados de diferentes tipos ocupam diferentes espaços na memória. A tabela abaixo apresenta os tipos mais comumente utilizados:

Tipo	Descrição	Tamanho	Intervalo
char	Único caracter	1 byte	Com sinal: -128 a 127 Sem sinal: 0 a 255
int	Número inteiro	4 bytes	Com sinal: -2147483648 a 2147483647 Sem sinal: 0 a 4294967295
bool	True ou false	1 byte	true (1) ou false (0)
double	Representa números em notação de ponto flutuante normalizada em precisão dupla de 64 bits	8 bytes	+/- 1.7e +/- 308 (15 dígitos)

Observações da tabela:

- Um inteiro com sinal (signed int) consegue representar números negativos. O inteiro sem sinal (unsigned int) nunca representará um valor negativo, assim é possível representar uma quantidade maior de valores positivos. A maioria dos compiladores assume o inteiro com sinal caso nada seja dito.

- Na verdade existem 3 tipos de inteiros: **short**, **int** e **long**, em ordem crescente de tamanho, onde o tipo int costuma ser um sinônimo para um dos outros dois. Geralmente, durante a programação, não é necessário se preocupar com o tipo de inteiro, a não ser que a quantidade de memória utilizada seja um requisito do programa ou que seja necessário representar valores muito grandes. O mesmo acontece para os três tipos em ponto flutuante **float**, **double** e **long double**, em ordem crescente de precisão.

- Os tamanhos/intervalos para cada tipo de dado não são totalmente padronizados; Os que foram apresentados aqui são os mais comuns em computadores de 32-bits.

Nota randômica: Um operador usualmente produz um dado do mesmo tipo dos seus operandos.

4 Variáveis

Durante a programação se faz necessário dar nome aos valores encontrados para que se possa utilizá-los posteriormente. Isso é feito através do uso de variáveis. **Uma variável é um nome para um local na memória.**

Por exemplo, supondo que desejamos utilizar o valor 4 + 6 diversas vezes. Podemos chamá-lo de x e utilizar como se segue:

```
1 #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6    int x;
7    x = 4 + 6;
8    cout << x / 5 << ' ' << x * 2;
9    return 0;
10 }
```

(Perceba como se pode imprimir uma sequência de valor encadeando o operador <<)

O nome de uma variável é um *token* identificador. Identificadores podem conter números, letras e *underscores*(_), não devem iniciar com um número.

Na linha 6 ocorre a **declaração** da variável. Deve-se informar ao compilador de qual tipo é a variável, para que ele possa alocar a quantidade de memória necessária e distinguir quais operações podem ser realizadas.

Na linha 7 ocorre a **inicialização** de x, ou seja, é fornecido um valor inicial a variável. Isso introduz um novo operador: **=** , é o operador de atribuição. Pode-se alterar o valor da variável x futuramente utilizando esse operador.

Podemos substituir a linha 6 e 7 por uma única que realiza tanto a declaração quanto a inicialização da variável:

```
int x = 4 + 6;
```

Essa forma de declaração/inicialização é uma prática recomendada.

5 Entrada de dados

Agora que já sabemos dar nome aos valores, podemos solicitar ao usuário para inserir alguns valores. Observe a linha 7:

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6    int x;
7    cin >> x;
8    cout << x / 5 << ' ' << x * 2;
9    return 0;
10 }
```

Assim como cout << é a sintaxe para mostrar valores, cin >> é a sintaxe para receber valores. Quando a linha 7 do código for executada o terminal (console) ficará aberto esperando alguma entrada do usuário. Se, por exemplo, o usuário digitar o valor 30 e apertar <enter> será impresso na tela '6 60'.

6 Strings

Em C++ o usuário pode empregar a palavra string para declarar uma string de tamanho variável. Porém, deve ser incluído o header string. A partir da declaração, a utilização de strings é simples.

```
#include <iostream>

using namespace std;

int main()

{

     string nome1("Fulano");       // Inicializa nome1

     string nome2("Beltrano");     // Inicializa nome2

     string nome3, nome4;          // Não inicializa nome3 nem nome4

     cout << "Os dois primeiros nomes são " << nome1 << " e " <<
nome2 << endl;

     nome3 = "Ciclano";       // Inicialização posterior de nome3

     nome4 = nome3;

     cout << "O terceiro e quarto nomes são " << nome3 << " e " <<
nome4 << endl;

}

return 0;
```

Uma String é um conjunto de variáveis tipo Char. Podemos fazer manipulações com uma String com: size(), +,var[x], < , > , ==, etc ...

7 IF-ELSE (e suas variações)

A estrutura de seleção (ou estrutura condicional) *if* é representada pelo seguinte formato:

```
if(condicional)

{

     comando1

     comando2

     ...
}
```

A condicional é uma expressão que terá seu valor (verdadeiro ou falso) testado. Se o valor da condicional for *true*, então os comandos (1,2,...) são executados antes que o programa continue. Caso

contrário, os comandos são ignorados. Se existir apenas um comando para ser executado, as chaves podem ser omitidas. Dando o formato:

```
if(condicional)
     comando
```

O formato *if-else* é utilizado para decidir entre duas sequências de comandos denominadas **blocos:**

```
if(condicional)
{
     comandoA1
     comandoA2
     ...
}
else
{
     comandoB1
     comandoB2
     ...
}
```

Se o condicional tiver o valor *true*, o bloco correspondente ao *if* é executado. Caso contrário, o bloco correspondente ao *else* é executado. Como o condicional só pode assumir dois valores (true ou false) um dos dois blocos certamente será executado. Se houver apenas um comando para qualquer um dos blocos, as chaves podem ser omitidas:

```
if(condicional)
     comandoA
else
     comandoB
```

Já, o *else if* é utilizado para decidir entre dois ou mais blocos através de múltiplas condições:

```
if(condicional-1)
{
     comandoA1
     comandoA2
     ...
}
else if(condicional-2)
{
     comandoB1
     comandoB2
     ...
}
```

Se a condicional-1 é verdadeira, o bloco correspondente ao if é executado. Senão, e somente se a condicional-2 for verdadeira, o bloco correspondente ao *else if* é executado. Podem haver mais de um bloco *else if*, cada um com sua condicional. Quando uma condicional de um bloco possui o valor verdadeiro o bloco é executado, e qualquer *else if* após é ignorado. Sendo assim, num estrutura *if-else-if*, pode ser executado um bloco ou nenhum.

Um *else* pode ser adicionado ao final de uma estrutura if-else-if. Se nenhuma das condicionais carregarem o valor verdadeiro, o bloco *else* é executado. Nessa estrutura, um bloco certamente será executado, como em um *if-else* "tradicional".

Um exemplo utilizado estruturas de seleção:

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6     int x = 6;
7     int y = 2;
```

```
8
9       if(x > y)
10              cout << "x é maior que y" << endl;
11      else if (y > x)
12              cout << "Y é maior que x" << endl;
13      else
14              cout << "x é igual a y" << endl;
15      return 0;
16  }
```

A saída desse programa é *x é maior que y.* Se trocarmos as linhas 6 e 7 por:

```
int x = 2; int y = 6;
```

A saída será *y é maior que x.* E se alterarmos para:

```
int x = 2; int y = 2;
```

A saída será *x é igual a y.*

Nota randômica: Existem dois tipos de erros em programas escritos em C++: erros de compilação e erros de execução. Erros de compilação são indicados pelo compilador, e normalmente, estão relacionados à violação as regras de sintaxe ou incompatibilidade de tipos. Erros de execução são problemas que só ocorrem quando o programa é executado – o programa foi compilado porém não faz o que deveria. Esses costumam ser mais difíceis de solucionar dado que o compilador não fornece nenhuma informação

EXERCÍCIOS

Vamos colocar em prática esses conceitos. No bloco main:

1. Implemente um código que solicite ao usuário dois valores, realize a subtração e imprima o resultado na tela.

2. Implemente um código que solicite do usuário um número inteiro e mostre na tela se o número é *primo* ou não.

3. Implemente um código que solicite o valor de dois catetos para o usuário, realize o cálculo e imprima o valor da hipotenusa.

4.O custo do aluguel de um automóvel é 1.75 reais por km até os primeiros 50 km, 1.65 reais por km para os 100 km seguintes e 1.50 reais por km acima de 150 km. Escreva um programa que leia a distância em quilômetros e calcule o valor total a pagar e o custo médio por quilômetro.

5.O algoritmo seguinte compara dois valores dados em a,b e coloca em m o maior deles:

Ler a,b

Se a > b então m ← a senão m ← b

fim

5.1 Dados três valores a,b e c coloque em m o maior deles

5.2 Dados três valores a,b e c coloque em m o valor do meio.

Mostre na tela ambos resultados.

6. São fornecidos dois números inteiros positivos *p* e *q*, sendo que o número de dígitos de *p* é menor ao número de dígitos de *q*. Verificar se p é um *subnúmero* de *q*. Escreva um programa onde *p* tenha 4 dígitos e *q* tenha 3 dígitos.

Exemplos:

p = 231, q = 5723, p é subnúmero de q

p = 231, q = 2583, p não é subnúmero de q

7. Em engenharia electrotécnica é comum exprimir a relação entre dois valores de potência em decibeis, ou dB, dada pela equação

$$\mathrm{dB} = 10 \log_{10} \frac{P_2}{P_1}$$

onde P_2 é o valor de potência a ser medido e P_1 é um valor entendido como referência.

Utilizando o valor de referência convencional (1 watt), escreva um programa que dado o valor de P_2 o converta em dB.

8. Escreva um algoritmo que, dada uma sequência com 8 números, indique quantos e quais os elementos (ordem na sequência) que são maiores do que a soma dos seus vizinhos.

Considere que os extremos não estão incluídos.

Exemplo: 8, 2, 4, 1, 6, 12, 5, 9 (Resposta: 2, o 3º e o 6º)

9. A área de qualquer triângulo cujos lados medem a,b,c pode ser calculada pela fórmula (Heron de Alexandria):

$$\mathrm{A} = \sqrt{s(s-a)(s-b)(s-c)}$$

Onde s = (a+b+c)/2

Escreva um programa que, dados os valores a,b,c, calcule a área do triângulo correspondente.

10. Escreva um algoritmo que solicite ao usuário que digite uma String com 5 caracteres e calcule quantas vezes e quais vogais aparecem nesta String. Mostre na tela estes dois resultados. Depois utilize o seguinte código de criptografia que modifica a vogal ´a´ por ´i´,a vogal ´e´ por ´o´ e a vogal ´i´ por ´u´ e faça as devidas alterações na String . Mostre na tela a nova String criptografada.

11. Implemente um código que realiza a multiplicação de dois valores e imprima o resultado na tela.

12. Implemente um código que solicite o valor de dois catetos para o usuário, realize o cálculo e imprima o valor da hipotenusa.

13. Implemente um código que solicite do usuário um número e imprima na tela a sequência de números ímpares limitado pelo valor do usuário. Por exemplo, caso o usuário informe o valor 10, o programa deve imprimir 1,3,5,7,9

14. Implemente um código que só permita que o usuário saia do programa, caso ele digite o número 10.

15. Implemente um código que fornecido um valor inteiro positivo *n*, calcule a soma dos *n* primeiros números inteiros positivos.

16. Dizemos que um número natural é *triangular* se ele é produto de três números naturais consecutivos. Implemente um código que fornecido um inteiro não-negativo *n*, verifique se o mesmo é triangular.

17. Dizemos que um número natural *n* é palíndromo se o 1º algarismo de *n* é igual ao seu último algarismo, o 2º algarismo de *n* é igual ao penúltimo, e assim sucessivamente. Exemplos:

567765 e 32423 são palíndromos.

567675 não é palíndromo.

Dado um número natural n > 9, verifique se *n* é palíndromo.

18. Qualquer número natural de quatro algarismos pode ser dividido em duas dezenas formadas pelos seus dois primeiros e dois últimos dígitos, exemplo:

1297: 12 e 97

5314: 53 e 14

Implemente um código que imprima todos os números naturais de 4 algarismos cuja a raiz quadrada seja a igual soma de suas dezenas.

Exemplo: raiz de 9801 = 99 = 98 + 01

Portanto 9801 é um dos números a ser impresso.