PETEE UFMG

UFMG

Learning.cpp

Uma apostila de introdução
à programação em C++

C++

2022

Programa de Educação Tutorial - Engenharia Elétrica
Universidade Federal de Minas Gerais

Prefácio

Autores
- Diego Vieira dos Santos
- Fabrinni Dias Bastos
- Lucas José de Souza Oliveira
- Lucas Santos Durães
- Marcos Gabriel Araújo Lima
- Pedro Otávio Fonseca Pires

Programa de Educação Tutorial da Engenharia Elétrica - Univerisdade Federal de Minas Gerais
HTTP://WWW.PETEE.CPDEE.UFMG.BR/

PETEE UFMG

/peteeUFMG petee.cpdee.ufmg.br @petee.ufmg

Grupo PETEE

O que é PET?

Os grupos PETs são organizados a partir de formações em nível de graduação nas Instituições de Ensino Superior do país orientados pelo princípio da indissociabilidade entre **ensino**, **pesquisa** e **extensão** e da educação tutorial.

Por esses três pilares, entende-se por:

- Ensino: As atividades extra-curriculares que compõem o Programa têm como objetivo garantir a formação global do aluno, procurando atender plenamente as necessidades do próprio curso de graduação e/ou ampliar e aprofundar os objetivos e os conteúdos programáticos que integram sua grade curricular.
- Pesquisa: As atividades de pesquisa desenvolvidas pelos petianos têm como objetivo garantir a formação não só teórica, mas também prática, do aluno, de modo a oferecer a oportunidade de aprender novos conteúdos e já se familiarizar com o ambiente de pesquisa científica.
- Extensão: Vivenciar o processo ensino-aprendizagem além dos limites da sala de aula, com a possibilidade de articular a universidade às diversas organizações da sociedade, numa enriquecedora troca de conhecimentos e experiências.

PETEE UFMG

O Programa de Educação Tutorial da Engenharia Elétrica (PETEE) da Universidade Federal de Minas Gerais (UFMG) é um grupo composto por graduandos do curso de Engenharia Elétrica da UFMG e por um docente tutor.

Atualmente, o PETEE realiza atividades como oficinas de robôs seguidores de linha, minicursos de Matlab, minicursos de LaTeX, Competição de Robôs Autônomos (CoRA), escrita de artigos científicos, iniciações científicas, etc.

Assim como outras atividades, o grupo acredita que os minicursos representam a união dos três

pilares: O pilar de ensino, porque ampliam e desenvolvem os conhecimentos dos petianos; O pilar da pesquisa, pois os petianos aprendem novos conteúdos e têm de pesquisar para isso; O pilar da extensão, porque o produto final do minicurso é levar à comunidade os conhecimentos adquiridos em forma de educação tutorial.

O Grupo

Ana Luiza da Silva Santos
Arthur Miranda do Vale Ribeiro
Caio Teraoka de Menezes Câmara
Christian Felippe Vasconcelos de oliveira
Clara Maria Candido Martins
Davi Faúla dos Santos
Diego Vieira dos Santos
Fabrinni Dias Bastos
Fernanda Camilo dos Santos Xavier
Gustavo Alves Dourado
Isabela Alves Soares
Letícia Duque Giovannini
Lucas José de Souza Oliveira
Lucas Santos Durães
Luciana Pedrosa Salles
Marcos Gabriel Araujo Lima
Pedro Otávio Fonseca Pires
Vinícius Batista Fetter
Yuan Dias Fernandes Pena Pereira

Agradecimentos

Agradecemos ao Ministério da Educação (MEC), através do Programa de Educação Tutorial (PET), à Pró-Reitoria de Graduação da Universidade Federal de Minas Gerais (UFMG), à Fundação Christiano Ottoni (FCO) e à Escola de Engenharia da UFMG pelo apoio financeiro e fomento desse projeto desenvolvido pelo grupo PET Engenharia Elétrica da UFMG (PETEE - UFMG).

Contato

Site:
`http://www.petee.cpdee.ufmg.br/`

Facebook:
`https://www.facebook.com/peteeUFMG/`

Instagram:
`https://www.instagram.com/petee.ufmg/`

E-mail:
petee.ufmg@gmail.com

Localização:
Universidade Federal de Minas Gerais, Escola de Engenharia, Bloco 3, Sala 1050.

Sumário

Introdução

1. Conhecendo a Linguagem

1.1 História

A linguagem C++ foi desenvolvida por Bjarne Stroustrup, da AT T Bell Laboratories, na década de 1980, com o objetivo de oferecer funcionalidades que a linguagem C não poderia proporcionar. Em questões de eficiência, a linguagem C se mostrava rápida, uma vez que ela podia tratar diretamente na memória. Outra vantagem de C era sua portabilidade, já que ela é capaz de ser compilada em qualquer sistema operacional, seja Windows, UNIX ou Linux. Usar ela poderia ser de grande vantagem se tratando de linguagens para uso pessoal, logo ela foi a melhor opção na hora de criarem o C++. Inicialmente, foi chamada de "C com classes" devido ao acréscimo das funcionalidades, mas logo depois foi nomeada para C++, com o intuito de se tornar única.

Para entendermos melhor as suas funcionalidades, precisamos conversar um pouco sobre a linguagem C. Ela é dita muitas vezes como sendo de alto-nível e outras vezes sendo de baixo-nível. Os níveis de linguagem servem para nos orientar do modo como o código pode ser lido, podendo pensar nas linguagens de alto nível sendo mais próximas da linguagem humana e as linguagens de baixo nível são mais próximas da linguagem de máquina.

Dizemos assim então que C é alto nível pela facilidade em escrever programas, comparado com outras linguagens da época, como cobol e fortran, e baixo por poder manipular diretamente a memória do hardware, sendo uma linguagem muito boa na tradução de códigos assembly. Isso a torna uma ótima linguagem para programação de máquina, porém, com o passar do tempo, novos conceitos surgiram no cenário das linguagens de programação, e a linguagem C se mostrou limitada em certos contextos Tendo isso em vista, começou a ser implementado o C++.

As vantagens de se usar C++ vão desde sua padronização de versões, que são disponibilizadas por um site próprio, o suporte as mais diversas bibliotecas, o que acaba agilizando o trabalho dos usuários, é uma linguagem que compila diretamente no código de máquina, fazendo dela uma das linguagens com implementação da orientação à objetos mais rápidas até então, a portabilidade que foi herdada do C, fazendo com que ela possa ser usada em qualquer sistema desde que tenha o compilador certo, o suporte à vários paradigmas de programação, tais como procedural(rodapé),

genérico(rodapé) e orientação à objetos(rodapé), e por fim, o uso de tipos tanto estáticos quanto dinâmicos, o que a consolida como sendo uma das mais eficientes também.

Pode-se dizer que a maioria dos programas em C podem ser lidos em C++, entretanto, nem todos os programas em C++ podem ser lidos em C. Tendo isso em mente, podemos fazer em C++ todas as funcionalidades que podem ser feitas em C, tais como manipulação de vetores e matrizes, leitura e escrita de arquivos, funções, dentre as várias outras aplicações da linguagem implementada por Dennis Ritchie.

1.1.1 Comitê de Padronização

A Organização Internacional para Padronização (ISO)[1] foi criada em 1947 com o objetivo de coordenar e unificar padrões industriais internacionalmente. A ISO é dividida em uma série de comitês técnicos e, entre eles, existe o WG21 (Working Group 21), que é o comitê responsável pela padronização da linguagem C++. O grupo é composto por uma série de especialistas dos vários países membros da organização.

Desde a primeira versão da linguagem C++, disponibilizada em 1985, várias mudanças ocorreram ao longo dos anos, mas a primeira padronização feita pelo comitê ocorreu em 1998, e a versão padronizada foi chamada informalmente de C++98. Logo depois, em 2003, o comitê publicou a versão C++03, contendo pequenas revisões.

Outros detalhes foram sendo publicados através de reportes técnicos ao longo dos anos subsequentes, porém uma nova versão padronizada foi publicada apenas em 2011 (C++11), e implementou tantas mudanças que essa versão e as versões subsequentes foram consideradas como fazendo parte do chamado C++ Moderno.

Desde 2011, novos padrões da linguagem foram sendo lançados a cada três anos, então surgiram também as versões C++14, C++17 e C++20. A próxima versão a ser lançada, no momento da publicação desta apostila, será a C++23.

1.1.2 C++ Moderno

Ao longo da maior parte desta apostila, utilizaremos a versão C++11 da linguagem para fins didáticos, porém, caso alguma funcionalidade das novas versões se relacione com algum tópico abordado, ela será mencionada brevemente, utilizando o seguinte formato:

> **C++ Moderno:** É importante conciliar as novas mudanças do C++ em seu estudo da linguagem!

Além disso, caso queira consultar um conteúdo mais aprofundado acerca dessas novas funcionalidades, teremos um capítulo exclusivo ao final desta apostila abordando um pouco dessas mudanças em cada padronização.

1.2 Motivações

Embora a linguagem C++ tenha sido criada há um tempo relativamente longo, tendo em vista o surgimento constante de novas linguagens nos dias atuais, ela ainda é uma das principais linguagens utilizadas atualmente.

[1] No Brasil, a ISO é representada pela Associação Brasileira de Normas e Técnicas (ABNT).

Entre os principais motivos pelos quais C++ se mantém fortemente relevante, destacam-se o seu poder, velocidade de execução dos programas, portabilidade e escalabilidade. Veremos mais sobre eles a seguir.

- **Poder**: C++ é uma linguagem muito poderosa no sentido de que, devido às suas características de baixo nível, ela permite ao programador utilizar a maioria dos recursos da máquina em que é programada;
- **Velocidade de execução**: por ser uma linguagem compilada, não precisa passar por um interpretador para que seja executada. Isso poupa bastante tempo e garante uma performance maior na execução das instruções;
- **Portabilidade**: existem compiladores C++ para vários tipos de máquinas e sistemas operacionais. Além disso, as bibliotecas da linguagem foram criadas com o intuito de poderem ser lidas por esses compiladores. Isso permite que o programa produzido para um Windows funcione também para um Linux, por exemplo;
- **Escalabilidade**: em algum momento de um projeto, geralmente existe a intenção ou a necessidade de fazê-lo crescer em escala. Para que isso ocorra, a linguagem em que ele é feito precisa possuir recursos que permitam com que o código cresça em tamanho sem aumentar muito a sua complexidade. C++ trouxe isso principalmente através do paradigma de orientação a objetos.

Por esses motivos, a linguagem C++ é a principal escolhida quando se deseja criar softwares de alta performance, tais como os da Microsoft, Google, Facebook, Adobe, etc. Também é utilizada majoritariamente na programação de sistemas operacionais, jogos que exigem alta performance através de engines como a Unity e Unreal Engine. Por fim, C++ também é utilizado na programação de sistemas embarcados, devido a requisitos de memória, na implementação de ferramentas financeiras, devido à sua velocidade, entre outras várias áreas de aplicação.

2. Preparação

2.1 Ambiente de Desenvolvimento

Para programar em uma linguagem, é preciso utilizar ferramentas computacionais no processo. Uma alternativa, é o uso de uma *Integrated Development Environment - IDE*, no português, Ambiente de Desenvolvimento Integrado.

As *IDE's* são softwares que unem em sua área de trabalho recursos de desenvolvimento de programas que originalmente são utilizados separados, ou seja, a cada vez que é necessário realizar uma determinada atividade, precisa-se recorrer a uma ferramenta diferente, se não for utilizado uma *IDE*. Dessa forma, utilizar uma *IDE* garante maior produtividade, pois em um único programa você conta com as principais ferramentas de desenvolvimento de código [1].

As principais ferramentas presentes nas IDE's estão listadas a seguir:

- **Editor de texto:** nele escrevemos o código da linguagem que está sendo utilizada;
- **Compilador:** transforma o código em linguagem de máquina;
- **Linker:** permite ligar as partes do código de máquina que foi compilado e cria um programa executável;
- **Debugger:** o debugger é uma ferramenta que permite que o programador encontre erros no código programado

Existem muitas opções de IDE's no mercado de software, uma das mais utilizadas para desenvolver código em C + + é o Code::Blocks. Tal ferramenta vai ser utilizada nesta apostila devido a ser Open Source e estar disponível nos sistemas operacionais mais populares do mercado: Windows, Linux e MacOS.

O Code::Blocks contém muitos recursos, nele podemos editar texto, compilar, fazer o processo de *Debugging*, etc [2]. O editor de texto do Code::Blocks, além de permitir escrever o código, contém ferramentas de auto completar a sintaxe do código, isso é útil para quando não se lembra de todos

os detalhes de sintaxe.

Já para compilar, temos suporte a diversos tipos de compiladores, vamos utilizar o *MinGW*. Por fim, no processo de debugging, a IDE utiliza o debugger da GNU, ele permite utilizar pontos de parada na execução do código e visualizar como os dados armazenados estão mudando durante a execução, esse procedimento permite encontrar erros no programa.

Ao longo desta apostila, mostraremos como utilizar cada um dessas funcionalidades no desenvolvimento de programas em C++. Portanto, não se preocupe se parecerem abstratas cada uma dessas funcionalidades.
[1] `http://www-usr.inf.ufsm.br/~alexks/elc1020/artigo-elc1020-alexks.pdf`
[2] `https://www.codeblocks.org/features/`

2.2 Instalação

2.2.1 Windows

Abra o navegador de sua preferência e acesse o link destacado: `http://www.codeblocks.org/downloads/binaries/#imagesoswindows48pnglogo-microsoft-windows` . Esse link direciona para a página de downloads do site do projeto Code::Blocks, onde é possível baixar a IDE.

Procure pelo modelo de sistema operacional do seu computador, nesse caso, estamos utilizando o Windows. É possível utilizar várias opções de arquivos de instalação do programa, como nosso curso é introdutório, assumimos que você não tem nenhuma ferramenta para desenvolver em C/C++ instalada em sua máquina, dessa forma, clique no link de sua preferência em frente ao nome "codeblocks-20.03mingw-setup.exe", o download começará automaticamente.

Ao finalizar o download, procure a pasta onde o arquivo foi baixado e clique em codeblocks-20.03mingw-setup. Ao clicar no arquivo baixado o processo de instalação vai começar, caso você entenda o que está fazendo pode alterar algumas das recomendações de instalação feita pelos desenvolvedores, porém, recomendamos que aceite as recomendações padrão.

Por fim, ao finalizar a instalação, abra o Code::Blocks. A figura 1 mostra como é a interface do programa.

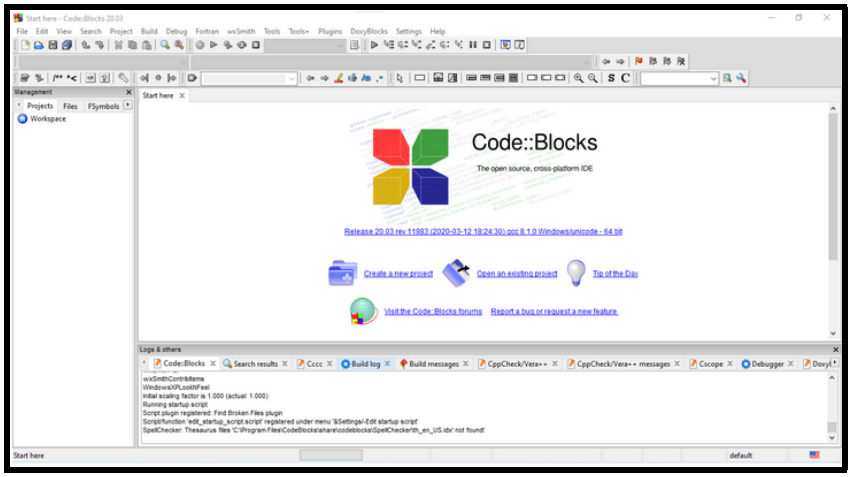

Figura 2.2.1: *Interface principal do Code::Blocks no Windows.*

Para concluir a instalação precisamos informar à IDE onde está o arquivo do compilador. Para isso, na parte superior da figura acima, clique em **Settings** e em seguida em **Compiler**. Vai ser aberta uma nova janela no programa, como na figura abaixo, siga os passos mostrados nas legendas das figuras a seguir para terminar o procedimento.

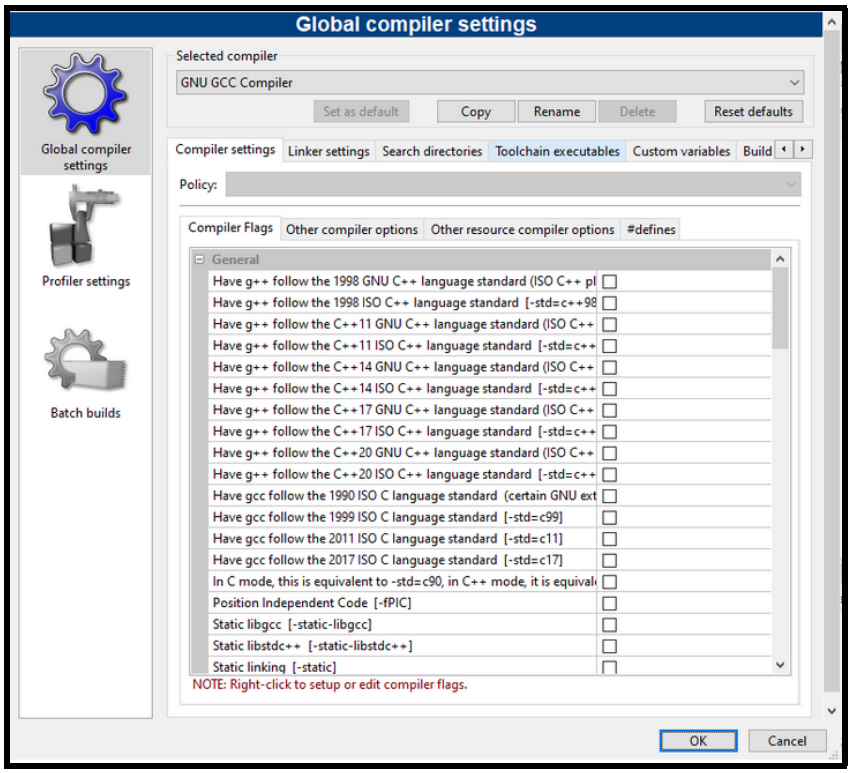

Figura 2.2.2: *Vai ser aberta a janela "Global Compiler Settings", clique em* ***Toolchain executables****.*

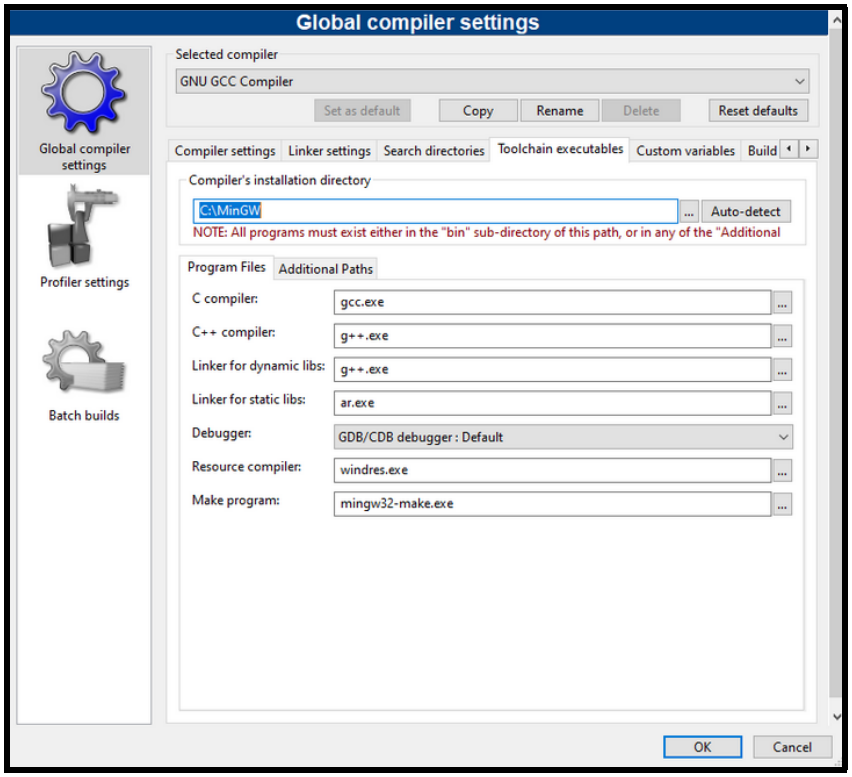

Figura 2.2.3: *Em sequência, clique em* ***Auto-detect*** *para encontrar o diretório do compilador.*

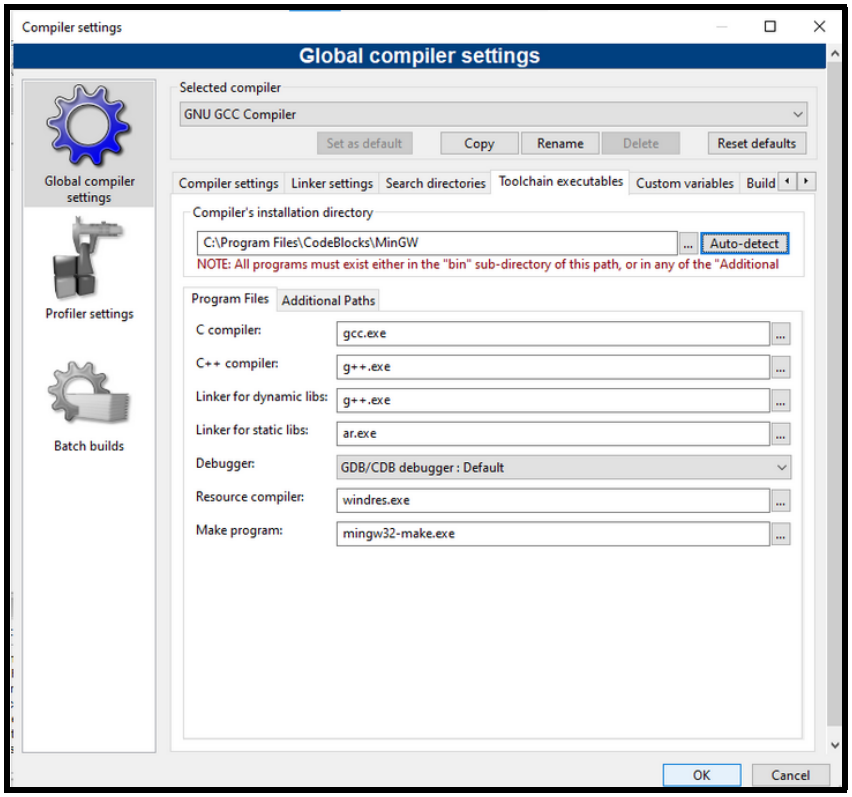

Figura 2.2.4: *Por fim, o diretório do compilador vai ser encontrado e você pode finalizar clicando em* ***OK****. Agora o Code::Blocks está pronto para ser usado no minicurso.*

2.2.2 Linux

No Linux, a instalação do CodeBlocks é feita por meio do terminal. É possível abrir um terminal na maioria das distribuições por meio do atalho CTRL+ALT+T.

Para instalar o Code::Blocks, execute (ou seja, copie, cole e aperte a tecla Enter) no terminal:

```
$ sudo apt install codeblocks
```

OBS: A senha do administrador será pedida pelo sistema para executar este comando.

É necessário também instalar as bibliotecas gcc, gpp, g++ e mingw-w64 por meio do seguinte comando:

```
$ sudo apt install gcc gpp g++ mingw-w64
```

OBS: Essas quatro bibliotecas são obrigatórias para o funcionamento do CodeBlocks.

Após a execução desses comandos, o Code::Blocks está instalado e pronto para ser utilizado.

2.3 Meu Primeiro Programa

Hello World

Com o Code::Blocks instalado podemos fazer um primeiro programa em C++. Trata-se de um código bem simples, que consiste em programar o texto “Hello World!"como saída na tela do computador. Antes de começarmos a programar devemos criar o arquivo do programa, “hello_world” Para isso, com o Code::Blocks aberto iremos em “File”, “New” e “Project...”.

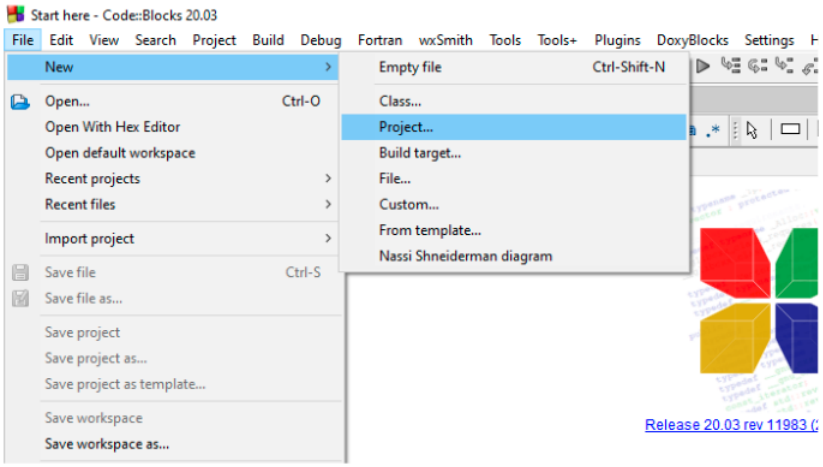

Figura 2.3.1: *Instruções de como criar o projeto de um programa no Code::Blocks*

Na tela seguinte escolhemos C++ como a linguagem na qual iremos programar.

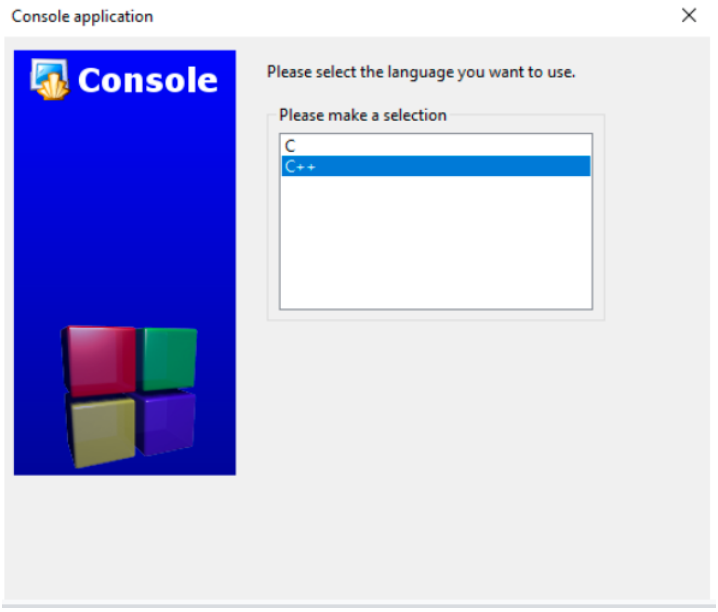

Figura 2.3.2: *Escolhendo C + + como linguagem de programação*

Seguindo, nomeamos o projeto como “hello_world” e escolhemos o local para salvá-lo.

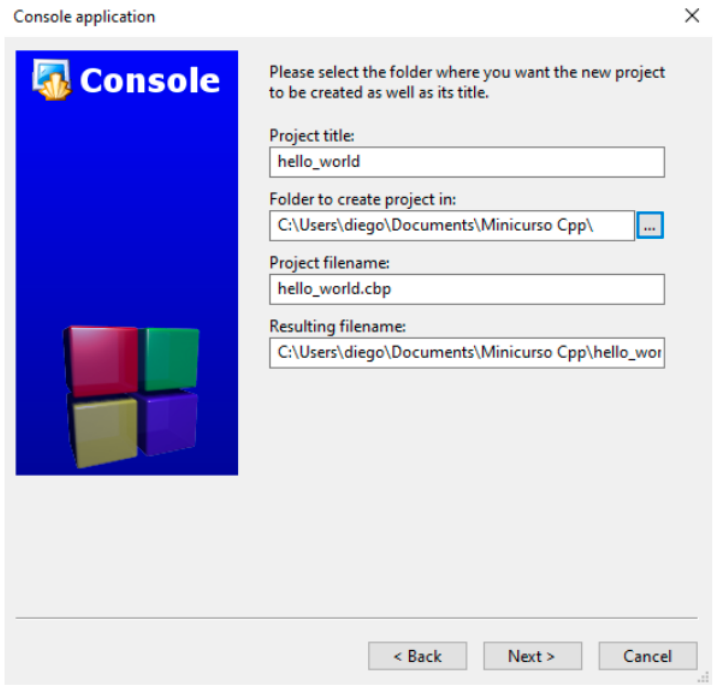

Figura 2.3.3: *Nomeando e escolhendo o local para salvar o projeto*

Na tela seguinte, precisaremos escolher qual compilador usar, a princípio deixaremos as opções sugeridas, processos de compilação serão abordados posteriormente.

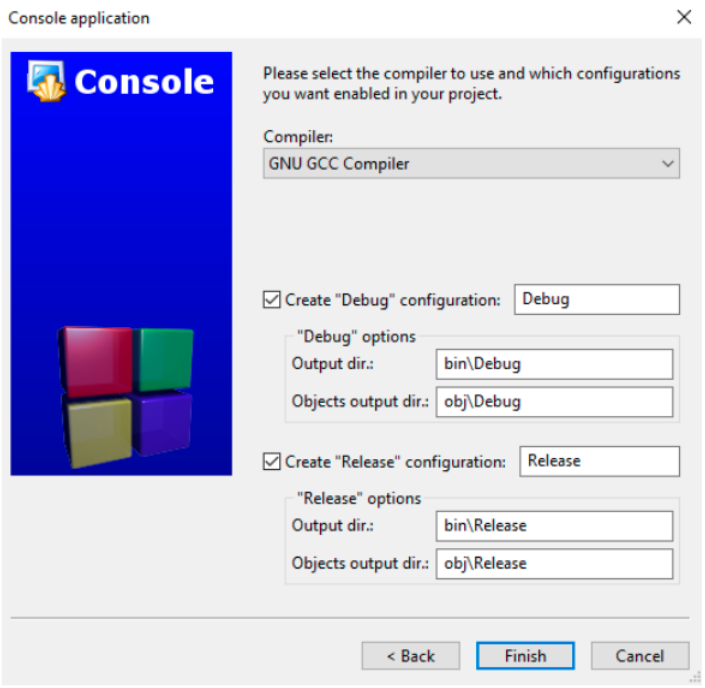

Figura 2.3.4: *Escolhendo o compilador padrão*

Feito isso, teremos o projeto de nosso programa criado .Para acessar o programa iremos em "Workspace", "hello_world", "Sources" e "main.cpp."

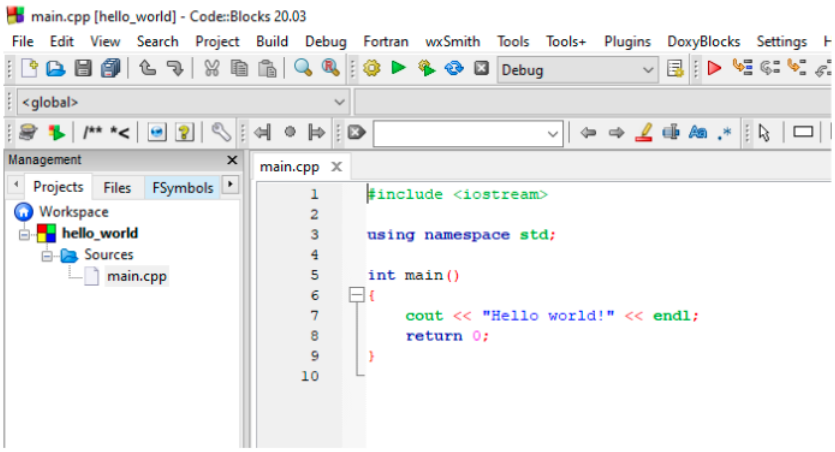

Figura 2.3.5: *Projeto criado para edição e compilação*

II Conceitos Básicos

3. Processos de Compilação e Linkagem

3.1 Processo de Compilação

Para compilar um código em C++ é necessário utilizar um compilador, que é um programa que possui duas tarefas principais:
A primeira é analisar todo o código descrito no arquivo (.cpp) e verificar se possui algum tipo de erro, isto é, verificar se existe algum erro de sintaxe, se existe alguma palavra no código, cujo significado não foi determinado e outros possíveis erros. Caso existam erros detectáveis pelo compilador, este irá abortar o processo de compilação e uma mensagem de erro será exibida no log do CodeBlocks, ou no log da IDE de sua preferência, indicando a linha onde o erro se encontra e o que gerou esse erro.
A segunda tarefa realizada pelo compilador é traduzir o código fonte em C++ para um arquivo de linguagem de máquina, cuja extensão é chamada de object (.o). Isso é necessário, pois o processador consegue executar apenas instruções binárias, de modo que o código em uma linguagem de programação qualquer precisa ser traduzido para a linguagem de máquina para que possa ser executado.
É comum, em C++, que um programa seja composto por diversas partes integradas entre si, por exemplo, podemos definir uma calculadora básica que realiza as operações de soma e adição no arquivo calculadora.cpp que possui as funções soma() e subtracao(), funções estas que estão implementadas em soma.cpp e subtracao.cpp, respectivamente. Dessa forma, ao compilar o código em calculadora.cpp, serão gerados os seguintes códigos de máquina: calculadora.o, soma.o e subtracao.o.

3.2 Processo de Linkagem

Após o processo de compilação são necessários outros passos para que possamos utilizar o programa implementado em C++, já que ainda não possuímos um arquivo capaz de ser utilizado no computador. Portanto, é necessário utilizar o Linker, que possui 2 tarefas principais:

1. Reunir todos os componentes object(.o), gerados pelo compilador, em um único executável (.exe), numa ordem que faça sentido para o processador. Voltemos ao exemplo de imple-

mentação de uma calculadora, nele temos que o arquivo calculadora.cpp possui uma relação com as componentes soma.cpp e subtracao.cpp, portanto o Linker deverá fazer uma conexão entre essas partes. Caso ele não consiga realizar essa conexão, o processo de linkagem será abortado e uma mensagem de erro será enviada ao programador.

2. Reunir a biblioteca padrão da linguagem e outras possíveis bibliotecas utilizadas durante a programação. **As bibliotecas possuem funções pré-determinadas para realizar manipulações de dados, de modo que muitas vezes é mais interessante utilizar as ferramentas implementadas em cada biblioteca ao ter que fazê-lo manualmente**.

Por fim, caso as duas etapas acima sejam realizadas com sucesso, um executável será gerado. O diagrama a seguir ilustra o processo de compilação e de linkagem:

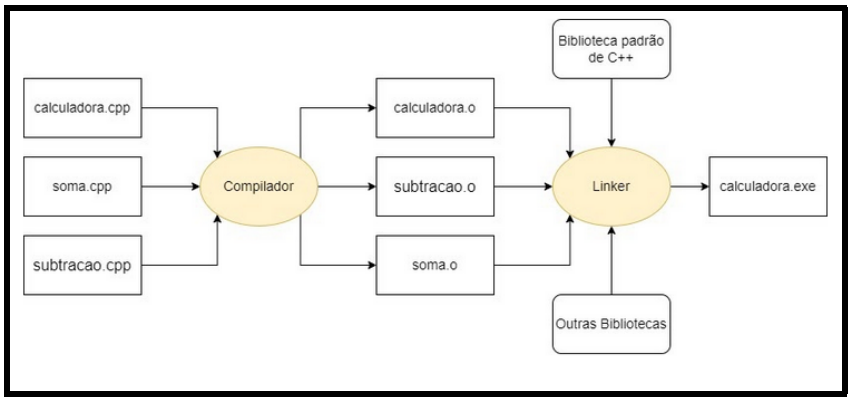

Figura 3.2.1: *Diagrama de compilação e linkagem*

4. Operadores

Operadores são símbolos utilizados em uma linguagem de programação que realizam algum processo em suas variáveis. Cada operador possui um símbolo associado.

4.1 Operadores Aritméticos (+,-,*,/,%)

Esses operadores representam operações matemáticas básicas. **Operadores Aritméticos não funcionam para o tipo de dados string**
Para todos os exemplos da tabela abaixo a,b,c são inteiros(ou seja, do tipo de dados int) e considere que a = 4, b = 5 e c é a variável que armazena o resultado da operação:

Operador	Operação	Sintaxe da Operação	Valor de c após a operação
<	Vê se um número é menor que o outro	c = a + b;	c = 9
-	Subtrai dois números	c = a - b;	c = -1
*	Multiplica dois números	c = a * b;	c = 20
/	Divide dois números	c = a / b;	c = 0
%	Retorna o resto da divisão de dois números	c = a % b;	c = 4

Note que o resultado da divisão foi 0. Isso ocorre porque no C++ valores após a vírgula não são mostrados na divisão de números inteiros. Para representar esses valores, é necessário utilizar o tipo de dados **float**.
O código abaixo mostra essas operações:

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

```
6       // Inicializando as variáveis
7       int a = 4;
8       int b = 5;
9       int c;
10
11      // Realizando as operações e imprimindo os resultados
12      c = a + b;
13      cout << "A soma de a e b é "<< c <<endl;
14      c = a - b;
15      cout << "A diferença de a e b é "<< c <<endl;
16      c = a * b;
17      cout <<"A multiplicação de a e b é "<< c <<endl;
18      c = b / a;
19      cout <<"A divisão de a por b é "<< c <<endl;
20      c = a \ b;cout « "O resto da divisão de a por b é «< c «
           endl;return 0;
```

4.1.1 Operadores Aritméticos com Atribuição (+=,-=,/=,*=,%=)

Os 5 operadores aritméticos básicos podem ser feitos com atribuição, ou seja:

```
a += b;
```

O código acima é equivalente a:

```
a = a + b;
```

Essa operação com atribuição pode ser feita com todos os 5 operadores básicos.

4.2 Operadores Aritméticos para Strings

Dos 5 operadores aritméticos, **apenas o operador de adição(+) funciona em strings**. Os outros operadores resultarão em erros. Para mais informações, veja a seção de **strings**.

4.3 Operadores Comparativos (<,<=,>,>=,==,!=)

Representam operações de comparação entre duas ou mais variáveis, ou seja, se um número é maior que, menor que, igual ou diferente do outro.
No C++ existem 2 possibilidades para o resultado de operadores de comparação: 0 ou 1. Se o valor for 1, isso significa que a operação é verdadeira. Portanto, se executarmos a seguinte linha de

No C++, o símbolo de = significa atribuição, ou seja, atribuir um valor a outro, enquanto que o símbolo de == representa a operação de comparação da igualdade entre dois valores. A troca entre esses dois símbolos provavelmente ocasionará erros de compilação e/ou lógicos no programa.
código:

```
c = 5 > 2;
```

O valor de c será igual a 1. Se o valor retornado pelo operador lógico for 0, significa que a operação é falsa. Portanto, se executarmos a seguinte linha de código:

```
c = 5 < 2;
```

O valor armazenado em c será igual a 0.
A tabela abaixo mostra os principais operadores comparativos, sua operação e o valor que é armazenado no inteiro c após a execução da operação:

Operador	Operação	Sintaxe da Operação	Valor final de c
<	Vê se o primeiro número é menor que o segundo	c = 3 < 5;	c = 1
<=	Vê se o primeiro número é menor ou igual ao segundo	c = 4 <= 4;	c = 1
>	Vê se o primeiro número é maior que o segundo	c = 5 > 7;	c = 0
>=	Vê se o primeiro número é maior ou igual ao segundo	c = 8 >= 7;	c = 1
==	Vê se o primeiro número é igual ao segundo	c = 9 == 9;	c = 1
!=	Vê se o primeiro número é diferente do segundo	c = 5 != 3;	c = 1

Algumas considerações importantes:

1. No C++, o símbolo de **=** significa atribuição, ou seja, atribuir um valor a outro, enquanto que o símbolo de **==** representa a operação de comparação da igualdade entre dois valores. A troca entre esses dois símbolos provavelmente ocasionará erros de compilação e/ou lógicos no programa.
2. Operadores comparativos são muito úteis para impor **condições** ao programa, ou seja, partes do código que serão executadas apenas quando uma relação matemática for verdadeira.

4.3.1 Operadores Comparativos para Strings

Uma **string** é uma estrutura de dados utilizada para armazenar blocos de texto. Os operadores comparativos também podem ser utilizados em strings e implicam diretamente na quantidade de caracteres entre as strings testadas, como mostrado na tabela abaixo:

Operador	Operação	Sintaxe da Operação	Valor de c
<	Tem menos caracteres?	c = "teste« "oi";	c = 0
<=	Tem **menos** ou o **mesmo tanto** de caracteres?	c = "teste«= "oi";	c = 0
>	Tem **mais** caracteres?	c = "teste» "oi";	c = 1
>=	Tem **mais** ou o **mesmo tanto** de caracteres?	c = "teste»= "oi";	c = 1
==	É **igual**?	c = "oi"== "oi";	c = 1
!=	É **diferente**?	c = "teste"!= "teste";	c = 0

Algumas considerações importantes:

1. Os operadores comparativos <,<=,>,>= comparam o **comprimento**(ou seja, a quantidade de caracteres) em uma string. Já os operadores == e != comparam o **conteúdo**(ou seja, se todos os caracteres são iguais e estão na mesma ordem) entre as duas strings.

4.4 Operadores Lógicos

São utilizados para determinar a relação lógica entre variáveis ou valores.
Esse tipo de operação retorna dois valores: 0 e 1. A operação retorna 0 caso a operação seja falsa, e 1 caso a operação seja verdadeira. **Operadores lógicos não funcionam em strings**.

Para a tabela abaixo, considere que a,b,c são do tipo int, que a = 0, b = 1 e c é a variável que armazena o resultado da operação.

Operador	Operação	Sintaxe da Operação	Valor de c após a operação
!	Não Lógico	c = !a;	c = 1
c = a && b;	**E** lógico		c = 0
||	**Ou** lógico	c = a || b;	c = 1

4.5 Outros Operadores

4.5.1 Atribuição =

Utilizado para armazenar valores em variáveis.

```
int a = 5;
```

No código acima, o valor 5 é **atribuído** à variável **a**. Isso significa que, a partir dessa linha de código, o programa reconhecerá que **a** é um inteiro de valor 5.

4.5.2 Chamada de Função ()

Utilizado para chamar funções no código. Para mais informações, veja a seção de **funções**

```
minhaFuncao(int parametro_a, int parametro_b);
```

4.5.3 Operadores de Memória e Operadores Relativos a Ponteiros

Operadores que são utilizados para acessar posições de memória e lidar com ponteiros. Esses operadores são mais analisados na seção de **memória**.

5. Entrada e saída de dados

5.1 E/S de dados

A entrada de dados, tal como a saída, é uma tarefa corriqueira em programação, por isso o C++ possui a biblioteca iostream, que já inclui os headers *istream* e *ostream*. Essas bibliotecas implementam objetos, como "cin" e "cout" para a entrada e para a saída de dados do programa, respectivamente, de modo que o programador não precise se preocupar em implementá-los.

Os dados de saída e de entrada do programa podem ser enxergados, na verdade, como fluxos de entrada e saída, que fluem por meio de objetos denominados streams, cuja tradução literal é "fluxo". Existem diversas formas do programa receber e escrever dados, sendo o terminal e arquivos externos os exemplos mais comuns. O presente capítulo terá o foco no uso do terminal para entrada e saída de dados.

5.2 Caracteres Especiais

C++, assim como outras linguagens de programação, possui caracteres especiais reservados, sendo que alguns deles são bem úteis para a formatação da saída de dados. A tabela a seguir contém esses caracteres:

Comando	Função
\n	Cria uma nova linha
\t	Tab horizontal (parágrafo)
\v	Tab vertical
\b	“Backspace”: apaga o último caractere
\r	Posiciona o cursor no início da linha
\f	Cria uma nova página
\a	Alerta, emite um pequeno som
\\	Cria uma barra invertida
\?	Cria uma interrogação
\’	Aspas simples
\"	Aspas duplas
\ooo	Representação octal
\xhhh	Representação hexadecimal
\0	Caractece nulo

5.3 Entrada de dados padrão

Todo valor de entrada é armazenado no buffer do teclado, que é uma parte da memória RAM dedicada a esta finalidade, antes de ser, de fato, recebido no programa. Isso é feito para evitar os atrasos existentes na transmissão dos dados, que poderiam acarretar em erros graves.

Os dados são lidos e escritos no buffer constantemente, porém há casos em que podem existir dados residuais nele, que podem acarretar em uma entrada indesejada no programa, resultando em erro em tempo de execução ou de lógica. Portanto, é necessário que o programador tome alguns cuidados, que serão detalhados na próxima seção.

Como dito anteriormente, os objetos utilizados para a entrada de dados estão definidos na classe *istream*. Nesta seção, serão detalhados dois desses objetos, por serem amplamente usados, e também será apresentada a classe “string”. O comando *“cin”* lê uma sequência de dados, inseridos por meio do teclado, e os armazena em uma variável de tipo apropriado, definido pelo programador. Portanto, sempre que seja necessário inserir um valor no programa pelo terminal pode-se utilizar esse comando. Os exemplos 1 e 2 ilustram o uso dele.

```
1
2  #include <iostream>
3
4  int main(){
5
6      // Exemplo 1:
7      int a,b;
8      std::cin >> a >> b;
9
10     // Exemplo 2:
11     char data[100];
12     int i = 0;
13         while( std::cin >> data[i]) {
14             i ++;
15         }
16
```

```
17      return 0;
18  }
```

No primeiro exemplo, são definidos dois inteiros “a” e “b” e é solicitado que o usuário insira 2 valores inteiros no programa, o primeiro valor será atribuído à variável “a” e o segundo à variável “b”.

No segundo exemplo, é definido um vetor de caracteres “data” que pode armazenar 100 valores. Em seguida, o *loop “while”*, é utilizado para preencher as posições desse vetor de tal forma que a cada valor inserido, a variável “i” é incrementada em uma unidade. Este exemplo pode parecer estranho para iniciantes em C++, pelo fato de que a função *“while”* demanda um teste condicional em seu argumento, para que o loop se repita enquanto alguma condição for verdadeira. O que acontece, de fato, é que o “cin” retorna um booleano, que é verdadeiro enquanto a entrada de dados acontece e é falso quando ela termina (ao pressionar ctrl + z).

Outro objeto muito utilizado em C++ para entrada de dados é o “getline”, que lê dados até encontrar o caractere “\n” e os salva em uma variável. Esse comando pode ser utilizado para leitura de dados do terminal, mas também é muito útil para dados advindos de arquivos externos. Os exemplos 3 e 4 ilustram o uso de “getline”. Exemplo 3 e 4

```
1   #include <iostream>
2   #include <string>  // Para Exemplo 2
3
4   int main (){
5
6       // Exemplo 3
7       char nome[256];
8       std::cin.getline(nome,256);
9
10      // Exemplo4
11      std::string strnome;
12      std::getline(std::cin, strnome);
13      return 0;
14  }
```

Para o exemplo 3, é declarado um vetor de caracteres com 256 posições e solicitado a entrada de dados utilizando o comando *“getline()”*. O primeiro argumento de *“getline()”* é o local onde serão armazenados os dados obtidos, enquanto que o segundo é o delimitador, ou seja, o tamanho máximo de caracteres que serão armazenados. Portanto, caso o usuário não insira uma quebra de linha (“\n”) antes do vetor “nome” encher, a entrada de dados será finalizada.
O exemplo 4 é um pouco mais complexo, mas faz, basicamente, a mesma tarefa. Nele é utilizada uma *“string”* que, colocando de forma simples, é uma variável que recebe uma sequência de caracteres. A *“string”* possui algumas diferenças básicas entre o vetor do tipo “char”, a primeira e mais evidente, é que uma *“string”* adapta seu tamanho à quantidade de caracteres salvos nela, o que é bem útil, já que, muitas das vezes, o programador não sabe quantos caracteres serão necessários em um vetor. A segunda diferença é que as *“string”* não terminam com o caractere “\0”, o que acontece no vetor do tipo char, em que a última posição do vetor possui o caractere nulo. Outra diferença, que também é a principal para que *“string”* sejam amplamente utilizadas, é que existem diversos métodos já implementados para a manipulação delas, facilitando e otimizando o trabalho do programador.

Retornando ao exemplo 4, é declarada uma variável do tipo *"string"* chamada "name" e, em seguida, solicitado que seja realizada a entrada de dados pelo comando *"getline()"*. A sintaxe de *"getline()"* para o presente exemplo se difere do exemplo 3; isso tem relação com a sobrecarga de métodos, que será discutida no parte 4 da apostila, sobre programação orientada a objetos.
Os exemplos a seguir mostram algumas formas de manipular strings:

```
1  #include <iostream>
2  #include <string>
3
4  int main(){
5      // Exemplo 5
6      std::string str1, str2;
7      str1 = "Uma string ";
8      str2 = "outra string";
9      std::cout << str1 + str2 << std::endl;
10
11     // Exemplo 6
12     std::swap(str1,str2);
13     std::cout << str1 << std::endl;
14     std::cout << str2 << std::endl;
15
16     // Exemplo 7
17     std::cout << " O tamanho de str1 eh igual a " <<
           str1.length()<< std::endl;
18     str1.erase(5,7);
19     std::cout << str1 << std::endl;
20 }
```

No exemplo 5, são declaradas duas strings "str1" e "str2", que contém os valores "Uma string " e "outra string", respectivamente. Em seguida, é realizada a operação de concatenação dessas strings, resultando em "Uma string outra string". No exemplo 6, é utilizado o método *"swap()"*, que troca os valores contidos em str1 pelos valores de str2, e vice-versa. Dessa forma, ao imprimir str1 aparecerá "outra string" e ao imprimir str2 aparecerá "Uma string ".

Por fim, no exemplo 7 é utilizado o método *"length()"*, que retorna o número de caracteres contidos em uma string. Também é utilizado o método *"erase()"*, que recebe 2 argumentos: o primeiro é a posição em que desejamos começar a apagar os dados da string; o segundo é o número de posições que desejamos apagar. Dessa forma, a saída da última linha do programa será "outra".
Existem diversos outros métodos já implementados para realizar operações com strings, estes podem ser acessados na documentação da biblioteca string.

5.4 Saída de dados padrão

Os objetos utilizados para a saída de dados estão definidos na classe *"ostream"*. Nesta seção será detalhada uma maneira de realizar a saída de dados, assim como as maneiras de formatá-la. A principal forma de imprimir algo no terminal é utilizando o objeto *"cout"*, que já foi utilizado em exemplos anteriores mas não foi devidamente apresentado. Ele é um objeto do tipo "ostream" bem simples de ser utilizado, de modo que seu uso será explicado por meio do exemplo 8.

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

int main(){
    std::string str1;
    str1 =  Bem   v i n d o ;
    std::cout << str1 <<     ao minicurso de C++    <<
       std::endl;
    return 0;
}
```

A saída deste programa é: “Bem vindo ao minicurso de C++”. Percebe-se então, que a impressão é sequencial, ou seja, em primeiro lugar vem o valor contido em “str1”, depois o texto inserido entre as aspas duplas, e por fim o objeto “endl”. Esse comando “endl” possui duas principais funções: a primeira é terminar a linha, semelhante ao caractere especial “”, e a segunda é limpar o buffer do teclado. Dessa forma, não é necessário que o desenvolvedor se preocupe em limpar o buffer sempre que for necessário utilizar a entrada de dados.

Muitas das vezes os usuários dos programas não se preocupam apenas com a corretude dos resultados, mas também com o modo em que estes dados são exibidos, principalmente para saídas muito extensas. Dessa forma, é necessário que o desenvolvedor se preocupe com a formatação da saída e, neste caso, os caracteres especiais serão grandes aliados, já que permitem fazê-lo com certa praticidade. Por exemplo, dado um vetor de strings que contenham o nome de um grupo de pessoas e um vetor de inteiros que contenha a idade de cada uma. Deseja-se imprimir estes dados, relacionando o nome e a idade de cada um. O exemplo 9 mostra como fazê-lo:

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

int main(){
    std::string nomes[6] = {"Carlos", "Fábio", "Maria",
       "Matheus", "Luiza", "Lucia"};
    int idade[6] = {13 , 19, 15, 25, 43, 22};
    for(int i = 0; i < 6; i++){
        std::cout << "Nome : " << nomes[i] <<    \ t   <<
           " Idade: " << idade[i];
        std::cout <<    \ n  ;
    }
    return 0;
}
```

A saída do exemplo 9 pode ser vista na imagem 1, a seguir. Nele foi utilizado o caractere especial “\t” para tabulação e o caractere “\n” para a quebra de linha.

5.5 Saída padrão de erros

A biblioteca ostream também possui um objeto para a saída de erros padrão, denominado cerr. Tanto o cout quanto o cerr funcionam de uma forma parecida, isto é, ambos imprimem mensagens

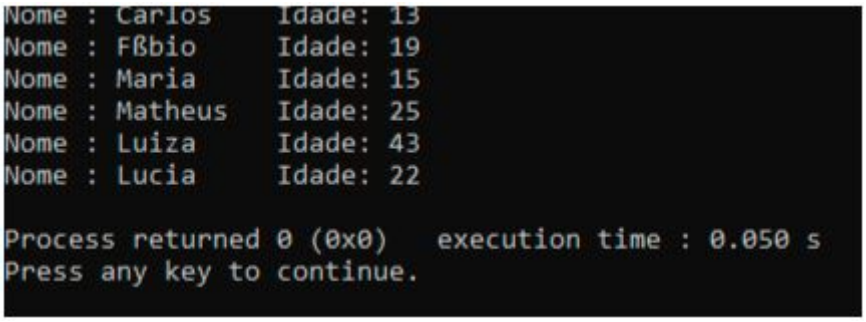

Figura 5.4.1: *Saída do exemplo 9*

na tela e não existe nenhum marcador que diferencie uma mensagem gerada por um cout ou por um cerr, o que pode levar programadores iniciantes a pensar que ambos servem ao mesmo propósito. A primeira diferença entre cout e cerr é que o cerr não utiliza o buffer para guardar a mensagem a ser exibida, já que ele serve para reportar erros imediatamente, de modo que não é necessário salvar a mensagem para exibi-la depois. Uma outra característica distinta será discutida no exemplo 10.

```
1 #include<iostream>
2
3 int main (){
4     std::cout <<   Isto   é um cout << std::endl;
5     std::cerr <<   Isto   é um cerr << std::endl;
6     return 0;
7 }
```

A saída para o programa contido no exemplo 10 é a seguinte:

```
Isto é um cout
Isto é um cerr
```

Percebe-se que, pela saída do terminal, ambos os comandos exibem mensagens da mesma forma, sem distinção entre um ou outro. No entanto, caso a saída do programa utilize arquivos, o que acontecerá é que os dados serão escritos em um arquivo e os erros em outro. Isso porque, em geral, interessa ao usuário de um determinado programa apenas os resultados obtidos por ele. Enquanto que os desenvolvedores do programa interessam, ou deveriam interessar, em possíveis erros gerados durante a execução do programa, tendo em mente que a análise desses erros permite aprimorar o código, tornando-o mais robusto.

6. Controle de fluxo

Na programação, associa-se ao controle de fluxo a ordem em que as instruções, expressões e chamadas de funções serão executadas e avaliadas. Pode-se dividir esse controle em 3 estruturas: Estrutura sequencial, Estrutura condicional e Estrutura de repetição. Nesta parte, será abordada apenas as duas últimas, uma vez que a estrutura sequencial é diluída nelas em C++.

6.1 Expressões Booleanas

6.1.1 Tipo *Bool*

Em uma breve recapitulação, as operações booleanas retornarão apenas 0 ou 1. Quando são feitas operações entre variáveis do tipo *bool*, nós podemos usar tabelas e expressões matemáticas, que serão representações já conhecidas da matemática, como o * e o +, mas em C++, será usado os **operadores**, que foram introduzidos anteriormente. A seguir, foi feita uma tabela para apresentar estes operadores:

&&	Retornará *true*, se as duas partes forem verdadeiras
||	Retornará *true*, se uma ou as duas partes forem verdadeiras
!	Será usada apenas uma parte e sempre retornará o contrário do que estiver nela.

6.1.2 Outros tipos

Nesta parte, serão introduzidas as operações de comparação entre duas partes de mesmo tipo. Essas são importantes para validar ou testar duas partes entre si. Entender como usá-las ajudará a compor outras estruturas mais adiante. Nesta parte também é feito o uso dos operadores, agora para comparar os dois tipos.

==	Retornará **true**, se as duas partes forem iguais
!=	Retornará **true**, se as duas partes forem diferentes
>	Retornará **true**, se a primeira for maior
>=	Retornará **true**, se a primeira for maior ou igual
<	Retornará **true**, se a primeira for menor
<=	Retornará **true**, se a primeira for menor ou igual

6.2 Estruturas Condicionais

6.2.1 If-Else

Nesta parte, será introduzido outro conceito bastante importante na dinâmica dos códigos: as estruturas condicionais. Essa parte compõem a tomada de decisões do código, podendo montar uma gama de cenários dadas as variáveis e possíveis respostas que ela possa ter. Essas estruturas farão uso das expressões booleanas em sua composição, por isso é importante ter pego o conceito na parte anterior. A primeira estrutura a ser inserida será o **If-Else**. Nesta estrutura, é dada forma a seguinte expressão: "Se **a** for **verdadeiro**, faça **x**, senão, faça **y**". Quando é passada uma expressão ao If, faz-se o programa guiar o código para dois caminhos chamados declarações, logo o **If** e o **Else**, terão declarações distintas. Por isso são utilizadas as expressões booleanas, pois é notável que **If** tenha só, e somente só, duas respostas possíveis: *true* ou *false* . É importante lembrar que não foi inserida uma condição em **Else**, já que ele irá executar sua funcionalidade sempre que o **If** não for *true*.

```
1 int entrada1, entrada2;
2
3 cin >> entrada1;
4 cin >> entrada2;
5
6 if(entrada1 > entrada2) return 1;
7 else return 0;
```

No exemplo acima, foi usada de uma característica interessante da programação: o uso dos blocos de código e das chaves. As chaves são usadas em C++ para conter um bloco, que por sua vez é o que irá compor uma declaração em uma estrutura. No código, foi usada apenas uma linha dentro do **If** e uma linha dentro do **Else**, então não precisamos usar os . Saber quando usar as chaves em programação pode tornar seu código mais limpo e, às vezes, até mais eficiente.
Uma funcionalidade interessante é a de usar o If sozinho, ou seja, sem o acompanhamento de Else, pois nem sempre será necessário uma segunda condição e será requerida apenas uma declaração nesta parte do código.

```
1     int idade;
2   cout <<   Qual   sua idade?    ;
3   cin >> idade;
4
5   cout <<   Lista   de filmes de super-heróis: \ n  ;
6   cout << 1   - Pantera negra \ n  ;
7   cout <<   2   - Homem-aranha no aranhaverso \ n  ;
8   cout << 3   - Guardiões das galáxias \ n  ;
9   if (idade >= 18){
10  cout <<   4   - Deadpool \ n  ;
11  cout << 5   - O esquadrão suicida \ n  ;
12 }
```

Neste terceiro exemplo, as linhas antes do **If** serão imprimidas independente da idade inserida, porém se a idade for *maior* ou *igual* a 18, todas as linhas serão imprimidas, incluindo as do bloco de **If**, já que a condição foi satisfeita. Note que não foi inserido um **Else**, já que não era requerido um bloco alternativo.
Outra funcionalidade que será inserida é utilizar a composição **Else if**. Esta será usada sempre que for preciso conduzir a partir de 3 condições possíveis, pois **Else if** irá receber uma condição além da que foi recebida pelo **if**.

```
1     int entrada1, entrada2;
2
3     cin >> entrada1;
4     cin >> entrada2;
5
6     if (entrada1 == entrada2) cout <<    Iguais    << endl;
7     else if (entrada1 > entrada2) cout <<    maior    <<
        endl;
8     else if (entrada1 < entrada2) cout <<    menor    <<
        endl;
9     else cout <<    ERROR    << endl;
```

No exemplo 4, será conduzido o código a partir das duas entradas recebidas. O código irá fazer 4 comparações e então executar a linha da condição satisfeita. Note que irá ser executada apenas uma linha, logo a última comparação nunca será mostrada, pois ocorrerá que condição dela é a união de outras duas comparações anteriores a ela.
Um conceito que será introduzido aqui é a diferença entre normal e encadeado. Enquanto no normal tem-se uma estrutura simples, utilizando apenas **If, Else if e Else** em um único fluxo, onde será tomada apenas uma decisão, no encadeado, tem-se múltiplos fluxos que nos possibilita fazer dentro de uma declaração outra estrutura condicional e assim ir compondo vários caminhos e construindo o que é chamada de árvore de condições

```
1     int opcao;
2
3     cout <<    Voc    gosta de filmes de terror? 1(sim)
        2(nao)    ;
```

```
4       cin >> opcao;
5
6       if(opcao == 1){
7         cout <<    Voc    gosta de filmes que dao susto?
             1(sim) 2(nao)    ;
8         cin >> opcao;
9         if(opcao == 1) cout <<   Assista   Invocacao do
             M a l  ;
10        else cout <<   Assista   Hereditario!
11      } else cout <<   Otimo  ! Eu também nao, prefiro os
          rom nticos!   ;
```

```
1   int opcao;
2   char jogo;
3
4   cout <<    Voc    gostaria de jogar video game? 1(sim)
       2(nao)    ;
5   cin >> opcao;
6
7   if(opcao == 1){
8       cout <<   Otimo  ! Voce gostaria de jogar futebol, luta
          ou tiro?    digite f, l ou t    ;
9       cin >> jogo;
10
11      if(jogo ==    f   ) cout <<   Certo  , voce quer jogar
          Fifa ou Pes?   ;
12      else if(jogo ==    l   ) cout <<   Certo  , voce quer
          jogar Mortal Kombat ou Injustice?   ;
13      else cout <<   Certo  , voce quer jogar Fortnite ou COD
          Warzone?   ;
14  } else cout <<  Que   pena! Eh tao divertido!
```

No quinto código, tem-se um encadeamento simples, onde tem-se apenas 2 decisões, sendo que uma não seria mostrada ao usuário dada a opção escolhida, o que resulta em 3 possibilidades diferentes. Já no sexto código, tinha-se mais opções, o que nos resultou em 4 possibilidades diferentes, mas podendo ser mais já que foi oferecida apenas 3 opções na decisão do jogo. Um fator que é importante ressaltar aqui é o caminho das decisões imprevistas ou não listadas, já que o usuário pode acabar escolhendo uma opção que não exista. Este caso foi contornado usando o comando **Else** apenas para essas possibilidades, uma vez que ele abrange qualquer escolha que não esteja condicionada num possível **If** ou **Else if**.

6.2.2 Switch-case

Essa segunda estrutura apresentada lidará com o conceito de fazer o fluxo em função de uma variável, uma vez que pode ser interessante não precisar declarar sempre grandes sentenças, como acontece no aninhamento de **If-Else**, que nada mais é que um excesso de encadeamento. Essa estrutura funcionará como uma composição de vários **operadores ==**, onde será passada a variável desejada para o **Switch** e em cada **Case**, será colocada uma variável. Caso a variável em **Case** tenha o mesmo valor que a variável de **Switch**, é feita a declaração agregada a ele e ´´e encerrado

este fluxo. Como proposto no caso do **If-Else**, pode-se colocar um **Switch** dentro de outro **Switch**, pois o conceito de encadeado também funciona para esta estrutura.

```
1  cout << " Voce gostaria de jogar futebol, luta ou tiro?"
       cout << "digite f, l ou t ";
2  cin >> jogo;
3
4  switch(jogo){
5      case (   f   ):
6      cout <<   Certo  , voce quer jogar Fifa ou Pes?   ;
7      break;
8          case (   l   ):
9      cout <<   Certo  , voce quer jogar Mortal Kombat ou
           Injustice?   ;
10     break;
11         case (   t   ):
12     cout <<   Certo  , voce quer jogar Fortnite ou COD
           Warzone?   ;
13     break;
14     default:
15       cout <<   Voce   nao digitou algo que eu pude
           identificar!   ;
16 }
```

Neste exemplo, o código irá passar por todos os **Case**, onde o primeiro a ter sua condição satisfeita, terá o seu bloco executado.

6.3 Estrutura de repetição

Um dos pilares mais importantes dentro das linguagens de programação são os laços, que são responsáveis pelos chamados loops, que são um comportamento usado para fazer tarefas onde é preciso executar várias vezes um mesmo comando. Na programação, chamaremos cada repetição de *iteração* por padrão, e se o laço conter mais de uma linha, chamaremos essa parte de bloco. Em C++, nós podemos usar o **For, While** e o **Do-while**, cada estrutura adequada para um certo tipo de código a ser implementado.

6.3.1 For

A primeira estrutura que abordaremos é o **For**. Esta estrutura receberá uma variável do tipo *Int*, que deverá começar em qualquer valor maior ou igual a 0, deve receber uma condição de comparação para indicar o momento de interrupção, e deve ser receber a ação de atualização, que deve ser a maneira como a variável do tipo *Int* deve crescer ou diminuir. Geralmente usa-se o **For** em situações onde sabemos exatamente quantas iterações queremos naquela estrutura.

```
1
2  for(int numero = 1; numero <= 6; numero++){
3    cout <<   Iteracao  :     << numero << endl;;
4  }
```

Neste código, será imprimir 6 vezes uma frase que irá mostrar sempre o número correspondente àquela iteração. Note que ali, o início é o número 1 e que foi incluído o 6 na condição, porém é comum começar pelo número 0, já que como será visto mais à frente, usamos essa função para preencher vetores.

6.3.2 While

A segunda estrutura que será abordada é o **While**. Esta estrutura receberá apenas uma condição de funcionamento, onde enquanto essa condição não for satisfeita, o laço continuará rodando. Esta condição pode depender do usuário se implementarmos uma entrada de valor, pode depender de uma função ou pode desempenhar um papel semelhante ao de **For**, onde já foi escrita a condição de parada.

```
1 int dias = 1;
2
3   while (dias < 365){
4     cout <<"Mais um dia se passa. Estamos no dia " << dias
        << " \n";
5     dias++;
6     }
7
8     cout << "Chegamos no dia " << dias << endl;
9     cout << "Completamos um ano, finalmente!!!\n";
```

Neste código, será imprimido o dia que se encontra, que seguirá crescendo a cada iteração. Note que ao chegar em 365, o loop se encerra e irá para a próxima linha de fora do bloco. Nesta estrutura, verifica-se a condição e só então ler a próxima linha, logo não entrará no bloco se os dias forem maiores que 365.

6.3.3 Do-While

Essa terceira estrutura é uma variante direta do **While** e serve basicamente para ser uma opção à parte por conta de uma particularidade no momento de execução. Enquanto em **Do-While**, a declaração será executada uma vez mesmo que a condição não seja satisfeita, no **While** a declaração não será executada nenhuma vez se a condição não for satisfeita.

```
1 int dias = 1;
2
3 do {
4     cout <<  Mais   um dia se passa. Estamos no dia     <<
        dias <<     \ n  ;
5     dias++;
6 }   while (dias < 365)
7
8 cout <<   Chegamos   no dia     << dias << endl;
9 cout <<   Completamos   um ano, finalmente!!!\ n  ;
```

Neste código, será imprimido o dia que estamos, que seguirá crescendo a cada iteração. Note que ao chegar em 365, o loop se encerra e irá para a próxima linha de fora do bloco. Note que neste

código, entramos no bloco da estrutura pelo menos uma vez, para então verificar a condição, logo terá imprimido a linha do bloco mesmo se dias fosse igual ou maior que 365

6.3.4 Loops Infinitos

Um conceito muito interessante são os loops infinitos, que traz o poder da recursividade. Com programas recursivos, pode-se inserir condições onde o código poderá rodar enquanto for necessário, podendo ser por vontade do usuário ou por alguma função ou condição de parada pré determinada. Este conceito nasce com a necessidade de programas cada vez mais autônomos e interativos.

```
1 char opcao =    s   ;
2
3 while( opcao ==    s     ){
4
5   cout <<   Se   voce deseja continuar, digite s. Caso
       queira sair, digite qualquer letra \ n  ;
6   cin >> opcao;
7 }
```

Neste código, o loop só irá terminar, se for digitado algo diferente de ‘s’, logo esse loop se comporta como infinito por conta dessa condição única.

6.3.5 Break e Continue

Os desvios são declarações que são usadas quando quer-se interromper algum laço. Neste caso, tem-se duas declarações possíveis: **Break** e **Continue**. O **Break** irá encerrar o laço no ponto onde foi colocado, partindo para o próxima linha fora do bloco dele, enquanto o **Continue** irá voltar para o início do bloco, não permitindo que ele chegue até o final.

```
1 for(int i = 0; i < 5; i++){
2
3   for(int j = 0; j < 5; j++){
4     if(i == j) continue;
5         cout << "Linha " << i << "Coluna " << j << endl;
6     }
7
8 }
```

Neste código, será imprimido o número da linha e o número da coluna. Observe que tem-se uma condição, onde se o número da linha for igual ao número da coluna, não vamos imprimir a linha que indica o número de cada, logo será aplicado o **Continue**, que irá encerrar o laço ali mesmo e irá pular direto para o início do laço.

```
1 for(int i = 0; i < 5; i++){
2
3     for(int j = 0; j < 5; j++){
4         if(i == j){
5             cout <<   Error  !    << endl;
```

```
6                 break;
7             }
8             cout << "Linha " << i << "Coluna " << j << endl;
9         }
10
11    }
```

Neste código, será imprimido o número da linha e o número da coluna. Observe que tem-se uma condição, onde se o número da linha for igual ao número da coluna, será usado o **Break**, que irá parar o laço que numera as colunas daquela linha.

7. Bibliotecas e Namespaces

Através do primeiro programa, *hello_world.cpp*, foi possível analisar a estrutura básica de um programa em C++. Neste capítulo, algumas das instruções que compõem essa estrutura serão abordadas mais a fundo.
Mesmo para os leitores que já tiveram contato com a linguagem predecessora, é interessante analisar esta discussão, pois embora a estrutura seja parecida com o que se vê na linguagem C, muitos pontos se diferem quanto à sintaxe, além de incluir novos conceitos, como os **Namespaces**.
Nesta seção, serão abordados os conceitos de **Bibliotecas**, **Diretivas de pré-processamento** e **Namespaces**. Alguns conceitos, como a função *main*, operadores e streams serão abordados em capítulos posteriores.
Abaixo, é mostrado o código do programa *hello_world.cpp* e todas as linhas dele são explicadas brevemente na tabela 2.1.

```
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7   cout << "Hello world!" << endl;
8   return 0;
9 }
```

Linha	Codigo	Descrição
1	#include <iostream>	Inclui no programa a biblioteca padrão da stream de entrada e saída de dados.
3	using namespace std;	É uma diretiva que faz com que o escopo do namespace std seja utilizado no escopo global do programa.
5	int main()	É a função principal, que recebe parâmetros do sistema e retorna a ele um valor inteiro.
6 e 9	{ }	Define o bloco que indica o escopo da função main. Tudo o que estiver entre as linhas 6 e 9 fazem parte da definição da função main.
7	cout <<“Hello world!” <<endl;	Escreve a sequência de caracteres na tela através da stream de saída de dados e salta uma linha ao fim da instrução.
8	return 0;	É o valor de retorno da função main. O valor 0 indica que não ocorreram problemas durante a execução do programa.

7.1 O Pré-processador

No próximo capítulo serão discutidas as etapas de processamento do código fonte, porém, existe uma etapa anterior chamada *pré-processamento*. Nela, ocorre a inclusão de **bibliotecas** e arquivos, condições de compilação, definição de **macros**, etc.

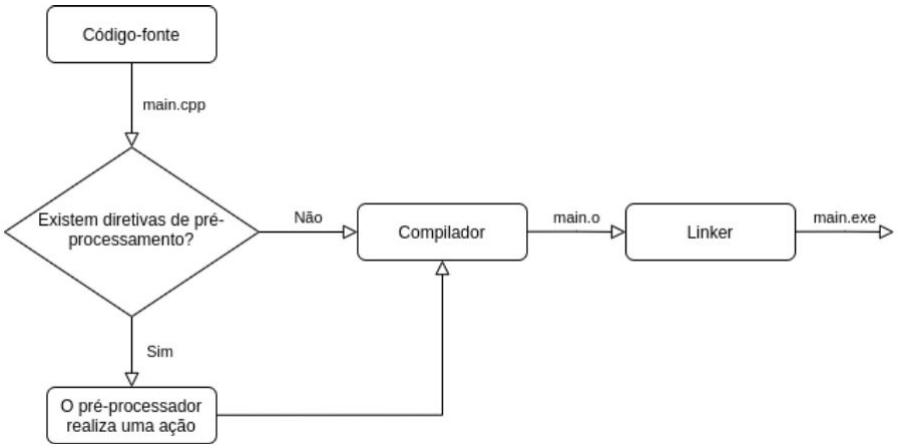

Figura 7.1.1: *Fluxograma*

As instruções do pré-processador são precedidas pelo símbolo ‘#’. Abaixo é mostrada uma tabela com algumas instruções e suas descrições:

Código	Descrição	Utilização
#include	Inclui o código de um arquivo externo no código-fonte.	#include <iostream> #include “cabecalho.h”
#define	Define uma macro (valor constante).	#define HORAS 24
#ifdef // se definido #ifndef // se nao definido #endif	Caso uma macro esteja ou não definida, realiza a compilação do código entre as diretivas #ifdef ou #ifndef e a diretiva #endif.	#ifdef HORAS int minutos = 60*HORAS; #endif #ifndef HORAS #define HORAS 24#endif

Ainda neste capítulo serão discutidas, principalmente, as instruções de include e declarações *using*.

7.2 Bibliotecas e Diretivas Include

As diretivas *include* são utilizadas para incluir **arquivos de cabeçalho** (bibliotecas) que contêm ferramentas úteis para o programador utilizar no código fonte, como a biblioteca **iostream**, que contém ferramentas que dão suporte a operações de entrada e saída de dados.

As bibliotecas utilizadas podem ser padrão da linguagem ou criadas pelo programador. As bibliotecas padrão se localizam no **diretório padrão de inclusão** e devem ser incluídas utilizando os símbolos "<" e ">", como é mostrado a seguir:

```
1 #include <iostream>
```

Algumas bibliotecas padrão são:

Biblioteca	Descrição
iostream	Fornece diversas funções que dão suporte à entrada e saída de dados.
algorithm	Fornece algoritmos genéricos, como os de busca e ordenação de dados.
string	Fornece a classe string, utilizada como um container para armazenar e manipular sequências de caracteres (string).
vector	Fornece a classe template vector, utilizada como um container para armazenar uma cadeia de valores de algum tipo.
cstdlib	Fornece várias funções de propósito genérico da linguagem C. Assim como outras bibliotecas adaptadas da linguagem C, a stdlib.h perde o ".h" ao final e recebe um "c" no início do nome.

Bibliotecas que sejam criadas pelo programador geralmente não são incluídas no diretório padrão e se situam no mesmo diretório do código fonte ou em subdiretórios. Por isso, ao realizar sua inclusão, o **caminho relativo** do arquivo deve ser fornecido e, ao invés dos símbolos maior e menor que, aspas devem ser utilizadas.

Neste exemplo, existe um arquivo de cabeçalho chamado "matematica.h", situado em um subdiretório. Nele estão definidas algumas funções de operações aritméticas. Ele é incluído no código fonte juntamente à biblioteca padrão iostream, para realizar a entrada e saída de dados.
Este é o main.cpp:

```
1
2 #include <iostream>
3 #include "subdiretorio/matematica.h"
4
5 using namespace std;
6
7 int main()
8 {
9   int a = 5;
```

```
10    int b = 3;
11
12    cout << "Sendo a = " << a << " e b = " << b << ",
          então:" << endl;
13
14    cout << "O resultado da soma de a e b é: " << soma(a,b)
          << endl;
15    cout << "O resultado da subtração de a e b é: " <<
          subtrai(a,b) << endl;
16
17    return 0;
18  }
```

Este é o matematica.cpp

```
1  int soma(int x, int y){
2      return (x+y);
3  }
4
5  int subtrai(int x, int y){
6      return (x-y);
7  }
```

A saída é:

```
1  Sendo a = 5 e b = 3, então:
2  O resultado da soma de a e b é: 8
3  O resultado da subtração de a e b é: 2
4
5  Program finished with exit code 0
```

7.3 Namespaces e Declarações Using

Ao inserir muitas bibliotecas em um código, é possível que algumas delas possuam funções com o mesmo nome e argumentos. Caso essas funções estejam em um mesmo **namespace**, isso causaria um conflito por existir mais de uma definição para a mesma função.

Os **namespaces**, então, servem para agrupar nomes (de variáveis, constantes ou funções) em um mesmo **escopo** e impedir que hajam conflitos entre outros nomes.

Para criar um namespace, basta escrever a palavra-chave **namespace** seguida de um bloco delimitado por chaves. Dentro do bloco, pode-se criar quaisquer nomes e, ao utilizá-los em outro momento, o namespace fará parte deles.

A palavra “manga” pode ser utilizada tanto no contexto em que representa uma fruta quanto no contexto que representa a manga de uma camisa. No contexto da programação em C++, essa situação poderia ser representada através de namespaces, como no código a seguir:

```
1
2  #include <iostream>
3  #include <string>
4
5  namespace roupa {
6    std::string manga(){
7      return "pedaço de tecido";
8    }
9  }
10
11 namespace fruta {
12   std::string manga(){
13     return"fruta";
14   }
15 }
16
17 int main(){
18   std::cout << "A palavra manga pode representar tanto um
        " << roupa::manga();
19   std::cout << " quanto uma " << fruta::manga() << "!" <<
        std::endl;
20
21   return 0;
22 }
```

Ao executar o programa, a saída deve ser:

```
1  A palavra manga pode representar tanto um pedaço de tecido
       quanto uma fruta!
2
3    Program   finished with exit code 0
```

No exemplo acima, é possível perceber que foi preciso utilizar o **operador de resolução de escopo** "::" para indicar que a função que desejávamos utilizar fazia parte de um namespace.

Um namespace que foi utilizado com bastante frequência até o momento é chamado **std**, uma abreviação da palavra "standart", que significa "padrão". Ele é, então, o namespace padrão, que armazena diversas ferramentas utilizadas normalmente na linguagem. No exemplo, é possível perceber que o **cout, endl** e a classe **string** pertencem a esse namespace.

Por vezes, utilizar a declaração **using namespace** poupa o trabalho de escrever o nome do namespace e utilizar o operador de resolução de escopo antes de cada nome que faça parte dele. Embora seja uma ferramenta útil, não é uma prática recomendada, pois ao abusar dela, pode ser que surja novamente o problema da redefinição de nomes, que os namespaces tentam resolver.

Ao tentar reescrever o código do exemplo 2 utilizando a declaração **using namespace**, o código ficaria visualmente mais "limpo", porém ao executar o programa, o compilador indicaria um erro.

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

using namespace roupa;
using namespace fruta;

namespace roupa {
  std::string manga(){
    return "pedaço de tecido";
  }
}

namespace fruta {
  std::string manga(){
    return"fruta";
  }
}

int main(){
  std::cout << "A palavra manga representa um(uma)     <<
     manga() << std::endl;

  return 0;
}
```

O erro indicado ao executar o programa deve ser:

```

error: call of overloaded   manga  ()    is ambiguous
```

indicando que o uso da função é ambíguo, retornando então ao problema inicial.

Using namespace: Não utilize a declaração *using namespace* em cabeçalhos, pois isso força a todos que incluam o cabeçalho a também utilizar o namespace em seus códigos. Isso pode resultar em erros de difícil identificação.

8. Nomes, tipos e valores

8.1 Nomes e Variáveis

Escrever programas de computador avançados exige a capacidade de armazenar dados na memória do computador. Para entender como armazenar e manipular dados em C++, é necessário compreender, de forma qualitativa, como os dados são armazenados na memória.

O espaço físico da memória do computador que armazena dados é conhecido como objeto. Um objeto possui três características fundamentais: tipo, nome e valor. O tipo de um objeto determina as características da informação que pode ser armazenada, os nomes são uma forma de identificá-los e o valor é o conteúdo armazenado neles.

O diagrama a seguir mostra uma representação, na memória de computador, de dois objetos de um programa do sistema de gestão estudantil de uma universidade, observe para entender melhor as características de um objeto:

Nome	"Pedro"’
Idade	21

Na tabela acima, cada retângulo representa um objeto. A primeira divisão é o nome, com ele é possível distinguir os objetos, o primeiro armazena o nome de um aluno e o segundo sua idade. A segunda divisão é o valor, o primeiro armazena o inteiro 21 e o segundo armazena o conjunto de caracteres “Pedro”, observe que as características das informações de cada objeto, nesse caso, são de tipos diferentes, pois temos números inteiros e conjunto de caracteres.

Portanto, fazendo uma analogia, um objeto é como uma caixa, ele pode armazenar informação assim como uma caixa armazena itens. O tipo do objeto pode ser comparado com o propósito de armazenamento da caixa, assim como existem caixas destinadas a armazenar livros e produtos eletrônicos, existem objetos com tipos específicos para lidar com números inteiros, números reais, caracteres e etc. O nome de um objeto é um rótulo para localizá-lo, assim como uma etiqueta de uma caixa convencional. O valor de um objeto é o conteúdo da caixa, apesar de existir várias caixas de mesmo propósito, o conteúdo delas pode ser distinto. Por exemplo, uma caixa feita para livros

pode conter apenas livros de romance ou ficção, da mesma forma , um objeto de tipo inteiro pode ter como valor o número 29 ou 31.

Definição Em C + + objetos que possuem um nome são chamados de variáveis, portanto, a partir de agora a apostila adota essa nomenclatura [1].

8.2 Como utilizar variáveis em C++?

A forma mais simples de entender como criar variáveis na linguagem C++ é pelo estudo de caso. Observe o código a seguir:

```
1
2  #include <iostream>
3
4  using namesspace std;
5
6  int main()
7  {
8
9      int idade;
10     String nome = "Pedro";
11     char conceito = "A";
12     double media_semestral = 99.9;
13     bool Aprovado = true;
14
15 }
```

As variáveis do código estão nas linhas 7 a 11. Veja que a estrutura geral para criar uma variável é semelhante nos cinco casos. Por exemplo, ao analisar a linha 7, o tipo da variável é *int* e seu nome "idade", porém, ela não tem um valor. É dito, nesse caso, que tal variável foi declarada mas não foi inicializada com um valor, isso significa que foi criado um objeto na memória mas não foi dado um valor inicial a ele.

Um caso diferente está na linha 10. O tipo da variável é *double*, o nome é "media_semestral" e o valor é 99.9, ou seja, além de declarar uma variável ela foi inicializada com o valor de 99.9. Ao declarar uma variável é preferível que o programador inicialize de imediato com um valor, isso evita erros em um programa, uma vez que variáveis não inicializadas contém valores aleatórios armazenados e podem levar a erros.

Portanto, pode-se resumir o processo de declaração e inicialização de uma variável pelo seguinte esquemático:

Tipo	Nome da variável	=	Valor	;

Como mostra a tabela, primeiro escolhe-se um tipo para a variável, em sequência é dado um nome, e , por fim, é dado um valor utilizando o operador "=".

Cuidado! Na linguagem matemática, o símbolo "=" é utilizado para representar igualdade, porém, em C++, como na maioria das linguagens de programação, tal símbolo é um operador de atribuição que permite ao programador atribuir um valor para uma variável.

8.3 Tipos na linguagem C++

No exemplo do código mostrado anteriormente, existem diversos tipos de dados da linguagem C++. Eles definem que tipo de dados podem ser armazenados em uma variável e também que operações podemos fazer com esses dados. Nesse tópico será abordado as características fundamentais dos principais tipos de dados nativos da linguagem como "int", "double", "string" , "boolean" e "char".

8.3.1 Tipo inteiro - *Int*

O tipo inteiro, como o próprio nome diz, tem a capacidade de representar o conjunto matemático dos números inteiros na linguagem C++. Ele pode armazenar números que vão desde -2.147.483.648 até 2.147.483.647, para trabalhar com tal tipo, deve-se utilizar a palavra reservada *int* quando for declarar uma variável,veja o exemplo a seguir:

```
1 int contador=0;
```

8.3.2 Tipo *Double*

O tipo *double* permite utilizar representações de números reais com maior precisão incluindo os intervalos 2,2*10^{-308} até 1,8*10^{308}. Para utilizar esse tipo deve-se acrescentar a palavra reservada *double* conforme o exemplo abaixo:

```
1 double pi = 3.1415926
```

Curiosidade: Existe um tipo semelhante ao *double* na linguagem C++ chamado de *float*, a diferença entre eles é que o *float* possui uma precisão menor que o *double*.

8.3.3 Tipo *Char*

O tipo *char*permite ao programador armazenar um único caractere (letras, números ou caracteres especiais) em uma variável desse tipo. A palavra reservada a esse tipo é *char* e é possível atribuir valores desse tipo de duas maneiras: '' com aspas simples ou usando o número da tabela ASCII que contém o caractere desejado. O exemplo a seguir demonstra diferentes formas de atribuir o valor "b" a uma variável.

```
1 char letra_b = "b" ;
2 char letra_b = 98;
```

8.3.4 Tipo *Bool*

O tipo booleano é um tipo com palavra reservada chamada *bool*. Pode receber apenas os valores 0 ou 1, também podemos usar as palavras reservadas *"true"* em equivalência a 1 e *"false"* para 0. Ele é muito utilizado para armazenar resultados de operações lógicas feitas nos programas. Exemplos:

```
1 bool greaterThan= 1;
2 bool greaterThan=true;
3 bool lowerThan=0;
4 bool lowerThan=false;
```

Curiosidade: Se o programador atribuir um valor diferente de zero para uma variável de tipo *bool*, tal valor é interpretado como 1.

8.3.5 Tipo *String*

O tipo string permite ao programador armazenar uma cadeia de caracteres em uma única variável, dessa forma é possível armazenar frases e textos em strings. Utilizar strings é um pouco mais complexo, a forma mais simples é colocar no arquivo de seu programa a frase *"using namespace std;"* e então use a palavra reservada *string* antes do nome da variável. Quanto ao valor que uma string recebe, ele sempre deve estar no interior de aspas duplas.

```
1 string nome_aluno= "Tobias da Silva";
2 string universidade= "Universidade Federal de Minas
     Gerais";
```

8.4 Operações suportadas pelos tipos

Nos programas de computador, constantemente é feita operações lógicas e aritméticas com os tipos de dados mostrados, porém, é restrito às operações suportadas por cada tipo. A tabela a seguir ilustra as operações suportadas de cada tipo.

Operação	int	double	char	bool	string
atribuição	=	=	=	=	=
soma	+	+	não há	não há	+
concatenação	não há	não há	não há	não há	+
subtração	-	-	não há	não há	não há
multiplicação	*	*	não há	não há	não há
divisão	/	/	não há	não há	não há
resto	%	não há	não há	não há	não há
incrementa 1	++	++	não há	não há	não há
decrementa 1	–	- -	não há	não há	não há
incrementa n	+=n	+=n	não há	não há	não há
acrescenta ao final	não há	não há	não há	não há	+=
decrementa de n	-=n	-=n	não há	não há	não há
multiplica e atribui	*=	*=	não há	não há	não há
divide e atribui	/=	/=	não há	não há	não há
resto e atribui	%=	não há	não há	não há	não há
igual	==	==	==	==	==
não igual	!=	!=	!=	!=	!=
maior do que	>	>	>	>	>
maior ou igual a	>=	>=	>=	>=	>=
menor do que	<	<	<	<	<
menor ou igual a	<=	<=	<=	<=	<=

8.5 Modificadores de tipo

Existem alguns recursos que permite alterar algumas características tipo de dado, são os chamados modificadores de tipo. Na linguagem C ++ estão na forma das palavras reservadas *short, long, unsigned , signed* e const. Veja a seguir como utilizá-los:

8.5.1 Short

O modificador *"short"* altera a faixa de valores para um tipo. Por exemplo, ao utilizar um *short int* o intervalo de trabalho vai para -32.768 até 32.767, ou seja, tal modificador reduz a quantidade de bits para representar um inteiro.

```
1 short int num = 140;
```

8.5.2 Long

Esse modificador de tipo é usado por padrão sempre que são declaradas variáveis, ele determina que vai ser usado a máxima quantidade de bits disponíveis para representar aquele tipo. Ao explicitar ele na frente do nome do tipo asseguramos que o tipo vai ter o máximo de bits na representação;

```
1 long int num = 2.147.483.647;
```

8.5.3 Signed

O modificador *"signed"* específica que a representação em binário do valor da variável cobre números com sinal. Dessa forma, ao utilizá-lo é possível trabalhar com números negativos;

```
1 signed int um= -4000;
```

8.5.4 Unsigned

O modificador *"unsigned"* permite apenas o uso de números sem sinal e, no caso de inteiros, permite trabalhar com números na faixa de 0 a 4.294.967.295, ou seja,obtém uma capacidade maior para trabalhar com números inteiros positivos.

```
1 unsigned int num =  4.294.967.290;
```

8.5.5 Const

O modificador *"const"* permite que o valor de uma variável seja sempre uma constante, não sendo possível alterar seus valores. Isso é um recurso muito útil em programas que usam um mesmo valor com muita frequência e que esse mesmo valor pode gerar problemas na execução caso seja alterado.

```
1 const double pi = 3.1415;
```

9. Tipos de erros

9.1 Erros em tempo de compilação

Durante o processo de compilação, o compilador analisa o código escrito pelo programador e verifica se existem erros de sintaxe, palavras ou funções que não foram definidas, entre outros. Caso o compilador encontre algo que não pareça correto, o processo de compilação será abortado e uma mensagem de erro será enviada ao usuário, avisando-o sobre a possível causa desse erro.

Portanto, os erros em **tempo de compilação** podem ser divididos entre **erros de sintaxe** e **erros de digitação**. A primeira subdivisão está relacionada a escrita incorreta de código, ou seja, alguma parte do código implementado não respeita as regras gramaticais da linguagem. Quanto aos erros de digitação, muitas vezes eles levam a variáveis ou funções que não foram definidas no código, argumentos com tipo incorreto, entre outros.

Por exemplo, define-se a função **soma(int a, int b)** que retorna um inteiro igual a soma de "a" e "b". Ao realizar a chamada de função, alguns erros comuns são listados a seguir, assim como a sua causa.

```
1 int c = soma(10, 5)       // Erro: Faltou um ";" ao
     terminar a instrução
2 int d == soma(10, 5);  // Erro: Atribuição de valores é
     realizada com apenas um "="
3 int e = soma(10 5);     // Erro: Faltou uma "," entre o 10
     e o 5
```

Os exemplos a seguir ilustram erros de digitação comuns:

```
1 int f = oma(10,5);         // Erro: "oma" não está definido
```

```
2 int g = soma(5);          // Erro: A função soma exige 2
    argumentos
```

É importante ressaltar que o compilador não infere nada sobre o que o programador deseja fazer, ele apenas analisa o código e julga se está razoável, de acordo com o esperado para a linguagem de programação utilizada, ou se existe algo que está incorreto. Portanto, é comum que a mensagem de erro não descreva o erro de forma precisa ou não acerte a linha em que ele está (acontece de mostrar a linha anterior ou a posterior). Por exemplo, para a chamada da função "soma" em **"int e = soma(10 5);**", alguns compiladores apontam que faltou um ")" após o número 10, mas note que fechar o parêntesis após o 10 resultaria em um outro erro de compilação.

9.2 Erros em tempo de Linkagem

Como discutido na seção *Processo de compilação, linkagem*3, um programa pode ser composto de diversos arquivos que comunicam entre si. Dessa forma, caso o linker não seja capaz de realizar a conexão entre esses arquivos, uma mensagem de erro de linkagem, será mostrada. Os erros em tempo de Linkagem possuem apenas duas causas:

- Declarações de tipos distintos para uma mesma função em arquivos diferentes;
- Funções não definidas ou com múltiplas definições no programa.

Por exemplo, ao compilar e executar o programa a seguir:

```
1 int soma(int a, int b); // Retorna a soma entre A e B
2
3 int main(){
4     int x = soma(10,5);
5     return 0;
6 }
```

Neste caso, note que a função soma foi apenas declarada, portanto a menos que tenhamos definido a função "soma" em algum outro arquivo do programa e linkado ele ao que faz a chamada da função, esse programa possuirá um erro de linkagem.

9.3 Erros em tempo de execução

Após o processo de compilação e o processo de linkagem ter sido finalizado com sucesso, o programa será executado. Porém, o fato do programa executar não quer dizer que ele está livre de erros, isso apenas indica que o programador seguiu as regras estabelecidas para códigos em C++. Na verdade, é comum que existam os chamados erros em **tempo de execução**, que estão relacionados a **erros de lógica** de programação e são mais difíceis de serem identificados, já que não são percebidos pelo compilador ou pelo linker.

Para os casos em que o erro em tempo de execução resulta na execução de alguma instrução ilegal, o programa será abortado e uma mensagem de erro será enviada para o usuário. Porém, há casos em que a execução do programa irá continuar e resultados inesperados vão aparecer.

Por exemplo, o programa a seguir será abortado ao executar instrução na linha 3, isso porque o valor contido em "n" é maior que o tamanho máximo para arrays em C++.

```
1 int main ()
```

```
2 {
3   int n = 100000000000;
4   int a[n];
5   std::cout << a[1];
6 }
```

Por outro lado, o programa contido no exemplo seguinte executará normalmente, sem que nenhuma mensagem de erro seja enviada ao usuário, apesar de acessar um endereço de memória inválido.

```
1 int main ()
2 {
3   int n = -15;
4   int a[10];
5   std::cout << a[n];
6 }
```

9.4 Erros de Lógica

Os erros de lógica de programação também são considerados erros em tempo de execução, apesar de estarem relacionados a resultados e/ou comportamentos inesperados para a funcionalidade implementada no programa.

Por exemplo, o programa a seguir implementa a função area(int a, int b), que retorna a área de um retângulo qualquer. Em seguida, é feita a chamada da função “area” e o resultado é armazenado no inteiro “c”.

```
1  int area(int a, int b)
2  {
3    return a * b;
4  }
5
6  int main ()
7  {
8    int x,y;
9    std::cin >> x;
10   std::cin >> y;
11   int c = area(x,y);
12   return 0;
13 }
```

De fato não existe erro algum no programa implementado acima, na verdade ele retorna exatamente o resultado esperado. Porém, pode ser que o usuário atribua o valor -5 ao x e o valor 6 ao y, por engano. Note que, neste caso, o programa irá compilar e executar normalmente, retornando o resultado -30, que está correto numericamente mas não possui sentido físico. Dessa forma, é função do programador prever estes tipos de erros e tratá-los, o que pode ser feito na chamada da função, ou no interior da função chamada.

Para o exemplo acima, podemos contornar o problema apenas inserindo uma expressão condicional na função “area” e, em seguida, retornar um valor que represente o que causou o erro, neste caso o valor -1 foi escolhido para argumentos inválidos.

```
1 int area(int a, int b)
2 {
3     if ( a  < 0 || b < 0 )
4     {
5         std::cerr(    Argumento inválido    ) << std::endl;
6         return -1;
7     }
8   return a*b;
9 }
```

Por questão de boas práticas de programação, é interessante que o programador, sempre que possível, opte por tratar os erros dentro da função implementada, pois isso torna o código mais legível e mais fácil de ser modificado posteriormente. Por outro lado, há casos em que não é possível fazê-lo, por exemplo quando o programador utiliza uma função implementada em uma biblioteca , ou quando o programador não tem acesso à documentação da função. Nesses casos, é necessário que os erros sejam tratados na chamada de função, como no exemplo a seguir:

```
1  int main ()
2  {
3    int x,y;
4    std::cin >> x;
5    std::cin >> y;
6    if (x < 0 || y < 0){
7          std::cout <<     Argumento inválido ;
8          return 0;
9      }
10   int c = area(x,y);
11     return 0;
12 }
```

9.5 Depuração ou “Debugar”

O processo de “debugar” o código tem o objetivo de identificar e tratar as fontes de erros no programa, de modo que ele execute as tarefas para o qual foi criado com assertividade. O procedimento para debugar o código é, muitas vezes, trabalhoso e tedioso, levando muito tempo para ser realizado, em especial para programas longos. Dito isso, é importante que o desenvolvedor tome alguns cuidados ao iniciar a programação, de modo a facilitar a posterior busca por **bugs**.

Curiosidade: A palavra bug, que significa "inseto", é um jargão muito utilizado nas áreas relacionadas à computação para se referir a erros. Na verdade, o primeiro bug foi, de fato, um inseto que ficou preso no relé do computador Mark 2, desenvolvido em Harvard, e impedia o funcionamento da máquina. O inseto, que era uma mariposa, foi encontrado pela analista de sistemas Grace Hopper. Ela o colocou em seu caderno de anotações com uma fita adesiva, escrevendo "Primeiro caso de bug encontrado", em tradução livre.

O primeiro deles é manter o código limpo e organizado, utilizando um padrão na diagramação, inserir comentários curtos e explicativos sobre determinada funcionalidade implementada, evitar números mágicos[1], nomear variáveis com nomes curtos e explicativos, entre outros.

Em segundo lugar, é interessante que o desenvolvedor utilize pequenas funções que realizam o mínimo de tarefas possíveis, ao invés de utilizar funções muito longas. Quebrar o código em pequenos blocos funcionais é uma prática que simplifica muito o processo de "debugar" o programa, tendo em mente que permite testar cada funcionalidade separadamente, o que leva à identificação mais rápida de erros.

Por fim, outra dica importante é: sempre que possível, utilizar funções pré-definidas em bibliotecas. Isso porque elas poupam bastante o tempo do programador, já que não será necessário implementar manualmente a mesma funcionalidade, e também porque elas são muito bem testadas e documentadas, de modo que dificilmente produzirão um resultado incorreto.

Após explicar o que é "debugar" o código e um pouco das boas práticas de programação[2] , que facilitam o longo processo de busca às causas de erro no programa, agora será mostrado como realizar a **depuração** do código.

Uma maneira simples e razoavelmente eficaz, para programas pequenos, é utilizar o próprio comando de saída **(std::cout)** para imprimir variáveis importantes ao longo do programa, para que o desenvolvedor possa comparar as saídas com o esperado para cada variável. No entanto, este método exige execuções repetidas do programa para testar se entradas específicas podem levar a algum erro, de modo que não é eficiente para programas maiores.

Outro método simples e um pouco mais interessante é criar programas compactos para testar funções específicas. Essa forma de teste é muito útil, principalmente quando o programador insere uma nova funcionalidade no programa, de modo que não seja necessário depurar tudo novamente, e sim apenas a nova parte inserida. Além disso, não será necessário compilar, linkar e executar o código diversas vezes, tornando o processo mais eficiente que o anterior. O programa a seguir exemplifica esse método, que testa a função "area".

```
1 int area (int a, int b)
2 {
3   if (a < 0 || b < 0) {
4     std::cout << "Argumento invalido";
```

[1] Números mágicos são valores que deveriam possuir um nome (declarados como constantes), mas foram inseridos diretamente no código, dificultando o entendimento sobre o motivo pela sua escolha, ou do funcionamento do programa.

[2] Boas práticas de programação são um conjunto de convenções, definidas pela comunidade de programadores, que busca uma padronização na escrita de programas, tornando-os mais fáceis de ler e compreender. Essa padronização permite a otimização da produtividade dos desenvolvedores, tendo em mente que, em geral, eles trabalham em equipe para a construção de um programa.

```
5       return 0;
6     }
7     return a * b;
8   }
9   int main(){
10    int x, y;
11    char resp;
12    do{
13          std::cout  << "Insira 2 inteiros\n";
14          std::cin >> x >> y;
15          std::cout << " A area eh:  "<< area(x,y) <<
              std::endl;
16          std::cout << "Testar Novamente? (s/n)";
17          std::cin >> resp;
18          std::cout << std::endl;
19          } while (resp == 'y' || resp== 'Y');
20    return 0;
21  }
```

Por fim, caso o programa apresente algum erro em tempo de execução que cause a interrupção do mesmo, ou o desenvolvedor precise debugar códigos grandes, é interessante utilizar uma ferramenta de depuração, que também está presente em muitas IDE’s. Essa ferramenta permite inserir pontos de parada ao longo do código, de modo que ele será executado até o primeiro ponto e a execução será pausada, permitindo o programador continuar linha a linha, ou até o próximo ponto de parada. Há também a possibilidade de acompanhar o valor, em tempo real, contido em cada variável durante a depuração, o que permite ao desenvolvedor analisar se os resultados obtidos até o momento estão de acordo com o esperado.

A seguir temos um passo-a-passo de como debugar utilizando o Code:Blocks.

1. Inserir pontos de parada no código, o que pode ser feito clicando no espaço ao lado do número da linha em que desejamos inseri-lo. Uma bolinha vermelha aparecerá, indicando que o ponto de parada foi definido. A imagem 1 ilustra dois pontos de parada inseridos, na linha 16 e 17 do programa.
2. Iniciar a depuração, o que pode ser realizado pela tecla de atalho F8. Feito isso, caso não exista nenhum erro em tempo de compilação ou de linkagem o programa será executado até o primeiro ponto de parada. Uma seta amarela aparecerá ao lado do número das linhas, indicando até qual parte do código ele já foi executado.
3. Em seguida, o desenvolvedor poderá escolher entre ir para a próxima linha, ir para o próximo ponto de parada, entre outros. Para poder ver as variáveis e os valores contidos em cada uma delas é necessário realizar o seguinte procedimento: Clicar em **“Debug”**, na barra de tarefas, em seguida clicar em **“Debug Windows”** e, por fim, em **“Watches”**.

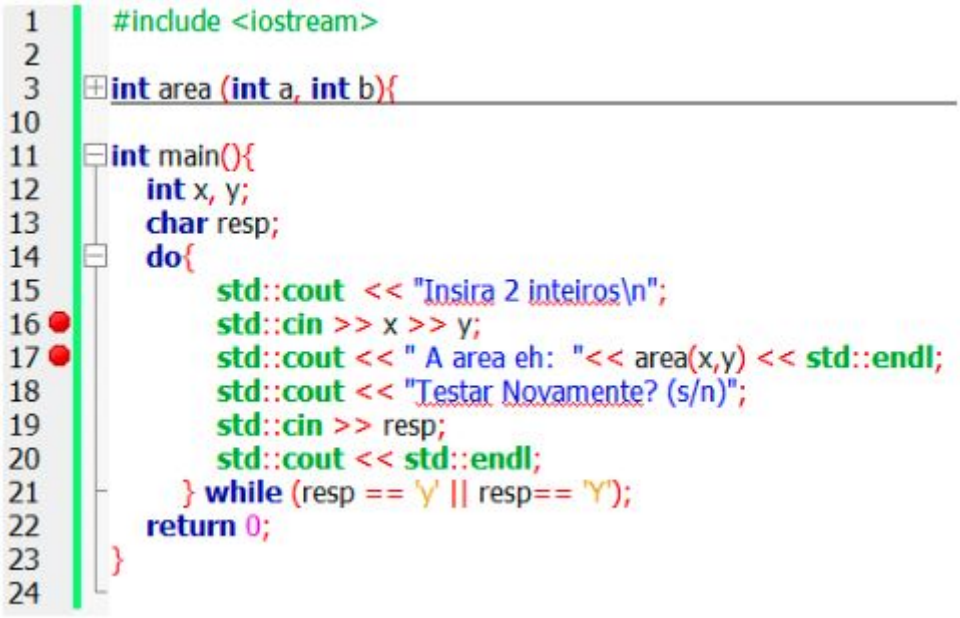

Figura 9.5.1: *Pontos de parada indicados pelo círculo em vermelho.*

Índice

Bibliografia

- W. Savitch. "Absolute C++", 3ª edição, 2008, Editora Addison Wesley
- B. Stroustrup. "Programming - Principles and Practice using C++", 1ª edição, Editora Addison Wesley

C++

PETEE UFMG

UFMG