

Lucas S. Vieira

Introdução à Programação em C++
Versão 0.7 – Compilação 20191206

Universidade Federal dos Vales do Jequitinhonha e do Mucuri
Diamantina, MG
6 de dezembro de 2019

Introdução à Programação em C++
Versão 0.7 – Compilação 20191206

Copyright (c) Lucas S. Vieira

Esta obra é licenciada sob a licença Creative Commons
Atribuição-NãoComercial-SemDerivações 4.0 Internacional.

O código contido neste livro-texto pode ser reproduzido sob a licença
BSD de duas cláusulas ("Simplificada"). O texto desta licença pode ser
encontrado nos apêndices.



Capa: Arte do autor. Simulação de um monitor de fósforo verde, feita com uma cor gradiente, e mostrando o código de um protótipo de um interpretador de uma linguagem de programação chamada Bel, escrito em C. Agradecimentos à Sarah Orlando pela ideia inicial da arte de capa.

SUMÁRIO

AGRADECIMENTOS	7
PREFÁCIO	9
1 Introdução a C++	11
1.1 Usando a IDE Code::Blocks	11
1.1.1 Instalação	12
1.1.2 Sobre Compiladores	16
1.1.3 Interface	17
1.1.4 Criando um projeto	23
1.2 Usando o console do Linux	29
1.2.1 Instalando o compilador	29
1.2.2 Sobre editores de texto	30
1.2.3 Criando e navegando por pastas	35
1.2.4 Escrevendo arquivos de código	37
1.2.5 Usando o compilador através do console	37
1.3 Programando em C++	42
1.3.1 O programa 'Hello World'	42
1.4 Variáveis	45
1.4.1 Tipos primitivos	45
1.4.2 Declaração de variáveis	47
1.4.3 Casos especiais de <code>int</code>	48
1.5 Operadores e aritmética	49
1.5.1 Atribuição (<code>=</code>)	50
1.5.2 Aritmética básica (<code>+</code> , <code>-</code> , <code>*</code> , <code>/</code> , <code>%</code>)	50
1.5.3 Parênteses (<code>(e)</code>)	50
1.5.4 Aritmética atributiva (<code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code>)	51
1.5.5 Incremento e decremento (<code>++</code> , <code>--</code>)	51
1.5.6 Comparação (<code>==</code> , <code>!=</code> , <code><</code> , <code>></code> , <code><=</code> , <code>>=</code>)	53
1.5.7 Lógicos (<code>&&</code> , <code> </code> , <code>!</code>)	54

1.5.8	Ternário (<code>?:</code>)	55
1.5.9	<i>Bitwise</i> (<code>&</code> , <code> </code> , <code>~</code> , <code>^</code> , <code>>></code> , <code><<</code>)	56
1.5.10	Comentários (<code>//</code> , <code>/*</code> e <code>*/</code>)	58
1.6	Precedência de operadores	59
1.6.1	Sobre agrupamento de expressões	60
1.7	Entrada e Saída (I/O)	61
1.8	Exercícios de Fixação I	63
1.9	Controle de fluxo	64
1.9.1	Estrutura condicional (<code>if</code>)	65
1.9.2	Estrutura de seleção (<code>switch</code>)	67
1.10	Exercícios de Fixação II	69
2	Bibliotecas	73
2.1	Diferenças entre C e C++	74
2.2	Bibliotecas da linguagem C	74
2.2.1	<code>cstdio</code>	75
2.2.2	<code>cmath</code>	79
2.2.3	<code>ctime</code>	88
2.2.4	<code>cstdlib</code>	92
2.2.5	<code>cassert</code>	97
2.2.6	<code>cstring</code>	97
2.2.7	<code>cerrno</code>	101
2.2.8	<code>climits</code>	102
2.2.9	<code>cctype</code>	102
2.3	Bibliotecas da linguagem C++	104
2.3.1	Sobre <i>namespaces</i>	105
2.3.2	<code>iostream</code>	106
2.3.3	<code>iomanip</code>	108
2.3.4	<code>fstream</code>	110
2.3.5	Standard Template Library (STL)	112
2.4	Exercícios de Fixação	116
3	Vetores	119
3.1	Primitivas, combinação e abstração	120

3.2	O que são vetores?	120
3.3	Usando vetores	121
3.3.1	Acessando elementos no vetor	123
3.3.2	Atribuindo valores ao vetor	124
3.3.3	Inicializando o vetor na declaração	125
3.4	Strings	128
3.4.1	Lendo <i>strings</i> informadas pelo usuário	130
3.4.2	Sobre leitura correta de <i>strings</i> em <i>streams</i>	131
3.5	Exercícios de Fixação I	133
3.6	Laços de repetição	134
3.6.1	Laços <code>while</code>	135
3.6.2	Laços <code>do...while</code>	137
3.6.3	Laços <code>for</code>	138
3.7	Exercícios de Fixação II	141
3.8	Autômatos celulares unidimensionais	144
3.8.1	Notação de Autômatos de Wolfram	145
3.8.2	Modelando AC's unidimensionais	147
3.9	Programando um AC unidimensional	148
3.9.1	Gerando a primeira geração do AC	148
3.9.2	Criando próximas gerações do AC	151
3.9.3	Embelezando a saída	152
3.9.4	Código completo	153
3.10	Projeto de Programação I	157
4	Matrizes	159
4.1	Declarando matrizes bidimensionais	161
4.1.1	Acessando e atribuindo elementos à matriz	162
4.1.2	Utilizando vetores individuais da matriz	162
4.1.3	Inicializando a matriz na declaração	163
4.1.4	Matrizes de mais dimensões	164
4.2	Iterando sobre matrizes	165
4.3	Exercícios de Fixação	166
4.4	Projeto de Programação II	168

5	Tipos Abstratos de Dados	171
5.1	Estrutura (struct)	172
5.1.1	Criando uma variável a partir do <i>tipo abstrato</i>	174
5.1.2	Acessando elementos de um <i>tipo abstrato</i>	175
5.1.3	Inicializando um struct	176
5.2	Vetores de tipos abstratos	177
5.3	Exercícios de Fixação I	179
5.4	Enumeração (enum)	180
5.4.1	Declarando enumerações	181
5.4.2	Usando enumerações	183
5.5	União (union)	184
5.5.1	Exemplo de uso	185
5.5.2	Código completo do exemplo	188
5.6	Exercícios de Fixação II	190
6	Considerações Finais	193
7	Soluções	195
7.1	Capítulo 1	195
7.2		205
	APÊNDICE A Licenciamento de Código	207
	APÊNDICE B Usando o GNU Debugger	209
B.1	Diagnóstico	210
B.1.1	Backtrace	211
B.1.2	Variáveis locais	211
B.2	Saindo do GNU Debugger	211
	APÊNDICE C Geração de Autômatos Celulares	213
	Referências Bibliográficas	215

LISTA DE FIGURAS

Figura 1	–	Página inicial do website do Code::Blocks	14
Figura 2	–	Página de downloads	15
Figura 3	–	Página de lançamentos binários	16
Figura 4	–	<i>Splash Screen</i> do Code::Blocks	18
Figura 5	–	Tela inicial Code::Blocks	18
Figura 6	–	Barra principal do Code::Blocks	19
Figura 7	–	Barra do compilador do Code::Blocks	20
Figura 8	–	Barra do debugger do Code::Blocks	20
Figura 9	–	Barra de completamento de código	21
Figura 10	–	Barra de gerenciamento de projetos	22
Figura 11	–	Barra de logs & outros	23
Figura 12	–	Menu para criação de projetos	23
Figura 13	–	Janela de seleção de projetos	24
Figura 14	–	Tela inicial do assistente de criação de projeto	25
Figura 15	–	Seleção de linguagem do assistente	26
Figura 16	–	Tela de dados do projeto	27
Figura 17	–	Tela de seleção de compilador	28
Figura 18	–	Árvore de projetos após a criação do projeto	29
Figura 19	–	Gedit editando código C	31
Figura 20	–	Nano editando código C	32
Figura 21	–	Vim editando código C	33
Figura 22	–	Emacs editando código C	34
Figura 23	–	Compilação e execução do programa anterior em um console do Linux	62
Figura 24	–	Ilustração de um vetor A , de seis elementos	121
Figura 25	–	Evolução da Regra 90 por 31 gerações.	144
Figura 26	–	Regras de vizinhança para a Regra 90	146
Figura 27	–	Evolução e definição da Regra 110	157

Figura 28 – Evolução e definição da Regra 150 169

RASCUNHO

AGRADECIMENTOS

Gostaria de agradecer a todos os que me apoiaram durante o processo de realização do curso de C++ que deu origem a este material. Primeiramente, quero agradecer à minha família, que me apoiou moralmente. Depois, gostaria de agradecer a Rafael Mainarti, que idealizou e divulgou o curso em primeiro lugar, com apoio da NextStep, a Empresa Júnior do curso de Sistemas de Informação da UFVJM.

Quero também agradecer ao professor Leonardo Lana de Carvalho, por fornecer sugestões valiosas com relação ao que este material poderia abordar, e também por apoiar a iniciativa; e por fim, ao grupo de pesquisas de *Linguagem, Cognição e Computação (LC2)*, companheiros de jornada com quem muito discuti a respeito da didática do ensino da programação, e que influenciaram diretamente este material.

PREFÁCIO

O presente trabalho é uma coletânea de materiais para o aprendizado de programação básica na linguagem C++. Este material é a descrição das notas de um curso inicialmente ministrado entre novembro e dezembro de 2019 por mim, de forma a complementar o aprendizado de programação básica no Ensino Superior, mais especificamente para o curso de Sistemas de Informação da Universidade Federal dos Vales do Jequitinhonha e Mucuri, localizada em Diamantina - MG, Brasil.

Este material também visa ser uma aproximação prática do que normalmente seria visto na disciplina de Algoritmos e Estruturas de Dados I, com um enfoque em alguns experimentos interessantes relacionados à área da inteligência computacional. Assim, o material possui um objetivo de fomento ao caráter exploratório na disciplina.

Este material possui um caráter primário de *notas de aula*, sendo complementado a rigor, à medida que aulas sejam ministradas. Assim, detalhes serão adicionados pouco a pouco.

1. INTRODUÇÃO A C++

A linguagem C++ é caracterizada como sendo uma linguagem de sistemas. Seu uso é comumente associado a aplicações que necessitam de gerência manual do uso de memória; mas, na realidade, C++ é uma linguagem extremamente flexível, podendo ser utilizada para praticamente qualquer propósito moderno, e até mesmo com ferramentas de gerenciamento automático de memória, em certas ocasiões.

Aplicações em C++ contam com o respaldo de uma biblioteca padrão poderosíssima, e a especificação da linguagem continua a receber novidades frequentemente. Neste capítulo, veremos o uso básico da IDE Code::Blocks, dos compiladores GCC e/ou Clang, e como compilar programas em C++ a partir do console do sistema operacional Linux.

1.1. USANDO A IDE CODE::BLOCKS

Code::Blocks é um *ambiente integrado de desenvolvimento* (IDE), idealizado para o desenvolvimento nas linguagens C, C++ e Fortran¹, podendo também ser configurado para uso com as linguagens

1. Fortran é uma linguagem criada por John Backus em 1957, sendo caracterizada por ser imperativa e de propósito geral, com aplicação principal em computação numérica e científica.

Java e D².

A IDE Code::Blocks é também *cross-platform*, o que significa que ela funciona em vários sistemas operacionais. Ela também é software livre³ e de código-aberto, sendo escrita em C++.

1.1.1. Instalação

Para instalar o Code::Blocks no seu sistema, devemos seguir um certo número de passos, que dependerá do sistema operacional utilizado pelo usuário.

No momento de escrita deste texto, a IDE Code::Blocks está em sua versão 17.12, podendo possuir uma numeração diferente no momento atual. O leitor está convidado a proceder de acordo com tal possibilidade.

1. Instalação no Linux

A instalação do Code::Blocks em sistemas Linux é simples e direta, podendo ser realizada através do terminal. Para tanto, precisaremos assegurarmo-nos da instalação dos seguinte pacotes:

- O pacote do Code::Blocks (normalmente `codeblocks`);

2. D é uma linguagem criada por Walter Bright em 2001, caracterizada como multiparadigma e sendo inicialmente pensada como um C++ reimaginado. Eventualmente a linguagem readaptou ao invés de reimplementar alguns conceitos de C++, tornando-se uma linguagem distinta desde então.

3. As denominações "*software livre*" e "*código-aberto*" dizem respeito ao método de licenciamento e de comercialização de um dado *software*. Os aspectos de maior relevância desta denominação estão no fato de que o código da IDE está disponível para que qualquer usuário possa consultá-lo, e que estes preceitos garantem certas liberdades individuais para o usuário final. Licenciamento de *software* que você cria é um aspecto muito importante no desenvolvimento, que merece uma discussão própria. Para mais informações, veja o website <<http://escolhaumalicenca.com.br>>.

- O pacote de compiladores para C/C++ (normalmente `gcc` e/ou `g++`, podendo ser substituído por `clang` e `clang++`, respectivamente).

A instalação destes pacotes varia de distribuição para distribuição. Caso você esteja utilizando uma distribuição baseada em Debian, como Ubuntu, Deepin e Mint, poderá utilizar o gerenciador de pacotes `apt` para baixar os programas necessários com os seguintes comandos:

```
1 sudo apt install build-essential
2 sudo apt install codeblocks
```

Caso você esteja utilizando uma distribuição da Red Hat, como Fedora, RHEL ou CentOS, utilize os seguintes comandos:

```
1 sudo dnf install gcc-c++
2 sudo dnf install codeblocks
```

Se você estiver utilizando uma distribuição baseada em Arch Linux, como Manjaro ou Antergos, utilize estes comandos:

```
1 sudo pacman -S gcc
2 sudo pacman -S codeblocks
```

Outras distribuições podem possuir formas diferentes de instalar pacotes. Neste caso, consulte a documentação da sua distribuição Linux para mais informações.

Uma vez que GCC esteja instalado, podemos testá-lo através do terminal com o comando a seguir.

```
1 g++ --version
```

Após realizar este teste, o Code::Blocks poderá ser aberto através de um clique no ícone do Code::Blocks na Área de Trabalho ou no Menu de Aplicativos (caso seu gerenciador de janelas torne-os visíveis), ou através da digitação do comando `codeblocks` no terminal.

Se você achar mais interessante usar apenas o console ao invés de uma IDE, veja a seção 1.2 para maiores detalhes.

2. Instalação no Windows

Para realizar a instalação do Code::Blocks no Windows, vá ao website oficial do Code::Blocks em <https://codeblocks.org>. Você deverá ver uma tela similar à Figura 1.



Figura 1 – Página inicial do website do Code::Blocks

Nesta página, clique na aba *Downloads*. Você será redirecionado para a página mostrada na Figura 2. Nesta página, clique em *Download the binary release*, uma vez que estamos apenas interessados em baixar os executáveis para a IDE.



Figura 2 – Página de downloads

Uma vez na nova página, role-a para baixo até chegar à seção representada na Figura 3. Escolha um dos links de download à direita, **na mesma linha do seguinte nome de arquivo:**

- `codeblocks-17.12mingw-setup.exe`

Note que é importante prestar bastante atenção em qual das opções está sendo baixada! A versão terminada com `mingw-setup` é a mais indicada, uma vez que ela virá não apenas com a IDE Code::Blocks, mas também com os compiladores da suite MinGW incluídos. Esta suite de aplicativos é essencial para a

compilação de aplicações, e será melhor explicada na seção a seguir.

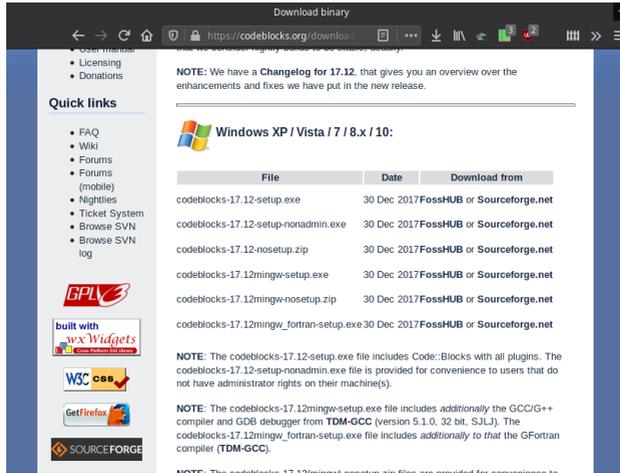


Figura 3 – Página de lançamentos binários

Após baixar o executável em questão, execute-o. Você será agraciado com um instalador para o Code::Blocks. Basta seguir os passos, e a IDE será instalada no seu sistema.

Após a instalação do Code::Blocks, este poderá ser aberto com um clique duplo em seu ícone na Área de Trabalho, ou sendo selecionado através do Menu Iniciar.

1.1.2. Sobre Compiladores

É muito comum ouvir falar sobre o Code::Blocks como sendo um *compilador* de C/C++. Esta denominação, todavia, é *incorreta*.

Como já apresentado, Code::Blocks é um *ambiente integrado de desenvolvimento*. Os *compiladores* são *software* especial, capazes de transformar código de uma linguagem em *código de máquina*, que será então executado pelo computador. Em outras palavras, **não é necessária uma IDE para compilarmos nossos códigos**; as IDEs simplesmente fazem uso de compiladores para criarem executáveis.

Caso você já esteja confortável com a utilização de algum editor de texto que possua suporte a C/C++ (como [Visual Studio Code](#), [Atom](#), [Sublime Text](#), [Emacs](#) ou [Vim](#)), fique à vontade para utilizá-los; neste caso, verifique se o editor de texto em questão possui plugins que facilitem a interação com a linguagem. A compilação poderá ser feita até mesmo através do próprio console; neste caso, leia mais a respeito na seção 1.2.

A instalação utilizada para o Code::Blocks anteriormente já possui a suite de compiladores GCC, sob a alcunha de MinGW. Estes compiladores são uma versão minimalista do compilador GCC, especificamente criada para uso em sistemas Windows.

1.1.3. Interface

Ao abrirmos a IDE Code::Blocks, o primeiro elemento a ser mostrado é sua tela de boas vindas da Figura 4. Logo em seguida, veremos a tela inicial mostrada na Figura 5.

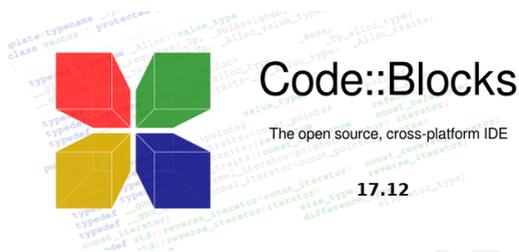
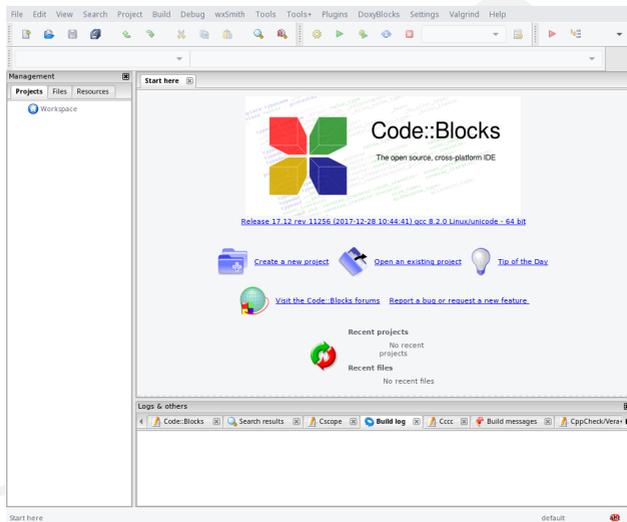
Figura 4 – *Splash Screen* do Code::Blocks

Figura 5 – Tela inicial Code::Blocks

Alguns dos recursos já presentes na interface merecem uma dose de atenção. Eles são:

1. Barra principal

Esta barra enumera funções triviais para qualquer editor de texto, incluindo salvar, abrir e criar novos projetos e arquivos, e também ferramentas de localização e troca de expressões.



Figura 6 – Barra principal do Code::Blocks

2. Barra do compilador

A barra do compilador enumera ações relacionadas à compilação e a execução do código. Estas ações são:

- a) *Build*: compila o código do projeto em questão, gerando um arquivo binário executável, caso seja necessário.
- b) *Run*: executa o arquivo binário executável já criado através do processo de compilação.
- c) *Build and run*: compila o código do projeto em questão, se necessário, e também executa-o imediatamente. Pode ser ativado pela tecla **F9**.
- d) *Rebuild*: ignora a verificação de necessidade da compilação, recompilando o projeto inteiro por completo.
- e) *Abort*: interrompe um processo de compilação.
- f) Barra de *Build Target*: permite mudar a configuração para a qual a compilação do projeto atual será realizada⁴.

4. O *target* de compilação de um projeto é um conjunto de configurações que visa alterar a forma como o mesmo é compilado. Esta mudança de configuração pode ser útil em situações específicas do estágio de desenvolvimento do *software*. Por exemplo, um *target Debug* poderia definir *flags* de compilação como `-g` para que símbolos de debug sejam exportados, afim de verificar passo-a-passo o andamento da execução do software durante o uso de um debugger. Um *target Release*, em contrapartida, pode especificar *flags* de otimização automática de código como `-O2`, por exemplo, garantindo execução mais rápida para algumas coisas.

- g) *Show the selected target dialog*: mostra a janela de configurações do *build target* selecionado na barra à esquerda.



Figura 7 – Barra do compilador do Code::Blocks

3. Barra do Debugger

A barra do Debugger enumera funcionalidades da ferramenta de *debug*, que costuma ser subestimada por desenvolvedores iniciantes.

Utilize as funcionalidades desta barra sempre que um erro de execução de código for incompreensível. Caso associada a um *build target* de Debug, o *debugger* possui o poder de indicar o local e o motivo exatos de alguns erros de compilação, sobretudo mostrando também a sequência de chamadas de *funções* para chegar a tal erro.

Usar o Debugger do Code::Blocks exige que o programa `gdb` esteja instalado no sistema.



Figura 8 – Barra do debugger do Code::Blocks

As funcionalidades desta barra são:

- Debug / Continue*: inicia ou continua o processo de Debug. Pode ser ativado pela tecla **F8**.
- Run to cursor*: executa o código até chegar à posição do cursor no mesmo.

- c) *Next line*: caso o processo de Debug esteja pausado, executa apenas a próxima linha de código.
- d) *Step into*: entra na definição de uma *função* sob o ponto de parada atual.
- e) *Step out*: retorna para a *função* superior que invocou a atual.
- f) *Next instruction*: vai para a próxima instrução a ser executada, independente da profundidade da chamada.
- g) *Step into instruction*: entra na definição da instrução atual.
- h) *Break debugger*: pausa imediatamente a execução no ponto atual.
- i) *Stop debugger*: interrompe o processo de execução.
- j) *Debugging windows*: mostra ou esconde janelas para inspeção de registradores, variáveis e outros valores específicos.
- k) *Various info*: mostra mais informações sobre o processo de debug.

4. Barra de completamento de código

A barra de completamento de código mostra dados da definição atual na qual o cursor do programador está. A caixa à esquerda mostra o *escopo* da *função* atual, e a barra da direita mostra informações sobre nome e tipo de retorno da *função* atual.

Esta barra também pode ser utilizada para navegar pelo código.

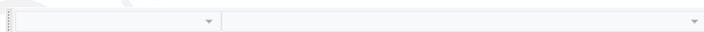


Figura 9 – Barra de completamento de código

5. Barra de gerenciamento de projetos

Esta é uma barra lateral, que normalmente é encontrada ao lado esquerdo do Code::Blocks.

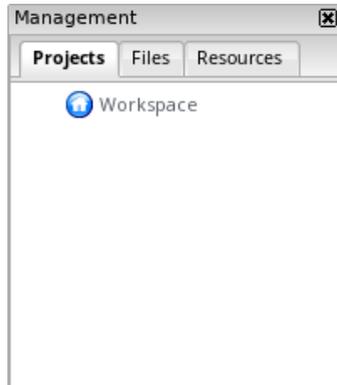


Figura 10 – Barra de gerenciamento de projetos

A barra de gerenciamento de projetos provê uma forma intuitiva de navegar em árvore pelos arquivos dos projetos atualmente abertos no Code::Blocks. Também possui uma ferramenta para navegar pelo sistema de arquivos do computador e pelos recursos utilizados pela aplicação.

6. Barra de logs e outros

A barra de logs e outros possui diversas abas, revelando o status da execução de diversas operações no Code::Blocks. Podemos destacar as abas *Build log*, que apresenta a saída da compilação de uma aplicação, e *Debugger*, que apresenta a saída da execução do programa de Debugger que o Code::Blocks utiliza.

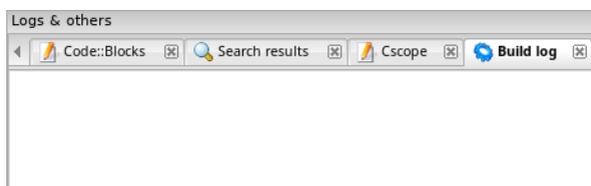


Figura 11 – Barra de logs & outros

1.1.4. Criando um projeto

Agora que estamos familiarizados com parte da interface do Code::Blocks, criaremos um projeto de uma aplicação C++ para execução no console.

Clique no menu **File**. Logo em seguida, vá para as opções **New > Project**.

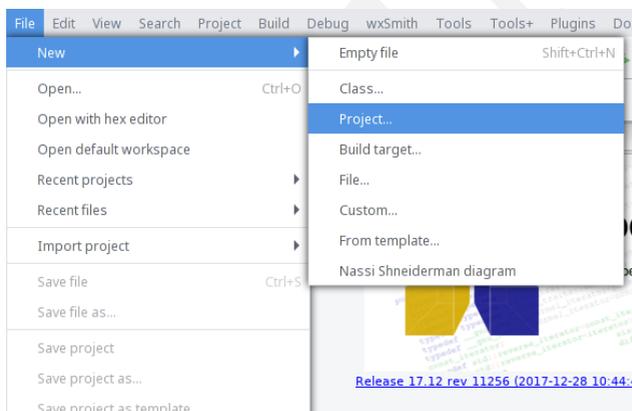


Figura 12 – Menu para criação de projetos

Uma janela de seleção de projetos será aberta. Nosso objetivo é

criar uma aplicação que será executada apenas no console, portanto selecione a opção **Console application**, e clique em **Go**.

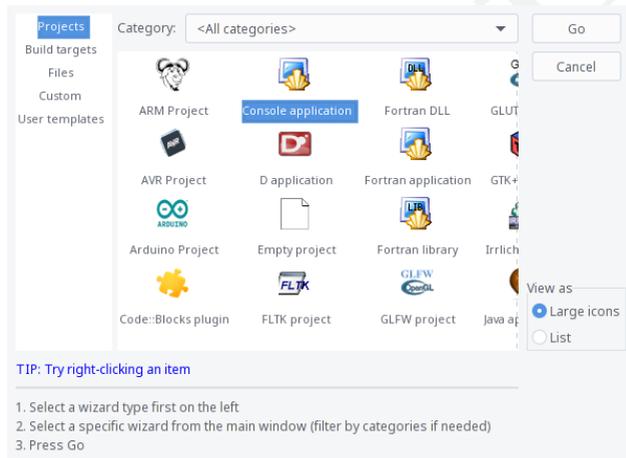


Figura 13 – Janela de seleção de projetos

Você será redirecionado para uma janela de assistente de criação de projetos. Na primeira tela (como mostrado na Figura 14), clique em **Next**.

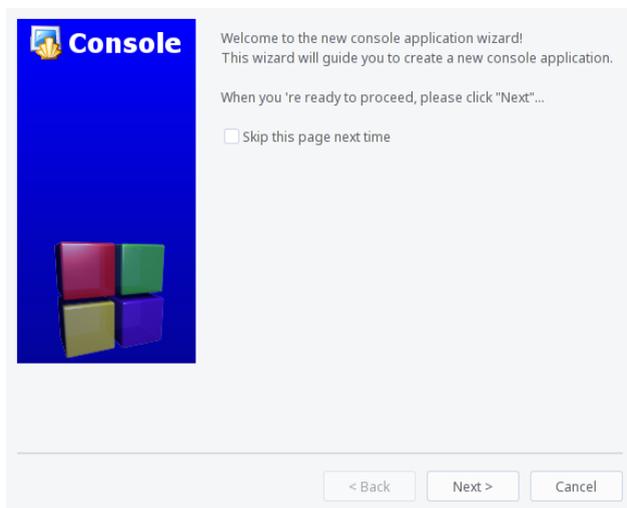


Figura 14 – Tela inicial do assistente de criação de projeto

Na tela seguinte, selecione a linguagem que você pretende utilizar para a aplicação. Deixe a opção **C++** selecionada⁵, e clique em **Next**.

5. As linguagens C e C++ podem ser radicalmente diferentes em alguns aspectos, por mais que seja verdade que C++ seja bastante retrocompatível com C. Este tópico será esmiuçado no próximo capítulo.

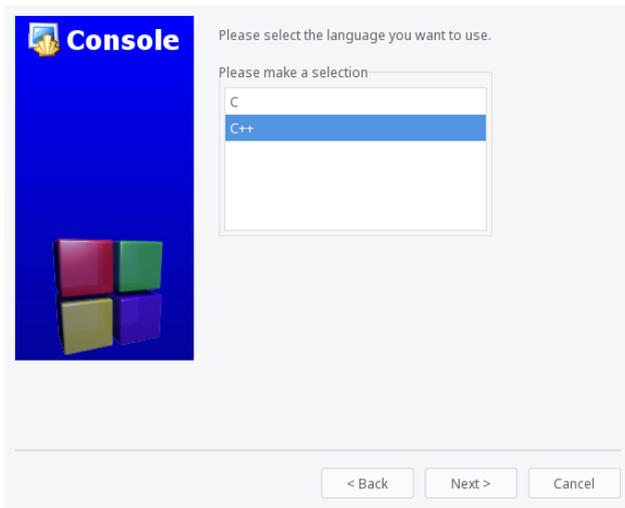


Figura 15 – Seleção de linguagem do assistente

A próxima tela pede por alguns dados do nome do projeto. Você poderá fornecer apenas o título do projeto (em *Project title*) e a pasta em que a nova pasta do projeto será criada (*Folder to create project in*). As demais caixas (*Project filename* e *Resulting filename*) serão preenchidas automaticamente de acordo com estas duas primeiras.

Por exemplo, suponhamos que nosso projeto, chamado `hello_world`, tenha seu nome ali escrito, e que selecionemos a pasta de criação como sendo `~/projects/cpp/`. Isto significa que será criada a pasta `hello_world` dentro da pasta de criação informada. Dentro desta nova pasta `hello_world`, será também criado um arquivo `hello_world.cpp`, que será o arquivo de configurações de projeto do Code::Blocks, bem como um arquivo de código `main.cpp` que servirá de *template*.

A saída a seguir demonstra a hierarquia de projeto criada para o exemplo supracitado.

```
1 projects
2 └─ cpp
3     └─ hello_world
4         └─ hello_world.cbp
5         └─ main.cpp
6
7 2 diretórios, 2 arquivos
```

A nova pasta `hello_world` abrigará tanto os detalhes do projeto quanto os arquivos de código-fonte e os binários compilados da aplicação. Após terminar o preenchimento, clique em **Next**.

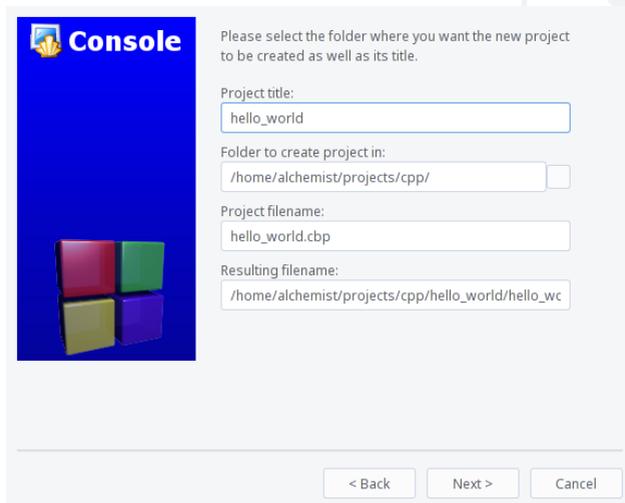


Figura 16 – Tela de dados do projeto

A última tela de criação de projeto é a tela de seleção de compilador e de criação de *targets* para compilação. Como estamos criando um projeto para C++, é interessante selecionar a opção **GNU GCC**

Compiler. Caso você tenha optado por utilizar o compilador Clang, poderá selecionar também **LLVM Clang Compiler**. As demais opções na lista dizem respeito ao desenvolvimento em Fortran, D, ou também a C/C++, porém para arquiteturas diferentes, o que não será abordado neste texto.

Caso a tela corresponda ao que está apresentado na Figura 17, basta clicar em **Finish**.

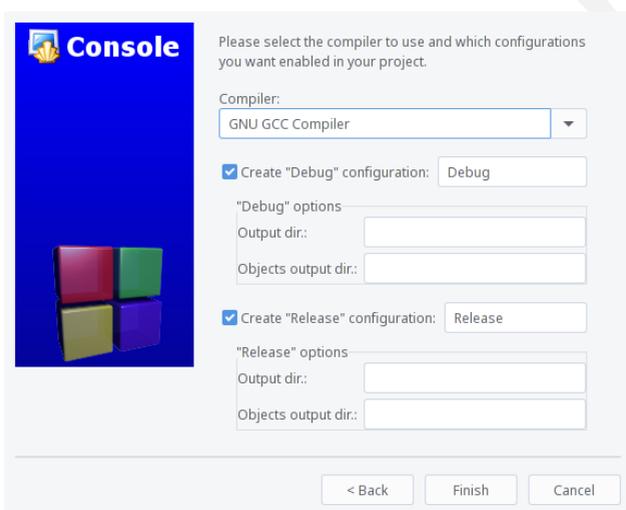


Figura 17 – Tela de seleção de compilador

Após estes procedimentos, o projeto será criado e imediatamente mostrado na árvore de gerenciamento de projetos, como mostra a Figura 18. Na pasta *Sources*, estará o arquivo de código `main.cpp`, que será utilizado como código.

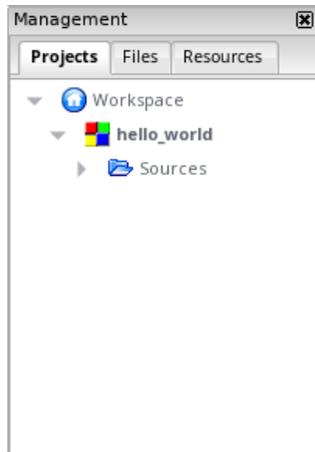


Figura 18 – Árvore de projetos após a criação do projeto

1.2. USANDO O CONSOLE DO LINUX

Você poderá achar mais conveniente utilizar diretamente o console do Linux ao invés do Code::Blocks. Para tanto, você precisará instalar o compilador de C++ de sua preferência.

1.2.1. Instalando o compilador

Em geral, a instalação de compiladores de C++ no Linux resumem-se aos seguintes comandos:

- Instalação em distribuições baseadas em Debian:

```
1 sudo apt install build-essential
```

- Instalação em distribuições da Red Hat:

```
1 sudo dnf install gcc-c++
```

- Instalação em distribuições baseadas em Arch Linux:

```
1 sudo pacman -S gcc
```

Caso você queira usar o *debugger*, também será necessário instalar a aplicação `gdb` (GNU Debugger). Esta ferramenta é compatível com GCC e LLVM Clang. Consulte a documentação da sua distribuição para encontrar o pacote para esta aplicação.

Informações sobre uso do Debugger podem ser vistas no Apêndice B. Para maiores informações sobre estes comandos, veja a seção 1.1.1.

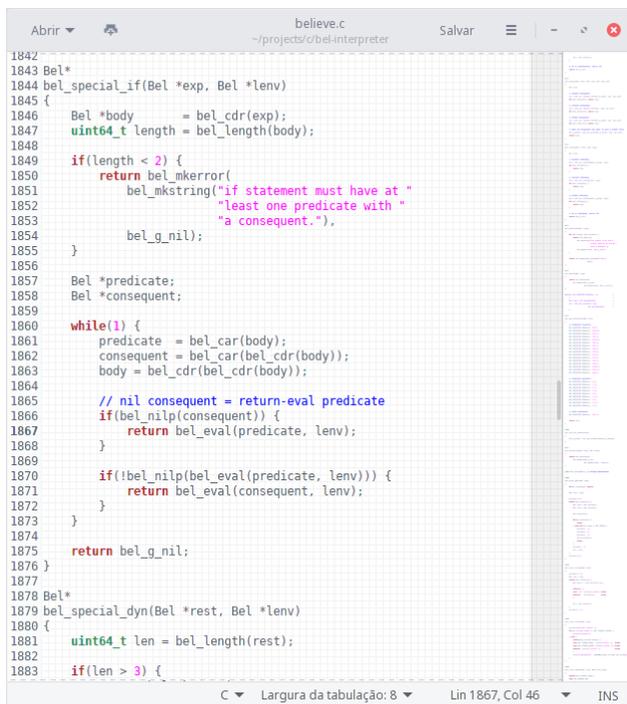
1.2.2. Sobre editores de texto

Como dito anteriormente, edição de código pode ser feita através de qualquer editor de texto. Obviamente, um editor de texto otimizado para a edição de código (em especial código C e C++) ajuda bastante na tarefa.

A escolha do editor fica a critério do programador. Como exemplo, mostraremos alguns editores de texto populares no mundo Linux. O leitor poderá opcionalmente pesquisar pelo editor de texto de sua preferência, caso este chame sua atenção.

1. Gedit

O editor de texto Gedit é o editor padrão de ambientes baseados em ambiente Gnome, sendo um editor gráfico com suporte a alguns plugins, que pode ser observado na Figura 19.



```
1842
1843 Bel*
1844 bel_special_if(Bel *exp, Bel *lenv)
1845 {
1846     Bel *body      = bel_cdr(exp);
1847     uint64_t length = bel_length(body);
1848
1849     if(length < 2) {
1850         return bel_mkerror(
1851             bel_mkstring("if statement must have at "
1852                 "least one predicate with "
1853                 "a consequent."),
1854             bel_g_nil);
1855     }
1856
1857     Bel *predicate;
1858     Bel *consequent;
1859
1860     while(1) {
1861         predicate = bel_car(body);
1862         consequent = bel_car(bel_cdr(body));
1863         body = bel_cdr(bel_cdr(body));
1864
1865         // nil consequent = return-eval predicate
1866         if(bel_nilp(consequent)) {
1867             return bel_eval(predicate, lenv);
1868         }
1869
1870         if(!bel_nilp(bel_eval(predicate, lenv))) {
1871             return bel_eval(consequent, lenv);
1872         }
1873     }
1874
1875     return bel_g_nil;
1876 }
1877
1878 Bel*
1879 bel_special_dyn(Bel *rest, Bel *lenv)
1880 {
1881     uint64_t len = bel_length(rest);
1882
1883     if(len > 3) {
```

Figura 19 – Gedit editando código C

2. Nano

O editor de texto Nano é particularmente útil para situações de edição de código rápido, diretamente através do terminal (vide Figura 20). Todavia, ele não possui customizações e muitas facilidades para digitação de código.

```

GNU nano 4.5 src/9mine.c
  for(i = 0; i < FIELD_SIDE; i++)
    print("%03d ", i + 1);
    print("\n");
  }

void
print_vertical_letter(char letter, int side)
{
    print((!side) ? "%c |" : "| %c", letter);
}

void
print_field_godmode(void)
{
    int i, j;
    print_horizontal_ruler();
    for(i = 0; i < FIELD_SIDE; i++) {
        print_vertical_letter('A' + (char) i, 0);
        for(j = 0; j < FIELD_SIDE; j++) {
            print(" ");
            switch(field_cell(i, j)) {
                case NOTHING: print(""); break;
                case BOMB:    print("+"); break;
                default:
                    print("%d", field_cell(i, j));
                    break;
            }
            print(" ");
        }
        print_vertical_letter('A' + (char) i, 1);
        print("\n");
    }
    print_horizontal_ruler();
}

void
print_field(void)
{
    int i, j;
    print_horizontal_ruler();
    for(i = 0; i < FIELD_SIDE; i++) {
        print_vertical_letter('A' + (char) i, 0);
        for(j = 0; j < FIELD_SIDE; j++) {

```

^{^G} Ajuda ^{^C} Gravar ^{^W} Onde está? ^{^R} Recort txt ^{^J} Justificar ^{^O} Pos atual
^{^X} Sair ^{^B} Ler o arq ^{^N} Substituir ^{^U} Colar txt ^{^T} VerfÓrtog ^{^I} In p/ linha

Figura 20 – Nano editando código C

3. Vim

O editor de texto Vim possui uma interface extremamente amigável e é altamente configurável, através da sua própria linguagem de configuração, sendo ideal tanto para terminal (Figura 21) quanto para interface gráfica (via GVim). Sua curva de aprendizado é um pouco mais íngreme que o Nano, mas não tanto. É uma ferramenta muito valiosa para desenvolvedores

de quaisquer linguagens.

```

#define __st_array_check_avail(array) \
    ((array->num_elements < array->array_size)

#define __st_array_put_at(array, index, element_ptr) \
    memcpy((array->elements + (index * array->element_size)), \
           element_ptr, array->element_size); \
    array->num_elements++;

static inline int
__st_array_grow_if_needed(st_array* array)
{
    if(!__st_array_check_avail(array)) {
        st_log_warn("attempt to grow array for adding more elements");
        if(st_array_grow(array, array->array_size + 2)) {
            st_log_err("cannot add element to array: out of memory");
            return 1;
        }
    }
    return 0;
}

int
st_array_put(st_array* array, size_t n, void* element)
{
    if(!array) {
        st_log_err("attempt to perform operation on NULL reference to array");
        return 1;
    } else if(!element) {
        st_log_err("cannot add a NULL element to array");
        return 1;
    }

    // If our array is full, make it grow
    if(__st_array_grow_if_needed(array))
        return 1;

    __st_array_put_at(array, n, element);
    return 0;
}

/* int */
/* st_array_add(st_array* array, size_t* out, void* element) */
/* { */
/*     if(!array) { */

```

Figura 21 – Vim editando código C

4. Emacs

Emacs é um editor de texto altamente configurável, podendo ser executado através de interface gráfica ou terminal. Sua curva de aprendizado é consideravelmente mais elevada que o Vim, e sua linguagem de configuração é uma linguagem de programação propriamente dita, fazendo parte da família de dialetos de Lisp. Todavia, esta é uma ferramenta extremamente

versátil e com milhares de plugins, sendo capaz tanto de editar e compilar código com facilidade, como também de exibir calendário, organizar agenda, acessar e-mail e até mesmo páginas da internet, sendo um verdadeiro *canivete suíço* dentre os editores de texto.

A Figura 22 demonstra o editor Emacs após uma grande modificação na interface. Foi feita a adição de elementos estéticos como fontes especiais, numeração de linhas e esquema de cores.

```
275 | st_vec3 neg;  
276 | __st_generic_vec_neg((float*)&neg, (const float*)&a, 3);  
277 | return neg;  
278 | }  
279 |  
280 | float  
281 | st_vec3_sqdist(st_vec3 a, st_vec3 b)  
282 | {  
283 |     float deltaX = a.x - b.x,  
284 |     deltaY = a.y - b.y,  
285 |     deltaZ = a.z - b.z;  
286 |     return (deltaX * deltaX)  
287 |         + (deltaY * deltaY)  
288 |         + (deltaZ * deltaZ);  
289 | }  
290 |  
291 | float  
292 | st_vec3_sqlen(st_vec3 a)  
293 | {  
294 |     return (a.x * a.x) + (a.y * a.y) + (a.z * a.z);  
295 | }  
296 |  
297 | float  
298 | st_vec3_dot(st_vec3 a, st_vec3 b)  
299 | {  
300 |     return (a.x * b.x) + (a.y * b.y) + (a.z * b.z);  
301 | }  
302 |  
303 | st_vec3  
304 | st_vec3_cross(st_vec3 a, st_vec3 b)  
305 | {  
306 |     st_vec3 cross;  
307 |     cross.x = (a.y * b.z) - (a.z * b.y);  
308 |     cross.y = -((a.x * b.z) - (a.z * b.x));  
309 |     cross.z = (a.x * b.y) - (a.y * b.x);  
310 |     return cross;  
311 | }  
312 |  
313 | void  
314 | st_vec3_print(const st_vec3* a)  
315 | {  
316 |     if(!a) {  
317 |         st_log_err("attempt on printing NULL reference to vector");  
318 |         return;  
319 |     }  
320 |     fputs("vec3 { ", stdout);  
321 |     __st_generic_vec_print((const float*)a, 3);  
322 | }
```

stadium -i--@- stmath.c 0/1 26% 271: 0

Figura 22 – Emacs editando código C

1.2.3. Criando e navegando por pastas

A utilização do console do Linux pressupõe que você conheça alguns comandos que são comumente usados, para que você possa se locomover através de pastas, criar e editar arquivos.

Estes comandos são, na realidade, programas que são executados com sua invocação. Eis uma lista destes programas e de seus usos comuns:

- `pwd`: mostra o caminho completo do diretório atual.
- `ls`: mostra todos os arquivos do diretório atual.
- `ls <dir>`: mostra todos os arquivos do diretório `<dir>`.
- `cd <dir>`: muda o diretório atual para o diretório `<dir>`.
- `touch <arq>`: cria um arquivo de texto vazio chamado `<arq>`.
- `rm <arq>`: remove permanentemente o arquivo `<arq>`.
- `rm -r <dir>`: remove permanentemente o diretório `<dir>`.

Para os comandos acima, os diretórios `<dir>` podem ser substituídos por um caminho absoluto de um diretório, um subdiretório no diretório atual, ou pelos símbolos `.` (significando o diretório atual) ou `..` (significando o diretório acima do atual).

Suponha a seguinte hierarquia de diretórios:

```
1  foo
2  └─ bar
3     └─ baz
4     └─ quux
5
6  1 diretório, 2 arquivos
```

Caso você esteja no diretório `bar`, eis a execução esperada para alguns comandos. Os comandos em questão estarão precedidos por um `$`, e sua saída será apresentada a seguir, caso o comando mostre algo na tela.

- Mostrar o diretório atual

```
1 $ pwd
2 /home/user/foo/bar
```

- Mostrar os arquivos no diretório atual

```
1 $ ls
2 baz quux
```

- Subir para um diretório acima

```
1 $ cd ..
2 $ pwd
3 /home/user/foo
```

- Criar um arquivo de texto vazio chamado `blah`

```
1 $ touch blah
2 $ ls
3 bar blah
```

- Remover o diretório `bar`

```
1 $ rm -r bar
2 $ ls
3 blah
```

1.2.4. Escrevendo arquivos de código

Você poderá utilizar qualquer editor de texto para escrever suas aplicações, como demonstrado na seção 1.2.2.

Certifique-se de que, caso você esteja escrevendo código C++, seus arquivos possuam a extensão `.cpp` ou `.cxx`⁶. Por exemplo, um arquivo de código qualquer poderia se chamar `main.cpp` ou `main.cxx` mas, por convenção, usaremos a extensão `.cpp` ao longo deste texto.

1.2.5. Usando o compilador através do console

O compilador GCC é um programa que, quando instalado em um sistema Linux, torna-se disponível globalmente no console. Isto significa que ele age como um *comando do console*, da mesma forma que os comandos da seção 1.2.3.

1. Criando um arquivo de objeto

Suponha uma situação em que temos um arquivo de código válido chamado `main.cpp` na pasta atual.

```
1 $ ls
2 main.cpp
```

6. Arquivos da linguagem C normalmente possuem extensões `.c` e `.h`, e arquivos da linguagem C++ normalmente possuem extensões `.cpp` ou `.cxx`, e `.hpp` ou `.hxx`. Arquivos `.h` e `.hpp` ou `.hxx` são arquivos especiais, representando *cabeçalhos* de código. Este tipo de arquivo será discutido no próximo capítulo.

Para compilar o arquivo, basta usar o programa `g++`⁷ para compilar.

O primeiro passo é gerar um arquivo de objeto, de extensão `.o`. Este tipo de arquivo ainda não é o nosso executável, mas é um arquivo com os *símbolos* compilados, ainda sem suas respectivas dependências.

Para criar o arquivo de objeto, execute o comando:

- `g++ -c arquivo.cpp -o arquivo.o`

Exemplificando:

```
1 $ g++ -c main.cpp -o main.o
2 $ ls
3 main.cpp main.o
```

2. Por que criar arquivos de objeto?

Arquivos de objeto são úteis para o processo de compilação. Quando lidamos com um *software* C++ muito grande, com muitos arquivos de código, é conveniente compilarmos todos os arquivos `.cpp` em seus respectivos arquivos de objeto `.o`.

Proceder desta forma evita recompilar o *software* por completo quando uma modificação é feita em apenas um dos arquivos: basta gerar o código de objeto do arquivo alterado, e então proceder com o processo de compilação. Isto torna a compilação mais rápida, dependendo da natureza do código.

A IDE Code::Blocks realiza este procedimento automaticamente, e verifica se um arquivo `.cpp` foi alterado para que isto seja feito.

7. Caso você opte por utilizar LLVM Clang, use o comando `clang++`.

3. Criando o executável

Agora que possuímos arquivos de objeto, podemos juntá-los em apenas um executável através de um processo conhecido como *linking*. Este processo é responsável por cruzar as dependências dos arquivos de objeto. É também durante este processo que bibliotecas externas podem ser utilizadas.

A criação do executável criará um arquivo executável com o nome informado através da linha de comando. No Windows, o nome do executável terá uma extensão `.exe` colocada ao final do seu nome de arquivo.

Para criar o executável a partir de arquivos de objeto, execute o comando:

- `g++ arq1.o arq2.o arq3.o... -o nome_do_executavel`

Exemplificando:

```
1 $ g++ main.o -o meu_programa
2 $ ls
3 main.cpp main.o meu_programa
```

4. Criando o executável diretamente

Em algumas situações, quando o projeto é muito pequeno, é possível não realizar o trabalho de gerar arquivos de código de objeto primeiro, para depois criar um executável; ao invés disso, podemos criar o executável diretamente.

Para fazer isto, basta executar o seguinte comando. Note a ausência da bandeira `-c`, e note também que o nome do executável é passado diretamente em um passo apenas:

- `g++ arq1.cpp arq2.cpp... -o meu_programa`

Exemplificando:

```
1 $ g++ main.cpp -o meu_programa
2 $ ls
3 main.cpp meu_programa
```

5. Bandeiras importantes na compilação

Você deve ter percebido que tanto o processo de geração de arquivos de objeto quanto o processo de *linking* envolvem o uso de algumas letras precedidas por hífen (-) nos comandos. Estas letras não são utilizadas ao acaso. Elas são chamadas de bandeiras ou *flags*, e algumas delas exigem escrever algo à sua frente.

- A *flag* `-o` instrui o compilador a gerar um arquivo de saída para a operação sendo feita. Por isso, é necessário informar o nome do arquivo de saída logo à sua frente;
- A *flag* `-c` instrui o compilador a gerar código de objeto para o arquivo a ser compilado. Ela não exige nada à sua frente mas, caso não seja usada, criará um arquivo executável diretamente.

Algumas flags extras podem ser usadas no processo de compilação:

- A *flag* `-g` compila o programa com *símbolos de debug*. É essencial a utilização desta *flag* durante o processo de teste da sua aplicação. Do contrário, seu *software* não será corretamente inspecionado pelo Debugger. A IDE Code::Blocks insere esta *flag* por padrão no *target* Debug;
- As flags `-O1`, `-O2`, `-O3` e `-Os` realizam otimizações na geração de código de objeto. As listas de otimizações podem

ser encontradas na documentação dos compiladores GCC e LLVM Clang. Recomendamos a utilização da flag `-O2` em situações de *lançamento* do software;

- A flag `-Wall` mostra vários avisos extras durante a compilação. Estes avisos são interessantes para evitar erros comuns de sintaxe, e costumam ser bem legíveis. É importante ler avisos do compilador para evitar imprevistos;
- A flag `-Werr` funciona como `-Wall`, porém não prossegue com o processo de compilação caso algum aviso apareça;
- *Flags* que começam com `--std=` são *flags* que instruem o compilador a utilizar uma versão específica da linguagem C++. Por padrão, os compiladores GCC e LLVM Clang (atualmente em suas versões 9.2.0 e 9.0.0, respectivamente), utilizam a especificação de 1998 por padrão. Caso você queira usar uma versão mais nova da linguagem, será necessário informá-la com esta *flag*.

Por exemplo, suponhamos que um *software* necessite ser compilado para Debug, segundo a especificação de C++ de 2014. Ele deverá ser compilado desta forma:

```
1 $ g++ -Wall -g --std=c++14 main.cpp -o meu_programa
```

Todavia, caso este *software* esteja sendo compilado para comercialização, ele poderia ser compilado sem símbolos de Debug, e com otimizações:

```
1 $ g++ -Wall -O2 --std=c++14 main.cpp -o meu_programa
```

Note que *flags* podem ser intercaladas, desde que *flags* que exigem alguma informação a mais não sejam separadas de tais informações (como é o caso de `-o meu_programa`).

6. Executando o programa

Em sistemas Linux, a execução de um programa pode ser realizada através de navegar até a pasta do mesmo, e escrever um comando iniciado com `./`, seguido do nome do executável.

Exemplificando:

```
1 $ ./meu_programa
```

1.3. PROGRAMANDO EM C++

Agora que estamos familiarizados com as ferramentas essenciais para programação, podemos nos familiarizar com a linguagem C++.

Nas próximas seções, destrincharemos o arquivo `main.cpp`, gerado na criação de um projeto C++ do Code::Blocks. Caso você esteja programando sem auxílio desta IDE, poderá digitar o código no editor de texto de sua preferência.

1.3.1. O programa 'Hello World'

O programa "Hello World" é um dos tipos mais simples de programa que pode ser feito com qualquer linguagem de programação. Muitas vezes, este programa é considerado uma Pedra de Rosetta⁸ durante o aprendizado de linguagens de programação.

O código a seguir é o código de um "Hello World", como gerado automaticamente pela IDE Code::Blocks. Caso você não esteja utilizando a IDE, basta digitar o código no arquivo `main.cpp`.

8. A Pedra de Rosetta é uma pedra de granodiorito descoberta em 1799, com inscrições de um decreto feito em Memphis, Egito, em 196 a.C. O decreto era escrito em Egípcio antigo (usando hieróglifos e demótico) e Grego antigo. O uso da Pedra foi fundamental na tradução de textos em egípcio antigo.

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

Quebraremos o código em elementos para explicarmos o que cada um deles significa.

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

Normalmente, no início de arquivos de código, definimos diretivas para a importação de funcionalidades a partir de outros lugares. Chamamos estes conjuntos de funcionalidades de *bibliotecas*.

A *biblioteca iostream* é uma biblioteca de C++ que define certas estruturas para impressão de texto no *console* e para recuperação de valores digitados no mesmo pelo usuário.

Com esta biblioteca, podemos usar os comandos `std::cout` para imprimir valores na tela, e `std::cin` para ler valores digitados.

```
1 using namespace std;
```

Esta é uma diretiva de importação de *namespace*. Na prática ela é utilizada para tornar a escrita do código mais sucinta. Com ela, ao invés de utilizarmos os comandos `std::cout` e `std::cin`, podemos utilizar apenas `cout` e `cin`, respectivamente.

A diretiva acima é uma *expressão* na linguagem de programação C++. A maioria das expressões de C++ precisam terminar com um ponto-e-vírgula (;). Omitir este caractere constitui um *erro de sintaxe*.

```
1 int main()  
2 {  
3     ...  
4 }
```

Todo o código remanescente no arquivo resume-se a um único bloco, delimitado pelas expressões acima. Este bloco é conhecido como *função principal*.

Esta *função* descreve todos os comandos que serão executados quando o programa inicia sua execução, por isso, conhecemos a *função main* como sendo o *ponto de entrada* da aplicação.

Veremos mais sobre funções no futuro.

```
1 cout << "Hello world!" << endl;
```

O comando `cout` é o comando relacionado à impressão de informações no *console*. Este comando normalmente vem seguido do *operador* `<<`. Após este operador, virá a informação a ser impressa na tela.

Como pode ser visto nesta linha de código, o uso de `cout` não é limitado a apenas um dado. Mais dados podem ser imediatamente escritos se colocarmos mais um operador `<<`.

A expressão acima termina com `endl`, que simboliza um caractere que faz com que o *console* pule para a linha abaixo. Como esta linha é uma *expressão comum*, ela termina em ponto-e-vírgula (`;`).

```
1 return 0;
```

A maioria das funções possui um valor de retorno. Isto será explicado com detalhes em outro momento.

Por padrão, a *função principal* encerra sua execução com uma expressão de *retorno*, mostrada acima. Esta expressão faz com que a *função principal* retorne um certo valor para o Sistema Operacional.

Por definição, o valor de retorno `0` indica que o programa foi executado corretamente. Esta expressão também é encerrada por um ponto-e-vírgula (`;`).

É interessante que a função `main` sempre possua uma expressão de retorno, para que o Sistema Operacional possa determinar se a execução do programa ocorreu da forma esperada.

- **NOTA:** Qualquer outra expressão após a execução do retorno da função *não será executada*. Portanto, cuidado para não executar um retorno antes de alguma outra expressão.

1.4. VARIÁVEIS

O principal objetivo de todos os programas de computador é a *manipulação de dados*. E para que manipulemos estes dados, precisamos primeiramente armazená-los em *variáveis*.

Variáveis são espaços de memória *nomeados pelo programador*, utilizados para armazenamento de informação. Em C++ estes espaços são alocados na memória RAM, com um tamanho exato correspondente ao *tipo* de variável igualmente declarado pelo programador.

A linguagem C++ possui diversas funcionalidades para armazenar e estruturar vários tipos de dados.

1.4.1. Tipos primitivos

Inicialmente, veremos os tipos mais primitivos na linguagem C++, que podem ser utilizados para declarar variáveis mais comuns.

1. `int`

O tipo `int` é capaz de armazenar valores inteiros.

2. `float`

O tipo `float` armazena um número real, também conhecido como ponto flutuante.

Literais de pontos flutuantes normalmente são escritas com um ponto separando as casas decimais, e um `f` ao final, indicando ser um valor `float` (por exemplo, `3.14f`).

3. `double`

O tipo `double` assemelha-se ao tipo `float`, todavia possui *precisão dupla*. Grosso modo, isto significa que `double` tem a capacidade de armazenar até o dobro das casas decimais de um `float`.

Literais de pontos flutuantes de precisão dupla assemelham-se às literais `float`, com a diferença de não possuírem o sufixo `f` (por exemplo, `3.14`).

4. `char`

O tipo `char` representa um caractere. Estes caracteres devem fazer parte da codificação ASCII (*American Standard Code for Information Interchange*), normalmente equivalente ao tamanho de um *byte*.

Durante este livro, não utilizaremos caracteres como vogais acentuadas, cedilha ou outros caracteres especiais no código, pois eles pertencem ao padrão Unicode, o que um mero `char` não suporta armazenar.

Para expressar um `char`, utilize o caractere em questão entre aspas simples. Por exemplo: `'a'`. As aspas duplas (`"`) estão reservadas para *strings*, uma coleção de caracteres, que serão melhor vistas posteriormente.

5. `bool`

O tipo `bool` representa uma variável *booleana*, ou seja, capaz de apresentar os estados `true` ou `false`. O tipo `bool` pode ser

trocado por um tipo `int`⁹, desde que mantenha-se a relação de que a constante `false` equivale ao número 0, e `true` equivale a 1.

Via de regra, uma verificação por *falsidade* em C++ também pode ser feita verificando se uma variável é *igual* a 0 ou `false`.

1.4.2. Declaração de variáveis

Tomemos a seguinte declaração no corpo da função principal:

```
1 int valor;
```

Esta expressão determina a existência de uma variável de nome `valor`, cujo tipo é `int`. Isto significa que esta variável só poderá armazenar um número inteiro.

Veja que o nome `valor` não possui caracteres especiais ou números em seu início. Fosse este o caso, o programa teria um *erro de sintaxe*. Ademais, veja também que o nome `valor` possui algum significado no escopo; evite utilizar nomes não-significativos em variáveis.

Por enquanto, a variável `valor` não possui nenhum valor armazenado. Quando isto acontece, dizemos que seu valor é *indefinido*, e equivale a *lixo de memória*¹⁰.

Podemos definir um valor inicial para esta variável através de uma *atribuição*. Para tanto, utilizamos o operador de atribuição (=):

```
1 int valor;  
2 valor = -5;
```

9. De fato, a linguagem C não possui um tipo `bool`; todo e qualquer valor *booleano* é armazenado em variáveis `int`.

10. Sempre realize atribuições às variáveis que criar, assim que for possível, antes de fazer qualquer operação com elas. Realizar uma operação com lixo de memória resulta em um *comportamento indefinido* no seu programa.

A partir da segunda linha de código, a variável `valor` armazenará o valor inteiro `-5`.

Estas duas linhas de código podem ser abreviadas para uma única linha:

```
1 int valor = -5;
```

Podemos, também declarar mais de uma variável de uma vez, desde que sejam do mesmo tipo. Por exemplo, criaremos duas variáveis do tipo `int` em apenas uma linha:

```
1 int valor1, valor2;
```

Também podemos realizar este tipo de declaração associando, opcionalmente, valores a cada uma das variáveis apresentadas:

```
1 int valor1 = -5, valor2 = -6;
```

1.4.3. Casos especiais de `int`

O tipo `int` possui alguns casos especiais que manipulam a capacidade de armazenamento ou a natureza dos números armazenados. Alguns destes casos serão listados a seguir.

1. `signed int`

Esta é uma forma redundante de declarar um `int`: um número inteiro com sinal. Um `int` tem a garantia de ter pelo menos 16 bits de tamanho, mas sistemas modernos normalmente implementam-no com 32 bits.

2. `unsigned` ou `unsigned int`

Um `unsigned` é um `int`, capaz de armazenar somente *números positivos*. O menor número que este tipo pode armazenar, portanto, é 0; tentar atribuir um valor menor que este causará um

estouro de memória, que fará com que a variável assuma um valor incorreto; todavia, este erro é silencioso e poderá realizar *comportamento indefinido* na execução de seu programa.

Caso você não tenha certeza de que sua variável será sempre maior ou igual a zero, utilize um `int`.

Os próximos casos especiais podem também ser precedidos de `unsigned`.

3. `short` ou `short int`

Um `short` é um `int` com metade da sua capacidade de armazenamento. Em alguns sistemas, o `short` pode possuir o mesmo tamanho de um `int`. Sistemas modernos costumam implementar o `short` com 16 bits de tamanho.

4. `long` ou `long int`

Um `long` é um `int` com o dobro da sua capacidade de armazenamento. Em alguns sistemas, o `long` pode possuir o mesmo tamanho de um `int`, como em sistemas modernos, que implementam o `long` com pelo menos 32 bits de tamanho.

5. `long long` ou `long long int`

Um `long long` é um `int` com o quádruplo de sua capacidade de armazenamento. Sistemas modernos costumam implementar o `long long` com 64 bits de tamanho.

1.5. OPERADORES E ARITMÉTICA

Ao escrevermos expressões em C++, precisaremos realizar certas operações relacionadas à manipulação de variáveis ou a aritmética básica. Para tanto, utilizaremos os *operadores* apresentados a seguir.

1.5.1. Atribuição (=)

Como demonstrada anteriormente, a *atribuição* é utilizada de forma a modificar o valor de uma certa variável, como nos exemplos a seguir:

```
1 int valor = -5;
2 float outro_valor = 1.0f;
3 bool foo = true;
```

1.5.2. Aritmética básica (+, -, *, /, %)

Podemos realizar operações de aritmética básica de soma, subtração, multiplicação e divisão usando os operadores para tal.

```
1 int a = 5;
2 int b = 6;
3
4 int soma = a + b;
5 int subtr = a - b;
6 int div = b / 3;
7 int mult = a * b;
```

O operador *mod* (%) lembra a divisão. Todavia, ao invés de retornar o *quociente* da divisão, retorna o *resto da divisão inteira*.

```
1 int mod = a % 2;
```

Para a expressão acima, a variável `mod` valerá 1, pois a divisão inteira do valor de `a` por 2 resulta em 2, com um resto 1.

1.5.3. Parênteses ((e))

Use parênteses para agrupar expressões e evitar ambiguidade nas expressões. Compare as duas linhas de código a seguir:

```
1 int expressao_1 = 3 * 5 + 2;  
2 int expressao_2 = 3 * (5 + 2);
```

O valor final de `expressao_1` será 17, e o valor final de `expressao_2` será 21. Quando operadores não são agrupados com parênteses, valem as regras de *precedência de operadores*, a serem discutidas na Seção 1.6.

1.5.4. Aritmética atributiva (+=, -=, *=, /=)

Algumas vezes, precisamos realizar uma operação em uma variável, e então atribuir o novo valor a ela. Veja este caso:

```
1 int teste = 5;  
2 teste = teste + 2;
```

Para tornar a segunda linha mais sucinta, podemos utilizar um operador de *aritmética atributiva*, que isenta o programador de reescrever o nome da variável. Assim, a segunda linha do código equivalerá a esta:

```
1 teste += 2;
```

O caso acima refere-se a uma soma entre uma variável e outro valor ou variável qualquer. Podemos reproduzir o mesmo para subtração (operador `-=`), multiplicação (operador `*=`) e divisão (operador `/=`).

Outros operadores ainda não mostrados também possuem uma versão *atributiva*, mas estes são os mais utilizados.

1.5.5. Incremento e decremento (++ , --)

Veja a seguinte expressão.

```
1 int teste = 5;  
2 teste += 1;
```

Quando uma variável é modificada, recebendo como valor o *incremento* (acréscimo de uma unidade) de si mesma, podemos tornar o código ainda mais sucinto utilizando o operador de *incremento*:

```
1 teste++;
```

Caso o objetivo seja remover uma unidade do valor de uma variável, podemos utilizar o operador de *decremento*. As seguintes linhas são equivalentes:

```
1 teste -= 1;  
2 teste--;
```

- **Sobre o uso de incremento e decremento**

Estes operadores são conhecidos como *unários*, pois executam uma operação em apenas um *operando*: a variável. Todavia, eles podem ser utilizados antes ou depois da variável em questão:

```
1 ++teste;  
2 --teste;
```

O detalhe é que **estas expressões não são equivalentes às mostradas anteriormente**, mas diferença torna-se evidente apenas quando *incremento* e *decremento* são feitos dentro de outra expressão. Considere o exemplo a seguir.

```
1 int a = 5;  
2 int b = 6;  
3
```

```
4 int c = a++;  
5 int d = ++b;
```

Os valores de `c` e `d` após a execução deste bloco de código serão 5 e 7, respectivamente.

Quando o operador de *incremento* vem **após** o nome da variável, isto significa que qualquer operação com a variável será feita *primeiro*, e só então o *incremento* será feito. Ou seja, primeiramente o valor de `a` é copiado para `c`, e só então `a` será incrementado.

Quando o operador de incremento vem **antes** do nome da variável, isto significa que o *incremento* antecede quaisquer operações com a variável. Assim, `b` é incrementado, e só então seu valor será copiado para `d`.

As mesmas regras aplicam-se ao operador de *decremento*.

1.5.6. Comparação (==, !=, <, >, <=, >=)

Em algumas situações, é necessário realizar comparações entre valores. Para tanto, podemos utilizar os operadores de comparação.

Um operador de comparação sempre retorna um valor booleano. No exemplo a seguir, comparamos duas variáveis para determinar se seus valores são iguais, e armazenamos esta informação na variável `resultado`:

```
1 int a = 5;  
2 int b = 5;  
3 bool resultado = (a == b);
```

Note a diferença entre os operadores de atribuição (=) e igualdade (==). Uma atribuição é sempre uma modificação de

uma variável, enquanto a **igualdade** compara dois valores. Cuidado com erros de sintaxe!

Outros operadores de comparação são as verificações de *desigualdade* (\neq), *menor que* ($<$), *maior que* ($>$), *menor ou igual a* (\leq) e *maior ou igual a* (\geq).

- **Atenção!** Operadores de comparação são *diádicos*, isto é, funcionam apenas entre duas variáveis. Uma relação na forma $a \leq x \leq b$, por exemplo, demanda o uso de um operador *lógico* e duas expressões de *comparação*, como veremos a seguir.

1.5.7. Lógicos (&&, ||, !)

Operadores *lógicos* normalmente associam-se a operadores de *comparação* em expressões, em especial para verificar "verdades" compostas. Por exemplo, dadas duas variáveis a e b , suponhamos que precisemos comparar uma variável x onde $a \leq x \leq b$. Em C++, faremos algo como:

```

1  int a = 2;
2  int b = 5;
3  int x = 3;
4
5  bool resultado = (a <= x) && (x <= b);

```

Veja que a expressão $a \leq x \leq b$ foi quebrada em duas expressões: $a \leq x$ e $x \leq b$ ¹¹. A variável `resultado` só será verdadeira se ambas estas expressões forem verdadeiras também.

Em suma, teremos:

- O operador *and lógico* (**&&**) constitui uma relação onde a expressão inteira será verdadeira, se e somente se ambos os membros forem verdadeiros ao mesmo tempo;

11. Matematicamente, $(a \leq x \leq b) \iff (a \leq x) \wedge (x \leq b)$.

- O operador *ou lógico* (`||`) constitui uma relação onde a expressão inteira será verdadeira, se e somente se pelo menos um dos membros for verdadeiro;
- O operador *not lógico* (`!`) **precede uma expressão** qualquer, invertendo o seu valor-verdade.

1.5.8. Ternário (`?:`)

O operador *ternário* é um operador especial, conhecido por ser um operador *triádico*, ou seja, que opera em *três* valores ao mesmo tempo.

Este operador pode ser utilizado para realizar rapidamente a verificação de *condições* no código.

O operador ternário segue a sintaxe

`condição ? consequente : alternativa`

onde

- `condição` é uma expressão booleana;
- `consequente` é o resultado da expressão ternária quando a `condição` é verdadeira;
- `alternativa` é o resultado da expressão ternária quando a `condição` é falsa.

Por exemplo, suponhamos o caso em que uma certa variável `resultado` precisa receber o maior valor entre `a` e `b`. Assim, podemos utilizar um operador ternário:

```
1 int a = 2;  
2 int b = 3;  
3  
4 int resultado = (a > b) ? a : b;
```

A expressão recebida por `resultado` poderia ser traduzida como o algoritmo a seguir, feito em pseudocódigo:

```
condição: se (a é maior que b)
então...
    consequência: retorne a
do contrário...
    alternativa: retorne b
```

Note que esta não é a forma mais simples de verificar por uma certa *condição*. A forma mais direta de realizar este tipo de verificação será abordada na Seção 1.9.

1.5.9. *Bitwise* (&, |, ~, ^, >>, <<)

Operações *bitwise* realizam manipulações diretamente nos *bits* de uma certa variável (por exemplo, variáveis do tipo `unsigned int`), e são equiparáveis a operações *lógicas*, todavia são, na verdade, operações *binárias*.

Estes operadores não serão comumente utilizados ao longo do livro, portanto, sua funcionalidade está aqui exposta apenas a título de curiosidade.

Operadores *bitwise* também possuem contrapartes atributivas, assim como os operadores de aritmética básica.

Suponhamos as seguintes variáveis:

```
1 unsigned int mask_1 = 1;
2 unsigned int mask_2 = 2;
```

A variável `mask_1` é uma variável de 16 bits que armazena o número 1, representado na base 10 (1_{10}). `mask_2`, por sua vez, representa o número 2_{10} .

Estes números poderão ser representados em uma base binária, se respeitarmos o tamanho padrão das variáveis `mask_1` e `mask_2` na

memória, como 0000000000000001_2 e 0000000000000010_2 , respectivamente.

Realizaremos operações *bitwise* nestas variáveis e, para melhor identificar o valor resultante de cada expressão, mostraremos suas representações binárias.

1. And (&)

O operador *and* (&) realiza uma operação "e" entre cada um dos bits de ambos os números.

```
1 unsigned int result_1 = mask_1 & mask_2;
```

$0000000000000001_2 \& 0000000000000010_2 \Rightarrow 0000000000000000_2$

2. Or (|)

O operador *or* (|) realiza uma operação "ou" entre cada um dos bits de ambos os números.

```
1 unsigned int result_2 = mask_1 | mask_2;
```

$0000000000000001_2 | 0000000000000010_2 \Rightarrow 0000000000000011_2$

3. Not (~)

O operador *not* (~) inverte toda a máscara de bits à qual se aplica, trocando 0's por 1's e vice-versa.

```
1 unsigned int result_3 = ~mask_1;
```

$\sim 0000000000000001_2 \Rightarrow 1111111111111110_2$

4. Xor (^)

O operador *xor* (^) realiza uma operação de "ou-exclusivo" entre cada um dos bits de ambos os números.

```
1 unsigned int result_4 = mask_1 ^ mask_2;
```

$0000000000000001_2 \wedge 000000000000010_2 \Rightarrow 000000000000011_2$

5. Arithmetic shift (>> e <<)

Os operadores de *arithmetic shift* deslocam cada um dos bits de uma variável para a direção que indicam.

Estes operadores não devem ser confundidos com os operadores de entrada e saída de `std::cout` e `std::cin`, a serem discutidos na Seção 1.7.

O operador *shift left* (<<) desloca os bits de uma variável um número n de casas para a esquerda:

```
1 unsigned int result_5 = mask_1 << 2;
```

$0000000000000001_2 \ll 2_{10} \Rightarrow 000000000000100_2$

Já o operador *shift right* (>>) também realiza o deslocamento de bits em n casas, porém para a direita:

```
1 unsigned int result_6 = mask_2 >> 1;
```

$000000000000010_2 \gg 1_{10} \Rightarrow 000000000000001_2$

1.5.10. Comentários (//, /* e */)

Às vezes, é necessário colocar informações textuais no código que escrevemos. Algumas operações podem não ficar claras apenas através da leitura do código, o que nos obriga a *comentá-lo*.

Para tanto, podemos utilizar marcadores de comentário, como no bloco abaixo:

```
1 // Este é um comentário de uma linha
2 int a = 5; // Podemos colocar comentários na frente de código
3
4 /*
5     Este é um comentário multilinha.
6 */
```

Tokens de comentários não são operadores em si, mas são extremamente utilizados durante o processo de codificação.

1.6. PRECEDÊNCIA DE OPERADORES

Assim como na Matemática, os operadores de C++ possuem regras de precedência. Para evitar que expressões tomem efeitos indesejados, utilize parênteses sempre que necessário.

Abaixo, mostra-se uma lista de precedência de operadores, da maior para a menor precedência. Quanto maior a precedência, maior a chance de uma expressão para aquela operação ser executada antes de outras. Operadores em uma mesma precedência serão sempre interpretados na ordem em que foram escritos, da esquerda para a direita.

1. Incremento (++) e decremento (--) sufixos;
2. Incremento (++) e decremento (--) prefixos, *not* lógico (!) e *not bitwise* (-);
3. Multiplicação (*), divisão (/), resto (%);
4. Adição (+) e subtração (-);
5. *Arithmetic shift* (>> e <<);
6. Comparação relacional (>, <, <=, >=);

7. Comparação relacional (`==` e `!=`);
8. *And bitwise* (`&`);
9. *Xor bitwise* (`^`);
10. *Or bitwise* (`|`);
11. *And lógico* (`&&`);
12. *Or lógico* (`||`);
13. Condicional ternário (`?:`), aritmética atributiva (todos).

1.6.1. Sobre agrupamento de expressões

Caso uma expressão seja escrita sem um agrupamento explícito usando parênteses, a regra de agrupamento de C++ determina que os grupos sejam sempre realizados *da direita para a esquerda*. Por exemplo, no seguinte código:

```
1 int a = 5;
2 int b = 6;
3
4 int resultado = (a > b) ? a : b + 1;
```

Podemos identificar uma soma ao final, portanto, o termo **alternativo** do ternário é `b + 1`.

A última expressão do código pode ser reescrita desta forma, sem alteração no valor semântico:

```
1 int resultado = (a > b) ? a : (b + 1);
```

1.7. ENTRADA E SAÍDA (I/O)

Escrever expressões aritméticas e armazenar seus resultados em variáveis certamente é útil, mas nenhuma destas operações envolve mostrar algo na tela, ou recuperar alguma informação digitada pelo usuário.

Para realizar estas operações, utilizaremos os comandos `std::cout` e `std::cin` para *imprimir* e *ler* informações no console, respectivamente.

Considere o programa a seguir, que incrementa um número informado pelo usuário.

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int numero;
7
8     cout << "Digite um numero inteiro:" << endl;
9     cin >> numero;
10
11     numero++;
12
13     cout << "O sucessor do numero eh " << numero << endl;
14     return 0;
15 }
```

Este programa recebe como entrada um número digitado pelo usuário no console, e mostra o sucessor de tal número na tela.

```
void% g++ -Wall -g sucessor.cpp -o sucessor
void% ./sucessor
Digite um numero inteiro:
5
O sucessor do numero eh 6
void% █
```

Figura 23 – Compilação e execução do programa anterior em um console do Linux

Assim como fizemos com o programa na Seção 1.3.1, destrincharemos este novo programa, mas vamos nos conter apenas às linhas dentro da *função principal*.

```
1 int numero;
```

Esta linha declara uma variável `numero` de tipo `int`. Esta variável armazenará, em breve, o valor digitado pelo usuário. Por enquanto, seu valor é indefinido.

```
1 cout << "Digite um numero inteiro:" << endl;
```

A próxima linha utiliza o comando `cout` para mostrar a cadeia de caracteres `"Digite um numero inteiro:"` na tela. Note que, imediatamente, uma linha é pulada no console, usando o comando `endl`.

```
1 cin >> numero;
```

Mostrado pela primeira vez, o comando `cin` realiza o trabalho implícito de *ler* o que foi digitado na tela pelo usuário, e então armazenar a informação apropriadamente na variável `numero`.

A entrada da informação no console envolve a digitação da informação, seguida do pressionamento da tecla Enter. `cin` "detecta" o

pressionamento de tal tecla, e então armazena o conteúdo digitado na variável mencionada.

Note que `cin` espera, neste caso, um valor apropriado para armazenamento na variável `numero`, ou seja, um valor *inteiro*. Caso outro tipo de informação seja digitada, o programa terá um *erro de lógica*, que não encerrará o programa imediatamente, mas poderá causar problemas inesperados.

```
1 numero++;
```

Este comando incrementa o valor inserido na variável `numero`.

```
1 cout << "O sucessor do numero eh " << numero << endl;
```

Este comando imprime na tela a cadeia de caracteres "O sucessor do numero eh ", seguida do novo valor da variável `numero`, após a operação de incremento; por fim, pula uma linha no *console*.

Note que há um espaço em branco ao final da cadeia de caracteres. Isto é proposital, para que o valor numérico de `numero` não apareça junto aos outros caracteres da frase.

```
1 return 0;
```

Encerra a *função principal* graciosamente, retornando um valor de sucesso.

1.8. EXERCÍCIOS DE FIXAÇÃO I

Somando números

Crie um programa que lê dois números reais digitados pelo usuário e imprime a soma deles.

Resto da divisão por dois

Crie um programa que lê um número inteiro digitado pelo usuário e imprime o resto de sua divisão por dois.

Área do círculo

Crie um programa que lê um número real digitado pelo usuário, simbolizando o raio de um certo círculo, e imprime a área do mesmo. A área de um círculo é representada pela fórmula

$$A_{\text{círculo}} = \pi r^2$$

Média aritmética

Crie um programa que lê três números reais digitados pelo usuário, calcula a média entre todos eles e imprime o resultado na tela.

Volume da esfera

Crie um programa que lê um número real digitado pelo usuário, simbolizando o raio de uma certa esfera, e imprime o volume da mesma. O volume de uma esfera é representado pela fórmula

$$V_{\text{esfera}} = \frac{4\pi r^3}{3}$$

1.9. CONTROLE DE FLUXO

As *estruturas de controle de fluxo* alteram o fluxo de execução de um programa

1.9.1. Estrutura condicional (if)

Uma *estrutura condicional* envolve a declaração de uma condição que determinará se outras condições serão executadas. Vejamos o código a seguir.

```
1  int a = 4;
2  if(a % 2 == 0) {
3      cout << "a eh par" << endl;
4  } else {
5      cout << "a eh impar" << endl;
6  }
```

O objetivo do código é *verificar se um número é par*, onde a paridade de um número é determinada quando verificamos se o resto de sua divisão inteira por 2 é igual a 0.

Nesta situação, temos uma clara mudança no fluxo da aplicação: a informação "a eh par" só será impressa se **a** for um valor par. Caso contrário, a informação "a eh impar" será impressa na tela.

Para tanto, utilizamos uma *condicional*, que normalmente segue a seguinte sintaxe:

```
if(predicado) {
    // Insira aqui o que fazer no caso verdadeiro
} else {
    // Insira aqui o que fazer no caso falso
}
```

Uma condicional não precisa verificar apenas um predicado. Podemos encadear vários predicados e uma alternativa usando **else**

if:

```
if(predicado1) {
    // Caso 1
```

```
} else if(predicado2) {  
    // Caso 2  
} else if(predicado3) {  
    // Caso 3  
} else {  
    // Alternativa  
}
```

O exemplo a seguir realiza uma verificação relacionada ao sinal de `a`. Para tanto, utilizamos uma comparação de várias cláusulas.

```
1  if(a < 0) {  
2      cout << "a eh negativo" << endl;  
3  } else if(a == 0) {  
4      cout << "a eh igual a zero" << endl;  
5  } else {  
6      cout << "a eh positivo" << endl;  
7  }
```

Não precisamos necessariamente utilizar `else if` e `else`. Podemos criar uma condicional que só executa algo mediante a validade de um predicado.

```
1  int a = 6;  
2  cout << "a eh igual a " << a << endl;  
3  
4  if(a % 2 == 0) {  
5      cout << "a tambem eh par" << endl;  
6  }
```

1.9.2. Estrutura de seleção (switch)

Podem ocorrer situações onde o controle de fluxo está ligado ao valor de uma variável em específico. Por exemplo, vejamos o programa a seguir.

```
1  #include <iostream>
2  using namespace std;
3
4  int
5  main()
6  {
7      int a, b, opcao;
8
9      cout << "Insira o numero a: ";
10     cin >> a;
11
12     cout << "Insira o numero b: ";
13     cin >> b;
14
15     cout << "Insira uma das seguintes opcoes:" << endl
16         << "1. Somar os dois numeros" << endl
17         << "2. Multiplicar os dois numeros" << endl
18         << "3. Calcular a media dos numeros" << endl
19         << "0. Sair" << endl
20         << "Sua opcao: ";
21
22     cin >> opcao;
23
24     if(opcao == 1) {
25         cout << "A soma eh " << (a + b) << endl;
26     } else if(opcao == 2) {
27         cout << "A multiplicacao eh " << (a * b) << endl;
```

```
28     } else if(opcao == 3) {
29         cout << "A media eh " << (a + b) / 2.0f << endl;
30     }
31
32     return 0;
33 }
```

Como você pode ter percebido, o programa pode ficar repetitivo no que tange à comparação dos valores da variável `opcao`. Para mitigar este problema, podemos substituir nossa análise de casos por uma estrutura de `switch`, desta forma:

```
1  switch(opcao) {
2      case 1:
3          cout << "A soma eh " << (a + b) << endl;
4          break;
5      case 2:
6          cout << "A multiplicacao eh " << (a * b) << endl;
7          break;
8      case 3:
9          cout << "A media eh " << (a + b) / 2.0f << endl;
10         break;
11     default:
12         break;
13 }
```

O objetivo do `switch` é realizar comparações sucessivas para a variável informada. Cada `case` apresenta um valor possível para a variável em questão e, caso o valor da variável seja aquele, então o bloco do `case` será executado.

O bloco de um *caso* arbitrário inicia-se ao final dos dois pontos, até a palavra-chave `break`. **Esta palavra-chave pode ser omi-**

tida¹², portanto tome sempre o cuidado de utilizá-la.

O caso simbolizado pela palavra `default` refere-se a todos os outros casos que não foram abordados, funcionando como uma espécie de `else` no `switch`.

1.10. EXERCÍCIOS DE FIXAÇÃO II

Par ou ímpar

Crie um programa que lê um número inteiro digitado pelo usuário, e informa se ele é *par* ou *ímpar*.

Boletim virtual I

Crie um programa que lê um número real representando a nota atual de um aluno em uma disciplina, variando de zero a cem. O programa deverá informar se o aluno foi *aprovado* ou *reprovado*. Caso a nota seja inválida, o programa deverá apresentar uma mensagem de erro.

Dias da semana

Crie um programa que recebe um número de 1 a 12 inclusive, e escreve na tela o nome do mês correspondente. Por exemplo, 1 equivale a "`janeiro`", e 12 equivale a "`dezembro`". Caso o número mês não faça sentido, imprima uma mensagem de erro na tela.

Distância entre dois pontos

Leia quatro valores reais, correspondentes aos eixos x e y de dois pontos, respectivamente, e calcule a distância entre estes dois pontos.

12. Omitir a palavra-chave `break` após um caso faz com que a execução das instruções continue, de forma que o fluxo "cai" para o próximo caso. O nome deste processo é conhecido como *fallthrough*. Este comportamento pode ser desejado ou indesejado.

Lembre-se de que a fórmula para a distância entre dois pontos (x_1, y_1) e (x_2, y_2) é:

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

- **DICA:** Você precisará calcular a raiz quadrada de um número real. Para tanto, utilize a biblioteca `<cmath>`, para ter acesso às funções `sqrtf` (para `float`) ou `sqrt` (para `double`). Por exemplo:

```
1 #include <cmath>
2 ...
3 float valor      = 25.0f;
4 float resultado = sqrtf(valor);
```

Conversão de tempo

Leia um número inteiro, correspondente a um certo valor em segundos, e imprima na tela o tempo no formato `h:m:s`.

Por exemplo, para a entrada `4800`, imprima `1:20:0`.

Equações de segundo grau

Uma equação de segundo grau com a forma

$$ax^2 + bx + c = 0$$

pode ter suas raízes calculadas pela fórmula de Bháskara:

$$x = \frac{-b \pm \sqrt{\Delta}}{2a}$$

onde

$$\Delta = b^2 - 4ac$$

lembrando que, se $\Delta < 0$, não existem raízes reais para a equação (ou seja, $\nexists x \in \mathbb{R}$).

Sendo assim, crie um programa que:

- Recebe valores para a , b e c ;
- Calcula o valor de Δ ;
- Imprime um erro se a equação não tiver raízes em \mathbb{R} , ou imprime valores para as duas raízes (mesmo que sejam iguais).

2. BIBLIOTECAS

Ainda que a programação de computadores possua diversas formas de abordar a computação de algum resultado esperado, a construção de todo e qualquer *software* para tal perpassa as instruções simples que foram apresentadas no capítulo anterior.

Todavia, é interessante garantir o *reuso* de *software* já programado, sobretudo quando precisamos desenvolver um *software* com um prazo já pré-estabelecido.

Um dos recursos utilizados no *reuso* de *software* é as *bibliotecas*. Bibliotecas são coletâneas de procedimentos, variáveis e outros elementos da linguagem que foram pré-programados e podem ser reutilizados em diversas aplicações. Estas bibliotecas podem ser padrão da linguagem, ou desenvolvidas por terceiros.

A linguagem C++ possui uma biblioteca-padrão extremamente vasta, e com muitas funcionalidades. Neste capítulo, vamos dar abordar algumas destas funcionalidades, principalmente a título de curiosidade e para consulta futura.

Não é essencial que absolutamente tudo neste capítulo seja compreendido neste momento, mas talvez seja pertinente consultá-lo em outras ocasiões.

2.1. DIFERENÇAS ENTRE C E C++

C++ é uma linguagem dita *retrocompatível* com C. Desenvolvida por Bjarne Stroustrup como uma camada sobre a linguagem C, esta característica deu a C++ a garantia de que código C pudesse ser reutilizado na mesma, demandando pouco trabalho para tal.

Devido a esta facilidade, um erro muito comum na programação é *assumir que C e C++ sejam linguagens iguais*, quando a verdade é que estas linguagens podem ser muito diferentes em alguns aspectos.

Um dos exemplos que demonstram isso é o próprio uso de bibliotecas de C em C++. Enquanto C demandaria o uso de um cabeçalho como `<stdio.h>`, em C++, recomenda-se substituir tal cabeçalho por `<cstdio>`, por exemplo.

Nas próximas seções, demonstraremos bibliotecas de C e C++ que podem ser utilizadas na programação em C++. No caso das bibliotecas de C, **lembre-se de remover o sufixo .h e prefixar o nome da biblioteca com c** quando for utilizá-la.

É importante lembrar que apenas funções e estruturas de maior relevância foram demonstradas a seguir, portanto é essencial que o leitor busque mais sobre cada biblioteca, segundo a necessidade.

Mais informações sobre tais bibliotecas podem ser encontradas nos websites <http://cplusplus.com> e <https://cppreference.com>, bem como nas páginas de manuais do sistema Linux.

2.2. BIBLIOTECAS DA LINGUAGEM C

As bibliotecas referenciadas a seguir vêm primariamente da linguagem C, sendo suportadas em C++ por questões de compatibilidade. Ainda assim, elas provêem ferramentas extremamente úteis para muitos aspectos.

É interessante notar que nem todas as funções das bibliotecas a seguir foram expostas. Será necessário o leitor procurar por mais

informações sobre elas.

2.2.1. `cstdio`

```
1 #include <cstdio>
```

Esta é a biblioteca de C para realizar operações de entrada e saída. Opera de forma similar a `iostream`, mas provê algumas ferramentas consideradas mais sucintas.

1. `putchar`

`int putchar(int character);` imprime um único caractere no console.

Caso haja algum erro na impressão, retorna o valor associado à constante `EOF`¹ e define um indicador de erro.

```
1 char c = '\n'; // Caractere de nova linha
2 putchar(c);
```

2. `puts`

`int puts(const char *str);` imprime uma cadeia de caracteres no console, e então pula automaticamente uma linha. Útil quando só há informação textual a ser impressa na tela.

Caso haja algum erro na impressão, retorna o valor associado à constante `EOF` e define um *indicador de erro*.

```
1 puts("Hello world!");
```

```
Hello world!
```

1. Também conhecido como o caractere de *fim-de-arquivo*.

3. `getchar`

`int getchar()`; lê um único caractere digitado no console. Em caso de sucesso, o caractere em questão é retornado. Em caso de erro, a função retorna `EOF`.

Quando a função entra em um estado de erro, pode ser que a *entrada do console* tenha sido interrompida bruscamente. Neste caso, a função define um *indicador de fim-de-arquivo*. Caso contrário, ela define um *indicador de erro*.

```
1 char c;  
2 c = getchar();
```

4. `fgets`

`char *fgets(char *str, int num, FILE *stream)`; lê uma cadeia de caracteres completa (incluindo espaços) a partir de um arquivo `stream` e armazena-a em um *vetor*² chamado `str`, de tamanho máximo `num`.

Em caso de sucesso, a função retorna o próprio *ponteiro*³ para a cadeia de caracteres `str`. Em caso de erro, a função retorna um *ponteiro nulo* (normalmente igual ao valor associado à constante `NULL`) e define um *indicador de fim-de-arquivo*, caso o arquivo `stream` esteja ao seu final, ou um *indicador de erro*, em outras situações.

É importante notar que, durante este curso, não abordaremos a leitura direta de *arquivos*; todavia, a *entrada* e a *saída* do

2. A ser discutido apropriadamente no próximo capítulo. Por enquanto, considere um *vetor* como sendo espaços de memória contínuos do mesmo tipo – neste caso, caracteres em conjunto na memória.

3. Não abordaremos ponteiros neste curso. Basta saber por enquanto que trata-se de um endereço de memória: assim como vamos ao endereço da casa de alguém para encontrar tal pessoa, variáveis também possuem endereços na memória.

console são tratadas em C e C++ como sendo tipos de *pseudo-arquivos*, representados pelas variáveis globais `stdin` e `stdout`, respectivamente.

```
1 int nome[80]; // Capaz de abrigar 80 caracteres
2 fgets(nome, 80, stdin); // Lê uma cadeia de caracteres
```

- **NOTA:** Algumas literaturas recomendam o uso da função `gets` para obter diretamente caracteres da linha de comando. **Jamais use esta função**⁴. Esta função foi removida de especificações mais novas das linguagens C e C++ por representarem um perigo de erro de *buffer overflow*.

Explicando rapidamente, imagine ter um espaço reservado para a digitação de oitenta caracteres, e seu usuário digitar cem; inevitavelmente, os vinte caracteres extras seriam escritos em *memória ilegal*, além da capacidade máxima de caracteres informada.

A função `fgets` não possui tal problema, por requisitar a quantidade de caracteres já no seu uso.

5. printf

`int printf(const char *format, ...)`; é uma das funções mais famosas de C, junto com `scanf`. Esta função é utilizada para imprimir *informação formatada* no console.

A cadeia de caracteres `format` é composta por texto comum e algumas *flags* de formatação, que utilizam outros argumentos passados após o formato, sequencialmente.

4. A maioria dos compiladores modernos já não permite a compilação de programas que usam este programa. Porém, se você estiver utilizando um compilador como GCC abaixo da versão 5, você ainda conseguirá utilizá-la.

Abaixo, temos um exemplo do uso de `printf` com alguns exemplos de formato. O código é seguido de sua saída no console, caso seja colocado em uma função *main*. Maiores explicações sobre os formatos podem ser consultadas em <<http://www.cplusplus.com/reference/cstdio/printf/>>.

```
1  int num        = 5;
2  float decimal = 5.25f;
3  double ddec   = 9.74;
4  char caractere = 'z';
5
6  puts("Teste de printf");
7  printf("Inteiro: %d\nFloat: %f\nDouble: %lf\n",
8         num, decimal, ddec);
9  printf("Decimal com uma casa: %0.1f\n", decimal);
10 printf("Decimal com repr. menor: %g\n", decimal);
11 printf("Bem-vindo, %s. Seu caractere: %c\n",
12        "Fulano", caractere);
```

```
Teste de printf
Inteiro: 5
Float: 5.250000
Double: 9.740000
Decimal com uma casa: 5.2
Decimal com repr. menor: 5.25
Bem-vindo, Fulano. Seu caractere: z
```

6. scanf

`int scanf(const char *format, ...)`; é outra função famosa de C, ao lado de `printf`. Esta função é utilizada para *ler* informação do console.

A natureza da informação a ser lida é dada pelo formato utilizado, e as variáveis a receberem a informação devem ser repassadas como *referências* (ou *ponteiros*) para a função `scanf`.

A função `scanf` também pode ser utilizada para

O *formato* de `scanf` segue vagamente o formato de `printf`. Para maiores explicações do formato de `scanf`, consulte <<http://www.cplusplus.com/reference/cstdio/scanf/>>.

```
1  int numero;
2  float decimal;
3  char caractere;
4  char primeiro_nome[80];
5
6  // Lendo variáveis simples
7  scanf("%d %f %c", &numero, &decimal, &caractere);
8
9  // Lendo uma cadeia de caracteres (sem espaços)
10 // "primeiro_nome" já é um ponteiro de memória
11 scanf("%80s", primeiro_nome);
```

2.2.2. `cmath`

```
1 #include <cmath>
```

Esta biblioteca declara conjuntos de funções para computar operações matemáticas e transformações, e também constantes com valores essenciais para alguns cálculos matemáticos.

1. Constantes

Os valores abaixo constituem constantes matemáticas importantes. Estes valores podem não estar disponíveis em todos os

compiladores de C++, portanto apresentamos também seus valores como utilizados no compilador GCC.

- a) `M_E`
Constante de Euler.
 $e \approx 2.7182818284590452354$
- b) `M_LOG2E`
Logaritmo da constante de Euler na base 2.
 $\log_2 e \approx 1.4426950408889634074$
- c) `M_LOG10E`
Logaritmo da constante de Euler na base 10.
 $\log_{10} e \approx 0.43429448190325182765$
- d) `M_LN2`
Logaritmo de 2 na base da constante de Euler
 $\log_e 2 = \ln 2 \approx 0.69314718055994530942$
- e) `M_LN10`
Logaritmo de 10 na base da constante de Euler
 $\log_e 10 = \ln 10 \approx 2.30258509299404568402$
- f) `M_PI`
Constante pi.
 $\pi \approx 3.14159265358979323846$
- g) `M_PI_2`
Constante pi dividido por 2.
 $\frac{\pi}{2} \approx 1.57079632679489661923$
- h) `M_PI_4`
Constante pi dividido por 4.
 $\frac{\pi}{4} \approx 0.78539816339744830962$
- i) `M_1_PI`
Recíproco de pi.
 $\frac{1}{\pi} \approx 0.31830988618379067154$

- j) `M_2_PI`
2 dividido por pi.
 $\frac{2}{\pi} \approx 0.63661977236758134308$
- k) `M_2_SQRTPI`
2 dividido pela raiz quadrada de pi.
 $\frac{2}{\sqrt{\pi}} \approx 1.12837916709551257390$
- l) `M_SQRT2`
Raiz quadrada de 2.
 $\sqrt{2} \approx 1.41421356237309504880$
- m) `M_SQRT1_2`
Recíproco da raiz quadrada de 2.
 $\frac{1}{\sqrt{2}} \approx 0.70710678118654752440$

2. `abs`

A função `abs(x)` recebe um valor `x` qualquer e retorna o valor absoluto deste número. Esta função pode ser utilizada para múltiplos tipos de variáveis.

```
1  int val = -2;  
2  
3  int valor_absoluto = abs(val);  
4  
5  std::cout << "abs(" << val << ") = "  
6      << valor_absoluto << std::endl;
```

`abs(-2) = 2`

3. `cos`, `sin`, `tan`

As funções `double cos(double x)`, `double sin(double x)` e `double tan(double x)` computam o *coosseno*, *seno* e *tangente* de um ângulo `x`, dado em *radianos*.

Um *radiano* é equivalente a $\frac{180}{\pi}$.

```

1  double angle = 60.0; // Ângulo em graus
2
3  // Conversão para radianos
4  angle *= M_PI / 180.0;
5
6  double seno      = sin(angle);
7  double cosseno   = cos(angle);
8  double tangente  = tan(angle);
9
10 std::cout << "Para o angulo " << angle
11           << ":" << std::endl;
12 std::cout << "Seno:      " << seno      << std::endl
13           << "Cosseno:  " << cosseno   << std::endl
14           << "Tangente: " << tangente  << std::endl;

```

```

Para o angulo 1.0472:
Seno:      0.866025
Cosseno:   0.5
Tangente:  1.73205

```

4. `acos`, `asin`, `atan`

As funções `double acos(double x)`, `double asin(double x)` e `double atan(double x)` computam o *arc cosseno*, *arc seno* e *arc tangente* de um valor x .

Estas operações são as inversas de *cosseno*, *seno* e *tangente*, sendo também conhecidas em Português como *secante*, *cossecante* e *cotangente*, respectivamente.

Estas operações funcionam de maneira "inversa" às trigonométricas originais. Por exemplo, ao computarmos `asin` de x ,

estamos procurando qual ângulo (em radianos) possui o *seno* no valor igual a x .

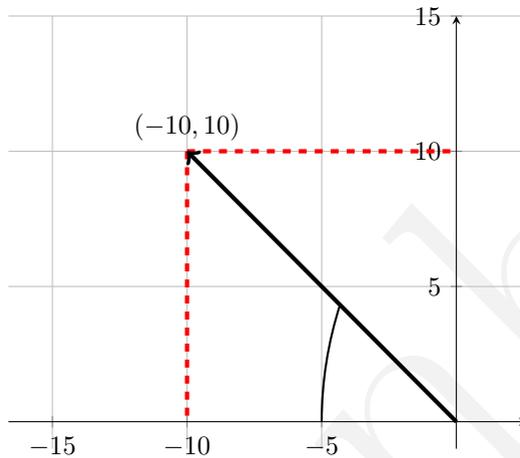
```
1  double cossec = asin(0.5);
2  double sec    = acos(0.5);
3  double cotan  = atan(1.0);
4
5  std::cout << "Cossecante de 0.5: "
6             << cossec << std::endl
7             << "Secante de 0.5:   "
8             << sec    << std::endl
9             << "Cotangente de 1:  "
10            << cotan  << std::endl;
```

```
Cossecante de 0.5: 0.523599
Secante de 0.5:   1.0472
Cotangente de 1:  0.785398
```

5. atan2

A função `double atan2(double y, double x)` deduz o ângulo α cuja cotangente seja o resultado da divisão x/y .

Suponhamos x com o valor -10 e y com o valor 10 . O par ordenado (x, y) descreve um vetor partindo da *origem*, como demonstrado a seguir.



Para o par ordenado apresentado, temos $\cotg(-10 \div 10) = \cotg(-1) \approx -0.785398\text{rad} \cong -45^\circ$.

Todavia, `atan2` realiza esta conversão de forma que o ângulo esteja inserido no intervalo $[-\pi, \pi]$, o que nos dá $-45^\circ \cong 135^\circ$.

```

1  double x = -10.0, y = 10.0;
2
3  double angle = atan2(y, x);
4
5  // Conversão para graus
6  angle *= 180.0 / M_PI;
7
8  std::cout << "A cotangente para a coordenada ("
9             << x << ", " << y << ") eh "
10            << angle << "°."
11            << std::endl;

```

A cotangente para a coordenada (-10, 10) eh 135° .

6. pow

`double pow(double base, double exponent)` eleva um número real `base` a um expoente `exponent`.

```
1 double number = 2.0;
2 double cube_of_two = pow(number, 3.0);
3
4 std::cout << number << "^3 = " << cube_of_two << std::endl;
```

$2^3 = 8$

7. exp

`double exp(double x)` computa o *valor exponencial* para `x`, ou seja, o valor do número de Euler elevado a `x` (e^x).

```
1 double param = 5.0;
2 double e_to_the_fifth = exp(param);
3
4 std::cout << "e^" << param << " = "
5           << e_to_the_fifth << std::endl;
```

$e^5 = 148.413$

8. log, log10

`double log(double x)` e `double log10(double x)` calculam o valor do *logaritmo* para `x`.

`log` calculará o valor tendo o número de Euler como base ($\ln(x) = \log_e(x)$), enquanto `log10` calculará um valor de logaritmo comum ($\log(x) = \log_{10}(x)$).

```

1  double natural_log = log(5.5);
2  double common_log  = log10(1000.0);
3
4  std::cout << "ln 5.5 = "   << natural_log << std::endl
5  << "log 1000 = " << common_log << std::endl;

```

ln 5.5 = 1.70475

log 1000 = 3

9. cosh, sinh, tanh

As funções `double cosh(double x)`, `double sinh(double x)` e `double tanh(double x)` calculam o *coosseno hiperbólico*, *seno hiperbólico* e *tangente hiperbólica* do ângulo hiperbólico x .

```

1  // logaritmo natural de 2
2  double param = log(2.0);
3
4  double cos_hip = cosh(param);
5  double sen_hip = sinh(param);
6  double tan_hip = tanh(param);
7
8  std::cout << "Para o angulo hiperbolico " << param
9  << ":" << std::endl;
10 std::cout << "Seno hip.:      " << sen_hip << std::endl
11 << "Cosseno hip.:  " << cos_hip << std::endl
12 << "Tangente hip.: " << tan_hip << std::endl;

```

Para o angulo hiperbolico 0.693147:

Seno hip.: 0.75

Cosseno hip.: 1.25

Tangente hip.: 0.6

10. `sqrt`, `sqrtf`

As funções `double sqrt(double x)` e `float sqrtf(float x)` calculam a raiz quadrada de um número real `x`.

```
1  double value = 1024.0;
2  float valuef = 25.0f;
3
4  double sqroot = sqrt(value);
5  float sqrootf = sqrtf(valuef);
6
7  std::cout << "A raiz quadrada de " << value
8             << " eh " << sqroot << std::endl;
9  std::cout << "A raiz quadrada de " << valuef
10            << " eh " << sqrootf << std::endl;
```

A raiz quadrada de 1024 eh 32

A raiz quadrada de 25 eh 5

11. `ceil`, `floor`

As funções `double ceil(double x)` e `double floor(double x)` calculam o *teto* e o *chão* de um número real `x`, respectivamente.

O *teto* de um número real `x` é o primeiro número inteiro que seja *maior* que `x`.

O *chão* de um número real `x` é o primeiro número inteiro que seja *menor* que `x`.

```
1  double num = 2.4;
2
3  double teto = ceil(num);
4  double chao = floor(num);
```

```
5
6 std::cout << "O teto e o chao de " << num
7     << " sao " << teto << " e "
8     << chao << std::endl;
```

O teto e o chao de 2.4 sao 3 e 2

Estas funções também podem ser utilizadas para realizar arredondamentos, como podemos ver no exemplo a seguir.

12. fmod

A função `double fmod(double numer, double denom)` age de forma similar ao operador *mod* (%), porém é capaz de retornar o resto da divisão inteira baseada em um *ponto flutuante*.

```
1 double res1 = fmod(5.3, 2.0);
2 double res2 = fmod(18.5, 4.2);
3
4 std::cout << "5.3 fmod 2 = "
5     << res1 << std::endl
6     << "18.5 fmod 4.2 = "
7     << res2 << std::endl;
```

5.3 fmod 2 = 1.3

18.5 fmod 4.2 = 1.7

2.2.3. ctime

```
1 #include <ctime>
```

Esta biblioteca contém definições de funções e estruturas para obter e manipular dados e informação relacionada a tempo.

1. `time_t`

Trata-se de uma *estrutura de dados*⁵ capaz de armazenar dados de um momento qualquer, geralmente relacionadas a *timestamps* dos sistema operacional Unix.

Por definição, esta estrutura armazena o tempo que passou, *em segundos*, desde as 0h de 1 de janeiro de 1970, no fuso-horário universal de Greenwich.

Esta estrutura não deve ser manipulada diretamente.

2. `struct tm`

Esta estrutura de dados é capaz de armazenar dados de um momento específico no tempo. Ela desdobra-se em membros, sendo eles elementos primitivos, como demonstrado na Tabela 1.

Use esta estrutura para representar momentos no tempo.

Tabela 1 – Membros da estrutura `struct tm`.

Membro	Tipo	Significado
<code>tm_sec</code>	<code>int</code>	Segundos após o minuto
<code>tm_min</code>	<code>int</code>	Minutos após a hora
<code>tm_hour</code>	<code>int</code>	Horas desde a meia-noite
<code>tm_mday</code>	<code>int</code>	Dia do mês
<code>tm_mon</code>	<code>int</code>	Meses desde janeiro
<code>tm_year</code>	<code>int</code>	Anos desde 1900
<code>tm_wday</code>	<code>int</code>	Dias desde domingo
<code>tm_yday</code>	<code>int</code>	Dias desde 1 de janeiro
<code>tm_isdst</code>	<code>int</code>	Indicador de horário de verão

5. `time_t` é um *tipo abstrato de dados* (TAD). Trata-se de uma estrutura que armazena um ou mais dados *primitivos* ou igualmente *abstratos*. Por enquanto, basta sabe que esta estrutura armazena, de forma pertinente e classificada, os dados referidos pela especificação da mesma.

```

1 // `tempo` representa a data:
2 // domingo, 1 de janeiro de 1900, 0h00m, UTC.
3 struct tm tempo = {0};
4 std::cout << 1 + tempo.tm_mday << "/"
5           << 1 + tempo.tm_mon << "/"
6           << 1900 + tempo.tm_year << ", "
7           << tempo.tm_hour << "h"
8           << tempo.tm_min << "m"
9           << tempo.tm_sec << "s" << std::endl;

```

1/1/1900, 0h0m0s

3. mktime

`time_t mktime(struct tm *timeptr)` converte uma estrutura `struct tm` para uma estrutura `time_t`.

```

1 struct tm y2k = {0};
2 time_t tempo = mktime(&y2k);

```

4. difftime

`double difftime(time_t end, time_t beginning)` retorna a diferença de tempo, em segundos, entre o tempo inicial `beginning` e o tempo final `end`.

```

1 struct tm time1 = {0};
2 struct tm time2 = {0};
3
4 // Uma hora de diferença
5 time2.tm_hour = 1;
6

```

```
7  time_t tempo1 = mktime(&time1);
8  time_t tempo2 = mktime(&time2);
9
10 double diff = difftime(tempo2, tempo1);
11
12 std::cout << "A diferenca eh de " << diff
13           << " segundos." << std::endl;
```

A diferenca eh de 3600 segundos.

5. time

`time_t time(time_t *timer)` retorna o tempo atual de acordo com o calendário. O parâmetro `timer` normalmente é associado a `NULL`, sendo ele a estrutura onde os dados do tempo atual serão opcionalmente armazenados.

A função retorna uma estrutura `time_t`.

```
1  time_t timer;
2  struct tm y2k = {0}; // 01/01/1900, 0h0m0s
3  double segundos;
4
5  // `y2k` passa a ser 01/01/2000, 0h0m0s
6  y2k.tm_hour = 0;
7  y2k.tm_year = 100;
8  y2k.tm_mday = 1;
9
10 time(&timer);
11 segundos = difftime(timer, mktime(&y2k));
12
13 std::cout << segundos << " segundos desde "
14           << "01/01/2000 0h0m0s." << std::endl;
```

6.26974e+08 segundos desde 01/01/2000 0h0m0s.

6. ctime

`char *ctime(const time_t *timer)` converte uma estrutura `time_t` para uma cadeia de caracteres *constante*, no formato "`Www Mmm dd hh:mm:ss yyyy\n`", onde:

- `Www` é o dia da semana;
- `Mmm` é o mês;
- `dd` é o dia do mês;
- `hh:mm:ss` é a hora exata;
- `yyyy` é o ano.

```
1 // `y2k` passa a ser 01/01/2000, 0h0m0s
2 time_t hora_atual = time(NULL);
3
4 std::cout << "A hora atual eh "
5           << ctime(&hora_atual) << std::endl;
```

A hora atual eh Wed Nov 13 15:31:54 2019

2.2.4. cstdlib

```
1 #include <cstdlib>
```

Esta biblioteca define várias funções de propósito geral, o que inclui gerenciamento dinâmico de memória, geração de números randômicos, aritmética com inteiros, procura, ordenação, conversões, etc.

1. `srand`, `rand`

`void srand(unsigned int seed)` e `int rand()` são responsáveis, respectivamente, pela inicialização de um gerador de números *pseudo-aleatórios* e pela produção dos mesmos.

O valor retornado por `rand` é sempre um número entre 0 e `RAND_MAX`. Para garantir a geração de um valor entre 0 e um certo limite, use o operador *mod* (%).

```
1  srand(time(NULL));
2
3  int valor1 = rand() % 100;
4  int valor2 = rand() % 100 + 1;
5
6  std::cout << "Valor no interv. [0, 99]: "
7             << valor1 << std::endl
8             << "Valor no interv. [1, 100]: "
9             << valor2 << std::endl;
```

```
Valor no interv. [0, 99]: 36
```

```
Valor no interv. [1, 100]: 72
```

2. `malloc`, `calloc`, `realloc`, `free`

As funções `malloc`, `calloc` e `realloc` são responsáveis por gerenciamento de memória dinâmica⁶:

- `malloc` aloca uma certa região de memória dinâmica sem inicialização;
- `calloc` aloca uma certa região de memória dinâmica, inicializada com *zeros*;

6. Neste livro, não abordaremos o uso de alocação de memória dinâmica. Por isso, o funcionamento destas funções não será detalhado.

- `realloc` toma um *ponteiro*⁷ para uma região de memória já alocada, e tenta ampliá-la para o novo tamanho informado.

Uma região de memória alocada pode ser liberada através da função `free`. É essencial que memória dinâmica seja liberada após o uso.

```
1  int *var_ptr;
2  int *vetor;
3  int *vetor_zero;
4
5  // Cria uma variável dinamicamente
6  var_ptr = malloc(sizeof (int));
7  // Cria um vetor de 5 elementos dinamicamente,
8  // contendo lixo de memória
9  vetor = malloc(5 * sizeof (int));
10 // Cria um vetor de 5 elementos dinamicamente,
11 // contendo zero em todas as posições
12 vetor_zero = calloc(5, sizeof (int));
13
14
15 // Aumenta o tamanho de `vetor` de 5 para 10
16 vetor = realloc(vetor, 10 * sizeof (int));
17
18
19 // Desalocando a memória
20 free(var_ptr);
21 free(vetor);
22 free(vetor_zero);
```

7. Um *ponteiro de memória* é, grosso modo, o *endereço* de uma região de memória (*vetor* ou *variável*) na memória RAM. Não abordaremos diretamente o uso deste tipo de informação neste livro.

3. `atof`, `atoi`

`double atof(const char *str)` e `int atoi(const char *str)` convertem uma cadeia de caracteres para uma variável `double` ou `int`, respectivamente.

Caso a primeira sequência de caracteres encontrada (excluindo espaços) não possua nenhum valor numérico válido, ou caso a cadeia de caracteres não possua nenhum valor numérico válido, nenhuma conversão será feita, e zero será retornado em ambos os casos.

```
1 double val1 = atof("13.5");
2 int    val2 = atoi("22");
3
4 std::cout << "Double: " << val1 << std::endl
5           << "Int:    " << val2 << std::endl;
```

```
Double: 13.5
```

```
Int:    22
```

4. `abort`, `exit`

`void abort()` aborta o programa, produzindo um término de programa anormal.

`void exit(int status)` termina o programa de forma normal.

Para a função `exit`, os valores de `status` podem ser, principalmente, `EXIT_SUCCESS` ou `EXIT_FAILURE`. Normalmente, associamos `EXIT_SUCCESS` ao valor 0 e `EXIT_FAILURE` ao valor 1.

- **NOTA:** Evite abortar a aplicação, a menos que saiba exatamente o que está fazendo.

5. system

`int system(const char *command)` executa um comando no console do sistema operacional. Os efeitos da execução do comando dependem do sistema operacional utilizado.

O retorno de `system` é o retorno do comando executado; caso o comando equivalha a uma aplicação em C/C++, este valor será o valor dado pelo retorno da função principal desta outra aplicação, por exemplo.

Caso `command` seja `NULL`, a função dirá se um processador de comandos está disponível para ser invocado em primeiro lugar.

```
1  if(system(NULL)) {
2      // A execução da função imprime a versão do kernel
3      // e o sistema operacional. Este comando só vale
4      // para Linux.
5      int retorno = system("uname -ro");
6
7      std::cout << "Saída do comando: " << retorno
8                  << std::endl;
9  }
```

5.3.8_2 GNU/Linux

Saída do comando: 0

- **NOTA:** É importante realizar o uso consciente da função `system`. Como esta função executa um comando arbitrário dentro da aplicação, é importante garantir que este comando seja seguro. Portanto, evite dar à função `system` um comando digitado diretamente pelo usuário, ou por alguém com outro tipo de acesso à aplicação (ex. em um servidor remoto).

Também é importante o uso moderado desta função: uma chamada para um comando arbitrário poderia executar alguma aplicação com código malicioso, por exemplo, se a aplicação sendo chamada pelo comando fosse trocada por um programa com *malware*, e com o mesmo nome.

2.2.5. `cassert`

```
1 #include <cassert>
```

Esta biblioteca define uma única função que pode ser utilizada como ferramenta de *debug*.

1. `assert`

`void assert(int expression)` verifica o valor-verdade passado em `expression`. Caso o valor seja *falso* (ou igual a zero), a aplicação é abortada **imediatamente**.

Toda e qualquer chamada a esta função **desaparecerá, caso a flag `NDEBUG` esteja definida no cabeçalho da aplicação**, de preferência antes da inclusão de `<cassert>`. Sendo assim, **não use `assert()` para aplicativos em produção**.

```
1 assert(2 == 3);
```

```
main.cpp:10: int main(): Assertion `2 == 3' failed.
```

2.2.6. `cstring`

```
1 #include <cstring>
```

Esta biblioteca implementa funções diversas para manipulação de *cadeias de caracteres* e *vetores*.

1. strcpy

`char *strcpy(char *destination, const char *source)` copia uma cadeia de caracteres `source` para dentro de um vetor de caracteres `destination`.

É muito importante garantir que `destination` tenha tamanho suficiente para armazenar os caracteres da cadeia e o caractere de terminador nulo ao final⁸.

```
1 char string[10];
2
3 // "Ola mundo" possui nove caracteres, e um
4 // caractere extra '\0' ao final
5 strcpy(string, "Ola mundo");
6
7 std::cout << string << std::endl;
```

Ola mundo

2. strcmp

`int strcmp(const char *str1, const char *str2)` compara as cadeias de caracteres `str1` e `str2`. Caso sejam iguais, retorna uma resposta **nula** (zero).

```
1 char resposta[255];
2 std::cout << "Voce esta bem?" << std::endl;
3 std::cin >> resposta;
4
5 if(!strcmp(resposta, "sim")) {
6     std::cout << "Que otimo!" << std::endl;
```

8. A forma como *strings* funcionam será melhor abordada no próximo capítulo.

```
7 } else {  
8     std::cout << "Que pena!" << std::endl;  
9 }
```

3. strchr

`const char *strchr(const char *str, int character)` procura pela primeira ocorrência de `character` na cadeia de caracteres `str`. A função retorna um *ponteiro* com a localização do caractere em questão. Se o caractere não for encontrado, retorna `NULL`.

```
1 const char *str = "Ola mundo!";  
2  
3 const char *ptr = strchr(str, 'm');  
4  
5 std::cout << "Localizacao de 'm' em \""  
6     << str << "\": ";  
7  
8 if(!ptr) {  
9     std::cout << "Indefinido";  
10 } else {  
11     std::cout << (int) (ptr - str + 1);  
12 }  
13 std::cout << std::endl;
```

Localizacao de 'm' em "Ola mundo!": 5

4. strstr

`const char *strstr(const char *str1, const char *str2)` procura pela primeira ocorrência da cadeia de caracteres `str2` na

cadeia de caracteres `str1`. A função retorna um *ponteiro* com a localização do primeiro caractere de `str2` em `str1`. Se a cadeia de caracteres não for encontrada, retorna `NULL`.

```
1  const char *str = "Ola mundo!";
2
3  const char *ptr = strstr(str, "und");
4
5  std::cout << "Localizacao de \"und\" em \""
6             << str << "\": ";
7
8  if(!ptr) {
9      std::cout << "Indefinido";
10 } else {
11     std::cout << (int) (ptr - str + 1);
12 }
13 std::cout << std::endl;
```

Localizacao de "und" em "Ola mundo!": 6

5. strlen

`size_t strlen(const char *str)` retorna o tamanho em quantidade de caracteres de uma cadeia de caracteres.

```
1  const char *str = "Ola mundo!";
2
3  int tamanho = strlen(str);
4
5  std::cout << "Tamanho de \"" << str << "\": "
6             << tamanho << std::endl;
```

Tamanho de "Ola mundo!": 10

6. strerror

char *strerror(int errnum) toma o valor de um erro errnum e retorna uma cadeia de caracteres que descreve o erro. Deve ser utilizado em conjunto com errno (vide a Seção .

```
1 FILE *fp;
2 fp = fopen("unexist.ent", "r");
3 if(!fp) {
4     std::cout << "Erro ao abrir arquivo: "
5               << strerror(errno) << std::endl;
6     return 1;
7 }
8 fclose(fp);
```

Erro ao abrir arquivo "unexist.ent": No such file or directory

2.2.7. cerrno

```
1 #include <cerrno>
```

Esta biblioteca define um único valor para o último número de erro emitido pela aplicação.

1. errno

Trata-se de uma variável int para um código de erro emitido pela aplicação. A mensagem de erro derivada do código pode ser obtida com a função strerror.

2.2.8. `climits`

```
1 #include <climits>
```

Esta biblioteca define valores constantes com os limites dos tipos de variáveis integrais da linguagem. Estes valores variam de acordo com a arquitetura e o compilador utilizado. É sempre melhor utilizar estas constantes, caso necessário.

As atribuições relacionadas a `float`, `double` e `long double` encontram-se na biblioteca `<cmath>`, que não será abordada neste material.

- `CHAR_BIT`: Número de bits em uma variável `char`;
- `CHAR_MIN`, `CHAR_MAX`: Valores mínimo e máximo numéricos armazenados por uma variável `char`;
- `SHRT_MIN`, `SHRT_MAX`: Valores mínimo e máximo numéricos armazenados por uma variável `short int`;
- `INT_MIN`, `INT_MAX`: Valores mínimo e máximo numéricos armazenados por uma variável `int` ou `signed int`;
- `LONG_MIN`, `LONG_MAX`: Valores mínimo e máximo numéricos armazenados por uma variável `long int`;
- `USHRT_MAX`, `UINT_MAX`, `ULONG_MAX`: Valores máximos armazenados por variáveis `unsigned short int`, `unsigned int` e `unsigned long int`, respectivamente.

2.2.9. `cctype`

```
1 #include <cctype>
```

Esta biblioteca declara conjuntos de funções para classificar e transformar *caracteres* individuais.

1. toupper, tolower

As funções `int toupper(int c)` e `int tolower(int c)` transformam caracteres individuais em *maiúsculas* e *minúsculas*, respectivamente.

```
1 char a = 'a', b = 'B';
2
3 std::cout << "Antes da transform.: "
4           << a << ' ' << b << std::endl;
5
6 a = toupper(a);
7 b = tolower(b);
8
9 std::cout << "Depois da transform.: "
10          << a << ' ' << b << std::endl;
```

Antes da transform.: a B

Depois da transform.: A b

2. isupper, islower

As funções `int isupper(int c)` e `int islower(int c)` verificam se caracteres individuais são letras *maiúsculas* ou *minúsculas*, respectivamente.

```
1 char a, b;
2
3 std::cout << "Insira dois caracteres: ";
4 std::cin >> a >> b;
5
6 std::cout << "O primeiro eh uma maiuscula? "
7           << (isupper(a) ? "Sim" : "Nao" << std::endl)
```

```
8  std::cout << "O segundo eh uma minuscula? "  
9  << (islower(b) ? "Sim" : "Nao" << std::endl);
```

3. isspace

A função `int isspace(int c)` verifica se um caractere individual é um *espaço em branco* (como um espaço comum, uma quebra de linha, um TAB horizontal, etc).

```
1  char a = ' ';  
2  char b = '\t';  
3  char c = '\n';  
4  
5  std::cout << "Os elementos sao espacos?" << std::endl  
6  << "a: " << (isspace(a) ? "Sim" : "Nao")  
7  << std::endl  
8  << "b: " << (isspace(b) ? "Sim" : "Nao")  
9  << std::endl  
10 << "c: " << (isspace(c) ? "Sim" : "Nao")  
11 << std::endl;
```

```
Os elementos sao espacos?  
a: Sim  
b: Sim  
c: Sim
```

2.3. BIBLIOTECAS DA LINGUAGEM C++

As bibliotecas a seguir são bibliotecas exclusivas para C++, não podendo ser utilizadas na linguagem C. Elas descrevem estruturas que fazem uso intensivo de elementos da linguagem como *templates* e

orientação a objetos, numa tentativa de generalização que culminou em um número gigantesco de utilidades.

A biblioteca padrão da linguagem C++ aumenta *com frequência*, sendo impossível de ver em sua completude com apenas uma literatura; um número gigantesco de recursos de C++ não será abordado neste livro.

Por isso, também é importante consultar sempre a especificação da linguagem.

É interessante notar que nem todas as funções das bibliotecas a seguir foram expostas. Será necessário o leitor procurar por mais informações sobre elas.

2.3.1. Sobre *namespaces*

Algumas das estruturas abaixo são prefixadas com `std::`. Isto indica que `std` seja o que chamamos de *namespace*, uma espécie de agrupamento para objetos e funções. O *namespace* `std` abriga estruturas padrão da linguagem C++.

O uso do prefixo `std::` pode ser omitido mediante a importação do *namespace* ao qual o objeto pertence:

```
1 // Importa tudo que for prefixado com `std`
2 using namespace std;
3
4 /* Restante do código... */
5
6 int a;
7 cin >> a;
8 cout << "Numero digitado: " << a << endl;
```

Também podemos importar apenas o objeto em questão:

```
1 using std::cin; // Importa apenas `cin`
2
3 /* Restante do código... */
4
5 int a;
6 cin >> a;
7 std::cout << "Numero digitado: " << a << std::endl;
```

2.3.2. iostream

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

Esta biblioteca define objetos-padrão de entrada e saída de informações.

1. cin

istream `std::cin` é um *objeto*⁹ que representa o *stream padrão de entrada* orientado a caracteres comuns, sendo correspondente ao objeto `stdin` da linguagem C.

```
1 int a;
2 std::cin >> a; // Lendo um numero do console
```

2. cout

ostream `std::cout` é um *objeto* que representa o *stream padrão de saída* orientado a caracteres comuns, sendo correspondente ao objeto `stdout` da linguagem C.

9. Esta informação diz primariamente respeito ao paradigma de Orientação a Objetos, que não será abordado neste material.

```
1  std::cout << "Hello world!" << std::endl;
```

Hello world!

3. cerr

`ostream std::cerr` é um *objeto* que representa o *stream padrão de erro* orientado a caracteres comuns, sendo correspondente ao objeto `stderr` da linguagem C.

Este *stream* é indicado para saída de informação relacionada a erros ocorridos na aplicação.

```
1  std::cerr << "Erro na execucao do codigo!"  
2      << std::endl;
```

4. clog

`ostream std::clog` é um *objeto* que representa o *stream padrão de logging* orientado a caracteres comuns, sendo correspondente ao objeto `stderr` da linguagem C.

Este *stream* não tem muita diferença de `std::cerr`, exceto por algumas peculiaridades, como *buffering* de saída. Todavia, esta estrutura pode ter a saída redirecionada para um arquivo ou mesmo para `stdout`, sem interferir em `std::cout` e `std::cerr`.

```
1  std::clog << "Erro na execucao do codigo!"  
2      << std::endl;
```

5. endl

`std::endl` é uma estrutura utilizada para representar um caractere de nova linha, podendo ser utilizado com `std::cout`, `std::cerr`, `std::clog`, ou qualquer *stream* para arquivos.

A diferença entre o uso de `std::endl` e imprimir o caractere `'\n'` na tela é que `std::endl` força qualquer saída ainda não tenha aparecido na saída a aparecer¹⁰.

```
1  std::cout << std::endl;
```

2.3.3. `iostream`

```
1  #include <iomanip>
```

Esta biblioteca provê *manipuladores paramétricos*, ou seja, estruturas de manipulação durante a saída de uma informação para uma estrutura como `std::cout`, `std::cerr`, `std::clog`, ou um outro *stream* de arquivo de C++.

Estes manipuladores também podem ser utilizados em *streams* de entrada como `std::cin`, para realizar entrada formatada.

1. `setbase`

`std::setbase(int n)` define a *base* de saída ou entrada de um número impresso ou lido, dependendo do valor informado. Os valores válidos são 8 (base *octal*), 10 (base *decimal*; padrão) e 16 (base *hexadecimal*). Valores diferentes resetam a base para 10.

```
1  int a = 255;
2
3  std::cout << "O valor hexadecimal de " << a
4          << " eh " << std::setbase(16) << a
5          << std::endl;
```

10. Em palavras mais técnicas, `std::endl` força um processo de *flush* no *buffer* de saída.

O valor hexadecimal de 255 eh ff

2. setw

`std::setw(int n)` define o parâmetro de *largura* para uma entrada ou saída para o tamanho de `n` caracteres.

```
1 int a = 7;
2
3 std::cout << "Sem setw(6): " << a << std::endl
4         << "Com setw(6): " << std::setw(6) << a
5         << std::endl;
```

Sem `setw(6)`: 7

Com `setw(6)`: 7

3. setfill

`std::setfill(char c)` define o *caractere de preenchimento* no *stream* atual.

Pode ser utilizado com `std::setw`.

```
1 int a = 7;
2 std::cout << "Sem setfill('0'): "
3         << std::setw(6) << a << std::endl
4         << "Com setfill('0'): "
5         << std::setfill('0') << std::setw(6) << a
6         << std::endl;
```

Sem `setfill('0')`: 7

Com `setfill('0')`: 000007

4. `setprecision`

`std::setprecision(int n)` define a precisão do objeto a ser impresso ou lido no *stream* atual.

```

1  const long double pi = std::acos(-1.0L);
2
3  std::cout << "Prec. normal (6): " << pi << std::endl
4      << "setprecision(10): "
5      << std::setprecision(10) << pi << std::endl;

```

```

Prec. normal (6): 3.14159
setprecision(10): 3.141592654

```

- **NOTA:** É possível escrever uma variável qualquer com a sua precisão máxima. Para isso, é necessário incluir a biblioteca `<limits>` (não confunda com `<climits>`):

```

1  const long double pi = std::acos(-1.0L);
2
3  int limite =
4      std::numeric_limits<long double>::digits10 + 1;
5
6  std::cout << "Prec. maxima: "
7      << std::setprecision(limite) << pi
8      << std::endl;

```

```
Prec. maxima: 3.141592653589793239
```

2.3.4. `fstream`

```
1 #include <fstream>
```

Esta biblioteca define estruturas básicas de entrada e saída que precedem o uso do *console*. Em especial, podem ser utilizadas para entrada e saída em arquivos.

1. ofstream

`std::ofstream` é uma estrutura capaz de abrir um arquivo-texto ou um arquivo binário, e *escrever* no mesmo, com a mesma sintaxe utilizada para `std::cout`, `std::cerr` e `std::clog`.

```
1 std::ofstream ofs;
2 ofs.open("arquivo.txt");
3
4 // Só escreva no arquivo se ele foi criado
5 // apropriadamente
6 if(ofs.is_open()) {
7     const long double pi = std::acos(-1.0L);
8     ofs << "Pi eh igual a " << pi << std::endl;
9
10    ofs.close();
11 }
```

2. ifstream

`std::ifstream` é uma estrutura capaz de abrir um arquivo-texto ou um arquivo binário, e *ler informações* do mesmo, com a mesma sintaxe utilizada para `std::cin`.

```
1 std::ifstream ifs;
2 ifs.open("arquivo.txt");
3
4 // Só leia do arquivo se ele foi aberto
```

```
5 // apropriadamente
6 if(ifs.is_open()) {
7     std::string nome, sobrenome;
8     ifs >> nome >> sobrenome;
9
10    std::cout << "Nome:      " << nome
11              << std::endl
12              << "Sobrenome: " << sobrenome
13              << std::endl;
14
15    ifs.close();
16 }
```

2.3.5. Standard Template Library (STL)

As próximas bibliotecas descrevem a *biblioteca padrão baseada em templates*. Trata-se de *estruturas de dados generalizadas*, de forma a interagirem com o resto do ecossistema de C++.

À exceção de `std::string`, nenhuma das estruturas a seguir será abordada diretamente neste material. Todavia, elas serão rapidamente expostas aqui como curiosidade.

1. `string`

```
1 #include <string>
```

Esta biblioteca (não confundir `<string>` com `<cstring>`!) define a classe `std::string`, uma abstração para uma *cadeia de caracteres*. Não utilizaremos esta estrutura, mas ela pode ser compreendida como uma forma mais sucinta de manipulação de cadeias de caracteres.

```
1  std::string texto = "Este eh o meu texto";
2
3  std::cout << "Meu texto: " << texto << std::endl;
```

Meu texto: Este eh o meu texto

2. list

```
1  #include <list>
```

Esta biblioteca define a classe `std::list`, capaz de armazenar uma *lista encadeada*¹¹ de objetos do tipo especificado durante sua declaração.

```
1  std::list<int> lista_de_numeros;
2
3  srand(time(NULL));
4
5  // Tres numeros aleatórios [0, 99] na lista
6  lista_de_numeros.push_back(rand() % 100);
7  lista_de_numeros.push_back(rand() % 100);
8  lista_de_numeros.push_back(rand() % 100);
9
10 // Imprime cada elemento da lista
11 for(int numero : lista_de_numeros) {
12     std::cout << numero << ' ';
13 }
14 std::cout << std::endl;
```

52 15 42

11. Esta estrutura não será abordada neste material.

3. map

```
1 #include <map>
```

Esta biblioteca define a classe `std::map`, capaz de armazenar associações *chave-valor* de objetos dos tipos especificados durante sua declaração.

Cada *compilador* implementa `std::map` de uma forma. O GCC utiliza uma implementação baseada em *árvore RB*¹².

```
1 // Associa cadeias de caracteres a um único número
2 std::map<std::string, int> numeros_naturais;
3
4 // Algumas atribuicoes
5 numeros_naturais["zero"] = 0;
6 numeros_naturais["um"] = 1;
7 numeros_naturais["dois"] = 2;
8
9 std::cout << "Atribuicoes: " << std::endl
10         << "zero => " << numeros_naturais["zero"]
11         << std::endl
12         << "um  => " << numeros_naturais["um"]
13         << std::endl
14         << "dois => " << numeros_naturais["dois"]
15         << std::endl;
```

```
Atribuicoes:
zero => 0
um  => 1
dois => 2
```

12. Árvores vermelho-preto não serão abordadas neste material.

4. stack

```
1 #include <stack>
```

Esta biblioteca define a classe `std::stack`, capaz de armazenar uma *pilha*¹³ de valores de um mesmo tipo.

Pilhas são estruturas de dados *last-in-first-out* (LIFO).

```
1 std::stack<int> pilha;  
2  
3 // Algumas operações  
4 pilha.push(1);  
5 pilha.push(2);  
6 pilha.push(3);  
7  
8 std::cout << "Topo da pilha: " << pilha.top()  
9         << std::endl;  
10  
11 std::cout << "Retirando elemento do topo..."  
12         << std::endl;  
13 pilha.pop();  
14  
15 std::cout << "Topo da pilha: " << pilha.top()  
16         << std::endl;
```

Topo da pilha: 3

Retirando elemento do topo...

Topo da pilha: 2

13. Pilhas não serão abordadas neste material.

2.4. EXERCÍCIOS DE FIXAÇÃO

Exponenciação

Escreva um programa que lê dois números reais: uma *base* e um *expoente*, e escreve na tela o resultado da exponenciação.

Teorema de Pitágoras

Construa um programa que recebe o tamanho dos catetos *oposto* e *adjacente* de um triângulo-retângulo como números reais, e calcula o tamanho da *hipotenusa* de tal triângulo-retângulo.

Lembre-se de que este cálculo pode ser feito através do Teorema de Pitágoras, dado pela fórmula

$$a^2 = b^2 + c^2$$

onde a é o tamanho da hipotenusa, e b e c são os tamanhos dos respectivos catetos.

Arredondamento

Construa um programa que recebe um número real e arredonda este número.

O arredondamento é feito de acordo com a parte decimal do número:

- Caso a parte decimal seja maior ou igual a $.5$, o número será arredondado para o primeiro número inteiro *acima* do mesmo;
- Caso a parte decimal seja menor que $.5$, o número será arredondado para o primeiro número inteiro *abaixo* do mesmo.

Há duas formas de realizar o arredondamento de um número real, de acordo com estas equações:

$\text{ceil}(x - 0.5)$ ou
 $\text{floor}(x + 0.5)$

Utilize-as, evitando direcionamentos baseados em controle de fluxo.

3. VETORES

Sabemos realizar operações e aritmética utilizando variáveis. Sabemos, também, realizar comparações que determinam o fluxo de execução da nossa aplicação; chamamos isto de *controle de fluxo*.

Mas existem situações onde o consumo de memória da aplicação necessita de "atalhos" ao ser declarado. Por exemplo, considere o problema a seguir:

Construa um programa que recebe as notas finais dos alunos de uma turma de trinta alunos e exibe a média de nota da turma.

A confecção do código de forma ingênua nos traz alguns problemas:

- É inviável criar *trinta variáveis*, uma para cada aluno. Isto deixaria o código visualmente complexo;
- Caso tivéssemos *trinta variáveis*, seria difícil referirmo-nos a elas como uma única estrutura;
- Em caso de consulta posterior, identificar a nota de um único aluno envolveria uma *gigantesca estrutura de controle de fluxo*: mesmo se utilizarmos a sintaxe sucinta do `switch`, teremos problemas da mesma forma.

Para resolver os problemas acima, é necessário que criemos uma sintaxe para a declaração de diversas variáveis do mesmo valor, ao mesmo tempo.

3.1. PRIMITIVAS, COMBINAÇÃO E ABSTRAÇÃO

Segundo Abelson *et al.* (1996), Quando aprendemos uma *linguagem de programação*, há alguns elementos sobre os quais procuramos nos inteirar para realizar a computação na mesma.

Estes elementos são:

- *Primitivas*: Elementos primitivos na linguagem, que podem ser rearranjados a gosto do programador. Por exemplo, temos os tipos básicos de variáveis: `int`, `float`, `double`, etc.
- *Meios de combinação*: Elementos formados através do agrupamento de estruturas primitivas, que comportam-se também como se fossem elementos primitivos. Por exemplo, temos *vetores* (a serem vistos a seguir) e *tipos abstratos de dados* (a serem vistos posteriormente).
- *Meios de abstração*: Elementos que manipulam *primitivas e combinações*, de forma a produzir abstrações que transcendem o mero armazenamento de dados. Por exemplo, temos *funções* (a serem melhor detalhadas posteriormente).

Neste capítulo, veremos o uso e a manipulação de *vetores*, que são *meios de combinação* de variáveis primitivas.

3.2. O QUE SÃO VETORES?

Vetores são *estruturas com n elementos do mesmo tipo*. Estes elementos ocupam *espaços contíguos na memória do computador*.

Isto significa que, dado um vetor de *cinco* números inteiros, por exemplo, é **garantido** que o segundo elemento do vetor esteja posicionado, na memória RAM, após o primeiro elemento, e antes do terceiro elemento. Ademais, nenhuma outra variável que não pertença ao vetor poderá estar entre nenhum destes elementos.

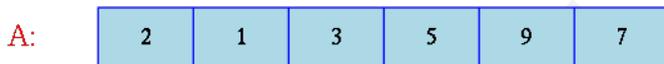


Figura 24 – Ilustração de um vetor *A*, de seis elementos

Os vetores ocupam *espaços contíguos*, ou seja, elementos que aparecem *em sequência e juntos* na memória.

3.3. USANDO VETORES

Em C++, podemos declarar um certo vetor de *n* elementos utilizando a sintaxe

```
tipo nome_do_vetor[n];
```

Onde:

- **tipo** é o *tipo* dos elementos do vetor, correspondendo a tipos *básicos*¹;
- **nome_do_vetor** é o símbolo para uma *variável*, que servirá para referenciar o vetor ao longo do programa;
- **n** é a *quantidade de elementos* que será armazenada no vetor.

1. Podemos também utilizar tipos declarados pelo usuário, mas este tipo de meio de abstração será melhor analisado posteriormente.

Por exemplo, dado o vetor de números inteiros da Figura 24, sua declaração em C++ seria algo como:

```
1 int A[6];
```

- **NOTA:** Jamais utilize uma variável para dimensionar um vetor. Por exemplo, o código a seguir é incorreto:

```
1 /* Este código é incorreto */  
2 int tamanho;  
3 std::cin >> tamanho;  
4 int meu_vetor[tamanho];
```

Os compiladores `gcc` e `clang` permitirão que você realize esta operação, mostrando, no máximo, um *aviso de compilação*. Todavia, compiladores como o compilador do Microsoft Visual C++ (`msvc`) proibem tal declaração, interrompendo o processo de compilação do programa.

O motivo para o problema está relacionado à *região na memória* onde um vetor deste tipo será alocado. Estamos lidando com **vetores estáticos**, portanto estes vetores existirão em uma região de memória **limitada** e reservada apenas para o seu programa.

Isto significa que o *tamanho do vetor* deverá ser conhecido **no momento em que o programa compila**.

Caso você queira utilizar um espaço de memória com tamanho informado pelo usuário, **deverá realizar alocação dinâmica**, um tópico que não abordaremos neste material.

- **NOTA:** Se você deseja *parametrizar o tamanho do vetor*, mantendo ainda o conhecimento do compilador sobre o tamanho

do mesmo, poderá utilizar um *macro*², ou seja, uma definição de cunho puramente sintático, que *substitui* uma certa palavra no código pelo valor do tamanho do vetor. Veja:

```
1 #define TAMANHO_DO_VETOR 6
2
3 int main()
4 {
5     int meu_vetor[TAMANHO_DO_VETOR];
6     return 0;
7 }
```

3.3.1. Acessando elementos no vetor

Para acessar um elemento do vetor, utilizamos uma sintaxe similar à sua declaração:

```
nome_do_vetor[indice]
```

Nesta sintaxe, é completamente *legal* utilizar uma variável inteira, por exemplo, no lugar de `indice`.

Dado um vetor de tamanho `tam`, o *índice de acesso* a um elemento do vetor varia entre `0` e `tam - 1`. Por exemplo, para um vetor `int A[6]`, serão acessíveis elementos de `A[0]` até `A[5]`, como podemos ver no exemplo a seguir.

```
1 int A[6];
2
```

2. *Macros* são estruturas úteis que podem desempenhar diversos papéis, deixando o código muito sucinto, ou modificando diretamente a forma como o programa é compilado. Todavia, *macros* devem ser usados com moderação, para não causar mais dano sintático que ajuda. Não abordaremos o uso extensivo de *macros* neste material.

```
3  /* Mais linhas viriam aqui... */
4
5  std::cout << "Primeiro elemento: " << A[0] << std::endl
6      << "Segundo elemento: " << A[1] << std::endl
7      << "Terceiro elemento: " << A[2] << std::endl
8      << "Quarto elemento: " << A[3] << std::endl
9      << "Quinto elemento: " << A[4] << std::endl
10     << "Sexto elemento: " << A[5] << std::endl;
```

- **ATENÇÃO:** É muito importante que você preste bastante atenção ao valor do índice sendo acessado. Acessar um índice fora do espaço permitido não necessariamente causará um *erro de execução* no seu programa, mas você estará acessando valores de outras partes da sua aplicação, que não deveriam ser lidos ou modificados. Tal erro é conhecido como *buffer overflow*, sendo da mesma categoria ocasionada pela função `gets` (discutida na Seção 4).

3.3.2. Atribuindo valores ao vetor

Podemos atribuir valores a um vetor referindo-nos ao índice do elemento no qual a atribuição será realizada.

Vejamos o código completo para construir o vetor `A`, visualmente demonstrado na Figura 24.

```
1  int A[6];
2
3  A[0] = 2;  A[1] = 1;
4  A[2] = 3;  A[3] = 5;
5  A[4] = 9;  A[5] = 7;
6
7  std::cout << "Primeiro elemento: " << A[0] << std::endl
```

```
8         << "Segundo elemento: " << A[1] << std::endl
9         << "Terceiro elemento: " << A[2] << std::endl
10        << "Quarto elemento: " << A[3] << std::endl
11        << "Quinto elemento: " << A[4] << std::endl
12        << "Sexto elemento: " << A[5] << std::endl;
```

```
Primeiro elemento: 2
Segundo elemento: 1
Terceiro elemento: 3
Quarto elemento: 5
Quinto elemento: 9
Sexto elemento: 7
```

Quando referenciamos uma posição específica de um vetor e realizamos uma *atribuição* deste tipo, o elemento referenciado age, efetivamente, como uma variável. Sendo assim **também é possível realizar aritmética** com os elementos do vetor, da mesma forma como é feito com variáveis.

3.3.3. Inicializando o vetor na declaração

Em C++, podemos inicializar um vetor com os itens expressos na forma de uma *literal*. Isto economiza tempo e torna desnecessário realizar as atribuições manualmente:

```
1 int A[6] = {2, 1, 3, 5, 9, 7};
2
3 std::cout << "Primeiro elemento: " << A[0] << std::endl
4         << "Segundo elemento: " << A[1] << std::endl
5         << "Terceiro elemento: " << A[2] << std::endl
6         << "Quarto elemento: " << A[3] << std::endl
```

```
7     << "Quinto elemento:  " << A[4] << std::endl
8     << "Sexto elemento:   " << A[5] << std::endl;
```

```
Primeiro elemento: 2
Segundo elemento:  1
Terceiro elemento: 3
Quarto elemento:   5
Quinto elemento:   9
Sexto elemento:    7
```

Caso sejam informados menos elementos que o esperado, as posições restantes serão completadas com zeros:

```
1  int A[6] = {2, 1, 3};
2
3  std::cout << "Primeiro elemento:  " << A[0] << std::endl
4          << "Segundo elemento:   " << A[1] << std::endl
5          << "Terceiro elemento:  " << A[2] << std::endl
6          << "Quarto elemento:    " << A[3] << std::endl
7          << "Quinto elemento:    " << A[4] << std::endl
8          << "Sexto elemento:     " << A[5] << std::endl;
```

```
Primeiro elemento: 2
Segundo elemento:  1
Terceiro elemento: 3
Quarto elemento:   0
Quinto elemento:   0
Sexto elemento:    0
```

Se o programador preferir, o vetor também poderá ser inicializado com zeros através de uma literal vazia:

```
1  int A[6] = {};  
2  
3  std::cout << "Primeiro elemento: " << A[0] << std::endl  
4      << "Segundo elemento: " << A[1] << std::endl  
5      << "Terceiro elemento: " << A[2] << std::endl  
6      << "Quarto elemento: " << A[3] << std::endl  
7      << "Quinto elemento: " << A[4] << std::endl  
8      << "Sexto elemento: " << A[5] << std::endl;
```

```
Primeiro elemento: 0  
Segundo elemento: 0  
Terceiro elemento: 0  
Quarto elemento: 0  
Quinto elemento: 0  
Sexto elemento: 0
```

Finalmente, para o caso de uma inicialização onde todos os elementos são informados, o tamanho do vetor poderá ser omitido. Assim, o compilador deduzirá o tamanho do mesmo pela quantidade de itens informados:

```
1  int A[] = {2, 1, 3, 5, 9, 7};  
2  
3  std::cout << "Primeiro elemento: " << A[0] << std::endl  
4      << "Segundo elemento: " << A[1] << std::endl  
5      << "Terceiro elemento: " << A[2] << std::endl  
6      << "Quarto elemento: " << A[3] << std::endl  
7      << "Quinto elemento: " << A[4] << std::endl  
8      << "Sexto elemento: " << A[5] << std::endl;
```

```
Primeiro elemento: 2
```

Segundo elemento: 1
Terceiro elemento: 3
Quarto elemento: 5
Quinto elemento: 9
Sexto elemento: 7

3.4. STRINGS

Vetores são especialmente úteis para um tipo de informação anteriormente discutida, porém não aprofundada: as *cadeias de caracteres*, também conhecidas como *strings*.

Strings nada mais são que vetores de elementos do tipo `char`, onde o último elemento é **sempre** um caractere de *terminador nulo* (*null terminator*), representado em C++ como `'\0'`.

Por exemplo, para a variável `str`, contendo a *string* "Teste", podemos representá-la da seguinte forma:

`str:`

T	e	s	t	e	\0
---	---	---	---	---	----

Note que, quando informamos à linguagem C++ uma literal de uma *string*, como é o caso de "Teste", a linguagem presume automaticamente que estamos nos referindo a este vetor de caracteres, incluindo o caractere `'\0'` ao final, que fica implícito.

Podemos declarar a *string* `str` em C++ desta forma. Note que o vetor de caracteres deve possuir um tamanho mínimo igual à quantidade de letras, acrescido de uma unidade para abrigar o *null terminator*:

```
1 char str[6] = "Teste";
```

Da mesma forma como fazemos com qualquer vetor, também podemos omitir a quantidade de caracteres e esperar que o compilador deduza o tamanho da *string*:

```
1 char str[] = "Teste";
```

Para o caso especial de um vetor que seja uma *string*, podemos simplesmente imprimi-lo no console usando `std::cout` ou `printf`.

```
1 char str[] = "Teste";  
2 std::cout << str << std::endl;
```

Teste

Por último, uma *string* não precisa, necessariamente, tomar todo o espaço de um vetor de caracteres. Todavia, o vetor de caracteres **precisa** ter espaço para abrigar a *string* por completo, incluindo o *null terminator*. Considere o exemplo a seguir:

```
1 char str[8] = "Teste";  
2 std::cout << str << std::endl;
```

Teste

A figura a seguir demonstra como a *string* está armazenada no espaço reservado.

str:

T	e	s	t	e	\0	(lixo)	(lixo)
---	---	---	---	---	----	--------	--------

Neste exemplo, teremos *duas posições extras* que não foram utilizadas no vetor. Todavia, a *string* possui um *null terminator* apropriado, portanto sua impressão na tela não apresentará erros inesperados.

- **NOTA:** É essencial que o programador use conscientemente os espaços alocados para uma *string*. Caso você queira uma abstração para uso mais seguro, poderá utilizar o objeto `std::string`, que requererá um pouco mais de estudo para manuseio apropriado.

Não nos aprofundaremos no objeto `std::string` neste material, uma vez que ele demanda um pouco mais de conhecimento de suas abstrações. Para algumas informações neste sentido, consulte a Seção 2.3.5.

3.4.1. Lendo *strings* informadas pelo usuário

O objeto `std::cin` pode ser utilizado para ler algumas *strings* digitadas pelo usuário. No entanto, o método trivial para isso está propenso a problemas.

Considere o código a seguir:

```
1 char nome[80];
2 std::cin >> nome;
```

Caso o usuário digite um nome como, por exemplo, *John Doe*, apenas o nome *John* será armazenado no vetor de caracteres `nome`.

O motivo para isso é que o uso cru de `std::cin` apenas recebe caracteres até o primeiro espaço em branco (que poderá ser um *espaço*, uma *quebra de linha* `'\n'`, um *tab* `'\t'`...).

Caso você queira receber a entrada de uma linha de caracteres através do console, e então armazená-la em uma *string* (assegurando também a existência do *null terminator* na mesma), poderá utilizar a instrução `std::cin.getline`:

```
1 char nome[80];
2 std::cin.getline(nome, 80);
```

Note que esta instrução também previne erros de *buffer overflow*, uma vez que realiza a conferência da quantidade máxima de caracteres a serem utilizados.

3.4.2. Sobre leitura correta de *strings* em *streams*

Algumas vezes, o uso de `std::cin.getline` pode ocasionar erros, gerando leituras de *strings* "vazias".

O motivo para isso é a forma como o uso cru de `std::cin` funciona, em relação ao uso de `std::cin.getline`.

A *entrada do console* funciona como uma entrada de um *arquivo* qualquer, o que chamamos de *stream*. Um *stream* é, de forma grosseira, um *fluxo* de caracteres a ser lido pelo programa. Sendo assim, a entrada do console (conhecida como `stdin`) é um *stream*, ou seja, uma espécie de pseudo-arquivo.

A digitação do usuário adiciona novos caracteres ao fim de `stdin`, para cada vez que o usuário aperta Enter. Ademais, o pressionamento de Enter em si não apenas "envia" os caracteres para o programa, como também envia um único caractere '\n' por pressionamento de tecla Enter.

Caso o programa esteja esperando uma entrada do usuário, ele esperará até que novos caracteres estejam disponíveis para que sejam lidos.

Quando um usuário digita, em sequência, um *número* (exemplo: 80) aperta a tecla Enter, e então digita um *texto* (exemplo: Foo) e aperta Enter novamente, podemos entender este fluxo como uma fila de caracteres, desta forma:

`stdin:`

8	0	\n	F	o	o	\n	...
---	---	----	---	---	---	----	-----

Se executarmos a seguir um código como este...

```

1  int valor;
2  std::cin >> valor;

```

...o *stream* `std::cin` terá todos os caracteres consumidos, até que seja encontrado um primeiro *caractere em branco* (como *espaço*, `'\n'`, `'\t'` e outros), e este *caractere em branco* não seria consumido:

`std::cin:`

<code>\n</code>	<code>F</code>	<code>o</code>	<code>o</code>	<code>\n</code>	<code>...</code>	<code>...</code>	<code>...</code>
-----------------	----------------	----------------	----------------	-----------------	------------------	------------------	------------------

Este texto consumido será, neste caso, implicitamente convertido para um valor numérico (uma vez que estamos armazenando a entrada em um `int`).

Ademais, se houvesse alguns caracteres *em branco* precedendo o valor `80`, estes caracteres **seriam consumidos e pulados**, sendo irrelevantes para a dedução do valor em si.

Esta é a origem do problema de `std::cin.getline`: este método lê todos os caracteres *até o próximo caractere em branco*. Além de consumir o caractere em branco de "parada" ao final da frase coletada (diferente do uso de `std::cin` "puro"), `std::cin.getline` **não ignora caracteres em branco antes da entrada válida**. Isto significa que o código a seguir, quando executado após o anterior...

```

1  char blah[80];
2  std::cin.getline(blah, 80);

```

...consumiria apenas o caractere `'\n'` no *stream* `std::cin`, o que resultaria neste caractere sendo *ignorado* quando a *string* fosse armazenada no vetor `blah`, gerando uma *string* vazia (nenhum caractere antes do *null terminator*, `'\0'`). Adicionalmente, a *string* que seria coletada, em si, **continuará intacta em** `std::cin`:

stdin:

F	o	o	\n
---	---	---	----	-----	-----	-----	-----

Felizmente, mitigar este problema é extremamente simples: basta consumir, de forma explícita, quaisquer caracteres em branco que estejam no início de `stdin`, antes de utilizarmos `std::cin.getline`. Para isso, podemos utilizar a estrutura de C++ conhecida como `std::ws`, da biblioteca `<iostream>`. Veja como reescrever o código:

```
1 char blah[80];
2 std::cin >> std::ws;
3 std::cin.getline(blah, 80);
```

A estrutura `std::ws` funciona como uma espécie de "buraco negro" que "suga" todos os caracteres em branco que estiverem à frente de um *stream*. Esta estrutura é exclusiva para C++, não podendo ser utilizada em C.

3.5. EXERCÍCIOS DE FIXAÇÃO I

'Hello World' Personalizado

Construa um programa que pergunta o nome do usuário, e imprime na tela a saudação "Ola, <nome>!", onde <nome> deverá ser substituído pelo nome informado pelo usuário.

Soma de três números

Construa um programa que lê três números reais digitados pelo usuário, e armazena-os em um vetor de três elementos. Em seguida, escreva na tela os três números digitados e, por fim, escreva também a soma destes três números.

Média aritmética II

Crie um programa que lê cinco números reais digitados pelo usuário, armazena-os em um vetor de cinco elementos, e então calcula a média aritmética entre todos eles, imprimindo o resultado na tela.

Citação formal

Construa um programa que receba as seguintes informações:

- O *título* de um livro;
- O *sobrenome* do autor;
- O *primeiro nome* do autor;
- O *ano* de lançamento do livro;
- A *editora* do livro;
- O *local* de lançamento do livro.

Em seguida, imprima as informações recebidas na tela, segundo este exemplo:

```
Sobrenome, Nome. Título do livro. Local: Editora, ano.
```

3.6. LAÇOS DE REPETIÇÃO

Agora que temos um vocabulário para *representar* variáveis contíguas, precisamos também de um vocabulário para *percorrer* estas variáveis. Do contrário, recorreríamos à abordagem pouco prática de acessar os elementos um a um, o que não seria muito melhor que declarar variáveis exclusivas para cada posição do vetor.

A resolução deste problema está nos *laços de repetição*, que abstraem a execução de um certo bloco de código. A ideia é fazer com que o bloco de código referido seja executado mais de uma vez.

Sendo assim, é importante ressaltar que, ainda que laços de repetição sejam extremamente úteis na manipulação de vetores, eles **também podem ser utilizados para outros fins**.

Laços de repetição são abstrações para *funções recursivas* muito especiais, que dizemos serem *iterações*. Portanto, laços executam repetições *iterativas* no código.

3.6.1. Laços `while`

Um laço de variedade `while` é similar, em sintaxe, a uma condicional.

Este laço executa continuamente o código especificado em seu escopo, até que o seu *predicado* torne-se falso.

Podemos utilizar um laço deste tipo para imprimir o vetor representado na Figura 24, desta forma:

```
1  int A[] = {2, 1, 3, 5, 9, 7};
2
3  // Índice do elemento iniciado em zero
4  int i = 0;
5
6  // Enquanto o índice não for maior que 5...
7  while(i <= 5) {
8      // Imprima o elemento na posição i
9      // e pule um espaço
10     std::cout << A[i] << ' ';
11
12     // Incremente a variável i
13     i++;
14 }
```

```
15
16 // Pule uma única linha ao final
17 std::cout << std::endl;
```

2 1 3 5 9 7

Caso você queira encerrar um laço qualquer antes de o *predicado* em questão tornar-se falso, utilize o comando `break;`, desta forma:

```
1 int A[] = {2, 1, 3, 5, 9, 7};
2 int i = 0;
3
4 // Repita infinitamente: true é sempre
5 // verdadeiro
6 while(true) {
7     std::cout << A[i] << ' ';
8     i++;
9
10    // Se i for maior que 5, quebre
11    // o laço
12    if(i > 5) {
13        break;
14    }
15 }
```

2 1 3 5 9 7

- **NOTA:** O uso de `break;` em um `switch`, caso este `switch` esteja dentro de um laço de repetição, não afetará a execução do laço.

3.6.2. Laços `do...while`

Um laço de variedade `do...while` é idêntico ao laço `while`, com uma única ressalva: ao invés de a verificação do *predicado* ser realizada *antes* da execução do bloco (como no `while`), esta verificação é realizada **após** a execução do bloco.

O uso de `do...while` não é muito comum, não sendo necessariamente encontrado em outras linguagens, mas continua sendo uma ferramenta extremamente útil.

Vejamos o exemplo a seguir:

```
1 int i = 1;
2
3 while(i < 1) {
4     std::cout << "Executou!" << std::endl;
5 }
```

No exemplo anterior, a aplicação não realiza nenhuma saída: como o laço `while` verifica que `i` não é menor que o valor `1`, o bloco de código do laço nunca é executado.

Todavia, no próximo exemplo...

```
1 int i = 1;
2
3 do {
4     std::cout << "Executou!" << std::endl;
5 } while(i < 1);
```

Executou!

...podemos ver que ocorre a impressão do texto "Executou!" uma única vez. Isto se dá porque a verificação do valor de `i` só ocorre *após a primeira execução* do bloco de código do laço de repetição.

3.6.3. Laços for

Laços de variedade `for` são especialmente úteis para percorrer vetores, porque determinam a *inicialização*, a *mudança* e a *verificação* na variável `i`, que normalmente chamamos de *iterador*.

Se nos recordarmos do primeiro exemplo de impressão do vetor `A` na tela, utilizando um laço de variedade `while`, temos:

```
1 int A[] = {2, 1, 3, 5, 9, 7};
2 int i = 0;
3
4 while(i <= 5) {
5     std::cout << A[i] << ' ';
6     i++;
7 }
8 std::cout << std::endl;
```

```
2 1 3 5 9 7
```

Podemos reescrever facilmente este laço utilizando a variedade de laço `for`:

```
1 int A[] = {2, 1, 3, 5, 9, 7};
2
3 for(int i = 0; i <= 5; i++) {
4     std::cout << A[i] << ' ';
5 }
6 std::cout << std::endl;
```

```
2 1 3 5 9 7
```

Veja que a aritmética de incremento da variável `i` passou a ser parte da declaração do laço em si, bem como a declaração da variável.

Podemos definir a sintaxe de um laço de variedade `for` da seguinte forma:

```
for(<declarações>; <predicado>; <incrementos e decrementos>)  
{  
    <corpo...>  
}
```

Também é possível realizar laços `for` complexos, utilizando mais de uma variável. O exemplo a seguir realiza uma operação deste tipo, enquanto também decreta as variáveis.

A declaração do laço `for` foi quebrada em linhas para melhor visualização.

```
1  int A[] = {2, 1, 3, 5, 9, 7};  
2  
3  for(int i = 5, n = 6; // Declarações  
4      i >= 0;          // Predicado  
5      i--, n--)        // Incrementos  
6  {  
7      std::cout << n << "º elemento: "  
8          << A[i] << std::endl;  
9  }
```

```
6º elemento: 7  
5º elemento: 9  
4º elemento: 5  
3º elemento: 3  
2º elemento: 1  
1º elemento: 2
```

É possível, também, percorrermos *strings* facilmente através do uso de laços `for`. Vejamos o exemplo a seguir, que conta a quantidade de *vogais* em uma *string*.

Para tanto, faremos uso das bibliotecas `<cstring>`, que possui a função `strlen` (que determina o tamanho da *string*, até o *null terminator*) e `<cctype>`, que possui a função `tolower` (que converte um caractere qualquer para *minúsculo*).

Também utilizamos uma estrutura de `switch`, o que nos economiza uma quantidade gigantesca de comparações entre literais e a variável `caractere`. O uso do *fallthrough* no `switch` ajuda a manter o código sucinto.

```
1 char texto[] =
2     "The Quick Brown Fox Jumps Over The Lazy Dog";
3
4 int num_vogais = 0;
5
6 for(int i = 0; i < strlen(texto); i++) {
7     char caractere = tolower(texto[i]);
8
9     switch(caractere) {
10        case 'a':
11        case 'e':
12        case 'i':
13        case 'o':
14        case 'u':
15            num_vogais++;
16            break;
17        default: break;
18    }
19 }
20
21 std::cout << "Numero de vogais: "
22            << num_vogais << std::endl;
```

Numero de vogais: 11

- **NOTA:** Na linguagem C (em sua especificação de 1989), laços `for` não admitem a declaração de variáveis em sua definição. Em contrapartida, o espaço onde esperaríamos por declarações é utilizado para a *atribuição de valores* antes da execução do laço.

O código a seguir apresenta uma declaração de um laço `for` que é válida³ para C e para C++:

```
1  int A[] = {2, 1, 3, 5, 9, 7};
2
3  int i;
4
5  for(i = 0; i <= 5; i++) {
6      printf("%d ", A[i]);
7  }
8  putchar('\n');
```

2 1 3 5 9 7

3.7. EXERCÍCIOS DE FIXAÇÃO II

Média aritmética III

Crie um programa que lê *dez* números reais digitados pelo usuário, armazena-os em um vetor de *dez* elementos, e então calcula a média aritmética entre todos eles, imprimindo o resultado na tela.

3. Este código é compatível principalmente com a especificação da linguagem C de 1989, também conhecida como C89. Ademais, este código precisa da biblioteca `<stdio.h>` (em C++, `<cstdio>`, discutida no capítulo anterior).

Boletim Virtual II

Resolva o problema proposto no início do capítulo:

Construa um programa que recebe as notas finais dos alunos de uma turma de *trinta* alunos e exibe a média de nota da turma.

Média aritmética IV

Crie um programa que lê *dez* números reais digitados pelo usuário, e então calcula a média aritmética entre todos eles, imprimindo o resultado na tela.

Não utilize vetores ao resolver este exercício. Também não utilize variáveis individuais para armazenar as notas.

Lanchonete

Uma lanchonete opera sob uma tabela de preços, que podem ser vistos a seguir⁴.

Código	Lanche	Preço
1	Cachorro-Quente	R\$ 4,00
2	X-Salada	R\$ 4,50
3	X-Bacon	R\$ 5,00
4	Torrada Simples	R\$ 2,00
5	Refrigerante	R\$ 1,50

Construa um programa que lê o código de um produto, e a quantidade do produto comprada. Em seguida, imprima o valor a ser pago pelo cliente. Caso o código ou a quantidade do produto sejam inválidos, imprima uma mensagem de erro.

4. Adaptado de <https://www.urionlinejudge.com.br/judge/en/problems/view/1038>.

Números de Fibonacci Pares

Cada novo termo da *sequência de Fibonacci* é gerado através da adição dos dois termos anteriores. Começando com 1 e 2, os *dez* primeiros termos serão:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89,...

Considerando os termos na sequência de Fibonacci cujo valor *não excede quatro milhões*, encontre a soma de todos os termos que sejam *pares*⁵.

Palíndromos

Uma *palavra* ou *frase* é considerada um **palíndromo** quando todas as suas letras, independentes de serem maiúsculas ou minúsculas, quando lidas de forma inversa, ainda possuem a mesma disposição.

Palavras como *asa*, *ovo*, *esse*, *reler* e *reviver* são consideradas palíndromos. Da mesma forma, frases como

Socorram me subi no onibus em Marrocos

também são consideradas palíndromos.

Construa um programa que recebe uma frase qualquer, e verifica se a mesma é um palíndromo. Você deverá verificar caracteres independente de serem maiúsculos e minúsculos, e deverá também saltar espaços em branco.

- **DICA:**

Você precisará manipular caracteres da *string*, bem como analisar espaços e verificar o tamanho da mesma. Veja as seções 2.2.6 (biblioteca `<cstring>`) e 2.2.9 (biblioteca `<cctype>`) para funções úteis.

5. Adaptado de <https://projecteuler.net/problem=2>.

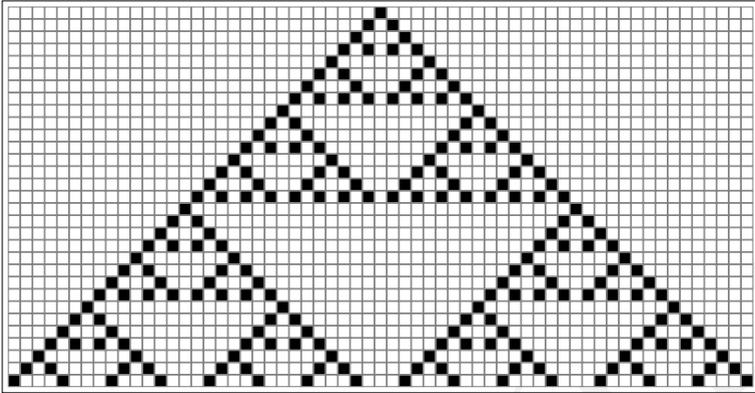


Figura 25 – Evolução da Regra 90 por 31 gerações.

3.8. AUTÔMATOS CELULARES UNIDIMENSIONAIS

Segundo Wolfram (2002), um autômato celular unidimensional consiste de uma *linha de células*, onde cada célula é colorida com as cores *preto* e *branco*. Em cada passo de tempo, há uma regra bem-definida que determina a cor de cada uma das células a partir da cor da mesma e de seus vizinhos *à esquerda* e *à direita*.

A evolução em cada *passo de tempo* gera uma nova *linha de células*, que chamamos de *geração* do AC. Os estados das células assumidos em cada *geração* originam-se se dos estados das células na primeira *linha* fornecida. Portanto, diz-se também que ACs são *sensíveis ao estado inicial*.

A Figura 25 mostra a evolução para um autômato celular unidimensional, conhecido como Regra 90. O gráfico da figura é gerado de forma que uma linha qualquer é resultado da aplicação das regras do autômato celular na linha anterior. Sendo assim, a primeira linha

do autômato é seu *estado inicial*; a segunda linha é sua *primeira geração*, a terceira linha é sua *segunda geração*, e assim por diante.

3.8.1. Notação de Autômatos de Wolfram

Wolfram (2002) instituiu uma visualização gráfica para as regras de autômatos celulares unidimensionais. A Figura 26 demonstra a visualização gráfica para a Regra 90 como exemplo.

Nesta visualização, cada "cláusula" é composta por três blocos superiores, indicando o estado da célula a ser alterada (no meio) e seus vizinhos (à esquerda e à direita). Esta *configuração* da vizinhança faz com que a célula a ser alterada assumo o estado indicado (o bloco solitário logo abaixo dos três blocos).

A Figura 26 torna possível inferir a seguinte lógica para as vizinhanças, de acordo com cada "cláusula":

- Caso a célula em questão e seus dois vizinhos estejam vivos, a célula morre na próxima geração;
- Caso a célula em questão e seu vizinho da esquerda estejam vivos, a célula sobrevive para a próxima geração;
- Caso a célula em questão esteja morta, e seus dois vizinhos estejam vivos, a célula continua morta na próxima geração;
- Caso a célula em questão esteja morta, e apenas seu vizinho esquerdo esteja vivo, a célula passa a viver na próxima geração;
- Caso a célula em questão e seu vizinho da direita estejam vivos, a célula sobrevive para a próxima geração;
- Caso a célula em questão esteja viva e nenhum de seus vizinhos esteja vivo, a célula morre na próxima geração;
- Caso a célula em questão esteja morta, e apenas seu vizinho direito esteja vivo, a célula passa a viver na próxima geração;



Figura 26 – Regras de vizinhança para a Regra 90

- Caso a célula em questão esteja morta e nenhum de seus vizinhos esteja vivo, a célula permanece morta na próxima geração.

Podemos observar que, na realidade, estas regras do autômato celular deixam claro que o estado da célula em questão é irrelevante para o estado da mesma na próxima geração, que será ditado por sua quantidade de vizinhos. Sendo assim, podemos resumir a Regra 90 desta forma:

- Se a célula em questão possui apenas um vizinho vivo, então ela sobrevive ou passa a viver na próxima geração;
- Se a célula em questão possui nenhum ou dois vizinhos vivos, então ela morre ou permanece morta na próxima geração.

O aspecto mais fascinante de um autômato celular como a Regra 90 está nas figuras que *emergem* da iteração de regras tão simples, em elementos tão triviais. Especificamente nesta regra, temos a *emergência* de um *fractal*⁶ muito famoso, conhecido como Triângulo de Sierpiński. Como aponta Mitchell (2009), este tipo de estrutura recorre muitas vezes na natureza, sobretudo em regiões como costas e litorais. Figuras como o Triângulo de Sierpiński e o Conjunto de Mandelbrot constituem fractais famosos.

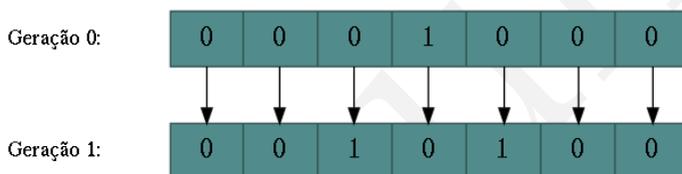
6. Fractais são figuras auto-similares, ou seja, que apresentam *repetição de si mesmas* em muitos de seus níveis.

3.8.2. Modelando AC's unidimensionais

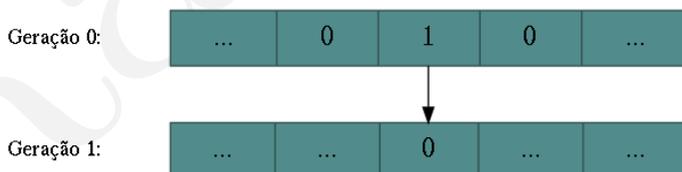
Podemos compreender autômatos celulares unidimensionais na forma de *vetores*. Para tanto, considere um *vetor* que seja populado unicamente pelos valores 0 e 1:



Considerando 0 como sendo o estado para uma célula *morta* e 1 como sendo o estado para uma célula *viva*, podemos aplicar as cláusulas da Regra 90 (representadas na Figura 26) para determinar a próxima geração do autômato celular:



Se cada *número* no vetor representa o estado de uma célula, então os números à esquerda e à direita desta mesma célula representam os *estados do seu vizinho*. Por exemplo, vejamos o *quarto elemento* do vetor anterior. A figura abaixo demonstra como a análise de toda a vizinhança (elemento atual e elementos vizinhos) define o estado do *quarto elemento* na próxima geração.



Alguns casos exigem um cuidado extra, em especial quando tratamos do *primeiro* e do *último* elementos. Neste caso, os elementos não possuem, respectivamente, vizinhos à esquerda e à direita.

Nesta situação, o programador poderá determinar como proceder. Pode-se ignorar a existência destes vizinhos, tratando-os como células permanentemente mortas.

Outra forma mais interessante de lidar com este caso é imaginar o *vetor* como uma *fita* cujas pontas estejam unidas, formando um anel. Dessa forma, podemos considerar os seguintes casos:

- O *vizinho esquerdo* do *primeiro elemento* do vetor torna-se o *último elemento do vetor*;
- O *vizinho direito* do *último elemento* do vetor torna-se o *primeiro elemento do vetor*.

Este tratamento de vizinhança é utilizado nas evoluções das Figuras 25 e 27.

3.9. PROGRAMANDO UM AC UNIDIMENSIONAL

Podemos programar um autômato celular unidimensional através da criação de um programa que faça uso de dois *vetores* e alguns laços de repetição.

3.9.1. Gerando a primeira geração do AC

Seguiremos o seguinte algoritmo:

1. Tome um vetor `gen0` responsável por armazenar a geração *inicial* do AC.
2. Crie um vetor `gen1`, com o mesmo tamanho de `gen0`, responsável por armazenar a *próxima* geração do AC.

3. Percorra cada um dos elementos de `gen0`.
 - Observe, através do índice do elemento no vetor, os valores de estado para o *elemento atual*, o *vizinho esquerdo* e o *vizinho direito*. A partir disso, monte a vizinhança.
4. A partir da configuração de vizinhança criada, escreva em `gen1` o novo estado do elemento. Note que este estado será inserido usando o mesmo índice do elemento em `gen0`.
5. Imprima, opcionalmente, `gen0` na tela.
6. Imprima os elementos de `gen1`.

O programa a seguir implementa o algoritmo acima, usando a Regra 90, como discutida na Seção 3.8.2. Realizamos estas operações para um vetor de sete elementos.

```
1  int gen0[7] = {0, 0, 0, 1, 0, 0, 0};
2  int gen1[7] = {};
3
4  // Iterando sobre cada elemento de gen0
5  for(int i = 0; i < 7; i++) {
6      int indice_esquerdo;
7      int indice_direito;
8
9      // Verifica os índices dos vizinhos
10     if(i == 0) {
11         indice_esquerdo = 6;
12         indice_direito = 1;
13     } else if(i == 6) {
14         indice_esquerdo = 5;
15         indice_direito = 0;
16     } else {
```

```
17     indice_esquerdo = i - 1;
18     indice_direito  = i + 1;
19 }
20
21 // Calcula a vizinhança
22 int esq = gen0[indice_esquerdo];
23 int dir = gen0[indice_direito];
24
25 // Aplica a Regra 90, escrevendo-a em gen1
26 if((esq == 1 && dir == 0) ||
27     (esq == 0 && dir == 1)) {
28     gen1[i] = 1;
29 } else {
30     gen1[i] = 0;
31 }
32 }
33
34 // Imprimindo gen0
35 std::cout << "Geracao 0: ";
36 for(int i = 0; i < 7; i++) {
37     std::cout << gen0[i] << ' ';
38 }
39 std::cout << std::endl;
40
41 // Imprimindo gen1
42 std::cout << "Geracao 1: ";
43 for(int i = 0; i < 7; i++) {
44     std::cout << gen1[i] << ' ';
45 }
46 std::cout << std::endl;
```

Geracao 0: 0 0 0 1 0 0 0

Geracao 1: 0 0 1 0 1 0 0

Como podemos observar, a saída do programa é a esperada, seguindo as regras mostradas na Figura 26.

3.9.2. Criando próximas gerações do AC

O algoritmo mostrado anteriormente demonstra como podemos gerar a *primeira geração* de um AC unidimensional.

Podemos reutilizar este código para mostrar mais de uma geração; basta garantir que a *nova* geração seja copiada para o vetor `gen0`.

Um algoritmo para copiar o vetor `gen1` para o vetor `gen0` poderia ser escrito assim:

```
1 for(int i = 0; i < 7; i++) {
2     gen0[i] = gen1[i];
3 }
```

Após inserir este código ao final do nosso programa, podemos tomar a maior parte do código que já possuímos dentro de outra *estrutura de repetição*.

Esta estrutura maior, que engloba todo o código de geração do AC, será repetida de forma a mostrar a saída para um número arbitrário de gerações. Aqui, definiremos este número como cinco.

```
1 int gen0[7] = {0, 0, 0, 1, 0, 0, 0};
2 int gen1[7] = {};
3
4 // Iterando para cada geração
5 for(int gen = 0; gen < 5; gen++) {
6     // Imprime gen0 antes de aplicar regras
7     std::cout << "Geracao " << gen << ": ";
```

```

8     for(int i = 0; i < 7; i++) {
9         std::cout << gen0[i] << ' ';
10    }
11    std::cout << std::endl;
12
13    // Iterando sobre cada elemento de gen0
14    for(int i = 0; i < 7; i++) {
15        // ...Restante do código omitido...
16    }
17
18    // Copiando gen1 para gen0
19    for(int i = 0; i < 7; i++) {
20        gen0[i] = gen1[i];
21    }
22 }

```

```

Geracao 0: 0 0 0 1 0 0 0
Geracao 1: 0 0 1 0 1 0 0
Geracao 2: 0 1 0 0 0 1 0
Geracao 3: 1 0 1 0 1 0 1
Geracao 4: 1 0 0 0 0 0 1

```

A partir de agora podemos ver que, mesmo com uma visualização puramente numérica, a disposição dos números 1 a cada geração lembra vagamente um Triângulo de Sierpiński.

3.9.3. Embelezando a saída

Nosso código já parece bom, todavia seria ainda mais interessante se nossa visualização mostrasse o triângulo propriamente dito.

Para tanto, realizaremos uma modificação na parte do código que imprime o vetor: ao invés de imprimirmos o número da geração

e seus valores, imprimiremos apenas os valores do vetor, substituindo os números da seguinte forma:

- Todo 0 será substituído por um *ponto*;
- Todo 1 será substituído por um *arroba*.

Também deixaremos de imprimir um espaço em branco após o elemento em questão.

O código ficará desta forma:

```
1 // Imprime gen0 antes de aplicar regras
2 for(int i = 0; i < 7; i++) {
3     if(gen0[i] == 0) {
4         std::cout << '.';
5     } else {
6         std::cout << '@';
7     }
8 }
9 std::cout << std::endl;
```

3.9.4. Código completo

Abaixo, temos o código completo para a aplicação e sua visualização final.

Para um melhor impacto visual, o tamanho do vetor e a quantidade de gerações foram parametrizados sob *macros*.

```
1 #include <iostream>
2 using namespace std;
3
4 #define TAMANHO 31
5 #define GERACOES 16
```

```
6
7  int main()
8  {
9      int gen0[TAMANHO] = {};
10     int gen1[TAMANHO] = {};
11
12     cout << "Regra 90 para " << TAMANHO
13           << " individuos em " << GERACOES
14           << " geracoes:" << endl << endl;
15
16     // Inicia o elemento central como vivo
17     gen0[TAMANHO / 2] = 1;
18
19     // Iterando para cada geração
20     for(int gen = 0; gen < GERACOES; gen++) {
21         // Imprime gen0 antes de aplicar regras
22         for(int i = 0; i < TAMANHO; i++) {
23             if(gen0[i] == 0) {
24                 cout << '.';
25             } else {
26                 cout << '@';
27             }
28         }
29         cout << endl;
30
31         // Iterando sobre cada elemento de gen0
32         for(int i = 0; i < TAMANHO; i++) {
33             int indice_esquerdo;
34             int indice_direito;
35
36             // Verifica os índices dos vizinhos
37             if(i == 0) {
```

```
38         indice_esquerdo = TAMANHO - 1;
39         indice_direito  = 1;
40     } else if(i == TAMANHO - 1) {
41         indice_esquerdo = TAMANHO - 2;
42         indice_direito  = 0;
43     } else {
44         indice_esquerdo = i - 1;
45         indice_direito  = i + 1;
46     }
47
48     // Calcula a vizinhança
49     int esq = gen0[indice_esquerdo];
50     int dir = gen0[indice_direito];
51
52     // Aplica a Regra 90,
53     // escrevendo-a em gen1
54     if((esq == 1 && dir == 0) ||
55        (esq == 0 && dir == 1)) {
56         gen1[i] = 1;
57     } else {
58         gen1[i] = 0;
59     }
60 }
61
62 // Copiando gen1 para gen0
63 for(int i = 0; i < TAMANHO; i++) {
64     gen0[i] = gen1[i];
65 }
66 }
67
68 return 0;
```


3.10. PROJETO DE PROGRAMAÇÃO I

Ao longo dos próximos capítulos, construiremos um projeto incremental em C++, que terá novas funcionalidades à medida que discutirmos novos aspectos da linguagem. Este projeto será baseado em autômatos celulares.

A Figura 27 a seguir descreve as "cláusulas" da regra de um autômato, bem como suas trinta e uma primeiras gerações (incluindo a *geração 0*). Este AC é conhecido como Regra 110.

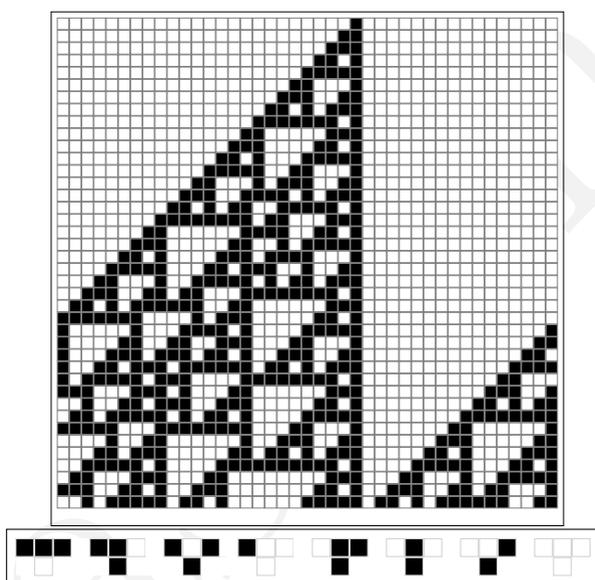


Figura 27 – Evolução e definição da Regra 110

O gráfico foi construído a partir de um estado inicial onde um vetor possuía todos os elementos no estado *morto*, exceto por um único elemento aproximadamente no meio, que possuía o estado *vivo*.

Ademais, foi utilizado um sistema de vizinhança *cilíndrico*, como nos exemplos demonstrados anteriormente.

A Regra 110 é conhecida por construir *estruturas localizadas*, assim como fractais, porém sob uma distribuição previsível. Este AC também é conhecido por ser equivalente a uma máquina de Turing, em termos de potência computacional.

Isto significa que, em tese, qualquer cálculo feito por um computador poderia ser feito também pela Regra 110. Porém, seu uso neste sentido requer esforço extra, já que um AC é uma ferramenta muito simples quando comparado a um computador pessoal.

Baseando-se nas informações da Figura 27, construa um programa que imprime na tela as primeiras gerações do autômato celular Regra 110 (incluindo a *geração 0*). Utilize o *estado inicial*, a *quantidade de gerações* e a *quantidade de células* que parecer mais conveniente.

4. MATRIZES

A utilização de vetores certamente é útil quando lidamos com o armazenamento de um certo número de variáveis do mesmo tipo. Vimos que o processo de armazenamento destas variáveis pode ser considerado um *meio de combinação* na linguagem C++.

Algumas vezes, o grande volume de dados a serem utilizados tornam inviável o armazenamento de valores do mesmo tipo em vetores. Considere, por exemplo, a seguinte situação:

Construa um programa que receba as porcentagens de notas de cinco provas dos alunos de uma turma de trinta pessoas. Calcule a média final para cada aluno e exiba a média final da sala.

Nesta situação, o objetivo inicial do programa não está em armazenar as médias finais de trinta alunos, mas sim em armazenar *cinco porcentagens de notas de provas, para trinta alunos*. Se pudéssemos modelar esta situação com um vetor, teríamos este vetor com *trinta* posições...

Alunos:

(notas)	(notas)	(notas)	(notas)	...	(notas)
---------	---------	---------	---------	-----	---------

...mas podemos constatar que cada posição no vetor também deveria desdobrar-se em mais *cinco* posições!

Uma alternativa seria armazenar todas as notas sequencialmente, e ter as notas de um único aluno a cada cinco posições, mas isto não confere uma boa separação nos dados.

Em contrapartida, poderíamos ter algum tipo de *tabela* para resolver este problema. Por exemplo, poderíamos supor uma *tabela* onde cada *linha* diz respeito a um aluno, e cada *coluna* diz respeito à nota de uma prova. Desta forma:

Aluno 1:	35%	58%	92%	27%	84%
Aluno 2:	22%	28%	91%	76%	86%
Aluno 3:	5%	4%	19%	24%	13%
...	...				
Aluno 30:	68%	85%	83%	37%	40%

Podemos, também, utilizar uma tabela propriamente dita para visualizar a distribuição destes dados:

	Prova 1	Prova 2	Prova 3	Prova 4	Prova 5
Aluno 1	35	58	92	27	84
Aluno 2	22	28	91	76	86
Aluno 3	5	4	19	24	13
...
Aluno 30	68	85	83	37	40

Matematicamente, apenas as porcentagens de notas são relevantes para acesso, então poderíamos utilizar uma estrutura conhecida como *matriz* para armazenar tais notas. Trinta alunos e cinco provas dão origem a uma *matriz* A de *trinta linhas* e *cinco colunas*:

$$A = \begin{bmatrix} 35 & 58 & 92 & 27 & 84 \\ 22 & 28 & 91 & 76 & 86 \\ 5 & 4 & 19 & 24 & 13 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 68 & 85 & 83 & 37 & 40 \end{bmatrix}$$

A linguagem C++ provê estruturas para a geração de *matrizes* como esta; elas também são conhecidas como *matrizes*, e funcionam como um *vetor de vetores*.

Para o caso apresentado, a ideia é que tenhamos algo similar ao esperado inicialmente: um *vetor* de *trinta* alunos, onde cada *posição* neste vetor "contém" um outro *vetor*.

4.1. DECLARANDO MATRIZES BIDIMENSIONAIS

Matrizes bidimensionais são estruturas que podem ser entendidas de duas formas:

- Uma estrutura com $m \times n$ elementos do mesmo *tipo*;
- Um *vetor* com m *sub-vetores*, cada qual com n elementos do mesmo *tipo*.

Isto significa que, para uma matriz bidimensional de 30×5 elementos, o elemento na posição $(15, 3)$ (com uma contagem iniciada por 0, assim como no vetor), indica a *quarta* nota (coluna) do *décimo-sexto* aluno (linha).

Podemos declarar esta matriz dessa forma:

```
1 int A[30][5];
```

Veja que usamos colchetes duas vezes: uma para declarar o número de *linhas*, outra para declarar o número de *colunas*. Assim, temos uma matriz A com 30 linhas e 5 colunas.

4.1.1. Acessando e atribuindo elementos à matriz

Acessar elementos em uma matriz é extremamente simples. Supondo que quiséssemos, de fato, acessar a *quarta* nota do *décimo-sexto* aluno na matriz A definida anteriormente, faríamos:

```
1 A[15][3];
```

Note que, primeiro, fornecemos o índice para a *linha* requerida, e então para a *coluna*.

Utilizando este mesmo tipo de expressão de acesso, assim como fazíamos com vetores, podemos atribuir valores a posições específicas do vetor, como podemos ver a seguir.

```
1 A[15][3] = 50;
```

4.1.2. Utilizando vetores individuais da matriz

Assumindo a matriz como um *vetor de vetores*, podemos aproveitar os *sub-vetores* como sendo elementos únicos em algumas situações.

Suponha, por exemplo, uma matriz 2×80 de elementos de tipo `char`. Podemos utilizar cada linha desta matriz como se fosse uma cadeia de caracteres individual.

No exemplo a seguir, as expressões de acesso `informacoes[0]` e `informacoes[1]` dizem respeito a *vetores de caracteres individuais*, portanto capazes de suportar uma *string* cada.

```
1 char informacoes[2][80];
2
3 std::cin >> std::ws;
4 std::cin.getline(informacoes[0], 80);
5
6 std::cin >> std::ws;
7 std::cin.getline(informacoes[1], 80);
8
9 std::cout << "Nome:      " << informacoes[0]
10           << std::endl
11           << "Sobrenome: " << informacoes[1]
12           << std::endl;
```

4.1.3. Inicializando a matriz na declaração

É possível inicializar uma matriz durante sua declaração, também. Vejamos a seguinte matriz de tamanho 2×2 como exemplo.

```
1 int A[2][2] = { {1, 2}, {3, 4} };
```

Partindo do preceito de que uma matriz pode ser compreendida como um *vetor de vetores do mesmo tamanho e tipo*, temos a matriz sendo primariamente inicializada como um vetor de elementos, onde cada um destes elementos em si é uma literal representando um vetor em si.

Veja que estes sub-vetores são constituídos por dois números *inteiros*, cada um.

É possível, também, omitir a quantidade de linhas da matriz, que será determinada na inicialização. Mas **apenas a quantidade de linhas pode ser omitida na declaração**, como vemos a seguir.

```
1 int A[][2] = { {1, 2}, {3, 4} };
```

Para inicializar uma matriz com zeros, basta informar uma literal vazia para cada sub-vetor.

```
1 int A[][2] = { {}, {} };
```

4.1.4. Matrizes de mais dimensões

É possível declarar matrizes com ainda mais dimensões em C++. Por exemplo, suponha uma situação onde precisássemos armazenar os dados de seis turmas, cada uma com trinta alunos, e cada aluno com cinco notas.

Isto exigiria uma matriz *tridimensional*, com índices para *turma*, *aluno* e *nota*.

Poderíamos compreender esta matriz *tridimensional* como um **vetor de tabelas bidimensionais**, onde cada tabela realiza a correspondência entre notas e alunos. A declaração seguiria algo nestas linhas gerais:

```
1 #define TURMAS 6
2 #define ALUNOS 30
3 #define NOTAS 5
4
5 int boletim[TURMAS][ALUNOS][NOTAS];
6
7 /* Mais linhas aqui... */
```

```
8
9 // Quarta turma (3), segundo aluno (1):
10 // Imprima a terceira nota (2)
11 std::cout << boletim[3][1][2] << std::endl;
```

4.2. ITERANDO SOBRE MATRIZES

Percorrer *matrizes* é uma operação extremamente similar a percorrer vetores. De fato, se ainda imaginarmos uma matriz bidimensional como sendo um *vetor de vetores*, podemos constatar que

- Precisamos percorrer o *vetor de vetores*, onde cada elemento será um *sub-vetor*;
- Para cada *sub-vetor*, precisamos percorrer seus *elementos*.

Esta dedução pode ser transformada em uma iteração *linha-a-linha*, que contém uma iteração *coluna-a-coluna*.

Em outras palavras, primeiro criamos um *laço de repetição* que percorre cada *linha da matriz*. Dentro deste laço, a variável *i* fixa o índice da linha em questão; ali, podemos percorrer cada *coluna da matriz*, cujo índice será *j*. Veja o código a seguir, que imprime todos os valores de uma matriz 4×5 .

```
1 // Matriz de quatro linhas e cinco colunas
2 int A[4][5] = {
3     {91, 67, 49, 74, 14},
4     {63, 37, 13, 41, 16},
5     {45, 3, 71, 42, 30},
6     {85, 95, 64, 80, 10},
7 };
8
```

```
9 // Percorrendo o "vetor de vetores", ou
10 // seja, cada linha
11 for(int i = 0; i < 4; i++) {
12     // Percorrendo o "sub-vetor", ou
13     // seja, cada coluna para a linha i
14     for(int j = 0; j < 5; j++) {
15         // Imprimimos cada elemento
16         std::cout << std::setw(2)
17             << A[i][j] << ' ';
18     }
19     // Pule uma linha após imprimir a
20     // linha da matriz
21     std::cout << std::endl;
22 }
```

```
91 67 49 74 14
63 37 13 41 16
45  3 71 42 30
85 95 64 80 10
```

4.3. EXERCÍCIOS DE FIXAÇÃO

Boletim Virtual III

Resolva o problema proposto no início do capítulo:

Construa um programa que recebe as porcentagens de notas de cinco provas dos alunos de uma turma de trinta pessoas. Calcule a média final para cada aluno e exiba a média final da sala.

Determinante 2×2

Suponha uma matriz 2×2 , como a matriz abaixo:

$$A = \begin{bmatrix} 5 & 9 \\ 7 & 3 \end{bmatrix}$$

O cálculo da *determinante* de uma matriz 2×2 envolve multiplicar os elementos da *diagonal principal*, e em seguida subtraímos desde resultado a multiplicação dos elementos da *diagonal secundária*.

Sabendo que o elemento na linha i e na coluna j da matriz A pode ser representado pela notação a_{ij} (onde i e j iniciam-se com 1), podemos computar a determinante de A de acordo com a equação:

$$\det(A) = (a_{11} \times a_{22}) - (a_{21} \times a_{12})$$

Assim, calculamos a determinante de A :

$$\det(A) = \begin{vmatrix} 5 & 9 \\ 7 & 3 \end{vmatrix} = (5 \times 3) - (7 \times 9) = -48$$

Crie um programa que lê uma matriz 2×2 digitada pelo usuário, e então calcula sua determinante.

Determinante 3×3

Suponha uma matriz 3×3 , como a matriz abaixo:

$$A = \begin{bmatrix} 97 & 2 & 53 \\ 80 & 78 & 70 \\ 92 & 9 & 13 \end{bmatrix}$$

O cálculo da *determinante* de uma matriz 3×3 envolve repetir as duas primeiras colunas da matriz ao final da mesma, criando a matriz estendida A' :

$$A' = \left[\begin{array}{ccc|cc} 97 & 2 & 53 & 97 & 2 \\ 80 & 78 & 70 & 80 & 78 \\ 92 & 9 & 13 & 92 & 9 \end{array} \right]$$

Sabendo que o elemento na linha i e na coluna j da matriz A' pode ser representado pela notação a'_{ij} , podemos computar a *determinante* de A segundo a equação:

$$\begin{aligned} \det(A) = & (a'_{11} \times a'_{22} \times a'_{33}) + (a'_{12} \times a'_{23} \times a'_{34}) + (a'_{13} \times a'_{24} \times a'_{35}) \\ & - ((a'_{31} \times a'_{22} \times a'_{13}) + (a'_{32} \times a'_{23} \times a'_{14}) + (a'_{33} \times a'_{24} \times a'_{15})) \end{aligned}$$

Crie um programa que lê uma matriz 3×3 digitada pelo usuário, e então calcula sua determinante.

4.4. PROJETO DE PROGRAMAÇÃO II

O projeto a seguir incrementa o que foi feito no projeto de programação da Seção 3.10.

O objetivo deste projeto é tomar a implementação da Regra 110 feita, e adicionar uma nova regra de autômatos celulares que possa ser impressa também. Esta nova regra será a Regra 150 (veja a Figura 28).

Adicionalmente, ao invés de imprimirmos o mesmo vetor linha a linha a cada evolução, vamos armazenar todo o histórico de evolução do Autômato Celular em uma *matriz*, e então imprimiremos esta matriz.

Este projeto será composto de quatro partes:

1. Incremente sua implementação da Regra 110, armazenando as gerações do autômato celular em uma *matriz*. Para tanto, trate a *primeira linha* da *matriz* como o *estado inicial*, e as

outras linhas como sendo a aplicação da regra do AC na linha anterior.

2. Programe a Regra 150 como alternativa à Regra 110. Uma ideia para isto é ter uma *variável* que controla qual das duas regras será aplicada. Assim, dentro da estrutura repetição, bastará verificar o valor da variável para determinar qual regra será aplicada. Use uma estrutura condicional para tal fim.
3. Pergunte, logo no início do programa, se o usuário deseja mostrar a Regra 110 **ou** a Regra 150, e então altere a *variável* sugerida apropriadamente.
4. Ao final do processamento da regra, imprima toda a matriz gerada.

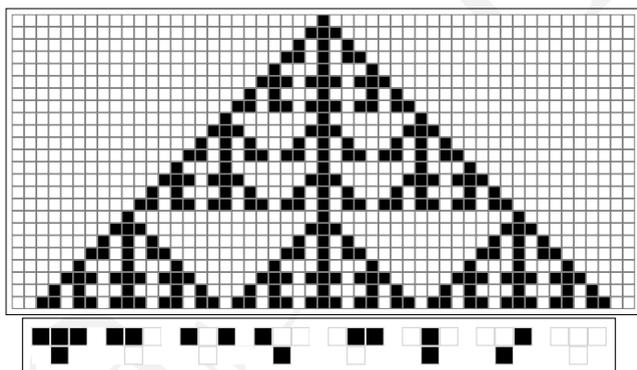


Figura 28 – Evolução e definição da Regra 150

5. TIPOS ABSTRATOS DE DADOS

Nos capítulos anteriores, vimos algumas formas de associar dados do mesmo *tipo*. Isto é, supondo situações onde temos uma uniformidade representacional para os dados: por exemplo, ainda que categorizemos notas entre turmas, alunos e provas, ainda são *notas*, representáveis como números reais.

Todavia, em algumas situações, precisamos lidar com tipos de dados *heterogêneos*, isto é, que precisam ser *agrupados*, mas que **não possuem o mesmo tipo**.

Considere, por exemplo, o cadastro de um aluno em uma escola. Este cadastro envolve vários tipos de dados. O aluno precisa de uma *matrícula* (normalmente numérica), possui um *nome* (cadeia de caracteres), idade (numérica), endereço (cadeia de caracteres)...

Também há uma situação interessante que pode ocorrer, onde temos uma quantidade seleta de *números inteiros* que são utilizados para identificar unicamente alguma coisa, ou seja, que agem como um *identificador*. Neste caso, é interessante que nomeemos estes números para uso dentro da aplicação; assim, fica muito mais fácil visualizar o significado de cada número.

Neste capítulo, abordaremos estas situações, e veremos *estruturas de dados abstratas* que lidam com este tipo de problema.

5.1. ESTRUTURA (STRUCT)

O primeiro *meio de abstração* que veremos é o chamado `struct`, também conhecido como *tipo abstrato de dados*.

O uso de *tipos abstratos* representa uma necessidade, quando precisamos agrupar informações que pertencem a uma mesma entidade. Caso não o fizéssemos, estas informações ficariam "espalhadas" no código, e seria difícil identificar a que entidade cada variável diz respeito.

A partir da definição de um `struct`, teremos uma "receita" para a criação de algo que assemelha-se a uma variável de um *tipo primitivo* mas, na verdade, representa diversos valores agrupados em uma só variável.

Em um *tipo abstrato de dados*, podemos reagrupar dados de vários tipos em uma única estrutura, constituindo um novo *tipo* que "funciona" como os tipos primitivos.

Na abertura deste capítulo, propusemos o cadastro de um tipo `Aluno`, que possui os seguintes dados:

- Matrícula (*numérica*);
- Nome (*string*);
- Idade (*numérica*).
- Endereço (*string*);

Podemos modelar o tipo `Aluno` através do uso de *structs*, desta forma:

```
1 struct Aluno
2 {
3     int matricula;
4     char nome[80];
```

```
5     int  idade;  
6     char endereco[80];  
7 };
```

Normalmente, `structs` como este devem ser declarados **fora da função principal**.

Vejamos o que significa cada parte desta definição.

```
1 struct Aluno
```

Esta linha determina o nome do *tipo abstrato* que estamos criando – neste caso, o *tipo abstrato* `Aluno`.

```
1 {  
2     ...  
3 };
```

O *escopo* de um `struct` é o local onde os componentes do mesmo são enumerados. Como o `struct` é um *meio de combinação* de dados primitivos da linguagem, neste escopo enumeramos elementos que serão considerados *membros* da estrutura. Cada elemento, portanto, poderá ser referenciado *por nome* quando precisarmos interagir com ele.

Mais tarde veremos este uso na prática.

Veja também que a definição do `struct` termina **obrigatoriamente** com ponto-e-vírgula.

```
1     int  matricula;  
2     char nome[80];  
3     int  idade;  
4     char endereco[80];
```

Cada "variável" declarada dentro do escopo é, na verdade, um *membro* da estrutura. Isto significa que cada *objeto*¹ do tipo `Aluno` criado deverá ter, dentro de si, uma *matricula*, um *nome*, uma *idade* e um *endereço*.

5.1.1. Criando uma variável a partir do *tipo abstrato*

Dada a definição do nosso *tipo abstrato* `Aluno`, criar uma variável para esta estrutura é extremamente simples. Veja o programa completo a seguir:

```
1  struct Aluno
2  {
3      int matricula;
4      char nome[80];
5      int idade;
6      char endereco[80];
7  };
8
9  int main()
10 {
11     Aluno a;
12
13     return 0;
14 }
```

Este programa não realiza nenhuma operação além de criar uma variável do tipo `Aluno`, na linha 11.

1. A palavra *objeto* costuma ser utilizada, no contexto da programação, como um elemento relacionado ao paradigma de Orientação a Objetos. Todavia, esta pauta não está no objetivo deste material. Portanto, toda e qualquer referência a "objeto" neste livro diz respeito a uma certa *abstração* que engloba vários membros, como é o caso do *tipo abstrato de dados*, e não ao "objeto" clássico da Orientação a Objetos.

A sintaxe para declaração mostrada acima cria uma variável chamada **a**, com o tipo **Aluno**. Esta sintaxe é muito similar à criação de qualquer variável.

Sendo assim, **a** é uma *instância* de um **Aluno**, que possui seus próprios componentes de **matricula**, **nome**, **idade** e **endereco**.

Quando criamos uma variável com um *tipo abstrato*, o conteúdo dos componentes, em sua inicialização normal, é *indefinido*², portanto não devemos contar com nenhum valor inicial nos componentes de **a**.

5.1.2. Acessando elementos de um *tipo abstrato*

Para acessar os elementos de um vetor, utilizamos um ponto (.) entre o *nome da variável* e o *nome do elemento*:

```
1 Aluno a;  
2  
3 // Lendo a idade do aluno "a":  
4 std::cin >> a.idade;
```

A sintaxe **a.idade** indica que queremos acessar o elemento **idade** na variável **a**.

O mesmo princípio poderia ser seguido para lermos o nome de um aluno com **std::cin.getline**. Veja:

```
1 Aluno aluno_1;  
2
```

2. A inicialização comum de um **struct**, na verdade, possui a mesma regra de inicialização de um vetor ou matriz: para uma variável estática (como as que estamos utilizando neste material), não há problema em assumir que ela será inicializada com "zeros". Todavia, trabalharemos com o pressuposto de que os valores são inicializados com conteúdo indefinido, uma vez que inicializar variáveis explicitamente é uma boa prática.

```
3 std::cin >> std::ws;
4 std::cin.getline(aluno_1.nome, 80);
```

Neste exemplo, acessamos o vetor `nome`, de 80 caracteres, que existe como elemento da variável `aluno_1`; neste vetor, armazenamos uma linha de caracteres lida através do console, como faríamos com qualquer *string*.

Mesmo a realização de *atribuições* ou *operações aritméticas* é extremamente simples com o uso da sintaxe baseada em ponto:

```
1 Aluno aluno_2;
2
3 aluno_2.idade = 25;
4
5 std::cout << "Sua proxima idade sera "
6           << aluno_2.idade + 1 << std::endl;
```

5.1.3. Inicializando um struct

A inicialização de um `struct` é muito similar à inicialização de um vetor: utilizamos uma notação com chaves (`{ }`) para atribuir os elementos. Veja:

```
1 struct Aluno
2 {
3     int matricula;
4     char nome[80];
5     int idade;
6     char endereco[80];
7 };
8
9 Aluno aluno_1 = { 123, "Fulano", 25, "Rua Alguma, 0" };
```

```
10
11 std::cout << "Dados do aluno:" << std::endl
12     << "Nome:      " << aluno_1.nome << std::endl
13     << "Matricula: " << aluno_1.matricula << std::endl
14     << "Idade:     " << aluno_1.idade << std::endl
15     << "Endereco:  " << aluno_1.endereco << std::endl;
```

```
Dados do aluno:
Nome:      Fulano
Matricula: 123
Idade:     25
Endereco:  Rua Alguma, 0
```

Note que a inicialização deve *respeitar a ordem de declaração dos elementos*. Sendo assim, como declaramos, em ordem:

1. `matricula`;
2. `nome`;
3. `idade`;
4. `endereco`

Devemos esperar, então, que o primeiro valor entre colchetes seja o número de matrícula; que o segundo seja o nome do aluno; e assim sucessivamente.

5.2. VETORES DE TIPOS ABSTRATOS

É interessante notar que, como `structs` agem, sintaticamente, como se fossem *tipos primitivos* (apesar de serem *tipos abstratos*), os mesmos podem ser portanto, reagrupados nos mesmos *meios de combinação* que vimos até agora (vetores, matrizes, novos tipos abstratos, e outros que veremos a seguir).

O exemplo a seguir enumera um *vetor de alunos*, onde cada aluno é uma instância de estrutura `Aluno`. Utilizamos a notação em chaves para inicializar o vetor e, para cada um dos elementos, utilizamos novamente a notação em chaves para iniciar cada estrutura.

```
1  struct Aluno
2  {
3      int matricula;
4      char nome[80];
5      int idade;
6      char endereco[80];
7  };
8
9  Aluno alunos[3] = {
10     { 123, "Fulano", 25, "Rua Alguma, 0" },
11     { 456, "Ciclano", 28, "Rua Alguma, 1" },
12     { 789, "Beltrano", 30, "Rua Alguma, 2" }
13 };
14
15 for(int i = 0; i < 3; i++) {
16     std::cout << "Dados do aluno " << i << ":"
17         << std::endl
18         << "Nome:      " << alunos[i].nome
19         << std::endl
20         << "Matricula: " << alunos[i].matricula
21         << std::endl
22         << "Idade:     " << alunos[i].idade
23         << std::endl
24         << "Endereco:  " << alunos[i].endereco
25         << std::endl << std::endl;
26 }
```

Dados do aluno 0:

Nome: Fulano
Matricula: 123
Idade: 25
Endereco: Rua Alguma, 0

Dados do aluno 1:

Nome: Ciclano
Matricula: 456
Idade: 28
Endereco: Rua Alguma, 1

Dados do aluno 2:

Nome: Beltrano
Matricula: 789
Idade: 30
Endereco: Rua Alguma, 2

`alunos` é, portanto, um *vetor de elementos do tipo Aluno*. Se referíssemos ao primeiro elemento do vetor (`alunos[0]`), estaríamos falando de *uma única estrutura* do tipo `Aluno`; o mesmo ocorreria para `alunos[1]`, e assim sucessivamente.

5.3. EXERCÍCIOS DE FIXAÇÃO I

Representando Frações

Frações são números racionais, definidos como números escritos na forma $\frac{a}{b}$, onde a é um *número inteiro*, e b é um *número inteiro diferente de zero*. Ou, matematicamente:

$$\mathbb{Q} = \left\{ \frac{a}{b} \mid (a \in \mathbb{Z}) \wedge (b \in \mathbb{Z}^*) \right\}$$

Construa um *tipo abstrato de dados* que representa *uma fração*.

Usando Frações

Utilizando o *tipo abstrato* criado no exercício anterior, crie uma variável com este *tipo abstrato*, receba do usuário seus respectivos valores para *numerador* e *denominador*, e em seguida, imprima a fração na tela, na forma `num/denom`.

5.4. ENUMERAÇÃO (ENUM)

Suponha que estivéssemos organizando os preços para os cinco tipos de comida diferentes de uma lanchonete:

- Cachorro- quente: R\$ 2,00;
- Pedaco de pizza: R\$ 5,00;
- Sorvete de chocolate: R\$ 1,50;
- X-Burguer: R\$ 12,00;
- Rosquinha: R\$ 4,00.

A forma mais trivial de armazenarmos estes valores é em um *vetor de cinco posições*, onde cada posição corresponde ao preço de um único produto, na sequência apresentada.

```
1 float prices[] = { 2.0f, 5.0f, 1.5f, 12.0f, 4.0f };
```

Para acessarmos e nos referirmos a um certo valor, realizamos, implicitamente, uma correspondência entre um *índice do vetor* e um produto, desta forma:

- 0 ⇒ Cachorro- quente;

- 1 ⇒ Pizza;
- 2 ⇒ Sorvete;
- 3 ⇒ X-Burguer;
- 4 ⇒ Rosquinha.

Todavia, esta informação, que é óbvia à primeira vista, poderia ficar *escondida no programa*, de forma que alguém com pouca experiência no código escrito pudesse não compreender a existência desta correspondência.

Para resolver este problema, criamos estruturas chamadas *enumerações*.

Em C++, *enumerações* nada mais são que um *pseudo-tipo abstrato*, que realiza a correspondência entre *números inteiros*, começando com 0, e certos *nomes*.

Para o compilador de C++, os *nomes* associados aos números são substituídos por estes mesmos números associados. Assim, o valor de uma *enumeração* está no uso sintático, e não na execução do programa em si.

5.4.1. Declarando enumerações

Para declararmos uma *enumeração*, usamos a sintaxe apresentada no bloco de código a seguir, em que implementamos uma estrutura `enum` para resolver o problema da lanchonete.

```
1 enum FOOD
2 {
3     FOOD_HOTDOG,      // Cachorro-quente
4     FOOD_PIZZA,      // Pizza
5     FOOD_ICECREAM,   // Sorvete
6     FOOD_CHEESEBURGER, // X-Burguer
```

```
7     FOOD_DONUT           // Rosquinha
8 };
```

Na primeira linha do código, estabelecemos o nome `FOOD` para a *enumeração*³. Este nome será usado da mesma forma que os *tipos abstratos de dados*, como uma espécie de *tipo*. Porém, teremos a consciência de quem um `enum` nada mais é que um `int` sintaticamente qualificado.

Note que, assim como no `struct`, o `enum` deve ser terminado com um ponto-e-vírgula após seu escopo, como demonstrado na linha 8.

As linhas 3 a 7 estabelecem, cada uma, um valor correspondente para cada elemento da enumeração. Sendo assim, temos `FOOD_HOTDOG` representando o cachorro-quente, `FOOD_PIZZA` representando a pizza, e assim sucessivamente.

É interessante ressaltar que **a ordem dos elementos da enumeração é importante**, uma vez que estes elementos terão um valor atribuído, começando em 0, relacionado à ordem em que aparecem. Em outras palavras, a ordem de declaração dos elementos da *enumeração* anterior torna-se esta:

- 0 ⇒ `FOOD_HOTDOG`;
- 1 ⇒ `FOOD_PIZZA`;
- 2 ⇒ `FOOD_ICECREAM`;
- 3 ⇒ `FOOD_CHEESEBURGER`;
- 4 ⇒ `FOOD_DONUT`.

3. Por convenção, utilizaremos nomes capitalizados para definir nomes para enumerações e também para seus elementos. Esta é uma decisão arbitrária e relacionada a estilo de código.

5.4.2. Usando enumerações

As enumerações possuem dois tipos de usos, servindo tanto como *pseudo-tipos* quanto como *constantes numéricas*.

Podemos criar enumerações como variáveis e atribuir valores⁴ a elas, desta forma:

```
1 FOOD comida = FOOD_ICECREAM;
```

Veja que o valor recebido pela variável do "tipo" `FOOD` é, necessariamente, um dos membros da enumeração.

Também seria possível realizar comparações por igualdade com a variável `comida`, inclusive utilizá-la em uma estrutura como `switch`, por exemplo.

Outro uso, mais interessante para o caso citado anteriormente, é o uso dos membros da enumeração como *constantes*. Vejamos como acessar, portanto, o preço de uma certa comida no vetor `prices`:

```
1 float prices[] = { 2.0f, 5.0f, 1.5f, 12.0f, 4.0f };
2
3 enum FOOD
4 {
5     FOOD_HOTDOG,
6     FOOD_PIZZA,
7     FOOD_ICECREAM,
8     FOOD_CHEESEBURGER,
9     FOOD_DONUT
10 };
11
12 std::cout << "Preço do sorvete: R$ "
```

4. Dito que a enumeração nada mais é que um tipo inteiro qualificado, é possível realizar aritmética com ela. Todavia, para efeitos de consistência sintática do código, evite fazer isto, a não ser que exista uma boa razão para tal.

```
13         << std::fixed << std::setprecision(2)
14         << prices[FOOD_ICECREAM] << std::endl;
```

Preço do sorvete: R\$ 1.50

Veja que a sintaxe `prices[FOOD_ICECREAM]` surte o mesmo efeito de `prices[2]`, uma vez que `FOOD_ICECREAM`, devido ao local de sua declaração no corpo da enumeração, é atribuído ao valor 2.

5.5. UNIÃO (UNION)

Unões são estruturas muito similares, sintaticamente, a *tipos abstratos de dados* (`struct`), porém são utilizadas não para garantir uma *separação entre membros*. Ao invés disso, uma *união* garante que **todos os seus elementos ocupem o mesmo espaço de memória**.

Devido a esta característica, é **incorreto** usar uma *union* como um armazenamento para *vários tipos de valores simultâneos*. Ao invés disso, devemos pensar na *union* como sendo capaz de armazenar *apenas um valor por vez*.

A sintaxe para declaração de uma *união* é a apresentada a seguir:

```
1 union GenericNumber
2 {
3     int    integer;
4     double real;
5     Complex complex;
6 };
```

A primeira linha do código apresenta o nome da *união* como sendo `GenericNumber`. Assim como o `struct` e o `enum`, esta estrutura também é encerrada com ponto-e-vírgula (vide a linha 6).

As linhas 3 a 5 mostram elementos da *união*, sendo eles um número do tipo `int`, um número do tipo `double`, e uma estrutura de tipo abstrato `Complex`, que será explicada mais tarde.

O acesso aos elementos da *união* é, sintaticamente, idêntico ao `struct`. Todavia, mais uma vez é importante lembrar que estas três estruturas *ocupam exatamente o mesmo espaço da memória*. Isto significa que realizar uma atribuição ao elemento `integer` também sobrescreveria os elementos `real` e `complex`, por exemplo.

A seguir, veremos um exemplo prático do uso desta *união*.

5.5.1. Exemplo de uso

Vejamos o exemplo a seguir, em que vamos construir um tipo genérico `Number`, capaz de abrigar três tipos **diferentes** de número.

Temos uma *enumeração* que identifica três tipos de números: *inteiro*, *real* e *complexo*.

```
1 enum NUMBER_KIND
2 {
3     INTEGER,
4     REAL,
5     COMPLEX
6 };
```

Números inteiros podem ser representados com `int`; reais, com `double`. Mas números complexos são compostos de dois números reais: uma parte *real* e uma parte *imaginária*. Podemos construir um número complexo desta forma:

```
1 struct Complex
2 {
3     double real_part; // Parte real
```

```
4     double imag_part; // Parte imaginária
5 };
```

Agora, construiremos nosso *número genérico*. Para tanto, precisamos nos assegurar de que *inteiro*, *real* e *complexo* ocupem **exatamente o mesmo espaço na memória**. Isto garante um certo nível de economia de memória:

```
1 union GenericNumber
2 {
3     int     integer;
4     double  real;
5     Complex complex;
6 };
```

Podemos criar, por último, mais um `struct`, que abriga duas coisas:

- Uma etiqueta indicando o tipo do número atual (`NUMBER_KIND`);
- A união dos números, constituindo o número genérico (`GenericNumber`).

```
1 struct Number
2 {
3     NUMBER_KIND  tag; // Etiqueta
4     GenericNumber g;  // União genérica
5 };
```

Podemos, também, criar um vetor de três elementos `Number`, e então inicializá-los manualmente:

```
1  Number numbers[3];
2
3  // Inicializando um inteiro
4  numbers[0].tag      = INTEGER;
5  numbers[0].g.integer = 5;
6
7  // Inicializando um real
8  numbers[1].tag      = REAL;
9  numbers[1].g.real   = 5;
10
11 // Inicializando um complexo
12 numbers[2].tag      = COMPLEX;
13 numbers[2].g.complex = {2.0, 5.0};
```

Finalmente, iteramos sobre cada elemento deste vetor, imprimindo-o de acordo com a etiqueta de tipo.

```
1  for(int i = 0; i < 3; i++) {
2      // Imprimindo o numero de acordo com a etiqueta
3      switch(numbers[i].tag) {
4          case INTEGER:
5              std::cout << "Imprimindo inteiro:" << std::endl;
6              std::cout << numbers[i].g.integer << std::endl;
7              break;
8          case REAL:
9              std::cout << "Imprimindo real:" << std::endl;
10             std::cout << numbers[i].g.real << std::endl;
11             break;
12          case COMPLEX:
13             std::cout << "Imprimindo complexo:" << std::endl;
14             std::cout << numbers[i].g.complex.real_part;
15             if(numbers[i].g.complex.imag_part >= 0.0) {
```

```
16         // Imprimir um '+' caso a parte imaginaria
17         // seja positiva
18         std::cout << '+';
19     }
20     std::cout << numbers[i].g.complex.imag_part
21         << 'i' << std::endl;
22     break;
23 default: break;
24 }
25 std::cout << std::endl;
26 }
```

5.5.2. Código completo do exemplo

Se utilizarmos todos os blocos de código anteriores, podemos instanciar números de diversos tipos de uma vez só. Veja:

```
1  enum NUMBER_KIND
2  {
3      INTEGER,
4      REAL,
5      COMPLEX
6  };
7
8  struct Complex
9  {
10     double real_part;
11     double imag_part;
12 };
13
14 union GenericNumber
15 {
```

```
16     int    integer;
17     double real;
18     Complex complex;
19 };
20
21 struct Number
22 {
23     NUMBER_KIND tag;
24     GenericNumber g;
25 };
26
27 Number numbers[3];
28
29 numbers[0].tag      = INTEGER;
30 numbers[0].g.integer = 5;
31
32 numbers[1].tag      = REAL;
33 numbers[1].g.real   = 2.0;
34
35 numbers[2].tag      = COMPLEX;
36 numbers[2].g.complex = {7.96, 30.0};
37
38 for(int i = 0; i < 3; i++) {
39     switch(numbers[i].tag) {
40     case INTEGER:
41         std::cout << "Imprimindo inteiro:" << std::endl;
42         std::cout << numbers[i].g.integer << std::endl;
43         break;
44     case REAL:
45         std::cout << "Imprimindo real:" << std::endl;
46         std::cout << numbers[i].g.real << std::endl;
```

```
47         break;
48     case COMPLEX:
49         std::cout << "Imprimindo complexo:" << std::endl;
50         std::cout << numbers[i].g.complex.real_part;
51         if(numbers[i].g.complex.imag_part >= 0.0) {
52             std::cout << '+';
53         }
54         std::cout << numbers[i].g.complex.imag_part
55             << 'i' << std::endl;
56         break;
57     default: break;
58 }
59 std::cout << std::endl;
60 }
```

Imprimindo inteiro:

5

Imprimindo real:

2

Imprimindo complexo:

7.96+30i

5.6. EXERCÍCIOS DE FIXAÇÃO II

Boletim Virtual IV

Construa um programa que recebe dados de uma turma de dez alunos, incluindo:

- O nome de cada aluno;

- A matrícula de cada aluno;
- A idade de cada aluno;
- As notas das cinco provas realizadas por cada aluno.

Ao final da coleta dos dados, imprima apenas os nomes de todos os alunos, e então calcule e mostre na tela a média de notas da turma.

Contagem de Vogais

Crie um programa que recebe um texto digitado pelo usuário, e verifique a quantidade de cada vogal existente no texto.

6. CONSIDERAÇÕES FINAIS

A ser feito em breve.

7. SOLUÇÕES

Abaixo, apresentamos as soluções para os problemas propostos nos capítulos anteriores.

Estas soluções não buscam ser a solução ótima para os problemas anteriores; ao invés disso, tomamos como base o conhecimento do leitor até tal ponto, e então elaboramos um código capaz de ser compreendido pelo mesmo.

A elaboração de soluções otimizadas ou mais sucintas permanecem como exercício para o leitor.

7.1. CAPÍTULO 1

Exercícios de Fixação I

- Somando números

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      double a, b;
7
```

```
8     cout << "Digite um numero: ";
9     cin >> a;
10
11    cout << "Digite outro numero: ";
12    cin >> b;
13
14    double resultado = a + b;
15
16    cout << a << " + " << b
17         << " = " << resultado << endl;
18
19    return 0;
20 }
```

- Resto da divisão por dois

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int numero;
7      cout << "Digite um numero: ";
8      cin >> numero;
9
10     cout << "O resto da divisao de " << numero
11          << " por dois eh " << (numero % 2)
12          << endl;
13
14     return 0;
15 }
```

- Área do círculo

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  int main()
6  {
7      double raio;
8
9      cout << "Insira o raio do círculo: ";
10     cin >> raio;
11
12     double area = M_PI * raio * raio;
13
14     cout << "Area: " << area << endl;
15
16     return 0;
17 }
```

- Média aritmética

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      double num1, num2, num3;
7
8      cout << "Digite tres numeros reais: ";
9      cin >> num1 >> num2 >> num3;
10 }
```

```
11     double media = (num1 + num2 + num3) / 3.0;
12
13     cout << "Media: " << media << endl;
14
15     return 0;
16 }
```

- Volume da esfera

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  int main()
6  {
7      double raio;
8
9      cout << "Insira o raio da esfera: ";
10     cin >> raio;
11
12     double volume =
13         (4.0 * M_PI * (raio * raio * raio)) / 3.0;
14
15     cout << "Volume: " << volume << endl;
16
17     return 0;
18 }
```

Exercícios de Fixação II

- Par ou ímpar

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      int numero;
7
8      cout << "Insira um numero: ";
9      cin >> numero;
10
11     if(numero % 2 == 0) {
12         cout << "O numero eh par" << endl;
13     } else {
14         cout << "O numero eh impar" << endl;
15     }
16
17     return 0;
18 }
```

- Boletim Virtual I

```
1  #include <iostream>
2  using namespace std;
3
4  int main()
5  {
6      float nota;
7
8      cout << "Insira a nota do aluno: ";
9      cin >> nota;
10
11     if(nota < 0.0f || nota > 100.0f) {
```

```
12     cout << "Nota invalida!" << endl;
13     return 1;
14 }
15
16 if(nota < 60.0f) {
17     cout << "Aluno reprovado" << endl;
18 } else {
19     cout << "Aluno aprovado" << endl;
20 }
21
22 return 0;
23 }
```

- Dias da semana

```
1 #include <iostream>
2 using namespace std;
3
4 int main()
5 {
6     int mes;
7
8     cout << "Insira o numero do mes: ";
9     cin >> mes;
10
11     switch(mes) {
12     case 1:
13         cout << "janeiro" << endl;
14         break;
15     case 2:
16         cout << "fevereiro" << endl;
17         break;
```

```
18     case 3:
19         cout << "marco" << endl;
20         break;
21     case 4:
22         cout << "abril" << endl;
23         break;
24     case 5:
25         cout << "maio" << endl;
26         break;
27     case 6:
28         cout << "junho" << endl;
29         break;
30     case 7:
31         cout << "julho" << endl;
32         break;
33     case 8:
34         cout << "agosto" << endl;
35         break;
36     case 9:
37         cout << "setembro" << endl;
38         break;
39     case 10:
40         cout << "outubro" << endl;
41         break;
42     case 11:
43         cout << "novembro" << endl;
44         break;
45     case 12:
46         cout << "dezembro" << endl;
47         break;
48     default:
```

```
49     cout << "Mes invalido!" << endl;
50     break;
51 };
52
53     return 0;
54 }
```

- Distância entre dois pontos

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  int main()
6  {
7      double x1, y1, x2, y2;
8
9      cout << "Insira as coordenadas do primeiro ponto: ";
10     cin >> x1 >> y1;
11
12     cout << "Insira as coordenadas do segundo ponto: ";
13     cin >> x2 >> y2;
14
15     double deltaX = x2 - x1;
16     double deltaY = y2 - y1;
17
18     double resultado =
19         sqrt((deltaX * deltaX) + (deltaY * deltaY));
20
21     cout << "Distancia entre os pontos: " << resultado
22         << endl;
23 }
```

```
24     return 0;  
25 }
```

- Conversão de tempo

```
1  #include <iostream>  
2  using namespace std;  
3  
4  int main()  
5  {  
6      int segundos;  
7      cout << "Insira os segundos: ";  
8      cin >> segundos;  
9  
10     int horas, minutos;  
11     int buffer;  
12  
13     // Tomando o número de horas  
14     // nos segundos  
15     horas    = segundos / 3600;  
16     // Removemos os segundos das  
17     // horas cheias  
18     segundos -= horas * 3600;  
19  
20     // Mesmo procedimento para minutos  
21     minutos  = segundos / 60;  
22     segundos -= minutos * 60;  
23  
24     cout << horas << ':' << minutos  
25           << ':' << segundos << endl;  
26
```

```
27     return 0;
28 }
```

- Equações de segundo grau

```
1  #include <iostream>
2  #include <cmath>
3  using namespace std;
4
5  int main()
6  {
7      double a, b, c;
8
9      cout << "Insira valores para a, b e c: ";
10     cin >> a >> b >> c;
11
12     double delta = (b * b) - (4.0 * a * c);
13
14     if(delta < 0.0) {
15         cout << "A equacao nao possui raizes reais!"
16             << endl;
17         return 1;
18     }
19
20     double x1, x2;
21     x1 = ((-b) + sqrt(delta)) / (2.0 * a);
22     x2 = ((-b) - sqrt(delta)) / (2.0 * a);
23
24     cout << "Raizes:" << endl
25         << "x' = " << x1 << endl
26         << "x'' = " << x2 << endl;
27 }
```

```
28     return 0;  
29 }
```

7.2.

Rascunho

A. LICENCIAMENTO DE CÓDIGO

A licença a seguir diz respeito a todos os códigos contidos neste livro. O livro-texto em si é redistribuído sob uma licença Creative Commons BY-NC-ND 4.0 Internacional, mas o código aqui contido possui um grau extra de liberdade para que o leitor possa reutilizá-lo para outros fins, mediante algumas condições.

Segue, abaixo, a licença permissiva que regulamenta a reutilização do código contido neste livro.

BSD 2-Clause License

Copyright (c) 2019, Lucas Samuel Vieira
All rights reserved.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

B. USANDO O GNU DEBUGGER

O *GNU Debugger* (GDB) é um programa extremamente útil durante a identificação de problemas no código, sobretudo quando estes problemas ocorrem durante a execução do programa, e quando fazem com que o programa se encerre de forma inesperada.

Os exemplos a seguir mostram rapidamente algumas operações que podem ser feitas através do GDB. Para tanto, foi criado um programa propositalmente com problemas.

O primeiro passo é executar o programa (de preferência, compilado com a *flag -g*), usando o seguinte comando:

```
1 $ gdb ./programa
```

Serão exibidas informações iniciais do Debugger, e então um prompt esperará por próximos comandos. Por exemplo:

```
$ gdb ./meu_programa
GNU gdb (GDB) 8.3.1
Copyright (C) 2019 Free Software Foundation, Inc.
[...texto omitido...]
For help, type "help".
Type "apropos word" to search for commands related to "word"...
```

```
Reading symbols from ./meu_programa..
```

```
(gdb)
```

O programa em questão não é executado imediatamente, portanto é importante que o comando `start` seja digitado.

O Debugger parará em um *ponto de parada* (*breakpoint*), antes da execução da primeira linha de código. Veja:

```
(gdb) start
```

```
Temporary breakpoint 1 at 0x117d: file main.cpp, line 7.
```

```
Starting program: /home/alchemy/foe/meu_programa
```

```
Temporary breakpoint 1, main () at main.cpp:7
```

```
7   int *a = (int*)87346826;
```

```
(gdb)
```

Podemos continuar a execução digitando o comando `continue`.

```
(gdb) continue
```

```
Continuing.
```

B.1. DIAGNÓSTICO

Em algum ponto da execução da aplicação, pode ser que o programa mostre um erro e seja encerrado inesperadamente. No exemplo a seguir, isto aconteceu com um programa.

```
Program received signal SIGSEGV, Segmentation fault.
```

```
__GI___libc_free (mem=0x534ce8a) at malloc.c:3102
```

```
3102 malloc.c: Arquivo ou diretório inexistente.
```

```
(gdb)
```

Neste caso, o Debugger voltará a pedir por mais um comando, e você poderá realizar diagnóstico manual. Mostraremos a seguir algumas das ferramentas a serem utilizadas em um diagnóstico.

B.1.1. Backtrace

O *backtrace* mostra o caminho e a execução do código problemático até o ponto de parada. Para mostrá-lo, digite o comando `bt` ou `backtrace` no prompt do Debugger.

O exemplo a seguir mostrar uma situação em que um arquivo `main.cpp` possui problemas na linha 8:

```
(gdb) bt
#0  __GI___libc_free (mem=0x534ce8a) at malloc.c:3102
#1  0x000055555555519b in main () at main.cpp:8
(gdb)
```

B.1.2. Variáveis locais

Podemos mostrar informações locais (incluindo variáveis, a serem discutidas posteriormente) durante um ponto de parada qualquer do Debugger. Para tanto, digite o comando `info locals` no prompt do mesmo. Exemplificando:

```
(gdb) info locals
ar_ptr = <optimized out>
p = <optimized out>
hook = 0x0
(gdb)
```

B.2. SAINDO DO GNU DEBUGGER

Para sair do Debugger, utilize o comando `quit`. Ele pedirá por uma confirmação, caso o programa executado não tenha sido finalizado corretamente. Basta confirmar com `y`.

```
(gdb) quit
```

A debugging session is active.

Inferior 1 [process 20313] will be killed.

Quit anyway? (y or n) y

C. GERAÇÃO DE AUTÔMATOS CELULARES

O código a seguir foi utilizado para a geração de figuras para autômatos celulares unidimensionais. Este código poderá ser reutilizado para novas gravuras.

Este código foi escrito na linguagem APL, criada por Kenneth E. Iverson em 1966. Seu propósito é conceder ao programador um vocabulário sucinto para manipular *vetores* e *matrizes*.

```
rule90←{>1=+/-1 1φ''cω}

fn←rule90
gen←{(fn*α)ω}
plotgen←{r←ω◇(=, /α (ρr))ρ>, /{ω gen r}''-1+ια}
drawfig←{'_.'[1+ω]}

findrule←{1+>+ /((-α-(⌊α÷2))+ια)φ''(φ2*-1+ια)*''cω}
findrule3←3◦findrule
do_rule3←{(findrule3 α)[]''cφ(ι8)εω}

rule110←{ω do_rule3 2 3 5 6 7}
rule150 ← {ω do_rule3 1 4 6 7}
```

fn+rule90 ◊ drawfig 32 plotgen 21=141

fn+rule110 ◊ drawfig 40 plotgen 25=141

fn+rule150 ◊ drawfig 24 plotgen 26=151

Referências Bibliográficas

ABELSON, H.; SUSSMAN, G. J.; SUSSMAN, J. **Structure and Interpretation of Computer Programs**. Cambridge: MIT Press, 1996. ISBN 978-0-262-51087-5. 120

MITCHELL, M. **Complexity: A guided tour**. New York: Oxford University Press, 2009. ISBN 978-0-19-979810-0. 146

WOLFRAM, S. **A New Kind of Science**. [S.l.]: Wolfram Media, Inc., 2002. 500 p. ISBN 1-57955-008-8. 144, 145