UNIVERSIDADE FEDERAL DO RIO GRANDE DO NORTE
ESCOLA DE CIÊNCIAS E TECNOLOGIA
CAMPUS CENTRAL

Apostila de Programação em C++

Para os estudantes da ECT

Sandro Bruno do Nascimento Lopes

Natal - RN
Dezembro de 2015

PUBLISHED BY PUBLISHER

BOOK-WEBSITE.COM

First printing, March 2013

Prefácio

Talvez você se pergunte “porque pagar uma disciplina de linguagem de programação, algo aparentemente tão específico (e chato para alguns), em um curso tão geral como o de ciência e tecnologia?”. A resposta para tal questão é simples: os programas utilizados em computadores são feitos com alguma linguagens de programação. E são os programas que possibilitam as pessoas utilizarem os computadores para escrever textos, jogar, ver vídeos ou mandar mensagens em redes sociais. De fato, você está lendo este text graças a um programa de computador. O próprio sistema operacional de um computador é, essencialmente, um programa que gerencia todos os elementos físicos de um computador.

Então, percebe-se que é algo bem importante para você diretamente. No entanto, programas afetam indiretamente as pessoas também. Quando se saca dinheiro em um caixa eletrônico, se recebe um produto de uma compra *on-line* ou se acessa o sistema de gestão acadêmico da universidade, na verdade se utiliza uma dentre inúmeras operações disponibilizadas por sistemas de gerenciamento de dados. Estes, por sua vez, envolvem vários programas, feitos em uma ou várias linguagens de programação (isto mesmo: existem várias linguagens de programação no mercado).

Mas a relação das pessoas com os programas vai além dos sistemas de gerenciamento de dados. A previsão do tempo envolve programas de análise estatística que rodam em supercomputadores. A construção de um carro conta com vários tipos de programas: para gerenciamento de insumos, da produção em chão de fábrica, da logística, entre outros. Programas são utilizados para obter e processar dados de pacientes a partir de sensores, na busca por diagnósticos mais rápidos e confiáveis. Processos físicos, químicos e biológicos podem ser simulados em programas de computador antes de serem realmente reproduzidos. O acesso a informação por meio da internet é continuamente aperfeiçoado com novos programas, para que o usuário se informe cada vez mais rápido e mais iterativamente...

Neste ponto, você já entendeu que tem muita coisa envolvendo linguagem de programação no a sua volta. Na verdade, já não é possível imaginar como seria a humanidade sem a existência dos computadores, especificamente dos programas de computadores. Mas, existe um detalhe que não é tão intuitivo neste texto: linguagens de programação permitem fazer muitas atividades que seriam difíceis de serem realizadas, mesmo para vários seres humanos. E a manipulação de uma

ferramenta como uma linguagem de programação pode capacitar um profissional a resolver um problema de forma mais rápida e eficiente, bastando preparar um programa de computador para isto. É este o maior intuito de uma disciplina desta neste curso de ciência e tecnologia: torná-lo hábil a resolver problemas de maneira rápida e eficiente pelo computador, seja para atender a uma necessidade de onde trabalha, seja para você abrir sua própria empresa, seja para fazer um concurso público.

Por isto, aproveite. :)

Conteúdo

I Parte 1: Introdução

1 Introdução a Linguagem C++ 11

1.1 O que é uma Linguagem de Programação? 11

1.1.1 O que manipula uma Linguagem de Programação? 13

1.2 Sobre a Linguagem de Programação C++ 14

1.3 Primeiro programa em C++ 15

1.3.1 Espaços em Branco e Indentação 16

1.3.2 Comentários em um código-fonte 16

1.3.3 Saída de Dados em C++ 17

1.4 Exercícios 18

2 Tipos, Variáveis, Constantes e Operadores 19

2.1 Variáveis 19

2.1.1 Tipos e Modificadores de Tipos 20

2.1.2 Declaração de Variáveis 23

2.1.3 Atribuição de Variáveis e Operador de Atribuição 24

2.1.4 Inicialização de Variáveis 25

2.2 Entrada de Dados em C++ 26

2.3 Constantes 27

2.4 Operadores 28

2.4.1 Expressões Aritméticas e Operadores Aritméticos 29

2.4.2 Expressões Lógicas 32

2.5 Conversão de Tipos 35
2.6 Operador `sizeof` 37
2.7 Exercícios 38

3 Comandos de Seleção 43
3.1 Comando `if` 43
3.2 Comando `if-else` 45
3.2.1 Encadeamento de `if-else` 48
3.3 Comando `switch-case` 49
3.4 Operador Ternário: `?  :` 51
3.5 Dicas Gerais 52
3.6 Exercícios 53

4 Comandos de Repetição 57
4.1 Introdução 57
4.2 Comando `for` 57
4.3 Comando `while` 61
4.4 Comando `do-while` 64
4.5 Comandos `break` e `continue` 65
4.6 Sobre os comandos de repetição 68
4.7 Exercícios 70

II Parte 2: Funções

5 Funções: Básico 79
5.1 Introdução 79
5.2 Chamada de Funções 80
5.2.1 Definição de Funções 80
5.3 Assinatura de uma Função 80
5.3.1 Consulta de assinaturas das Bibliotecas STL (C++) 82
5.4 Definição de uma Função 84
5.5 Escopo de Variáveis 86
5.6 Funções sem Retorno 88
5.7 Exercícios 89

6 Funções: Passagem por Referência 93
6.1 Introdução 93
6.2 Funções que chamam outras Funções 97
6.2.1 Modularização com arquivos `.h` 99
6.3 Exercícios 102

7 Recursão 107
7.1 Introdução 107
7.2 Recursão indireta 115
7.3 Exercícios 117

III Parte 3: Blocos de Memória

8 Vetores 121
8.1 Introdução 121
8.1.1 Indexação de Vetores 122
8.1.2 Inicialização de Vetores 125
8.2 Funções com Vetores 126
8.3 Ordenação 128
8.4 Exercícios 132

9 Matrizes 137
9.1 Introdução 137
9.2 Definição e Indexação de Matrizes 137
9.3 Inicialização de Matrizes 140
9.4 Funções com Matrizes 141
9.5 Exercícios 143

10 Strings 147
10.1 Introdução 147
10.2 Inicialização de uma *string* 148
10.3 Impressão e Leitura de *strings* 149
10.4 Uso de *strings* em funções 150
10.5 Tratamento de Texto com *strings* 153
10.6 Exercícios 156

IV Parte 4: Tipos Derivados

11 Tipos Estruturados 161
11.1 Introdução 161
11.2 Definição e uso de tipos estruturados 163
11.3 Funções com tipos estruturados 166
11.4 Exercícios 168

V Tópicos Especiais

12 Números Aleatórios 173
12.1 Introdução 173
12.2 Exercícios 176

13 Arquivos 177
13.1 Introdução 177
13.2 Operações com Arquivos 178
13.3 Exercícios 180

VI Bibliografia

Parte 1: Introdução

1 Introdução a Linguagem C++ 11
1.1 O que é uma Linguagem de Programação?
1.2 Sobre a Linguagem de Programação C++
1.3 Primeiro programa em C++
1.4 Exercícios

2 Tipos, Variáveis, Constantes e Operadores 19
2.1 Variáveis
2.2 Entrada de Dados em C++
2.3 Constantes
2.4 Operadores
2.5 Conversão de Tipos
2.6 Operador `sizeof`
2.7 Exercícios

3 Comandos de Seleção 43
3.1 Comando `if`
3.2 Comando `if-else`
3.3 Comando `switch-case`
3.4 Operador Ternário: `? :`
3.5 Dicas Gerais
3.6 Exercícios

4 Comandos de Repetição 57
4.1 Introdução
4.2 Comando `for`
4.3 Comando `while`
4.4 Comando `do-while`
4.5 Comandos `break` e `continue`
4.6 Sobre os comandos de repetição
4.7 Exercícios

1. Introdução a Linguagem C++

Trabalhar com uma linguagem de programação implica em instruir um computador a realizar uma determinada tarefa. Embora possa parecer uma associação grosseira, é possível pensar em instruir um computador de maneira semelhante ao que se faz para uma pessoa. Isto porque quando se deseja instruir uma pessoa a fazer uma determinada atividade, é necessário explicar claramente o passo-a-passo da execução de atividade. Mas do que isto, cada etapa desta atividade deve ser possível de ser realizada pela pessoa. Esta ideia também serve para o computador, com uma restrição importante: o passo-a-passo não pode ser ambíguo ou dependente de interpretações próprias da máquina, pois ele será executado exatamente como solicitado.

1.1 O que é uma Linguagem de Programação?

A princípio, uma **Linguagem de Programação** pode ser visto como um conjunto de termos válidos e regras de formação que permite montar instruções executáveis em um computador. Esta é uma noção parecida com a que temos de uma linguagem como português e inglês, por exemplo. Porém, ao contrário destas linguagens, as instruções geralmente são descritas em um sistema próprio de codificação que irá depender do equipamento.

No caso específico dos computadores digitais, o tipo mais comum de computador, as instruções são utilizadas para acionar circuitos lógicos booleanos, que são baseados no sistema de numeração binário. Este tipo de sistema é composto apenas por dois dígitos: 0 e 1. Qualquer informação codificada neste sistema será uma palavra de 0s e 1s. Este tipo de linguagem é comumente conhecido como *linguagem de máquina*, sendo o nível mais baixo e direto de iteração com o computador.

```
0000100100000001
0000101000000001
0001000100000000
0001001000000001
0011000000010000
0001100000000001
```

Linguagens de máquina geralmente não são manipuladas diretamente pelos programadores, pela dificuldade de leitura e dependência do equipamento (computadores de modelos diferentes podem ter linguagens de máquina diferentes). Ao invés disto, as operações disponíveis pelo computador geralmente são associadas a palavras que as descrevem, e valores numéricos e lógicos envolvidos são descritos no sistemas decimal ou hexadecimal. Este último possui 16 dígitos: de 0 a 9 e de A até F). Este tipo de linguagem é denominada **linguagem *assembly***, e sua disposição facilita a leitura do programa, porém exige uma etapa intermediária de tradução da descrição textual para a correspondente palavra em binário.

```
write 1 01
write 2 01
move 0 1
move 1 2
sum 0 1 2
move 2 01
```

A programação em linguagem *assembly* geralmente é empregada na instrução de pequenos sistemas. Na maioria dos casos, as linguagens empregadas possuem instruções mais elaboradas, cuja descrição e construção é mais próxima da linguagem do ser humano (geralmente inglês), porém com maiores restrições, para evitar ambiguidades. Além disto, estas instruções permitem manipular vários recursos do computador de uma vez. Este conjunto é denominado genericamente de **linguagens de alto nível**, em oposição a linguagem *assembly*, considerada baixo nível.

```
cout << 1 + 1 << endl;
```

A classificação em alto nível e baixo nível está associados a conversão para a linguagem de máquina, que ainda precisa ser feita. No caso das linguagens de alto nível, o processo torna-se bem mais elaborada, pois uma instrução de alto nível pode envolver muitas palavras de linguagem de máquina. Tipicamente, este processo pode ser feito de duas maneiras:

- **Compilação:** As instruções envolvidas são convertidas diretamente em linguagem de máquina;
- **Interpretação:** As instruções envolvidas são convertidas em uma linguagem intermediária, e desta para linguagem de máquina.

Não há um nível ideal de abstração para se resolver todos os problemas que envolvem computadores. Por causa disto, muitas linguagens de programação foram desenvolvidas e estão disponíveis na literatura. Cada uma delas atende a um determinada área de aplicação, como computação web, científica, de sistemas embarcados. No entanto, em todas elas, o conjunto de instruções é convertido para um conjunto de instruções em linguagem de máquina. Este último constitui um **programa** ou **aplicação** em computador. Na figura 1.1, é possível ver algumas existentes.

Figura 1.1: Algumas Linguagens de Programação existentes.

1.1.1 O que manipula uma Linguagem de Programação?

Basicamente, uma linguagem de programação deve permitir ao programador acesso a todos os componentes do computador de maneira confortável. Na prática, nem todos os elementos de um computador são disponibilizados ao programador, por uma questão de segurança. Existem vários tipos de componentes associados a um computador que costumam ser explorados em programas de computador. Eles podem ser resumidamente descritos a seguir:

- **Processador:** Elemento principal de um computador, é nele que são processadas e administradas as informações oriundas de outras partes do computador. É composto por duas partes. A primeira é a unidade lógica-aritmética (ULA), onde as operações são realizadas. A outra parte é a unidade de controle, onde as instruções de máquina estão armazenadas. Além destes elementos, ainda são disponibilizados registradores, que são elementos de memória de acesso muito rápido, porém com capacidade bem reduzida. Quando se fala em computadores *multicore*, fala-se em computadores com mais de um processador em um único circuito integrado;
- **Memória principal:** Dispositivo utilizado para armazenar dados utilizados durante a execução de um programa. Possui um processo rápido de leitura e escrita de dados, porém volátil - os dados são perdidos com o desligamento do computador;
- **Memória secundária:** Dispositivo utilizado para armazenar grandes quantidades de dados que podem ser utilizados posteriormente por um ou mais programas, como os arquivos. As operações de leitura e escrita de dados associada a estes são bem mais lentas, porém são persistentes - o dados armazenados são preservados mesmo depois do computador ser desligado. Inclui os elementos de armazenamento eletro-mecânicos (como o discos rígidos), os dispositivos ópticos (CDs, DVDs, Blue Rays) e os dispositivos baseados em memória Flash (como os *pendrives*);

- **Dispositivos de entrada:** São aqueles utilizados para receber dados do usuário e incluem periféricos clássicos como teclado, *mouse* e telas *touchscreen* (no caso de *smarthphones* e *tablets*). No entanto, entram nesta lista outros elementos como microfones, câmeras, *scanners* e leitores de códigos de barra;
- **Dispositivos de saída:** São aqueles utilizados para exibir dados para o usuário. O mais conhecido deles é o monitor de vídeo. No entanto, existem outros periféricos de saída bastante utilizados, como projetor de vídeo, caixas de som, fones de ouvido e impressora;

Além dos dados, todas as instruções e elementos de controle utilizadas em um programa são armazenadas na memória principal, e salvas em um arquivo na memória secundária. O gerenciamento da região da memória disponibilizado para cada item é feita pelo sistema operacional.

1.2 Sobre a Linguagem de Programação C++

C++ foi desenvolvida por Bjarne Stroustrup na década de 1980 como uma evolução da linguagem C. Mas especificamente, ele incorpora boa parte da estrutura de C e acrescenta mecanismos para programação orientada a objetos. Este último não será visto neste material, embora exista uma extensa bibliografia sobre ele. Ao invés disto, será tratada apenas a parte mais próxima de C, especialmente um modelo de programação denominada modular, que utiliza funções e tipos estruturados.

Primeiramente, C++ é uma linguagem compilada. Isto significa que ela é convertida diretamente para linguagem de máquina. Para isto, primeiramente é preciso construir um arquivo com as instruções em linguagem C++. Este é denominado **arquivo-fonte**, e pode ser caracterizado pela extensão `.cpp`. A partir dele, um programa denominado compilador faz a conversão do código-fonte para a linguagem de máquina. Teoricamente, ao final do processo, é gerado um programa em código binário que pode ser executado. A figura 1.2 mostra o processo simplificado de compilação.

Figura 1.2: Processo simplificado de compilação.

Na prática, a conversão do código-fonte para o arquivo binário envolve outros programas, além do compilador:

- **Pre-processador:** Programa que automaticamente realiza operações especificas antes da compilação. Estas operações tipicamente são determinadas pelo programador através do uso de diretivas de pré-processamento e costumam envolvem a manipulação de arquivos ou dados;
- **Conector** ou ***Linker***: Programa que adiciona informações não reconhecidas pelo compilador por serem fornecidas por outras fontes. Geralmente, são provenientes de **bibliotecas**, conjunto de códigos-fonte desenvolvidos pelo próprio programador ou por terceiros disponíveis em outros arquivos.

O esquema completo de geração do arquivo binário pode ser visto na figura 1.3.

O fato de C++ incorporar orientação a objetos em sua sintaxe a torna uma linguagem bastante poderosa. Por ser compilada, gera executáveis mais rápidos e leves que outras linguagens, além de apresentar um processo de compilação simples e rápido. Possui um padrão, definido pela ISO (`https://www.iso.org/standard/64031.html`), que permite portabilidade entre os vários sistemas operacionais e computadores disponíveis. Também possui mecanismos para agregar várias

Figura 1.3: Processo completo de compilação.

funcionalidades as aplicações, em boa parte dos sistemas operacionais disponíveis no mercado. Por isto, C++ é uma linguagem amplamente utilizada na programação.

1.3 Primeiro programa em C++

Como uma forma de familiarizar o programador com algumas das características e funcionalidades de alguma linguagem, um programa simples chamado "Hello world!" ("Alô mundo!", em português) é disponibilizado com exemplo. Neste tipo de programa, o objetivo é gerar a frase `Alô mundo` em algum dispositivo de saída, como o monitor de vídeo. Em C++, um programa "Hello world.cpp" pode ser definido a seguir:

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

Para compreender o funcionamento deste programa, uma análise por instrução será necessária:

- `#include <iostream>`
  Este trecho de código é uma *diretiva de pré-processamento*. Como já comentado anteriormente, este tipo de instrução não é processada diretamente pelo compilador, e sim pelo pré-processador, que opera antes da compilação. As diretiva de pré-processamento são caracterizadas pela presença do caractere # antes e junto ao comando, sendo este obrigatório. Neste, o comando *#include* solicita ao pré-processador para inserir a este programa o código-fonte presente na biblioteca "iostream". Por ser uma biblioteca nativa da linguagem, pertencente a ***standard template library* - STD** - biblioteca padrão de modelos, ela pode ser escrita entre <>;
- `using namespace std;`
  Este trecho de código está associado ao conceito de espaço de nomes. Este é um mecanismo utilizado pela linguagem para distinguir elementos que pertentem a bibliotecas diferentes mas que apresentam a mesma nomenclatura. Neste trecho, é solicitado, que os comandos definidos na STD sejam referenciados diretamente pelo compilador. Isto se faz necessário pelo emprego dos comandos `cout` e `endl` no código. Caso este comando não fosse empregado, seria necessário explicitar qual a biblioteca que definira o comando fonte no programa. Isto pode ser feito com o comando `std::cout`;
  Um detalhe muito importante: Este trecho de código, associado ao compilador, apresenta um ponto-e-vírgula ao final. Este caractere sinaliza o final do comando, **sendo obrigatório o seu emprego**. Caso isto não aconteça, o compilador pode acusar erro de sintaxe;
- `int main() {...}`
  Esta instrução será melhor compreendida no decorrer do curso, mas trata-se da definição

da função principal - por isto, o nome main - do programa. A sua existência é obrigatória em qualquer código-fonte para gerar um arquivo binário, pois é por esta instrução que o compilador começa o processo de conversão. As chaves { } também são obrigatórias, pois elas delimitam o conjunto de instruções que pertencem, de fato, a função principal, do resto do código fonte;

- `cout << "Alô mundo!" << endl;`
  Esta instrução é a que, de fato, imprime a mensagem "Alô mundo!" no monitor de vídeo. O comando `cout` indica que o dado a seguir deverá ser impresso na saída padrão, que é o monitor. A inserção é sinalizada pelo operador de inserção (< <). Além disto, como o dado a ser mostrado na tela é um texto, ela é envolta por aspas duplas (" "), sendo estas obrigatórias para sinalizar um texto. O comando `endl` insere uma quebra de linha ao final do texto; Assim como no `using namespace std`, o ponto-e-vírgula deve ser inserido após a instrução;
- `return 0;`
  Embora esta instrução não influencie diretamente na visualização da mensagem, ela é importante no gerenciamento do programa pelo compilador, pois será utilizada para informar se o programa saiu corretamente ou não. A sua omissão não ocasiona problemas na hora da compilação, mas é uma boa prática inserir este comando ao final do bloco da função principal (antes do }). Isto será melhor explicado adiante, na parte de funções.

1.3.1 Espaços em Branco e Indentação

Durante o processo de compilação, os espaços em branco são ignorados. Isto significa que o espaçamento do código-fonte não alteram o programa. A única exceção é feita nas diretivas de processamento, onde é exigido um espaço entre o comando e os dados associados. Por causa disto, o programa "Hello world.cpp" pode ser reescrito das seguintes maneiras:

```
#include <iostream>
using namespace std;
int main(){
cout << "Alo mundo!" << endl;
return 0;
}
```

```
#include <iostream>
using namespace std; int main(){
cout << "Alo mundo!" << endl; return 0; }
```

Entretanto, as duas formas são problemáticas, pois a leitura adequada do código-fonte fica comprometida. Par evitar este problema, regras de distribuição das instruções, com tabulações, espaçamento e quebras de linha adequadas, devem ser adotadas. Denomina-se **indentar** a prática empregar esta distribuição do código fonte, sendo esta uma boa prática de programação e **devendo ser adotada**.

1.3.2 Comentários em um código-fonte

Em um código-fonte, é possível inserir blocos de texto que serão ignorados pelo compilador. Estes são denominados de **comentários** e comumente são empregados para explicar, delimitar visualmente ou desabilitar trechos de um código. Geralmente, o trecho a ser tratado como comentário é identificado pela linguagem por meio de delimitadores.

C++ disponibiliza duas formas de definir comentários:

- Apenas em uma linha: Neste tipo de comentário, utiliza-se barras duplas (//) no início da linha a ser comentada a ser comentado. Qualquer texto escrito na linha onde este operador estiver empregado será tratado como parte do comentário;

```
#include <iostream>

using namespace std;

int main(){
    //A proxima linha imprime a frase "Alo mundo"..
    cout << "Alo mundo!" << endl;
    return 0;
}
```

- Em múltiplas linhas: Neste caso, define-se o início do comentário com o operador /* e o fim deste com o operador */. Qualquer texto escrito entre estes dois operadores será tratado como comentário.

```
#include <iostream>

/* Por causa deste comando, define-se qual biblioteca a
   qual se refere o comando 'cout'. Caso ele nao
   estivesse presente, o programador deve explicitar a
   biblioteca a qual aquele comando esta definida. Assim,
    o comando para imprimir a mensagem seria:
std::cout << "Alo mundo!" << std::endl;
*/
using namespace std;

int main(){
    cout << "Alo mundo!" << endl;
    return 0;
}
```

  É possível perceber que, embora exista um comando válido na linguagem dentro do comentário, ele não será executado.

Os dois formatos de comentário podem ser empregados no mesmo código-fonte, e em mais de uma oportunidade. Com o decorrer do curso, e o aumento da complexidade dos códigos-fonte, pode ser necessário explicar a que se refere cada parte do programa. Por isto, o emprego de comentários é considerada uma boa prática, devendo ser explorada.

1.3.3 Saída de Dados em C++

Como foi visto anteriormente, a instrução que de fato imprime a mensagem "Alo mundo!" na tela é:

```
cout << "Alo mundo!" << endl;
```

Nele, o comando `cout` sinaliza a impressão dos dados no canal de saída padrão, o monitor de vídeo, de tudo que está após o operador de inserção "< <" até o fim do comando, que é indicado pelo ponto-e-vírgula. É possível trabalhar com mais de um operador de inserção associado ao

mesmo `cout`: o próprio exemplo mostra um caso, com `endl`. No entanto, esta abordagem pode ser utilizada para imprimir valores de tipos diferentes na tela. Nada impede, também, de ser imprimir a mensagem de outra forma, como é visto a seguir:

```
cout << "Alo" << " " << "mundo!" << endl;
```

O resultado obtido é semelhante ao anterior. A única questão é que foi associado um operador de inserção para cada palavra da mensagem. Também foi associado um operador de inserção para o espaço em branco que, caso seja necessário, **deve ser explicitamente indicado no** `cout`. Isto pode ser feito por meio da inserção de um texto com um espaço em branco como feito caso anterior, ou a associação do espaço a uma das palavras impressas, como é mostrado a seguir. Neste caso, isto só é possível se antes ou depois do espaço também vier um texto.

```
//Note o espaco apos o "Alo".
cout << "Alo " << "mundo!" << endl;
```

O comando a seguir não terá o mesmo resultado dos anteriores. Ao invés disto, o resultado será a impressão da frase “Alomundo!” na tela, com as palavras juntas.

```
cout << "Alo" << "mundo!" << endl;
```

1.4 Exercícios

1. Escreva um programa que imprima a frase “Primeiro programa em C++ na tela”.
2. Escreva um programa que imprima as seguintes formas com asteriscos:

```
*********     ***        *         *
*       *    *   *      ***       * *
*       *   *     *    *****     *   *
*       *   *     *      *      *     *
*       *   *     *      *     *       *
*       *   *     *      *      *     *
*       *   *     *      *       *   *
*       *    *   *       *        * *
*********     ***        *         *
```

2. Tipos, Variáveis, Constantes e Operadores

Para que as linguagens de programação de alto nível lidem com os dados produzidos por humanos, é importante estabelecer formas para representação, armazenamento e modificação destes valores. Isto tem que ser feito levando em conta que um computador (assume-se aqui digital) opera sobre a representação binária deste valores. Este processo é importante pois é manipulando estas informações que os computadores podem produzir resultados, ao invés de reproduzir, como foi feito até agora.

2.1 Variáveis

Um dos recursos manejados pelas linguagens de programação é a memória principal, onde os dados utilizados durante a execução de um programa são armazenados, além do próprio programa em si. Para que a linguagem de programação possa distinguir entre os vários dados envolvidos em um programa, identificadores são utilizados para dados cada um deles. A este identificador, dá-se o nome de **variável**. Com as memórias podem ser vistas como matrizes de 0s e 1s, cada linha desta matriz tem um valor já associado no sistema operacional, um *endereço de memória*. Assim, a linguagem de programação toma um conjunto de linhas da memória principal para representar uma variável.

No entanto, um identificador não é suficiente para descrever o dado de maneira satisfatória na memória. É preciso também manter informações sobre o que o valor neste espaço da memória principal representa para o programador: uma letra, um número, uma palavra, uma unidade booleana. Assim, se permite aplicar manipular e operar a informação armazenada como esperado pela pessoa. Para isto, se estabelece um **tipo** para esta variável.

Pos. Mem.	Var.	Valor
00	x	1
01	media	5.0
02	v[0]	'L'
03	v[1]	'i'
04	v[2]	'P'

Para que uma variável possa ser utilizada no programa, duas etapas precisam ser realizadas antes pelo programador:

1. Declaração da variável;
2. Atribuição de um valor a ela;

Na *declaração* da variável, são definidos o seu nome e seu tipo. Este último **é obrigatório em C++** porque esta é um linguagem *fortemente tipada*. Linguagens deste tipo não permitem que a variável tenha seu tipo modificado durante a execução do programa. O processo de declaração de uma variável é feito uma vez apenas, de forma que o compilador indica um erro de sintaxe caso haja mais de uma tentativa de declaração de uma mesma variável: `variável já definida no escopo`.

O processo de *atribuição de valor* à variável pode ser feito em qualquer instante após a declaração, sendo uma boa prática realizá-lo no momento ou imediatamente após. Isto exige do programador verificar se a variável envolvida em uma atribuição já foi declarada, para evitar erros de sintaxe do tipo `variável não declarada no escopo`. Ao contrário da declaração, pode-se atribuir um valor a uma variável quantas vezes for necessário. Para tanto, utiliza-se um **operador de atribuição** em cada nova atribuição.

2.1.1 Tipos e Modificadores de Tipos

Como C++ exige que as variáveis apresentem um tipo durante a declaração, é importante que o programador conheça os tipos disponibilizados pela linguagem. A própria linguagem estabelece um conjunto de tipos básicos para representar informações simples, como letras e números. Diz-se básicos porque não exigem a inserção de outro códigos-fonte para que sejam reconhecidos pelo compilador: ele são próprios da linguagem. São eles:

- `bool`: Representa um valor lógico (verdadeiro ou falso). Este tipo também é chamado booleano;
- `char`: Representa numericamente um caractere;
- `int`: Representa um número inteiro;
- `float`: Representa um número real com precisão simples;
- `double`: representa um número real com precisão dupla;

Além destes, C++ disponibiliza um tipo especial denominado `void`. Ele representa uma informação sem valor associado, sendo empregado em casos específicos como funções sem valor de retorno ou ponteiros genéricos.

Um dado representado em um tipo básico possui um tamanho pré-definido na memória principal. Este tamanho geralmente é expresso em bytes, que são conjunto de oito bits. Chama-se de bit a unidade binária da memória, que assume valor 0 ou 1. De acordo com o tipo, a leitura dos bits

também é feita de maneira distinta, implicando em um conjunto distinto de valores entre os tipos. Na tabela 2.1 é possível ver o tamanho e os valores dos tipos básicos para maioria dos compiladores de C++.

Tipo	Bytes	Valores
`bool`	1	`false` (0) ou `true` (1)
`char`	1	-128 a 127
`int`	4	$-2,14 \times 10^9$ a $2,14 \times 10^9$
`float`	4	$\pm 3,4 \times 10^{-38}$ a $\pm 3,4 \times 10^{38}$
`double`	8	$\pm 1,7 \times 10^{-308}$ a $\pm 1,7 \times 10^{308}$

Tabela 2.1: Número de bytes e intervalo de valores para os tipos básicos de C++.

Em alguns programas, os valores envolvidos podem ser muito grandes ou muito pequenos, ou apresentarem restrições não cobertas pelos tipos básicos. Neste caso, é possível alterar o conjunto de valores que um tipo pode representar com o uso de **modificadores**. Estes são palavras-chaves associadas ao tipo que alteram o tamanho ou a forma de representação original de um valor. Os modificadores existentes em C++ são:

- `signed`: Assume-se valores com sinal. Permite uso nos tipos `char` e `int`;
- `unsigned`: Assume-se valores sem sinal (não-negativos). Permite uso nos tipos `char` e `int`;
- `short`: Assume-se valores com representação reduzida (numero de *bytes* é reduzido, geralmente a metade). Permite uso apenas no tipo `int`;
- `long`: assume-se valores com representação estendida (numero de *bytes* é aumentado, geralmente ao dobro). Permite uso nos tipos `int` e `double`;

Para o compilador, a inserção de um modificador a um tipo gera um novo tipo. Assim é possível expandir a tabela de tipos básicos para apresentar o efeito dos modificadores. Isto pode ser visualizado na tabela 2.2.

Observe que, em caso de apenas o modificador ser especificado, assume-se que o tipo associado é `int`. Então `long`, `short` e `unsigned` equivalem a `long int`, `short int` e `unsigned int`, respectivamente.

Tipo int

O tipo `int` armazena valores numéricos inteiros, aqueles que não apresentam parte fracionária. Em C++, o sistema adotado por padrão para a representação destes valores é sistema decimal; no entanto, é possível escrever um valor inteiro em outras bases:

- Base octal: Insere-se o prefixo `0` a frente do número. Como exemplos, os números `045`, `061` e `011` estão em base octal;
- Base hexadecimal: Insere-se o prefixo `0x` a frente do número. Como exemplos, os números `0x1F`, `0x4C3A` e `0x11` estão em base hexadecimal.

Tipos float e double

Os tipos `float` e `double` são utilizados para armazenar valores numéricos reais. Para isto, o valor é representado sob a forma de ponto flutuante. O tipo `float` utiliza a representação com precisão simples, enquanto que o tipo `double` utiliza a representação com precisão dupla, que é maior. C++ utiliza a notação americana na representação do número real, onde a parte inteira é separada da parte fracionária por um ponto final. É uma boa prática explicitar o ponto para representar um número real, mesmo que este não tenha parte fracionária, para que não seja confundido com um número inteiro.

Embora possa parecer inútil a existência de um tipo com precisão simples, operações aritméticas envolvendo números em ponto flutuante costumam ter erros numéricos associados. E dependendo

Tipo	Bytes	Valores
`bool`	1	`false` (0) ou `true` (1)
`char`	1	-128 a 127
`unsigned char`	1	0 a 255
`short int` ou `short`	2	`signed`: -32768 a 32767; `unsigned`: 0 a 65535
`int`	4	`signed`:$-2,14 \times 10^9$ a $2,14 \times 10^9$; `unsigned`: 0 a $4,29 \times 10^9$
`long int` ou `long`	4	`signed`:$-2,14 \times 10^9$ a $2,14 \times 10^9$; `unsigned`: 0 a $4,29 \times 10^9$
`float`	4	$\pm 3,4 \times 10^{-38}$ a $\pm 3,4 \times 10^{38}$
`double`	8	$\pm 1,7 \times 10^{-308}$ a $\pm 1,7 \times 10^{308}$
`long double`	8	$\pm 1,7 \times 10^{-308}$ a $\pm 1,7 \times 10^{308}$

Tabela 2.2: Número de bytes e intervalo de valores para os tipos básicos de C++ com modificadores.

dos valores envolvidos, uma precisão elevada pode aumentar as chances destes erros se acumularem e prejudicarem o resultado final. Um exemplo são operações monetárias, onde geralmente são utilizados valores fracionários com até duas casas decimais. Neste caso, o emprego do tipo `float` é mais adequado do que o do tipo `double`. O último é mais apropriado para lidar com números que apresentam muitas casas decimais.

Em alguns casos, um número real pode ser melhor representado em notação científica. Neste caso, um número neste esquema terá um valor para a mantissa e um valor para o expoente (que é um valor inteiro). A notação em ponto flutuante também segue esta regra. Em C++, a mantissa e o exponente de um número são separados por um *E* ou *e*, que sinaliza a base dez. Como exemplos, os números `2.23`, `-129.01` e `0.0001` em notação decimal equivalem, respectivamente aos valores `0.223E1`,`-0.12901e3` e `1E-3` em notação científica.

Tipo char

O tipo char é utilizado para representar caracteres, que são letras, números ou símbolos especiais como '`]`' e '`&`'. Aqui, as aspas simples não são acessórios: **todo valor em caractere deve vir envolto de aspas simples**. Como exemplos, são caracteres que representam os símbolos `a`, `0` e `#` são, respectivamente, 'a', '0' e '#'.

Internamente, o valor armazenado em memória de um tipo caractere é um número inteiro. Para que o programador possa realizar as operações assumindo um caractere de fato, é feito um processo de conversão do valor inteiro para o caractere correspondente. Esta conversão é baseada em uma tabela de símbolos, sendo a mais conhecida delas a tabela ASCII. Esta é a tabela utilizada como padrão em C++. Do emprego de uma tabela de conversão, decorre os seguintes fatos:

- Um caractere correspondente a um número não é a mesma coisa que o seu valor numérico. Como exemplo, o caractere '7' vale 55 na tabela ASCII;
- Um caractere correspondente a uma letra minúscula não é a mesma coisa que o seu equivalente em maiúsculo, por se tratarem de símbolos diferentes. Como exemplo, o caractere 'a' vale 97 na tabela ASCII, enquanto que o caractere 'A' vale 65;
- O espaço em branco possui um caractere associado, definido como ' ', e que vale 32 na tabela ASCII.
  Além de sinais gráficos, alguns caractere especiais funcionam como sinais de controle, não sendo impressos na tela. Os mais conhecidos deste grupo são:
  - '`\n`': quebra de linha;
  - '`\t`': tabulação horizontal;

- '\r': retorno de carro;
- '\a': alerta sonoro;
- '\\': barra invertida;
- '\0': delimitador de final de cadeia de caracteres;

Tipo bool

O tipo `bool` é associado a valores lógicos, que podem ser expressos por apenas dois valores, verdadeiro (`true`) ou falso (`false`). O emprego deste tipo de dado será visto adiante, quando será discutida a parte de operadores relacionais e lógicos. A representação destes valores, assim como no tipo caractere, é feita com número inteiros, da seguinte forma:

- O valor inteiro 0 representa o valor falso (`false`);
- Qualquer outro valor inteiro representa o valor verdadeiro (`true`);

2.1.2 Declaração de Variáveis

A sintaxe para a declaração de variáveis em C++ dada como:

```
<tipo_da_variavel> <nome_da_variavel>;
```

Onde:

- `<tipo_da_variavel>` são os tipos básicos;
- `<nome_da_variavel>` é o identificador associado.

Existem algumas restrições impostas ao nome da variável:

1. Deve começar com uma letra ou '_' (conhecido como *underline* ou *underscore*). Números e caracteres especiais não são permitidos;
2. Deve ser composto por letras, dígitos ou '_'. Outros caracteres não são permitidos;
3. Não pode ser uma palavra reservada (p. ex. `if`)

Uma lista com as palavras-chave da linguagem pode ser vista na figura :

Além disto, C++ é uma linguagem sensível ao tamanho da letra. Isto significa que `x` e `X` representam duas variáveis diferentes. Isto permite ao programador poder adotar o mesmo nome para variáveis diferentes, bastando mudar o tamanho da letra. No entanto, não é uma boa prática, pois pode dificultar a leitura.

Uma boa pratica de programação é utilizar nomes significativos para o problema. Como exemplo, se uma variável irá receber o valor da média aritmética entre três números, nomes como `media` ou `mediaAritm` são adequados. Também, recomenda-se utilizar palavras em português para não haver conflito com alguma palavra-chave da linguagem, pois a base linguística do C++ é o inglês. No entanto, caso haja a possibilidade do seu código-fonte ser disponibilizado publicamente, é importante empregar nomes de variáveis em inglês, para facilitar a leitura.

Exemplo

Quais dos seguintes nomes estão corretos para variáveis?

teste	TESTE	_tESTe_	Var0	_1234
delta_bhaskara	soma	resultado1	raizDelta	temp
resultado-final	10_binario	divisão	raiz.quadrada	teste@

Solução:

De todos os nomes, apenas os seguintes não são permitidos como nome de variável em C++:

- 10_binario, por começar com um número;
- divisão, por apresentar um símbolo especial(o 'ã' é considerado um caractere especial);

- raiz.quadrada, por apresentar um símbolo especial (o '.');
- teste@, por apresentar um símbolo especial (o '@').

Uma observação importante: Nomes que começam com dois caracteres **underline** (como __teste) ou com um caractere **underline** seguido de uma letra maiúscula (como _Teste) são reservados para uso pelo compilador e pelos recursos que ele usa. Além disto, nomes que começam com um caractere **underline** são reservados para serem usados como identificadores globais pelo compilador. Portanto, esta classe de nomes deve ser evitada.

Exemplo

Declare variáveis para os seguintes casos:

1. Chamada de `x`, para armazenar um inteiro
2. Chamada de `c`, para armazenar um caractere
3. Chamada de `temp`, para armazenar uma temperatura
4. Para armazenar o resultado de uma média aritmética

Solução:

1. `int x;`
2. `char c;`
3. `float temp;`
4. `float media;`

Para cada variável a ser declarada, um comando de declaração precisa ser chamado. Porém, no caso da declaração de várias variáveis de um mesmo tipo, é possível listar todos os nomes das variáveis após o tipo, separando-os por vírgula. Isto proporciona economia de espaço, mas pode dificultar a leitura em alguns casos. Como exemplo, a seguinte sequência de declarações:

```
int x;
int y;
int z;
int w;
```

Pode ser reescrita como:

```
int x, y, z, w;
```

2.1.3 Atribuição de Variáveis e Operador de Atribuição

A sintaxe da atribuição de uma variável é dada com:

```
<nome_da_variave> = <novo_valor>;
```

Onde:

- `<nome_da_variavel>` é o identificador associado.
- `<novo_valor>` é o valor que desejamos atribuir à variável;
- '=' é o operador de atribuição;

O símbolo '=' sinaliza ao compilador que uma atribuição está sendo realizada. A presença dele divide a instrução em duas partes: o lado direito e o lado esquerdo, de forma que obrigatoriamente

uma variável deve ficar do lado esquerdo do operador. Isto acontece porque o processo de atribuição sempre associa o lado esquerdo do operador como valor do lado direito deste. A descrição textual do comando de atribuição pode ser dada como "a variável recebe o valor". Como exemplo, a instrução `temp = 15;` é lida como "A variável temp recebe o valor 15".

C++ permite sucessivas atribuições em uma mesma instrução, onde mais de um operador de atribuição é utilizado. Neste caso, o compilador realiza a leitura da direita para a esquerda da instrução, de forma que a atribuição mais a direita será realizada primeiro, e a atribuição mais a esquerda será realizada por último. Como exemplo, a instrução `x = y = z = 1` será interpretada pelo compilador da seguinte forma:

1. Primeiro, será feito z = 1, ou " a variável z recebe o valor 1";
2. Depois, será feito y = z, ou " a variável y recebe o valor z". Como z é igual a 1, então y terá valor 1;
3. Por último, será feito x = y, ou " a variável x recebe o valor y". Como y é igual a 1, então x terá valor 1;

Ao final, as três variáveis receberão valor 1.

Exemplo

O que é impresso pelo programa a seguir?

```
int main(){
        //declaracoes
    int x;
    int y;
    int z;
    //atribuicoes
    x = 0;
    y = 2;
    z = 3;
    x = y = z; //atribuicao: direita para a esquerda
    cout << x << " " << y << " " << z << endl;
    return 0;
}
```

Solução:

O programa imprime os valores de x, y e z, em sequência. Estes valores dependem, ultimamemnte, da dupla atribuição `x = y = z`. Este comando pode ser interpretado como:

1. y = z, ou " a variável y recebe o valor z". Como z é igual a 1, então y terá valor 3;
2. x = y, ou " a variável x recebe o valor y". Como y é igual a 1, então x terá valor 3;

Ao final da execução, o programa irá imprimir a seguinte linha:

```
3 3 3
```

2.1.4 Inicialização de Variáveis

Quando uma variável é declarada em C++, as variáveis apresentam valores aleatórios (lixo em memória), pois o valor representado no espaço alocado é automaticamente convertido, e este não é pre-determinado. Por causa disto, C++ oferece o recurso de atribuir um valor a uma variável ao mesmo tempo em que ela é declarada. A este processo de declaração e atribuição de valor dá-se o nome de inicialização de uma variável.

A sintaxe de inicialização de uma variável é definida como:

```
<tipo_da_variavel> <nome_da_variavel> = <valor_inicial>;
```

Onde:

- `<tipo_da_variavel>` são os tipos básicos;
- `<nome_da_variavel>` é o identificador associado.
- `<valor_inicial>` é o valor que desejamos atribuir como valor inicial à variável;
- '=' é o operador de atribuição;

Uma boa prática de programação é sempre inicializar as variáveis. Caso não haja um valor inicial pré-especificado para aquela variável, é possível inicia-la com o valor 0, qualquer que seja o tipo.

Exemplo

Esboce a inicialização das seguintes variáveis:

1. Chamada de `x`, para armazenar um inteiro
2. Para armazenar o resultado de uma média aritmética
3. Para armazenar um inteiro, com valor inicial igual a 50
4. Para armazenar um caractere, com valor inicial igual a 's'

Solução:

1. `int x = 0;`
2. `float media = 0.0;`
3. `int y = 50;`
4. `char c = 's';`

2.2 Entrada de Dados em C++

Um dos casos de uso de variáveis é a manipulação de entrada de dados em C++. Este tipo de processamento aparece em problemas onde é necessário receber informações do usuário, como o nome do usuário, um caractere de controle ou o número de elementos de um vetor. Neste caso, utiliza-se o comando de entrada de dados `cin`. A sintaxe empregada é parecido a do comando `cout`: o comando `cin` armazena o valor digitado via teclado, que é o dispositivo de entrada padrão em C++, na variável que está após o operador de extração "> >". Não por coincidência, o seu uso também requer biblioteca `iostream`. Um exemplo de funcionamento pode ser visto no código a seguir, que apenas mostra o valor digitado pelo usuário:

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      int x = 0;
7      cout << "Digite um valor: ";
8      cin >> x;
9      cout << "O valor digitado foi " << x << endl;
10     return 0;
11 }
```

Assim como o `cout`, o `cin` permite que o operador de extração possa ser utilizado de maneira

sucessiva. Neste caso, o comportamento será parecido com o de várias chamadas sequenciais ao `cin`. Como exemplo, o trecho de código a seguir:

```
int x, y, x;
cin >> x >> y >> z;
```

Pode ser interpretado da mesma maneira que o seguinte trecho:

```
int x, y, x;
cin >> x;
cin >> y;
cin >> z;
```

É importante mencionar que a chamada de sucessiva de operadores de extração não necessariamente precisa ser feita com variáveis do mesmo tipo. Ela também pode ser empregada em casos de variáveis de tipos diferentes. O tratamento para os valores inseridos irá depender do tipo associado a variável que irá receber a entrada.

```
int x;
char y;
float x;
cin >> x >> y >> z;
```

Exemplo

Implemente um programa que lê dois valores inteiros e imprime-os na tela.

Solução:

```
1  #include <iostream>
2  using namespace std;
3  int main(){
4      int x = 0, y = 0;
5      cout << "digite dois valores: ";
6      cin >> x >> y;
7      cout << "O valor de x e " << x << endl;
8      cin << "O valor de y e " << y << endl;
9      return 0;
10 }
```

2.3 Constantes

Constante são usadas em comandos para representar valores fixos de um dado tipo. A ideia é impedir a alteração de um determinado valor *durante a execução do programa*.

A declaração de uma constante pode ser feita de duas formas:

- Modificador `const`;
- Diretiva de pré-processamento `#define`.

Na definição de uma constante com o modificador `const`, é inicializada uma variável cuja leitura no espaço de memória associada é desabilitada pelo compilador.

```
const <tipo_da_const> <nome_da_const> = <valor_da_const>;
```

Exemplos de definição de uma constante com o modificador `const` pode ser visto a seguir:

- `const int A = 10;`
- `const float GRAVIDADE = 9.8;`
- `const double T = 1e-10;`

No uso da diretiva de pré-processamento `#define`, são necessários dois termos: nome da constante e o valor associado. Para a execução deste diretiva, o pré-processador percorre o código-fonte e substitui cada ocorrência do nome definido para a constante pelo valor dado. Por causa disto, este tipo de constante é conhecida como constante simbólica.

```
#define <nome_da_const> <valor_da_const>
```

Um exemplo de definição de uma constante com `#define` pode ser visto no programa a seguir:

```
1  #include <iostream>
2
3  #define A 10
4  #define GRAVIDADE 9.80665
5  #define T 1e-10
6
7  using namespace std;
8
9  int main()
10 {
11   cout << "Aceleracao da gravidade: " << GRAVIDADE << endl;
12   return 0;
13 }
```

2.4 Operadores

Na grande maioria das vezes, os programas precisa realizar algum tipo de processamento dos dados disponíveis. Um dos processamento mais recorrentes são os cálculos numéricos e/ou lógicos. Para realizá-los, as linguagens de programação disponibilizam meios para a construção de *expressões*. Um expressão é uma sequência bem-definida de uma ou mais *operações*. Uma operação é uma instrução que manipula dados de entrada, gerando um resultado (outro dado) de acordo com um sinalizador. A construção de uma operação envolvem dois elementos:

- Operadores: Os dados de entrada;
- Operandos: O sinalizado da operação.

Os operandos podem ser literais, constantes ou variáveis. Os operadores geralmente são pré-especificados pela linguagem, e servem para que o compilador determine a sequencia de instruções de máquina deve ser realizada com o valor daqueles operandos. C++ possui cerca de 50 tipos de operadores básicos, grande parte deles manipulando dados numéricos (`int`, `float`, `double`) ou lógicos `bool`. Pelo tipo de processamento, os operadores podem ser divididos em três grupos em C++:

- Operadores aritméticos: Receber e retornam dados numéricos;
- Operadores relacionais: Recebem dados numéricos e/ou lógicos, mas retornam dados lógicos;

- Operadores lógicos: Recebem e retornam dados lógicos.

Também, as expressões podem ser classificadas em grupos de expressões básicas em C++:

- Expressões aritméticas: O resultado é um dado numérico;
- Expressões lógicas: O resultado é um dado lógico.

Exemplos de expressões:

1. `10 + 10;`
2. `x/y <= 5;`
3. `x*y && x%y;`
4. `h + 2 >= 7 || y++ == u + v;`
5. `sizeof(int) == 4;`

2.4.1 Expressões Aritméticas e Operadores Aritméticos

As expressões são sequências bem-definidas de operações aritméticas. Estas, por sua vez, se caracterizam por receberem um valor numérico (operandos numéricos) em valores numéricos (resultado é um número). Operações aritméticas fazem uso de operadores aritméticos, sendo comum que alguns deles já estejam especificados pela linguagens. Este são então denominados *operadores básicos*. Os operadores aritméticos básicos em C++ são mostrados na tabela 2.3.

Operador	Funcionalidade
`()`	Delimitador de operação
`+`	Adição
`-`	Subtração
`*`	Multiplicação
`/`	Divisão
`%`	Resto de divisão inteira

Tabela 2.3: Operadores aritméticos básicos da linguagem C++.

Observações:

- O operador `%` é definido apenas para valores do tipo `int`;
- O operador `()`, embora não indique uma operação de fato, é importante para a definição da *precedência* das operações;

É possível perceber que não operadores básico para potenciação e radiciação. Para realizá-las, é preciso incluir no programa a biblioteca matemática `cmath` para chamar às seguintes funções:

- `pow(b,e)` para potência b^e;
- `sqrt(x)` para a raiz quadrada de x;

Uma noção importante na hora de lidar com expressões aritméticas é de precedência de operações. Basicamente, são regras que estabelecem a prioridade das operações, ou seja, qual operação será realizada primeiro. Matematicamente, existe uma precedência de operações especificada:

- Soma e subtração possuem a mesma prioridade;
- Multiplicação e divisão tem a mesma prioridade, e são realizadas antes da soma e subtração;
- Potenciação e radiciação tem a mesma prioridade, e são realizadas antes das quatro anteriores;
- Expressões compostas por operações de mesma prioridade: realizações as operações da esquerda para a direita (no seguinte sentido: $\rightarrow$);
- Qualquer mudança de prioridade nas operações, deve ser sinalizada por parênteses (), colchetes[] e chaves {};

Em C++, a precedência envolvendo operadores aritméticos é semelhante da noção matemática. A tabela faz um resumo da ordem de computação para grupo de operadores.

1. Realiza-se, primeiro, as operações indicadas pelos parênteses, na seguinte sequência:
   - Primeiramente, dos parênteses mais internos aos mais externos;

- Em parênteses de mesmo nível, realiza-se da esquerda para a direita;

2. Depois, realiza-se a multiplicação, a divisão e o resto da divisão inteira. Em expressões com ambos os operadores, realiza-se as operações da esquerda para a direita;
3. Por fim, realiza-se a adição e a subtração. Em expressões com ambos os operadores, realiza-se as operações da esquerda para a direita;

Um resumo da precedência dos operadores pode ser visto na tabela 2.4.

Operador	Precedência no Grupo
`()`	Primeiro a ser avaliado
`+`	Último a ser avaliado
`-`	Último a ser avaliado
`*`	Avaliado após os parêntesis
`/`	Avaliado após os parêntesis
`%`	Avaliado após os parêntesis

Tabela 2.4: Precedência para o grupo de operadores aritméticos.

Uma boa prática de programação é estabelecer parênteses para indicar a sequência desejada, caso haja dúvida sobre a sequência de operações que será realizada.

Exemplo

Indique qual o resultado das expressões aritméticas a seguir em C++:

1. `1*2+6%3-2/2;`
2. `1*(2+6)%(3-2)/2;`
3. `1*(2+(6%3-2))/2;`

Solução:

1. `1*2+6%3-2/2` = `2+6%3-2/2` = `2+0-2/2` = `2+0-1` = `2-1` = 1;
2. `1*(2+6)%(3-2)/2` = 1*8%`(3-2)`/2 = 1*8%1/2 = 8%1/2 = 0/2 = 0;
3. `1*(2+(6%3-2))/2` = `1*(2+(0-2))/2` = `1*(2 + -2)/2` = `1*0/2` = `0/2` = 0;

Operadores Aritméticos de Atribuição

Existe um grupo de operadores aritméticos disponibilizado pala linguagem C++ que combinam duas operações em uma instrução:

1. Uma operação aritmética;
2. Uma operação de atribuição.

Este grupo é denominado operadores aritméticos de atribuição. Durante o processo de compilação, primeiro é realizada a operação indicada entre o valor da variável e a expressão dada, e depois a atribuem o resultado à variável `var`, que está a esquerda da operador de atribuição. A tabela 2.5 mostra os operadores aritméticos de atribuição disponíveis em C++:

Operador	Expressão	Expresssão Equivalente
`+=`	`var += exp`	`var = var + exp`
`-=`	`var -= exp`	`var = var - exp`
`*=`	`var *= exp`	`var = var * exp`
`/=`	`var /= exp`	`var = var / exp`
`%=`	`var %= exp`	`var = var % exp`

Tabela 2.5: Operadores aritméticos de atribuição disponíveis em C++.

É possível perceber que operadores aritméticos de atribuição podem ser substituídos por expressões aritméticas sem prejuízo ao programa. Como exemplo `x += 5` é equivalente a expressão `x = x + 5`. O emprego de operadores deste tipo é um recurso avançado de C++, usualmente não encontrado em outra linguagens. Por causa disto, costuma trazer problemas de leitura mesmo para programadores experientes. Recomenda-se neste caso, o uso das expressões aritméticas convencionais em caos de dúvida.

Exemplo

Sabendo que `x = 1, y = 2, z = 3`, calcule as seguintes expressões:

1. `x += 2 * 2;`
2. `x += y -= z;`
3. `z %= y %= x;`

Solução:

1. `x += 2 * 2;` → `x = x + (2 * 2);` → `x = 1 + (2 * 2);` → `x = 1 + 4;` → `x = 5;`
2. `x += y -= z;` → `x = x + (y = y - z);` → `y = y - z; x = x - y;` → `y = 2 - 3; x = x + y;` → `y = -1; x = x + y` → `y = -1; x = 1 + (-1)` → `y = -1; x = 0;`
3. `z %= y %= x;` → `z = z % (y = y % x);` → `y = y % x; z = z % y;` → `y = 2 % 1; z = z % y;` → `y = 0; z = z % y` → `y = 0; z = 3 % 0` → `y = 0; Erro: Divisão por zero;`

Operadores Unários: Incremento e Decremento

Operadores unários são aqueles que só precisam de um operando. Neste caso, o operando deve ser uma variável, pois eles efetuam uma troca de valor. Em C++, os operadores unários apresentam duas operações possíveis:

- Incremento: O valor armazenado é aumentado em uma unidade;
- Decremento: O valor armazenado é reduzido em uma unidade;

Na tabela 2.6, é possível ver os operadores unários em C++:

Operador	Expressão	Expresssão Equivalente
++ (pré-fixado)	`++var`	`var = var + 1`
- - (pré-fixado)	`- -var`	`var = var - 1`
++ (pós-fixado)	`var++`	`var = var + 1`
- - (pós-fixado)	`var- -`	`var = var - 1`

Tabela 2.6: Operadores unários de C++.

Estes operadores possuem precedência mais alta do que a multiplicação.

É possível perceber que há dois modos de escrever o operador de incremento, bem como o de decremento: a forma pré-fixada (`++x`, `- -x`) e a forma pós-fixada (`x++`, `x- -`. A diferença básica entre as duas é a precedência da operação de incremento/decremento dentro de uma expressão.

- No operador pré-fixado, primeiro é feito o incremento/decremento do valor da variável, e depois o uso do novo valor na expressão;
- No operador pós-fixado, primeiro é usado o valor corrente da variável na expressão e depois realiza-se o incremento/decremento;

Para ilustrar esta diferença, tome como exemplo duas variáveis `x = 1` e `y = 0` e a seguinte expressão:

```
y = 4 * x++;
```

Como o operador pré-fixo realiza a emprego do valor da variável antes do incremento, então a sequência de operações equivalente é dada como:

```
y = 4 * x;
x = x + 1;
```

Que dará como resultado `y = 4*1 = 4` e `x = 1+1 = 2`.

Agora, empregando as mesmas variáveis `x = 1` e `y = 0`, mas a seguinte expressão:

```
y = 4 * ++x;
```

Como o operador pré-fixo realiza a emprego do valor da variável antes do incremento, então a sequência de operações equivalente é dada como:

```
x = x + 1;
y = 4 * x;
```

Que dará como resultado `x = 1+1 = 2`, mas `y = 4*2 = 8`.

Percebe-se que o valor atribuído a `x` não foi alterado. De fato, a diferença não é notada na variável a ser modificada, pois a operação de incremento permanece inalterada. Ela é percebida apenas nas expressões que envolverem a operação.

2.4.2 Expressões Lógicas

Assim com as expressões aritméticas, as expressões lógicas também são sequência bem-definida de operações. Neste caso, porém, as operações envolvidas tanto podem ser aritméticas quanto lógicas. Para estruturas de controle, que modificam o fluxo do programa baseado em uma condição lógica, expressões deste tipo são essenciais. Por isto, é frequente que este grupo de expressões seja a parte ou uma das partes mais importantes de um programa.

Denomina-se de operações lógicas aquelas que resultam em valores lógicos, aqueles representados por pelo tipo `bool`. Este conjunto de operações podem receber tanto valores lógicos (operandos booleanos) quanto numéricos (operandos numéricos), neste último incluindo os grupo dos caracteres. Para tanto, fazem uso de dois grupos de operadores:

- Operadores relacionais: recebem valores numéricos, retorna valores lógicos;
- Operadores lógicos: recebem e retornam valores lógicos;

Operadores Relacionais

Os operadores relacionais avaliam as seguintes condições entre dois números ou expressões numéricas:

- Igualdade ou desigualdade;
- Superioridade ou inferioridade;

O conjunto de operadores relacionais definidos em C++ é mostrado na tabela 2.7.

O resultado delas é um valor lógico, de maneira que elas são comumente empregadas em expressões lógicas. Como envolvem números, elas podem receber expressões aritméticas como operando. Como exemplo, `4+1 < 9/3`, expressão bastante comum na matemática, cujo valor é `false`. Por causa disto, possuem precedência mais alta que o operador de atribuição, porém mais

Operador	Funcionalidade	Precedência no Grupo
==	Igual	Último a ser avaliado
!=	Diferente	Último a ser avaliado
<	Menor	Primeiro a ser avaliado
<=	Menor ou igual	Primeiro a ser avaliado
>	Maior	Primeiro a ser avaliado
>=	Maior ou igual	Primeiro a ser avaliado

Tabela 2.7: Operadores relacionais em C++.

baixa do que os operadores aritméticos. Em caso de expressões que apresentam apenas operadores relacionais, as operações são realizadas de forma que:

- As operações de superioridade/inferioridade precedem as de de igualdade/desigualdade (== e !=);
- As operações de superioridade/inferioridade tem mesma precedência, e são avaliadas da esquerda para a direita;

Exemplo

Verifique o resultado das seguintes expressões lógicas:

1. `1 > 2`
2. `2 != 2`
3. `x == x`
4. `x = 4*3 > 5+1`

Solução:

1. `1 > 2` → `false`
2. `2 != 2` → `false`
3. `x == x` → `true`
4. `x = 4 * 3 > 5 + 1` → `x = ((4 * 3) > (5 + 1))` → `x = (12 > (5 + 1))` → `x = (12 > 6)` → `x = true`

Operadores Lógicos

Os operadores lógicos, como o nome sugerem, recebem e resultam em valores lógicos. Comumente são empregados como conectores para resultados de outros operadores, como os relacionais. Por causa disto, possuem precedência mais baixa do que os operadores relacionais, exceto a negação, que tem precedência igual ao operador de incremento). Na tabela 2.8 é possível ver os operadores lógicos disponíveis em C++.

Operador	Funcionalidade	Precedência no Grupo
!	Negação (não lógico)	Primeiro a ser avaliado
||	Disjunção (ou lógico)	Último a ser avaliado
&&	Conjunção (e lógico)	Avaliado após o não lógico

Tabela 2.8: Operadores lógicos disponíveis em C++.

Sobre o significado dos operadores lógicos:

- O operador de **disjunção** retorna falso apenas quando os dois operandos são falsos. Visto de outra forma, basta que um dos operandos seja verdadeiro para que o resultado seja verdadeiro. A tabela verdade da disjunção pode ser vista na tabela 2.9;

x	y	x || y
0	0	0
0	1	1
1	0	1
1	1	1

Tabela 2.9: Tabela-verdade para o operador de disjunção.

- O operador de **conjunção** retorna verdadeiro apenas quando os dois operandos são verdadeiros. Visto de outra forma, basta que um dos operandos seja falso para que o resultado seja falso. A tabela verdade da disjunção pode ser vista na tabela 2.10;

x	y	x && y
0	0	0
0	1	0
1	0	0
1	1	1

Tabela 2.10: Tabela-verdade para o operador de conjunção.

- O operador de **negação** é um operador unário que apenas inverte o valor lógico do operando. Isto é, uma entrada com valor lógico verdadeiro gera um resultado falso, e vice-versa. A tabela-verdade da negação pode ser vista na tabela 2.11.

x	!x
0	1
1	0

Tabela 2.11: Tabela-verdade para o operador de negação.

Exemplo

Sabendo que `x = 1, y = 2, z = 3`, resolva as seguintes expressões lógicas:

1. `y-x == false`
2. `x > 0 && x < 10`
3. `z >= y || true`

Solução:

1. `y-x == false` → `(y-x) == false` → `2-1 == false` → `1 == false` → `1 == 0` → `false`
2. `x > 0 && x < 10` → `(x > 0) && (x < 10)` → `(1 > 0) && (1 > 10)` → `true && (1 > 10)` → `true && false` → `false`
3. `z >= y || true` → `(z >= y) || true` → `(3 >= 2) || true` → `true || true` → `true`

Exemplo

Construa expressões lógicas em C++ a partir das seguinte informações:

1. $x \leq -4$ ou $x > 2$;
2. $-1 < x \leq 1$;

3. número de iterações n menor que 100 ou erro relativo er maior do que 10^{-5};

Solução:

1. `(x <= -4) || (x > 2);`
2. `(x > -1) && (x <= 1);`
3. `(n < 100) || (er > 0.00001);`

Resumo da Precedência

De maneira geral, a ordem de precedência é a seguinte:

1. Parênteses;
2. Operador de incremento/não lógico;
3. Operadores aritméticos;
4. Operadores relacionais;
5. Operadores lógicos;
6. Atribuição;

Sempre que houver dúvida, é importante parênteses para indicar a procedência correta.

2.5 Conversão de Tipos

Como foi dito anteriormente, C++ é uma linguagem que não permite alteração do tipo de uma variável durante a execução do programa. No entanto, isto impõe a linguagem uma série de questões acerca da interação entre valores em expressões.

Primeiramente, porque há a necessidade de tratar valores de tipos diferentes, mas com mesmo significado. O caso mais emblemático envolve os tipos `int`, `float` e `double`. Isto porque o conjunto dos valores inteiros é um subconjunto dos valores reais: matematicamente, os valores inteiros são valores reais sem parte fracionária. Além disto, embora seja tipos diferentes, `float` e `double` lidam com o mesmo conjunto de dados. Em uma expressão que envolve valores numéricos, a relação entre estes três tipos é matematicamente válida e deve ser suportada pela linguagem.

Também, há a necessidade de resolver o tipo para o resultado de uma expressão composta por operandos de tipos distintos. Isto acontece com os tipos `bool` e `char`, que possuem representação interna por um valo numérico. Por causa disto, é possível empreálos em expressões que envolvem operações inteiras como os casos abaixo:

```
false >= true
```

```
'c' == 87
```

No primeiro caso, toma-se a representação numérica do conjunto lógico, onde `false` é igual a zero, e `true` é igual a um. Na segunda, utiliza-se o valor numérico associado ao caractere `'c'` na tabela ASCII para a comparação com o número inteiro. Este último tipo de consulta, inclusive, costuma ser bastante explorado em avaliações que envolvem caracteres. Resumidamente, a transformação de tipos com representação numérica interna para inteiro segue as seguintes regras:

- Booleano para inteiro: `false` vale `0`, `true` vale `1`;
- Inteiro para booleano: `0` vale `false`, qualquer outro valor vale `true`;
- Caractere para inteiro: conversão pela tabela ASCII;
- Inteiro para caractere: conversão pela tabela ASCII. Neste caso, há perda de informação, pois o tipo inteiro possui capacidade de representação interna maior que o tipo caractere;

O processo de transformação de valores entre tipos diferentes é denominado de **conversão de tipos**. Em C++, este processo é orientado por três regras básicas:

Regra I: Em C++, o tipo do resultado de uma expressão aritmética é sempre convertido no tipo do operando com maior precisão. Isto significa que o retorno de uma expressão será associado ao tipo que apresentar maior intervalo de valores.

Como exemplo, no trecho abaixo:

```
char ch;
int i;
float f;
double d;
result = (ch/i) + (f*d) - (f+i);
```

1. `ch/i` envolve a divisão de uma variável do tipo `char` com uma do tipo `int`. Como o intervalo de valores do tipo `int` é maior que o do tipo `char`, então o resultado da expressão será um valor do tipo `int`;
2. `f*d` envolve a multiplicação de uma variável do tipo `float` com uma do tipo `double`. Como o intervalo de valores do tipo `double` é maior que o do tipo `float`, então o resultado da expressão será um valor do tipo `double`;
3. `f+i` envolve a soma de uma variável do tipo `float` com uma do tipo `int`. Como o intervalo de valores do tipo `float` é maior que o do tipo `int`, então o resultado da expressão será do tipo `float`;
4. `(ch/i) + (f*d)` envolve a expressão 1, com resultado do tipo `int`, e expressão 2, com resultado do tipo `double`. Como o intervalo de valores do tipo `double` é maior que o tipo `int`, então o resultado da expressão será do tipo `double`;
5. `result` será o resultado da expressão obtida em 4, de tipo `double`, e a expressão 3, de tipo `int`. Como o intervalo de valores do tipo `double` é maior que o tipo `int`, então o resultado final será do tipo `double`.

É possível perceber que a linguagem estabelece o seguinte grau de prioridade para expressões de tipos distintos.

1. `double`;
2. `float`;
3. `int`;
4. `char`.

Regra II: Em C++, o tipo da expressão do lado direito de uma atribuição é convertido no tipo do lado esquerdo, mesmo quando este tem menor precisão. Isto acontece pois, na atribuição, o resultado do lado direito será armazenado da variável associada ao lado esquerdo. Logo, é preciso que o dado final esteja em formato adequado para ser armazenado na variável da atribuição.

Como exemplo, seja os seguintes casos:

```
float g = 9.8;
//valor de g e convertido para int
int x = g;
```

```
int x = 3, y = 3, z = 4;
//valor da expressao NAO e convertido para float
float media = (x + y + z)/3;
```

No primeiro caso, é possível verificar que o lado esquerdo é uma constante do tipo `float`. Porém, como a variável que irá receber o valor é do tipo `int`, então o valor armazenado será reduzido de `float` para `int`. No segundo caso, o resultado da expressão é do tipo `int` - basta verificar que não há ponto decimal definido em nenhum momento; portanto, `(x + y + z)/3 = 3`. Este valor será expandido para ser armazenado em uma variável do tipo `float`. Situações como esta são bastante indesejáveis e problemáticas, não sendo indicadas pelo compilador. Para evitar este tipo de problema, recomenda-se que algum valor da expressão seja explicitamente definido com **float**. Por exemplo, podia-se definir o valor 3 com o ponto decimal, de forma deixar a expressão como `(x + y + z)/3.0`. O valor desta seria, aproximadamente, `3.3333`.

Regra III: Em C++, é possível forçar uma expressão a ser interpretada como um tipo informado, utilizando um operador de molde ou `typecast`. Isto permite alterar manualmente a sequência de conversão em uma operação. Há duas sintaxes possíveis para realizar este tipo de conversão:

```
<novo\_tipo> ( <expressao> );
```

```
( <novo\_tipo> ) expressao;
```

Onde `<novo_tipo>` é o tipo na qual o valor de `<expressao>` será convertido. `<expressao>` pode ser uma expressão propriamente dita, como também um valor constante ou uma variável. Exemplos de conversão por operador de molde podem ser visto a seguir:

```
int x = 3, y = 3, z = 4;
float media = float(x + y + z)/3;
```

```
char ch = 'A';
cout << ch << " " << int(ch) << endl;
```

Uma questão importante ao se lidar com forma de conversão que não seguem a ordem de conversão da linguagem é que há perda de informação quando se converte de um tipo de de maior precisão para outro de menor precisão, como nos seguinte casos:

- inteiros para caracteres;
- inteiros longos para inteiros;
- inteiros para inteiros curtos;

Além disto, pode haver perda de precisão quando se faz conversão envolvendo números em ponto flutuante.

2.6 Operador `sizeof`

Habitualmente, os valores dos tipos envolvidos possuem uma quantidade de bytes pré-especificada pelas tabelas 2.1 e 2.2. No entanto, alguns compiladores podem alterar esta dimensão de acordo com a arquitetura do computador em que está executando. Além disto, novos tipos podem ser definidos em C++,e o tamanho deles pode não ser bem-especificado ou pode ser variável, sofrer alteração durante o programa. Para permitir ao programador determinar o tamanho em bytes uma variável, tipo ou valor de expressão, a linguagem C++ possui o operador de tamanho `sizeof`.

A sintaxe deste operador é a seguinte:

```
sizeof(<tipo>);
```

```
sizeof(<expressao>);
```

```
sizeof(<variavel>);
```

Um exemplo de execução pode ser visto a seguir:

```
#include <iostream>
using namespace std;
int main() {
  int a = 10, tamanhoA, tamanhoExpressao, tamanhoB;
  float b;
  tamanhoA = sizeof(int);
  tamanhoB = sizeof(b);
  cout << "'a' ocupa " << tamanhoA << " bytes em memoria."
   << endl;
  cout << "'b' ocupa " << tamanhoB << " bytes em memoria."
   << endl;
  tamanhoExpressao = sizeof(10.1*3.2);
  cout << "O resultado da expressao '10.1*3.2' ira ocupar "
   << tamanhoExpressao << " em memoria.";
  return 0;
}
```

2.7 Exercícios

1. Os itens a seguir fazem parte da linguagem C++. Descreva textualmente o que faz cada um deles.
   Ex.: (l) `media = (n1 + n2 + n3 + n4 + n5)/5;` → calcula a média aritmética entre cinco variáveis de tipo desconhecido e armazena o resultado em uma variável chamada `media`

   (a) `int x;`
   (b) `float salario = 1000.0;`
   (c) `char c = 'z';`
   (d) `bool achou = false;`
   (e) `float X, Y, Z;`
   (f) `int a = 1, b = 2;`
   (g) `cout << "Programacao" << endl;`
   (h) `cin >> tamanho;`
   (i) `cout << "Eu tenho " << anos << "de idade\n";`
   (j) `cin >> X >> Y >> Z;`
   (k) `A = 17%5;`
   (l) `media = (n1 + n2 + n3 + n4)/4;`
   (m) `j = j + 1;`
   (n) `j++;`
   (o) `a > b;`
   (p) `a || b;`
   (q) `a <= x && x <= b;`
   (r) `s == 'm' || s == 'M';`
   (s) `a = b == c;`

2. Indique o que será impresso em cada um dos comandos de saída (`cout`) a seguir.

(a) `cout << pow(3,2) + 1 << endl;`
(b) `cout << 5 + 0.5 << endl;`
(c) `cout << 'a' << endl;`
(d) `cout << (int) 'a' << endl;`
(e) `cout << 1 + '1' << endl;`
(f) `cout << (char) ('a' + 5) << endl;`
(g) `cout << (char) (1 + '1') << endl;`
(h) `cout << (5 > 3) << endl;`
(i) `cout << (2 || 1) << endl;`
(j) `cout << (0 && 'K') << endl;`
(k) `cout << (2 < 5 && 15/3 == 5) << endl;`
(l) `cout << ('j' + 3 > 'z' || 'z' > 'a') << endl`

3. Indique a ordem de avaliação dos operadores em cada uma das instruções em C++ e a seguir mostre o valor de x para cada alternativa:
   (a) `int x = 7 + 3 * 6 / 2 - 1;`
   (b) `int x = 2 % 2 + 2 * 2 - 2 / 2;`
   (c) `int x = (3 * 9 * ( 3 + (9 * 3 / ( 3 ) ) ) );`
4. O que é exibido na tela quando o programa a seguir é executado?

```
#include <iostream>
#include <cmath>

using namespace std;

int main() {
    int x = 4, y = 8;
    double d = 1.5;
    float e = 5.0;
    int r1, r2;
    double r3, r4;
    r1 = ++x % y;
    r2 = d * e + x / y--;
    r3 = pow(e, 2.0) / 3;
    r4 = abs(d-e) - ceil(4 + r1 % (int)r3);
    cout << "Saida do programa: " << endl;
    cout << r1 << " " << r2 << " " << r3 << " " << r4 <<
       endl;
    return 0;
}
```

5. O que é exibido na tela quando o programa a seguir é executado?

```cpp
#include <iostream>

using namespace std;

int main() {
    bool a = 1, b = 0;
    int c = 2, d = 4.2;
    float e = 2.1, f = 7.1;
    bool r1, r2;
    int r3, r4, r5;
    r1 = a && b || e > f + !b;
    r2 = f == a || (int)d / (2*2);
    r3 = ((f == a == c && d != (int)d) + c % d);
    r4 = c++ + ++e;
    r5 = (a || b || c || d || f) - a + ++b * c % d;
    cout << r1 << " " << r2 << " " << r3 << " " << r4 <<
        " " << r5 << endl;
    return 0;
}
```

6. Escreva um programa que leia o raio de um círculo e informe o diâmetro, a circunferência e a área do círculo.
7. Escreva um programa que leia um número inteiro de 5 dígitos, separe o número em dígitos individuais e imprima os dígitos separados um do outro por três espaços cada um. Por exemplo, se o número lido for `42139`, seu programa deverá imprimir:

   ```
   4   2   1   3   9
   ```

8. Calcula-se o IMC (Índice de Massa Corporal) de uma pessoa por meio da seguinte equação:

$$\text{IMC} = \frac{\text{peso, em quilogramas}}{(\text{altura, em metros})^2}$$

   Crie uma aplicação para calcular o IMC de uma pessoa. Essa aplicação deverá ler a partir do teclado o peso do usuário em Kg e a altura em m, imprimindo o valor calculado do IMC. Sua aplicação também deverá imprimir na tela a seguinte tabela:

   ```
   VALORES DE IMC
   Abaixo do peso: menor que 18,5;
   Normal: entre 18,5 e 24,9;
   Acima do peso: entre 25 e 29,9;
   Obeso: 30 ou mais.
   ```

9. Crie uma aplicação que calcule a sua despesa diária com o automóvel, para que você possa estimar quanto dinheiro poderia economizar com o transporte solidário, que também tem outras vantagens, como reduzir as emissões de carbono e congestionamentos. A aplicação deverá solicitar as seguintes informações ao usuário:
   (a) Total de quilômetros dirigidos por dia;
   (b) Custo por litro de combustível;
   (c) Média de quilômetros por litro;
   (d) Preço de estacionamento por dia;
   (e) Gastos diários com pedágios;

10. Desenvolva um programa em C++ que determina a quantidade de crédito de um cliente em uma loja de departamentos. Os seguintes dados estão disponíveis para cada cliente:
    (a) Número da conta;
    (b) Saldo no início do mês;
    (c) Total de todos os débitos deste cliente no mês;
    (d) Total de todos os créditos aplicados à conta deste cliente no mês;
    (e) Limite de crédito autorizado;

3. Comandos de Seleção

Os comandos vistos até agora lidam exclusivamente com dados. No entanto, as linguagens de alto nível disponibilizam também mecanismos para controle de execução das instruções do código-fonte em si. Tradicionalmente, o compilador ou interpretador executa as instruções do início ao fim do código-fonte ou de alguma estrutura especial (como é o caso de C++, onde se começa da função `main`). As estruturas de controle de fluxo alteram esta sequência de execução, permitindo voltar a alguma instrução já calculada ou seguindo direto a uma instrução adiante, pulando instruções intermediárias.

Neste capítulo será visto comandos de seleção, que permitem a execução de uma ou mais instruções (bloco de código) a partir do resultado de uma expressão lógica. Na maioria dos casos, como é o caso de C++, a execução é feita quando a expressão é verdadeira. Isto permite que, de certo modo, o próprio programa seleciona a sua execução, de acordo com o valor de variáveis e expressões.

3.1 Comando `if`

O `if` é um comando de seleção simples, ou seja, depende apenas de uma condição. Se a condição for satisfeita (verdadeira), o comando a seguir é executado. Caso contrário, este é ignorado, e a próxima instrução fora do escopo é executada. A condição, neste caso, é interpretada como uma expressão lógica, embora não seja obrigatória que a expressão da condição seja um expressão lógica (com operadores lógicos ou relacionais). Caso a expressão tenha valores não lógicos, comumente, ela gera valores inteiros. Neste caso, o compilador realiza a conversão de inteiro para booleano pela seguinte regra, já vista:

- Verdadeiro: qualquer valor diferente de zero
- Falso: zero

A sintaxe padrão do comando, que envolve várias instruções, é dada por:

```
if(condicao){
  comando1;
  ...
  comandoN;
}
```

Quando há apenas uma instrução, é possível utilizar a sintaxe alternativa:

```
if(condicao)
  comando;
```

Para ilustrar o emprego deste comando, considere um programa que lê um número digitado pelo usuário e imprime a frase "`Numero invalido`" caso o usuário digite zero.

```
int main(){
int x;
cin >> x;
if(x == 0)
    cout << "Numero invalido" << endl;
return 0;
}
```

Como a condição de execução do `if` é um expressão lógica, nada impede se de utilizar constantes ou expressões que envolvam valores constantes, cujo resultado será sempre o mesmo. Neste caso, se o valor da expressão foi constante verdadeira `true`, ele sempre será executado. Caso o valor da expressão foi constante verdadeira `true`, ele nunca será executado. Estes são caso atípicos, e geralmente indicam falha na programação.

```
if(1){ // Este comando sempre sera executado
cout << "Sempre imprime" << endl;
}
```

```
if(2 >= 3){ /* Este comando nunca sera executado, pois a
   expressao vale false. */
cout << "Nunca imprime" << endl;
}
```

Um cuidado especial com a construção das expressões de verificação do `if` deve ser tomado quando se envolve comparações de igualdade. Isto porque matematicamente, a comparação de igualdade é feita com o operador =. No entanto, em C++, este operador é utilzado na atribuição de variáveis. A comparação de igualdade é feita pelo operador ==. Logo `a = b` pode ter resultado diferente de `a == b`. Isto enfatiza a necessidade do programador de ficar atento às expressões estabelecidas para que elas se comportem como desejado.

```
if(x = 0){ //ha um problema aqui
cout << "Numero invalido" << endl;
}
```

Exemplo

Faça um programa que lê uma letra digitada pelo usuário e imprime a frase "Vogal digitada" caso o usuário digite uma vogal.

Solução:

```
#include <iostream>

using namespace std;

int main(){
    char c;
    cin >> c;
    if(c == 'A' || c == 'E' || c == 'I' ||
       c == 'O' || c == 'U' ||
       c == 'a' || c == 'e' || c == 'i' ||
       c == 'o' || c == 'u'){
        cout << "Vogal digitada" << endl;
    }
    return 0;
}
```

3.2 Comando if-else

O if-else é um comando de seleção composto, pois apresenta dois blocos que dependem de uma mesma condição. Quando a condição é verdadeira, o bloco correspondente ao if é executado. Quando a condição é falsa, o bloco correspondente ao else é executado.

A sintaxe padrão deste comando, que envolve várias instruções, é dada por:

```
if(condicao){
  comandoI.1;
  ...
  comandoI.N;
}
else{
  comandoE.1;
  ...
  comandoE.N;
}
```

Para ilustrar o emprego deste comando, considere um programa que lê um número digitado pelo usuário e imprime a frase "Numero Invalido" caso o usuário digite zero, e a frase "Número Válido" caso o usuário digite um um valor diferente de zero.

```
int main(){
int x;
cin >> x;
if(x == 0){
    cout << "Numero Invalido" << endl;
}
else{
    cout << "Numero Valido" << endl;
}
return 0;
}
```

Como há duas possibilidades de execução que dependem de uma mesma condição, uma questão importante é a possibilidade de invertemos a condição. Neste caso, uma alternativa é utilizar o operador lógico de negação sobre a condição original. Para ilustrar, considere o mesmo programa que lê um número digitado pelo usuário e imprime a frase "`Numero Válido`" caso o usuário digite um valor diferente de zero, e a frase "`Número Inválido`" em caso contrário.

```
int main(){
int x;
cin >> x;
if(!(x == 0)){
    cout << "Numero Valido" << endl;
}
else{
    cout << "Numero Invalido" << endl;
}
return 0;
}
```

Perceba que a condição foi alterada para a negação dela. Para este exemplo, é fácil perceber que a condição invertida poderia ser substituída por `x != 0`. No entanto, outros casos podem ser mais difíceis de se perceber. Especificamente para expressões que envolvem operadores de conjunção (`&&`) ou disjunção (`||`), uma propriedade bastante útil para estes casos é a lei de De Morgan:

Teorema 3.2.1 **Lei de De Morgan**: Seja duas expressões lógicas `p` e `q`. É possível afirmar que:

- `!(p && q) = !p || !q;`
- `!(p || q) = !p && !q;`

Para ilustrar, considere o exemplo de um programa que imprime na tela imprime a frase "`Numero Inválido`" caso o usuário digite um valor maior que 10 ou menor que -7, e a frase "`Número Válido`" em caso contrário. or causa das lei de De Morgan, os dois programas a seguir são equivalentes:

```cpp
int main(){
int x;
cin >> x;
if((x > 10) || (x < -7)
   ){
    cout << "Numero
       Invalido" <<
       endl;
}
else{
    cout << "Numero
       Valido" << endl;
}
return 0;
}
```

```cpp
int main(){
int x;
cin >> x;
if((x <= 10) && (x >=
   -7)){
    cout << "Numero
       Valido" << endl;
}
else{
    cout << "Numero
       Invalido" <<
       endl;
}
return 0;
}
```

Exemplo

Faça um programa que lê uma letra digitada pelo usuário e imprime a frase "`Vogal digitada`" caso o usuário digite uma vogal ou a frase "`Consoante digitada`" caso o usuário digite uma consoante

Solução:

```cpp
#include <iostream>
#include <cctype>

using namespace std;

int main(){
    char c;
    cin >> c;
    if(isalpha(c)) {
        if(c == 'A' || c == 'E' || c == 'I' ||
        c == 'O' || c == 'U' || c == 'a' ||
        c == 'e' || c == 'i' || c == 'o' ||
        c == 'u'){
        cout << "Vogal digitada" << endl;
        }
        else{
                cout << "Consoante digitada" << endl;
        }
    }
    return 0;
}
```

3.2.1 Encadeamento de `if-else`

Quando existe várias opções disponíveis, apenas um `if-else` não é suficiente. Neste caso, C++ permite o uso de comando de seleção aninhado. Trata-se da associação de vários `if-else`. Neste comando, Cada `else` só é executado se o `if` anterior for falso. Dito de outra maneira, a condição de um `if` é testada somente se todos os anteriores forem falsos. Assim, garante-se que os blocos de comandos sejam mutuamente exclusivos: apenas um deles pode ser verdade.

A sintaxe para o encadeamento de `if-else` é dada como:

```
if(condicao1){
  comando1.1;
  ...
  comando1.N;
}
else if(condicao2{
  comando2.1;
  ...
  comando2.N;
}
else{
  comando3.1;
  ...
  comando3.N;
}
```

Como exemplo de aplicação, considere um programa que lê a nota de um aluno e imprime a frase "`Aprovado`", caso a nota seja maior ou igual a `5.0`, "`Em recuperacao`", caso a nota esteja entre texttt3.0 e `5.0` e "`Reprovado`", caso a nota esteja seja menor do que `3.0`.

```
/* O Programa nao funciona corretamente para valores acima de
   5.0
(Por exemplo, 6.0) */
int main(){
    float nota;
    cin >> nota;
    if(nota >= 5.0){
        cout << "Aprovado" << endl;
    }
    else if(nota >= 3.0){
        cout << "Em recuperacao" << endl;
    }
    else{
        cout << "Reprovado" << endl;
    }
    return 0;
}
```

A noção de condições mutuamente exclusivas disjuntos é muito importante para o encadeamento de `if-else`, pois é ela que diferencia o seu emprego de uma sequência de comandos `if` independentes. Neste último, não há a garantia de de que os blocos de `if` sejam independentes. Para ilustrar, considere um tentativa de construção do programa anterior, quando se emprega uma sequência de `if`.

```
int main(){
    float nota;
    cin >> nota;
    if(nota >= 5.0){
        cout << "Aprovado" << endl;
    }
    if(nota >= 3.0){
        cout << "Em recuperacao" << endl;
    }
    if(nota < 3.0) {
        cout << "Reprovado" << endl;
    }
    return 0;
}
```

Neste caso, não há garantia de exclusão entre todos os blocos condicionais, pois o funcionamento de cada `if` é independente um do outro. Isto possibilita que dois blocos de `if` possam ser executados para o mesmo valor. Isto pode ser verificado, para este exemplo, se o usuário digitar um valor maior que 5.0: este conjunto de valores satisfaz a expressão `nota >= 5.0`, e também a condição `nota >= 3.0`, imprimindo erroneamente as duas mensagens associadas.

Para se evitar isto sem o emprego do encadeamento de `else-if`, uma alternativa é montar as expressões de forma a garantir que cada bloco de `if` seja mutuamente exclusivo. Neste caso, passa a ser responsabilidade do programador determinar quais as expressões. Para ilustrar, considere o programa anterior que emprega uma sequência de `if` mutuamente exclusivos.

```
int main(){
    float nota;
    cin >> nota;
    if(nota >= 5.0){
        cout << "Aprovado" << endl;
    }
    if(nota < 5.0 && nota >= 3.0){
        cout << "Em recuperacao" << endl;
    }
    if(nota < 3.0) {
        cout << "Reprovado" << endl;
    }
    return 0;
}
```

Perceba a mudança no segundo comando `if`. Para caso mais complexos, determinar as condições mutuamente exclusivas pode ser trabalhoso.

3.3 Comando `switch-case`

O `switch-case` é um comando de seleção de múltipla escolha, que serve de alternativa ao uso de vários `if-else`. No entanto, ele só pode ser empregado em situações onde as opções são constantes e pré-definidas. Isto significa que opções expressas por um único valor podem ser empregadas em um comando `switch-case`; porém, opções que envolvem intervalos de valores não podem ser empregadas. Além disto, a expressão deve ser qualquer expressão em C++ que retorne um inteiro

ou caractere: não há emprego do `switch-case` para valores em ponto flutuante ou lógicos.

A sintaxe do `switch-case` é dada como:

```
switch(expressao){
case constante1:
    comando 1.1;
    ...
    comando 1.N;
    break;
...
case constanteN:
    comando N.1;
    ...
    comando N.N;
    break;
default:
    comando d.1;
    ...
    comando d.N;
}
```

Onde:

- Caso a expressão seja igual ao da constante1, o bloco 1 é executado;
- Caso a expressão seja igual ao da constante2, o bloco 2 é executado;
- Caso a expressão seja igual ao da constante3, o bloco 3 é executado, e assim sucessivamente;

Na estrutura, o comando `break` serve para evitar que os blocos subsequentes sejam também executados. Isto garante que as condições sejam mutuamente exclusivas. Ele é opcional: sem ele, o blocos de código de opções posteriores também são executados.

A cláusula `default`, também opcional, permite a execução de um bloco de instruções quando a expressão não é igual a nenhuma das opções constantes enumeradas no `switch-case`. Ela é útil em casos onde apenas uma parcela dos valores dados é de interesse do programa: o restante dos valores é considerado um caso `default`, sendo tratado de maneira única.

Como exemplo de aplicação:, considere um programa que escreve por extenso na tela o nome de cada dígito de 0 até 2.

```
int main(){
int d;
cin >> d;
switch(d){
    case 0:
        cout << "zero" << endl;
        break;
    case 1:
        cout << "um" << endl;
        break;
    case 2:
        cout << "dois" << endl;
        break;
    default:
        cout << "numero nao reconhecido" << endl;
        break;
}
return 0;
}
```

Exemplo

Utilizando o `switch-case`, faça um programa que lê uma letra digitada pelo usuário e imprime a frase "`Vogal digitada`" caso o usuário digite uma vogal ou a frase "`Consoante digitada`" caso o usuário digite uma consoante

Solução:

```
int main(){
    char c;
    cin >> c;
    switch(c){
        case 'a':
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            cout << "Vogal digitada" << endl;
            break;
        default:
            cout << "Consoante digitada" << endl;
    }
    return 0;
}
```

3.4 Operador Ternário: ? :

Como o nome sugere, o operador ternário ? : é o único da linguagem C++ a lidar com três operandos. É um comando de seleção com comportamento semelhante ao de um comando `if-else`.

Sua sintaxe é dada por:

```
expr1 ? expr2 : expr3;
```

Onde:

- `expr1` é á expressão a ser avaliada;
- `expr2` é a expressão executada, cujo resultado é associado ao resultado do operador ternário, se `expr1` for verdadeira;
- `expr3` é a expressão executada, cujo resultado é associado ao resultado do operador ternário, se `expr1` for falsa.

Como exemplo de aplicação, considere um programa que atribui à `y` o valor 1 se `x` for maior ou igual a zero ou o valor -1 se `x` for menor que zero.

```
int main(){
    int x, y;
    cin >> x;
    y = x >= 0 ? 1 : -1;
    cout << "y: " << y << endl;
    return 0;
}
```

Exemplo

Utilizando o operador ternário (`?  :`), faça um programa que lê dois números e imprime qual é o maior dos dois.

Solução:

Exemplo:

```
int main(){
    int x, y, maior;
    cin >> x >> y;
    x > y ? maior = x : maior = y;
    cout << maior << " e o maior" << endl;
    return 0;
}
```

3.5 Dicas Gerais

Existem várias tipos de comandos de seleção em C++, alguns com empregos bem distintos. Na hora de programar, é importante prestar atenção em alguns detalhes do problema antes de escolher o comando mais adequado:

- Todo `switch-case` pode ser feito com `if-else`, mas não o contrário;
- `switch-case` só pode ser utilizado quando:
  - A expressão retorna um inteiro ou caractere;
  - Todos os valores possíveis que a expressão pode assumir são conhecidos;
- Caso esteja lidando com intervalos, o `switch` não funciona;

- O operador ternário pode ser utilizado em casos simples de `if-else`;
- No final das contas: `if-else` é mais genérico e portanto, mais indicado.

3.6 Exercícios

1. (**Questão similar do URI: 1044**) Diz-se que um número `x` é múltiplo de `y` se o resto da divisão inteira de `x` por `y` for igual a zero. Implemente um programa que recebe como entrada dois números inteiros, `x` e `y`. Em seguida, o seu programa deve imprimir uma mensagem informando se `x` é múltiplo de `y` ou não. Exemplo:

```
-- Exemplo 1:
Insira dois valores
5 2
5 nao e multiplo de 2
-- Exemplo 2:
Insira dois valores
9 3
9 e multiplo de 3
```

2. Implemente um programa que recebe como entrada um ano e informa se ele é bissexto ou não. Os anos bissextos são aqueles que são múltiplos de 4, mas que não são múltiplos de 100, com exceção daqueles que são múltiplos de 400. Exemplos:

```
1999 nao e um ano bissexto
2000 e um ano bissexto
1998 nao e um ano bissexto
1900 nao e um ano bissexto
2016 e um ano bissexto
400 e um ano bissexto
```

3. (**Questão similar do URI: 1043, 1045**) Implemente um programa que recebe como entrada três números inteiros. O seu programa deve imprimir uma mensagem na tela informando se os números fornecidos constituem os lados de um triângulo ou não. Além disso, o programa deve imprimir também qual é o tipo do triângulo formado: equilátero (três lados iguais), isósceles (dois lados iguais) ou escaleno (três lados diferentes). Considere que para constituir um triângulo, cada um dos lados tem que ser menor do que a soma dos outros dois. Exemplos:

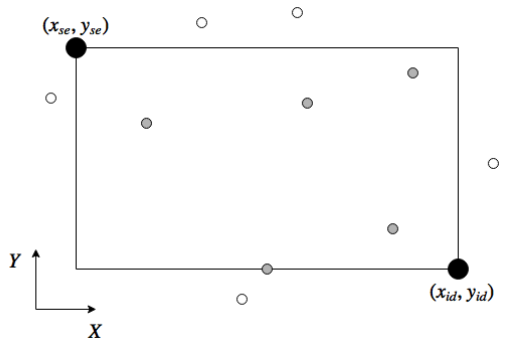
```
6 3 3: Numeros nao formam triangulo
2 2 2: Numeros formam triangulo equilatero
5 10 9: Numeros formam triangulo escaleno
6 6 9: Numeros formam triangulo isosceles
```

4. Implemente um programa para computar as quatro operações básicas (adição, subtração, multiplicação e divisão) com operandos inteiros. O seu programa deve receber como entrada, além de dois números inteiros, um caractere correspondente à operação (`'+'`, `'-'`, `'*'`, `/'`), sendo que eles devem ser informados em uma mesma linha. Exemplos:

```
-- Exemplo 1:
2+3
5
-- Exemplo 2:
3-8
-5
-- Exemplo 3:
7/8
0
-- Exemplo 4:
7*3
21
```

5. Implemente um programa que imprime uma mensagem na tela informando se um ponto 2D faz parte de um retângulo ou não. O seu programa deve receber como entrada os seguintes dados:
   - As coordenadas `x` e `y` do ponto
   - As coordenadas `xse` e `yse` do vértice superior esquerdo do retângulo
   - As coordenadas `xid` e `yid` do vértice inferior direito do retângulo

   Considere a ilustração a seguir para desenvolver o seu raciocínio.



   Dica: observe a direção de crescimento do sistema de coordenadas e também que os vértices informados delimitam uma região. A partir disto, tente construir uma expressão lógica que retorna verdadeiro sempre que `x` e `y` estiver dentro dos limites desta região.
6. Implemente um programa que recebe como entrada um caractere e informa se ele é uma vogal, uma consoante ou um símbolo qualquer. Assuma que as letras digitadas serão sempre minúsculas. Exemplos:

```
a e uma vogal
f e uma consoante
6 e um simbolo qualquer
[ e um simbolo qualquer
```

7. (**Questão similar do URI: 1036**) Implemente um programa que calcula as raízes de uma equação do segundo grau utilizando a fórmula de Bhaskara. O seu programa deve receber

como dados de entrada três números, a, b e c, correspondentes aos coeficientes de uma equação do segundo grau

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

e calcular as raízes x_1 e x_2 de acordo com

$$x_{1,2} = \frac{-b \pm \sqrt{\Delta}}{2a},$$

onde

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

O seu programa deve imprimir os valores de $x_{1,2}$ ou a mensagem "`Equacao com raizes complexas`", se $\Delta < 0$. Exemplos:

```
-- Exemplo 1:
1 3 2
x1 = -1, x2 = -2
-- Exemplo 2:
1 1 -2
x1 = 1, x2 = -2
-- Exemplo 3:
1 4 4
x1 = -2, x2 = -2
```

8. Aprimore o programa anterior, de modo que ele seja compatível com o caso em que a equação resulta em uma raíz repetida ($\Delta = 0$) e também com o caso em que a equação resulta em raízes complexas. No primeiro caso, o seu programa deve imprimir o valor de $x_{1,2}$ apenas uma vez, seguido pela mensagem "`(duas raizes iguais)`". No segundo caso, o seu programa deve imprimir os valores de $x_{1,2}$ separados entre parte real r e parte imaginária k, considerando $i = \sqrt{-1}$, no seguinte formato:

   ```
   x1 = r + ki, x2 = r - ki
   ```

   Exemplos:

   ```
   -- Exemplo 1:
   1 4 4
   x = -2 (duas raizes iguais)
   -- Exemplo 2:
   1 0 4
   x1 = 0 + 2i, x2 = 0 - 2i
   ```

9. (**Questão similar do URI: 1042**) Implemente um programa que recebe três caracteres como dados de entrada e os imprime em ordem alfabética. Exemplos:

   ```
   -- Exemplo 1:
   Entrada
   a b c
   Saída
   a b c
   -- Exemplo 2:
   ```

```
Entrada
b c a
Saída
a b c
-- Exemplo 1:
Entrada
z j a
Saída
a j z
```

4. Comandos de Repetição

Uma das atribuições mais importantes de uma linguagem de programação é automatizar tarefas, principalmente aquelas mais desgastantes para o usuário. Neste contexto, processos de repetição de tarefas costumam ser exemplo. As estruturas de repetição permitem ao programador estabelecer a tarefa e o número de vezes que ela pode ser repetida. Trata-se de um dos conceitos mais importantes e explorados em programação.

4.1 Introdução

Estruturas de repetição são essenciais para a automatização de tarefas repetitivas, e comumente são utilizadas em programas. Isto porque permitem repetir os comandos que estejam no seu bloco de código. Por causa disto, são também chamados de laços ou *loops*.

As linguagens de programação costumam oferecer dois tipos de comandos de repetição. O primeiro é comando de repetição contado ou numérico, que permite repetir por um determinado número de vezes. Neste conjunto, enquadra-se o comando `for`. Segundo é o comando de repetição condicional ou lógico, que repete instruções enquanto uma condição é verdadeira. C++ apresenta dois comandos deste tipo: o `while` e o `do-while`

4.2 Comando `for`

O comando `for` é adequado para a construção de laços numéricos, aqueles cuja quantidade de repetições é conhecida a priori. A sintaxe padrão, para várias instruções, é dada como:

```
for(inicializacao; condicao; atualizacao){
    comando1;
    ...
    comandoN;
}
```

No laço `for`, as repetições do laço são estabelecidas sobre uma variável, chamada de contador, variável de iteração ou variável de controle. Para tanto, o comando utiliza três expressões que gerenciam o comportamento do contador durante a execução:

- Inicialização: Atribui um valor inicial ao contador do laço. Este é o primeiro valor a ser utilizado durante a primeira passagem do laço. Ela é executada somente na primeira vez em que o `for` é executado;
- Condição: Ela é uma expressão lógica envolvendo o contador, que é testada para permitir ou não a execução do bloco de código. Ela é avaliada no início de cada iteração;
- Atualização: Expressão utilizada para modificar o valor do contador. Ela é executado no final de cada iteração;

Todas as expressões são opcionais, embora não seja usual omiti-las. A ordem de execução do laço `for` pode ser resumida da seguinte forma:

1. A expressão de inicialização é executada;
2. A expressão de condição é avaliada:
   - Caso seja verdadeira: o bloco de comandos é executado;
   - Caso seja falsa: o laço é encerrado;
3. A expressão de atualização é executada;
4. O passo 2 é executado.

Como exemplo de aplicação, considere um programa que imprima os 1.000.000-ésimos números naturais não-nulos.

```
#include <iostream>

using namespace std;

int main(){
    int i;
    for(i = 1; i <= 1000000; i++){
        cout << i << endl;
    }
    return 0;
}
```

A princípio, qualquer expressão lógica que envolva o contador pode ser utilizada como condição e qualquer expressão numérica que altere o contador pode ser empregada como expressão de atualização. Isto implica que, embora seja comum utilizar operações de incremento, é possível declarar o comando `for` de várias maneiras, cada uma gerando uma sequência de atribuições ao valor do contador:

- `for(i = 0; i <= 9; i++)` → { 0, 1 , 2, 3, 4, 5, 6, 7, 8, 9 };
- `for(i = 10; i > 0; i- -)` → { 10, 9 , 8, 7, 6, 5, 4, 3, 2, 1 };
- `for(i = 0; i < 20; i += 2)` → { 0, 2 , 4, 6, 8, 10, 12, 14, 16, 18 };
- `for(i = 2; i <= 1024; i *= 2)` → { 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024 };

Além disto, declaração do comando `for` pode envolver outras variáveis que não sejam o contador. Como exemplo, considere um programa que lê um número `n` e em seguida, imprime os `n` primeiros números pares na tela. Considere o número 0 como o primeiro par.

```
int main(){
    int i, n;
    cout << "Digite n\n";
    cin >> n;
    for(i = 0; i < 2*n; i += 2){
        cout << i << endl;
    }
    return 0;
}
```

Percebe-se que a expressão de condição do laço depende da variável `n` definida anteriormente.

É possível encontrar várias definições para o comando `for` que geram a mesma sequência para o valor do contador. Geralmente, este comportamento ocorre quando os laços possuem condições envolvidas que passam a ser falsas no mesmo mesmo valor, e apresentam o mesmo valor inicial e mesmo mecanismo de atualização.

- `for(i = 0; i <= 10; i++)` $\rightarrow \{0, 1, 2, ..., 10\}$;
- `for(j = 0; j < 11; j = j+1)` $\rightarrow \{0, 1, 2, ..., 10\}$;

Também é possível encontrar várias definições para o comando `for` que geram o mesmo número de repetições. Isto permite estabelecer definições equivalentes do comando `for` nos casos em que o bloco de comandos depende da quantidade de vezes a ser repetida, mas não do valor da variável de iteração. Um exemplo de situação deste tipo pode ser vista em um programa que lê as notas de 10 alunos e exiba a média das notas na tela. São apresentadas quatro soluções possíveis para este problema:

```
int main(){
    int i;
    float nota, media =
        0;
    for(i = 1; i <= 10;
        i++){
        cout << "Insira
            a nota de
           um aluno: ";
        cin >> nota;
        media += nota;
    }
    media /= 10;
    cout << "media da
       turma: " <<
       media << endl;
    return 0;
}
```

```
int main(){
    int i;
    float nota, media =
        0;
    for(i = 0; i < 10;
       i++){
        cout << "Insira
            a nota de
           um aluno: ";
        cin >> nota;
        media += nota;
    }
    media /= 10;
    cout << "media da
       turma: " <<
       media << endl;
    return 0;
}
```

```cpp
int main(){
    int i;
    float nota, media =
        0;
    for(i = 10; i < 0;
       i--){
        cout << "Insira
            a nota de
           um aluno: ";
        cin >> nota;
        media += nota;
    }
    media /= 10;
    cout << "media da
       turma: " <<
       media << endl;
    return 0;
}
```

```cpp
int main(){
    int i;
    float nota, media =
        0;
    for(i = 0; i < 20;
       i += 2){
        cout << "Insira
            a nota de
           um aluno: ";
        cin >> nota;
        media += nota;
    }
    media /= 10;
    cout << "media da
       turma: " <<
       media << endl;
    return 0;
}
```

Isto foi possível porque o que precisava ser repetido era a leitura das notas. Esta operação estava vinculada a variável `nota`, e não ao contador do laço `for`.

Observações importantes

- `for(i = 0; i <= n; i++)`:
  Este laço NÃO é executado `n` vezes, mas sim n+1 vezes;
- `for(i = n; i > 0; i++)` e `for(i = n; i < 0; i--)`:
  Como as condições de parada jamais serão verdadeiras, estes laços ficam sempre em execução, impedindo a progressão das linhas subsequentes do programa. Estes tipos de laços são conhecidos como **laços infinitos** ou **eternos**;
- É possível criar o contador diretamente na declaração do `for`:

```cpp
for(int i = n; i > 0; i++){
    ...
}
cout << "i: " << i << endl;
```

  Porém, o contador declarado dentro do `for` estará visível apenas para as instruções dentro do `for`, deixando de existir após o fim do programa. Isto significa que qualquer outra chamada ao contador após o fim do laço `for` implicará em erro.
- Com o uso de um laço contado, é possível realizar a leitura de uma quantidade determinada de valores de entrada. Como exemplo, considere um programa que lê um número `n` e em seguida, lê `n` caracteres. Também, ele exibir a quantidade de caracteres digitados que são letras minúsculas.

```
int main(){
    char c;
    int n, cont = 0;
    cout << "Informe a quantidade de caracteres\n";
    cin >> n;
    for(int i = 0; i < n; i++){
        cout << "Informe um caractere\n";
        cin >> c;
        if(c >= 'a' && c <= 'z'){
            cont++;
        }
    }
    cout << cont << " letras digitadas\n";
    return 0;
}
```

O emprego combinado de comandos de repetição e entrada de dados permite ao usuário será bastante explorado no decorrer do curso.

Exemplo

Implemente um programa em C++ que imprime na tela em forma de tabela as seguintes contagens:

- De 1 a 100
- De 10 a 1000 (incrementando o contador de 10 em 10)
- De 100 a 1 (em ordem decrescente)

1	10	100
2	20	99
3	30	98
...	...	...
100	1000	1

Solução:

```
int main(){
    int i;
    for(i = 1; i <= 100; i++){
        cout << i << " "
             << 10*i << " "
             << 101-i << endl;
    }
    return 0;
}
```

4.3 Comando while

O comando while é adequado para a construção de laços não-contados, aqueles onde o número de iterações desejado é desconhecido. A sintaxe padrão, para várias instruções, é dada como:

```
while(condicao){
    comando1;
    ...
    comandoN;
}
```

No `while`, condição é qualquer expressão que possa ser avaliada como expressão lógica, sendo obrigatório neste comando. Esta condição é testada antes da execução do bloco de comandos, que será executado enquanto a condição for verdadeira. A ordem de execução do laço `for` pode ser resumida da seguinte forma:

1. A expressão de condição é avaliada:
   - Caso seja verdadeira: o bloco de comandos é executado;
   - Caso seja falsa: o laço é encerrado;
2. O passo 1 é executado;

É importante notar que, ao contrário do `for`, o `while` não apresenta as definições da inicialização e da atualização explicitamente. Neste caso, **o programador é responsável por definir tais informações**. De outra maneira: Dois itens precisam estar garantidos para o correto funcionamento do laço `while`:

1. Que a condição vai ser verdadeira ao ser testada pela primeira vez;
2. Que a condição vai se tornar falsa, em algum momento;

Como exemplo, considere o programa anterior que imprima os 1.000.000-ésimos números naturais não-nulos. Agora, uma implementação com o comando `while` é mostrada a seguir:

```
1 int main(){
2     int i = 1;
3     while(i <= 1000000){
4         cout << i << endl;
5         i++;
6     }
7     return 0;
8 }
```

É importante notar que a inicialização do valor da variável `i`, envolvida na condição, foi feita antes da chamada do comando `while`. Além disto, a garantia de parada do laço é realizada pelo incremento da variável `i`, feito dentro do bloco de comandos do `while`.

Exemplo

Faça um programa que recebe dois números inteiros não-negativos `a` e `b` e realize a potenciação a^b.

Solução:

```
int main(){
    unsigned int a = 0, b = 0;
    cout << "Informe dois numeros inteiros nao-negativos\
       n";
    cin >> a >> b;
    unsigned int y = 1;
    while(b > 0){
        y *= a;
        b--;
    }
    return 0;
}
```

Importante notar que a inicialização do valor da variável b, envolvida na condição, foi feita antes da chamada do comando while, através do valor recebido pelo usuário. Também, a garantia de parada do laço é realizada pelo decremento da variável b, feito dentro do bloco de comandos do while.

O laço lógico, por não estar obrigatoriamente associado a um contador, pode permitir o uso de condições que dependem da variação de duas ou mais variáveis:

- `while(w > y && x <= z);`
- `while(a == 'x' || b == 'r' || i < j);`
- `while(n < 100 && erro >= 0.0001);`

Assim como o laço contado, o laço lógico é possível realizar a leitura de uma quantidade determinada de valores de entrada. No entanto, ele permite associar a sua condição diretamente com o valor recebido, mesmo que ele não seja propriamente numérico. Isto possibilita realizar a leitura de uma quantidade indeterminada de valores de entrada, bem como implementar interrupções do laço pelo usuário. Como exemplo, considere um programa que aguarda até o usuário digitar um número natural e, depois, imprima os primeiros n-ésimos números naturais não-nulos.

```
#include <iostream>

using namespace std;

int main (){
        int n = 0;
        while(n <= 0) {
                cout << "Digite um numero: ";
                cin >> n;
        }
        for(i = 1; i <= n; i++){
                cout << i << endl;
        }
        return 0;
}
```

São erros comuns no emprego do while:

- Não garantir que a condição é verdadeira ao ser testada pela primeira vez;

- Não garantir que a condição vai se tornar falsa na lógica do algoritmo;
- Inverter condição: a expressão lógica deve ter valor verdadeiro para que o algoritmo execute. Este tipo de situação está relacionada ao dilema condição de parada *versus* condição de prosseguimento: alguns problemas definem a condição do laço diretamente, outros estabelecem qual deve ser o mecanismo de parada, de forma que a expressão do laço precisa a negação desta. Como exemplo, considere um programa que lê **vários** números e informa se cada um deles é par ou ímpar, sendo que o programa deve se encerrar quando for digitado um número negativo.

```
1  int main(){
2      int x;
3      cout << "Informe um numero\n";
4      cin >> x;
5      while(x >= 0){ //Observe a condicao do laco.
6          if(x % 2 == 0){
7              cout << x << " e um numero par\n";
8          }
9          else{
10             cout << x << " e um numero impar\n";
11         }
12         cout << "Informe um numero\n";
13         cin >> x;
14     }
15     return 0;
16 }
```

Perceba que é solicitado que o laço deixe de executar quando `x < 0`. No entanto, a condição do laço determina o *prosseguimento* do bloco de instrução, e não diretamente o seu fim. Logo, a condição inversa a de parada deverá ser empregada, que seria `x >= 0`.

4.4 Comando `do-while`

Além do comando `while`, a linguagem C++ oferece também o comando `do-while` (faça enquanto). A sintaxe padrão, para várias instruções, é definida como:

```
do{
    comando1;
    ...
    comandoN;
} while(condicao);
```

O funciona deste comando é semelhante ao do `while`. A única - e importante - diferença é que, no `do-while`, a condição é testada após a execução do bloco de comandos. Neste caso, a ordem de execução é modificada, sendo apresentada a seguir:

1. A expressão de condição é avaliada:
   - Caso seja verdadeira: o bloco de comandos é executado;
   - Caso seja falsa: o laço é encerrado;
2. O passo 1 é executado;

Um exemplo de execução do `do-while` pode ser visto no programa anterior que imprima os 1.000.000-ésimos números naturais não-nulos.

```
int main(){
    int i = 1;
    do{
        cout << i << endl;
        i++;
    }while(i <= 1000000)
    return 0;
}
```

A reordenação da execução do `do-while` possui algumas implicações práticas. No comando `while`, é possível que o bloco de comandos não seja executado nenhuma vez, bastando que o valor das variáveis associadas no comando não torne a expressão de condição verdadeira. No caso do `do-while`, mesmo que tal situação aconteça, o bloco de comandos é executado pelo menos uma vez. Por causa disto, o `do-while` permite que as variáveis envolvidas na condição de prosseguimento não precisem satisfazer as condições impostas logo na primeira iteração. Neste caso, a definição delas pode ser levada para dentro do bloco de instruções.

Como exemplo, considere o exemplo a seguir. Ele implementa um programa que lê um número inteiro positivo `n` e em seguida, imprime o quociente e o resto da divisão de `n` por 10. Para garantir que o número seja positivo, o programa deve checar se o usuário digitou um número inteiro positivo, solicitando ao mesmo para redigitá-lo se não for o caso.

```
1  int main(){
2      int x;
3      do{
4          cout << "Insira um numero inteiro
5                    e positivo\n";
6          cin >> x;
7      } while(x < 0);
8
9      cout << "q: " << x/10
10          << ", r:" << x%10 << endl;
11
12     return 0;
13 }
```

4.5 Comandos `break` e `continue`

Existem dois comandos especialmente utilizados dentro do escopo de laços `for`, `while` e `do-while`, para controle de execução em casos especiais.

O primeiro deles é o comando `break`, que é utilizado para encerrar o laço em que o comando `break` se encontra. Ele é frequentemente utilizado em laços infinitos, para sinalizar a interrupção do programa. Um exemplo de emprego deste comando pode ser visto no exemplo a seguir, que implementa um menu.

```
int main(){
    char o;
    do{
        cout << "Insira uma opcao:\n";
        cout << "\t (1) opcao1\n";
        cout << "\t (2) opcao2\n";
        cout << "\t (3) opcao3\n";
        cout << "\t (s) air\n";
        cin >> o;
        if(o == '1'){
            cout << "opcao1 escolhida\n";
        }
        else if(o == '2'){
            cout << "opcao2 escolhida\n";
        }
        else if(o == '3'){
            cout << "opcao3 escolhida\n";
        }
        else if(o == 's'){
            cout << "Vc escolheu sair\n";
            break;
        }
    }while(true);
    return 0;
}
```

É possível perceber que o comando `break` pode ser omitido do comando, bastando levar a condição associada ao `break` para dentro da condição de avaliação do laço. Este é o mecanismo padrão para permitir a exclusão do `break`, sem alterar a lógica do programa. Considere o mesmo exemplo, sem o uso do break.

```
int main(){
        char o;
        do{
        cout << "Insira uma opcao:\n";
        cout << "\t (1) opcao1\n";
        cout << "\t (2) opcao2\n";
        cout << "\t (3) opcao3\n";
                cout << "\t (s) air\n";
        cin >> o;
        if(o == '1'){
            cout << "opcao1 escolhida\n";
        }
        else if(o == '2'){
            cout << "opcao2 escolhida\n";
        }
        else if(o == '3'){
            cout << "opcao3 escolhida\n";
        }
        else if(o == 's'){
            cout << "Vc escolheu sair\n";
        }
    }while(o != 's');
    return 0;
}
```

O outro comando `continue`, que é utilizado para ignorar a iteração atual do laço e forçar a próxima iteração. Um exemplo de execução deste comando pode ser visto no exemplo a seguir, que calcula a soma de todos os pares até `n`.

```
int main(){
    int n, i, s = 0;
    cout << "Informe um numero\n";
    cin >> n;
    for(i = 0; i < n; i++){
        if(i % 2 != 0){
            continue;
        }
        s += i;
    }
    cout << s << endl;
    return 0;
}
```

O mesmo programa pode ser implementado sem `continue`. Para tanto, é necessário estabelecer um comando de seleção que envolve os comandos que não estão associados ao continue, onde a condição é a negação da condição associada ao `continue`. Para ilustrar, considere o programa anterior sem o uso de `continue`.

```
int main(){
    int n, i, s = 0;
    cout << "Informe um numero\n";
    cin >> n;
    for(i = 0; i < n; i++){
        if(i % 2 != 0){
            s += i;
        }
    }
    cout << s << endl;
    return 0;
}
```

4.6 Sobre os comandos de repetição

Diferenças entre `while` e `for`:

- O comando `while` é apropriado para repetição condicional;
  - É possível utilizá-lo para repetição contada, entretanto fazer isto é mais sujeito a erros;
  - Assim como no caso do `for`: diversas expressões lógicas podem ser usadas para o mesmo efeito;
- O comando `for` é apropriado para repetição contada;
  - É possível utilizá-lo para repetição condicional, entretanto o código resultante fica menos legível;

Exemplo

Implemente um programa que lê dois números inteiros e imprime na tela o MDC (Máximo Divisor Comum) entre os dois números.

Solução:

```
1  int main(){
2      int x, y, mdc;
3      cout << "Insira dois numeros:\n";
4      cin >> x >> y;
5      if(x < y){
6          mdc = x;
7      }
8      else{
9          mdc = y;
10     }
11     while(x % mdc != 0 || y % mdc != 0){
12         mdc--;
13     }
14     cout << "O MDC entre os dois numeros e "
15          << mdc << endl;
16     return 0;
17 }
```

Exemplo

Implemente um programa que lê a população no ano corrente de dois países, A e B. O país A tem taxa de crescimento de 2% ao ano, enquanto o país B cresce 4% ao ano. Supondo que a população do país A é maior do que a do país B, o seu programa deve informar em quantos anos a população do país B ultrapassará a população do país A.

Solução:

```
int main(){
    int anos = 0;
    int pop_a, pop_b;

    cout << "Insira as populacoes iniciais:\n";
    cin >> pop_a >> pop_b;

    while(pop_b <= pop_a){
        pop_a *= 1.02;
        pop_b *= 1.04;
        anos++;
    }
    cout << "A populacao do pais B ultrapassa
             a do pais A em " << anos
                              << " anos.\n";

    return 0;
}
```

Exemplo

Faça um programa que lê um número `n` e um número `m`. Em seguida, o seu programa deve exibir na tela as `m` primeiras potências positivas dos `n` primeiros números positivos. As potências devem estar separadas por linha.
Exemplo:

Solução:

```
Informe a quantidade de numeros
5
Informe a quantidade de potencias
4
Potencias:
1 1 1 1
2 4 8 16
3 9 27 81
4 16 64 256
5 25 125 625
```

- **Dica:** comece com um programa que imprime as 3 primeiras potências positivas de `n` e em seguida, expanda este código para o programa solicitado

Solução:

```
int main(){
    int i, j, n, m;
    cout << "Informe a quantidade de numeros\n";
    cin >> n;
    cout << "Informe a quantidade de potencias\n";
    cin >> m;
    for(i = 1; i <= n; i++){
        for(j = 1; j <= m; j++){
            cout << pow(i,j) << " ";
        }
        cout << endl;
    }
    return 0;
}
```

4.7 Exercícios

1. Implemente um programa que lê um número `n` e em seguida, lê `n` caracteres. O seu programa deve exibir a quantidade de caracteres digitados que são letras, considerando que o usuário pode digitar letras minúsculas ou maiúsculas. Exemplo:

   ```
   Insira a quantidade de caracteres:
   5
   Insira os caracteres:
   A a 7 & J
   Total de letras digitadas: 3
   ```

2. (**Questão similar do URI: 1080**) Implemente um programa que recebe como entrada um número inteiro `n` e em seguida, lê `n` números inteiros informados pelo usuário. O seu programa deve imprimir na tela o menor e o maior número informados. Exemplo:

   ```
   Insira a quantidade de numeros
   5
   Insira os numeros
   2 7 -10000 988 3
   Menor: -10000, maior: 988
   ```

3. Implemente um programa que recebe como entrada um número inteiro `n` e em seguida, computa e imprime na tela o valor da seguinte fórmula:

   $$\sum_{i=1}^{n} i^2$$

   Para depurar o seu programa, imprima o resultado no seguinte formato:

   ```
   Insira um valor para n
   4
   Para n = 4: 1 + 4 + 9 + 16
   Total = 30
   ```

4. (**Questão similar do URI: 1153**) Mais fórmulas – Implemente um programa que receba como entrada um número real `x` e um número inteiro `n`. O seu programa deve computar e imprimir na tela as seguintes fórmulas. **Não utilize a biblioteca de funções matemáticas**.
   (a) x^n
   (b) $x! = \prod_{i=1}^{x} i$
   (c) $\sum_{i=1}^{n} 2x$
5. (**Questão similar do URI: 1078**) Implemente um programa que recebe como entrada um número inteiro `x` e imprime na tela a tabuada para o número `x`, formatada como no exemplo a seguir:

```
Insira um numero inteiro
5
5 x 1 = 5
5 x 2 = 10
...
5 x 10 = 50
```

6. Modifique o programa anterior, de modo que ele imprima a tabuada para todos os números, formatado como no exemplo a seguir (observe a tabulação):

```
Tabuada para o numero 1
1 x 1 = 1
1 x 2 = 2
...
1 x 10 = 10
...
Tabuada para o numero 10
10 x 1 = 10
10 x 2 = 20
...
10 x 10 = 100
```

7. Implemente um programa que lê um número `n` e um número `m`. Em seguida, o seu programa deve exibir na tela as `m` primeiras potências positivas dos `n` primeiros números positivos. As potências devem estar separadas por linha. Exemplo:

```
Informe a quantidade de numeros
5
Informe a quantidade de potencias
4
Potencias:
1 1 1 1
2 4 8 16
3 9 27 81
4 16 64 256
5 25 125 625
```

8. Implemente um programa que recebe como entrada dois números inteiros `l` e `h`. Estes números devem representar, respectivamente, a largura e a altura de um retângulo de asteriscos (`'*'`) a ser desenhado na tela. Exemplos:

```
-- Exemplo 1:
Insira a largura e a altura do retangulo:
4 2
****
****
-- Exemplo 2:
Insira a largura e a altura do retangulo:
10 5
**********
**********
**********
**********
**********
```

9. Implemente um programa que recebe como entrada dois números inteiros e positivos `x` e `y`. O seu programa deve computar e imprimir na tela o Máximo Divisor Comum (MDC) entre `x` e `y`, isto é, o maior número inteiro do qual `x` e `y` são múltiplos. Exemplos:

```
-- Exemplo 1:
Insira os valores de x e y:
6 8
O MDC entre 6 e 8 e 2
-- Exemplo 2:
Insira os valores de x e y:
16 36
O MDC entre 16 e 36 e 4
-- Exemplo 3:
Insira os valores de x e y:
70 90
O MDC entre 70 e 90 e 10
-- Exemplo 4:
Insira os valores de x e y:
0 5
O MDC entre 0 e 5 e 5
```

10. (**Questão similar do URI: 1160**) Implemente um programa que recebe como entrada a população no ano corrente de dois países, A e B. O país A tem taxa de crescimento de 2% ao ano, enquanto o país B cresce 4% ao ano. Supondo que a população do país A é maior do que a do país B, o seu programa deve informar em quantos anos a população do país B ultrapassará a população do país A. Exemplos:

```
-- Exemplo 1:
Insira as duas populacoes atuais:
1000 400
Populacao de B alcanca a de A em 48 anos
-- Exemplo 2:
Insira as duas populacoes atuais:
400 300
Populacao de B alcanca a de A em 16 anos
```

11. Implemente um programa que lê vários caracteres digitados pelo usuário. Para cada caractere digitado, o seu programa deve checar se o mesmo é uma letra minúscula e pedir ao usuário

que redigite o caractere se não for o caso. O seu programa deve informar se cada letra minúscula digitada é vogal ou consoante, até que o usuário digite o caractere '$', que deve fazer o programa se encerrar. Exemplo:

```
Informe uma letra minuscula:
T
Informe uma letra minuscula:
0
Informe uma letra minuscula:
c
>Consoante.
Informe uma letra minuscula:
a
>Vogal.
Informe uma letra minuscula:
~
Informe uma letra minuscula:
$
```

12. (**Questão similar do URI: 1018**) Implemente um programa que recebe como entrada vários números inteiros, sendo cada um deles correspondente a um montante total em reais. O seu programa deve decompor o montante informado nas cédulas de reais (notas de 100, 50, 20, 10, 5, 2 e 1), isto é, computar a quantidade máxima de cada cédula contida no montante informado, considerando primeiro as cédulas de maior valor. O programa deve se encerrar quando o usuário digita um número negativo. A saída do programa deve ser igual a do exemplo a seguir:

```
Insira a quantidade total de R$ (nr. negativo para sair)
312
3 notas de 100
0 notas de 50
0 notas de 20
1 notas de 10
0 notas de 5
1 notas de 2
0 notas de 1
Insira a quantidade total de R$ (nr. negativo para sair)
491
4 notas de 100
1 notas de 50
2 notas de 20
0 notas de 10
0 notas de 5
0 notas de 2
1 notas de 1
Insira a quantidade total de R$ (nr. negativo para sair)
-1
```

13. Implemente um programa para calcular o MDC (Máximo Divisor Comum) entre dois números inteiros e positivos `x` e `y` utilizando o algoritmo de Euclides. Este algoritmo funciona executando os seguintes passos:

- **Passo1**: calcule o resto `r` da divisão inteira entre `x` e `y`
- **Passo2**: faça `x` receber o valor de `y` e `y` receber o valor de `r`
- **Passo3**: se `r` for igual a 0, o MDC entre os dois números é igual ao valor atual de `x`. Caso contrário, volte ao Passo1

14. Implemente um programa que recebe como entrada dois números inteiros e positivos, `x` e `y`. O seu programa deve verificar se os números digitados pelo usuário são válidos e solicitar ao mesmo que os insira novamente caso contrário. Por fim, calcule e mostre na tela o Mínimo Múltiplo Comum (MMC) entre `x` e `y`, isto é, o menor número inteiro do qual `x` e `y` são divisores. Exemplos:

```
-- Exemplo 1:
Insira os valores de x e y:
6 8
O MMC entre 6 e 8 e 24
-- Exemplo 2:
Insira os valores de x e y:
16 36
O MMC entre 16 e 36 e 144
-- Exemplo 3:
Insira os valores de x e y:
12 6
O MMC entre 12 e 6 e 12
```

15. (**Questão similar do URI: 1164**) Chama-se de número perfeito um número que seja igual a soma dos seus divisores (excluindo ele mesmo). Implemente um programa que receba como entrada um número inteiro positivo `x` e informe se `x` é um número perfeito ou não. Exemplos:

```
-- Exemplo 1:
Informe um numero
4
4 nao e um numero perfeito
-- Exemplo 2:
Informe um numero
6
6 e um numero perfeito
-- Exemplo 3:
Informe um numero
28
28 e um numero perfeito
```

16. (**Questão similar do URI: 1151**) A sequência de Fibonacci é dada por

$$1, 1, 2, 3, 5, 8, 13, 21, ...,$$

isto é, ela é uma sequência infinita de números inteiros e positivos onde cada termo é formado pela soma dos dois anteriores, sendo os dois primeiros termos iguais a 1. Implemente um programa que imprima na tela todos os termos da sequência menores ou iguais do que um número `x` digitado pelo usuário, assumindo-se que este é maior ou igual a 1. Imprima a sequência em uma mesma linha, como mostrado nos exemplos a seguir:

```
-- Exemplo 1:
Informe um numero
```

```
1
1 1
-- Exemplo 2:
Informe um numero
50
1 1 2 3 5 8 13 21 34
```

17. Implemente um programa que receba como entrada um número inteiro positivo `x`. O seu programa deve imprimir na tela a soma dos dígitos que compõem `x`. Exemplos:

```
-- Exemplo 1:
Informe um numero
10
Soma dos digitos do numero: 1
-- Exemplo 2:
Informe um numero
1337
Soma dos digitos do numero: 14
```

18. (**Questão similar do URI: 1165**) Um número primo p é considerado um número primo de Sophie Germain se $2p+1$ também for primo. Por exemplo, $p = 2$ é um número primo de Sophie Germain, já que $2p+1 = 2 \times 2+1 = 5$ também é primo. Implemente um programa que receba como entrada um número inteiro positivo `p` e informa se o número:
    - Não é primo
    - É primo, mas não de Sophie Germain
    - É primo de Sophie Germain

    Exemplos:

```
-- Exemplo 1:
Informe um numero
2
2 e primo de Sophie-Germain
-- Exemplo 2:
Informe um numero
7
7 e apenas primo
-- Exemplo 3:
Informe um numero
4
4 nao e primo
```

II Parte 2: Funções

5 Funções: Básico 79
5.1 Introdução
5.2 Chamada de Funções
5.3 Assinatura de uma Função
5.4 Definição de uma Função
5.5 Escopo de Variáveis
5.6 Funções sem Retorno
5.7 Exercícios

6 Funções: Passagem por Referência . . 93
6.1 Introdução
6.2 Funções que chamam outras Funções
6.3 Exercícios

7 Recursão 107
7.1 Introdução
7.2 Recursão indireta
7.3 Exercícios

5. Funções: Básico

A partir deste ponto, será introduzido um modelo de programação denominado modular. Ele se baseia na ideia de que um problema mais complexo pode ser realizada pela integração adequada de soluções para problemas menores. Cada uma destas soluções menores é associada a uma tarefa específica, sendo possível reutilizá-la em outras atividades.

5.1 Introdução

Foi visto que variáveis guardam valores de diferentes tipos (`char`, `int`, `float`, `bool`). O conceito de funções permite utilizar tal abordagem para um trecho inteiro de código: associar um identificador a um determinado trecho de código. Denomina-se de **função** ou **subprograma** um conjunto de comandos (trecho de código) que juntos, desempenham uma tarefa em particular. Cada função recebe um nome e através deste, pode ser ativada por meio de uma chamada. O uso de funções organiza, simplifica e reduz programas.

O conceito de funções já vem sendo empregado de maneira não comentada até agora. São exemplos de funções conhecidas:

1. `sqrt`, `pow`, `sin`, `cos`, `tan` e `abs`, da biblioteca `cmath`;
2. `setioflags`, `setw` e `setfill`, da biblioteca `iomanip`;
3. `isalpha`, `isnum`, `tolower` e `toupper`, da biblioteca `cctype`.

O emprego de funções é importante por várias razões, entre elas:

- Tornar o código mais curto;
- Ajudar na legibilidade do código;
- Facilitar a portabilidade do código;
- Ajudar na depuração do código.

A aplicação de funções em programas é feito em duas etapas:

1. Na primeira, a função é criada, ou **implementada**;
2. Na segunda, a função é empregada ou **chamada** em um programa de fato;

É importante que uma função seja definida antes de ser utilizada. Caso contrário, o compilador indicará um erro.

5.2 Chamada de Funções

Funções já vêm sendo chamadas em vários programas, sendo importante notar que não é necessário saber como uma função está implementada para utilizá-la. Como exemplo, considere a função `cos(x)`, que computa o cosseno de `x`. Matematicamente, ela é definida a partir de um série de potências, como um polinômio de MacLaurin:

$$cos(x) = \sum_{n=0}^{\infty} \frac{(-1)^n}{(2n)!} x^{2n}$$

Embora seja uma informação relevante, pois o emprego deste tipo de recurso possui limitações numéricas associadas, ela não impede que um usuário leigo a empregue para calcular o cosseno de um número. Esta capacidade de "esconder" a implementação de uma solução para o suário é denominada *encapsulamento* e é um conceito bastante desejado em programação. A chamada de uma função funciona exatamente como o uso de uma variável ou constante; porém, ao invés de um valor, ela chama o trecho de código associado a ela.

A sintaxe da chamada de uma função é dada como:

```
<nome_da_funcao> (<valores_de_entrada>)
```

Onde:

- O valor de entrada pode ser um valor constante, variável, expressão ou mesmo a saída de outra função;
- A saída de uma função pode ser atribuída a uma variável, utilizada em uma expressão ou utilizada como entrada de outra função.

5.2.1 Definição de Funções

Linguagens de programação estruturadas oferecem uma sintaxe para implementação de funções. Entretanto, somente saber a sintaxe não garante a implementação correta de funções. Implementar funções corretamente envolve identificar:

1. As entradas necessárias do subprograma (parâmetros ou argumentos da função);
2. O que o subprograma deve computar (código da função);
3. As possíveis saídas do subprograma;

A implementação de uma função é dividida em duas partes:

- **Assinatura** ou declaração da função;
- **Definição**, que equivale ao conjunto de instruções que compõem a função propriamente dita.

5.3 Assinatura de uma Função

A assinatura de uma função também é chamada de cabeçalho ou protótipo da função. A partir dela são definidos:

- O nome da função;
- A lista de parâmetros de entrada da função;
- O tipo do retorno da função;

Por causa disto, ela tem impacto direto na chamada da função. A sintaxe da assinatura de uma função é dada como:

```
<tipo_da_funcao> <nome_da_funcao> (<lista_de_parametros>);
```

Onde:

- `tipo_da_funcao`: O tipo do valor a ser computado ou retornado pela função. Ele pode ser um tipo básico (`int, char, float, bool`), um tipo criado pelo usuário ou mesmo não apresentar tipo. Neste caso, diz-se uma função sem retorno, sendo indicada pelo tipo `void` no campo de tipo;
- `nome_da_funcao`: É o nome definido para a função, e segue as mesmas regras que usamos para nomear variáveis;
- `lista_de_parametros`: São variáveis definidas para que a função possa computar o seu retorno. A lista é composta de um tipo e um nome para cada variável, separados por vírgula.

São exemplos de assinaturas de funções:

- `int fatorial (int n);`
- `bool _isValid (bool a, bool b)` - Geralmente nomes que começam com _ estão associados a funções internas. Devem ser usadas com cautela;
- `double norm2(double x, double y, double z);`
- `double operation(int i, char j, double k, bool t)` - Uma função pode exigir parâmetros de tipos diferentes;
- `void printSum(double a, float b)` - Uma função pode não apresentar retorno;
- `int showConstant()` - Uma função pode não apresentar parâmetros de entrada.

Exemplo

Escreva assinaturas para as funções abaixo.

1. `sqrt`
2. `sin`
3. `pow`
4. Função que converte temperatura em graus Celsius para Fahrenheit
5. Função que converte um número no caractere da tabela ASCII correspondente;
6. Função que retorna o menor dentre três números reais;
7. Função que retorna verdadeiro caso o número seja par e falso caso o número seja ímpar;

Solução:

1. `float sqrt(float x);`
2. `float sin(float x);`
3. `float pow(float b, float e);`
4. `float converte_temp(float c);`
5. `char converte_caractere(int cod);`
6. `float menor(float a, float b, float c);`
7. `bool eh_par(int num);`

Não é necessário que toda a implementação da função esteja antes da sua chamada. No entanto, é obrigatório que a declaração de uma função esteja **antes** de funções que a usam (p. ex. antes da função `main`).

A assinatura de uma função define uma relação entre entrada e saída, algo parecido com o conceito matemático de funções. Neste caso, a entrada da função seria os parâmetros da função, enquanto que a saída da função estaria associada ao tipo da função. Por causa disto, alguns cuidados devem ser tomados durante a chamada de uma função.

- **O número de variáveis na chamada de uma função DEVE SER IGUAL ao número de parâmetros definido na assinatura da função**. Caso isto não aconteça, um erro será emitido pelo compilador;

- **As sequência imposta pela assinatura aos tipos da lista de parâmetros DEVE SER A MESMA para a chamada da função**. Caso isto não aconteça, o compilador pode indicar erro ou realizar uma conversão implícita, que seria o caso mais grave;
- **A variável ou estrutura que irá receber o valor de saída da função, caso seja necessário, DEVE SUPORTAR o tipo indicado pela assinatura da função.**

Para ilustrar estas observações, considere o seguinte exemplo de chamada de uma função.

- Assinatura da função: `double operation(int i, char j, double k, bool t);`
- Chamada da função:

```
int main() {
        int v4 = 15;
        double v3 = 4.75;
        bool v2 = true;
        char v1 = 'R';
        /*Sequencia incorreta. Compila, mas pode
        gerar um comportamento inadequado.*/
        cout << operation(v1, v2, v3, v4);
        /*Sequencia correta.*/
        cout << operation(v4, v1, v3, v2);
        /*Erro: falta parametros*/
        cout << operation(v4, v1, v3);
        return 0;
}
```

Observações:

- Algumas funções não necessitam de entrada;
- É possível utilizar parâmetros da função para armazenar saída (próximas aulas);

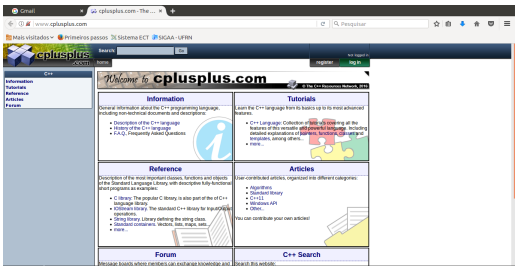
Uma questão importante é como fazer para verificar a chamada correta de uma função. Neste caso, se a função for implementada em uma biblioteca de terceiros, será necessário consultar o manual disponibilizado pelo desenvolvedor. Porém, caso a função a ser utilizada pertença a uma biblioteca STL é possível consultar a sua assinatura no site *cplusplus.com*. Isto mostra a importância de documentar BEM o código.

5.3.1 Consulta de assinaturas das Bibliotecas STL (C++)

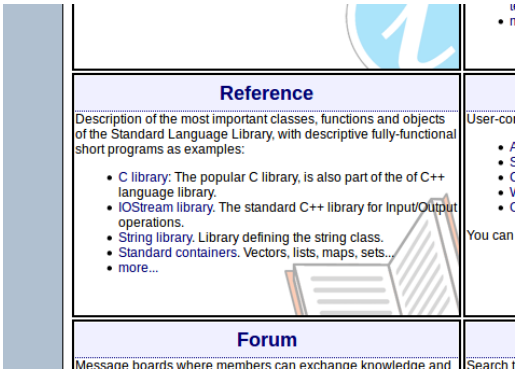
STL é uma abreviação para *standard template library*, ou biblioteca padrão de modelos. Trata-se um conjunto de bibliotecas nativo da linguagem, onde se encontram as funções mais comuns em C++. Fazem parte dela bibliotecas como:

- `iostream`;
- `cmath`;
- `cctype`;
- `iomanip`;

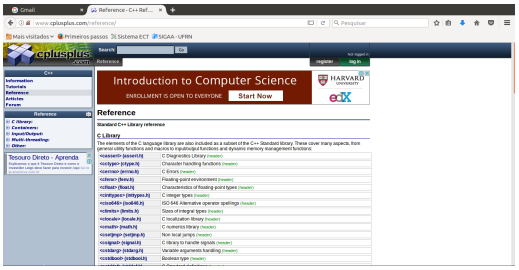
Existem bastante material sobre na *Internet*. O principal deles é o *site cplusplus.com* (que está em inglês). Para obter informações sobre uma função da STL, basta seguir os seguintes passos:



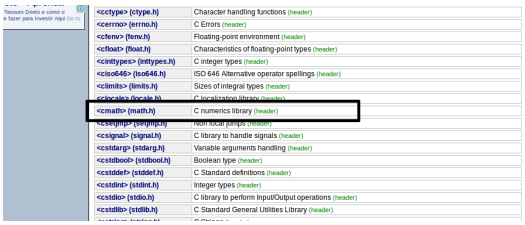
1: Acessar o *site* `cplusplus.com`.



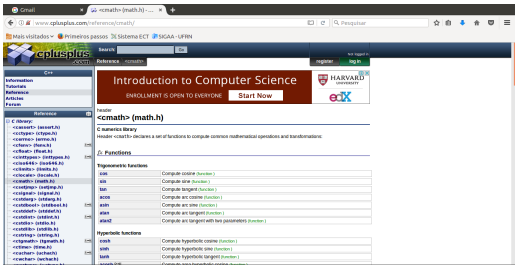
2: Procurar o campo *References*



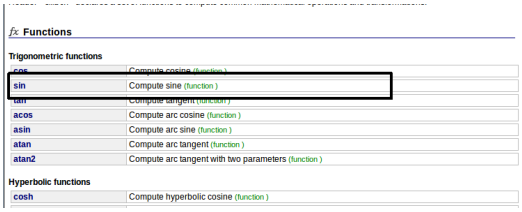
3: Neste ponto, haverá um redirecionamento para a seção das bibliotecas.



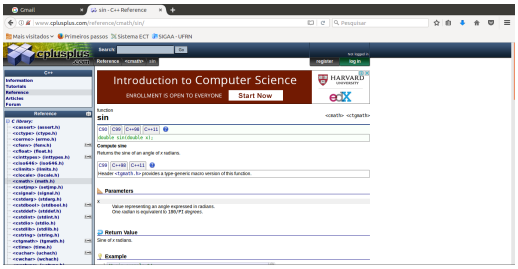
4: Acesse a biblioteca desejada para consulta.



5: Haverá um redirecionado para a seção das funções disponibilizadas na biblioteca.



6: Acessar a função desejada (é importante ler a descrição da função, caso não a conheça).



7: Haverá um redirecionamento para a seção de descrição da função.

5.4 Definição de uma Função

Uma função é definida quando é programado o seu comportamento com instruções válidas em C++. Nesta etapa, além dos comandos, há também a necessidade da assinatura desta. A sintaxe para a definição de uma função em C++ é dada como:

```
tipo_da_funcao nome_da_funcao(lista_de_parametros){
    corpo_da_funcao
}
```

A princípio, qualquer instrução é permitida no corpo da função, inclusive chamadas a outras funções (ou a ela mesma). Quando uma função apresenta um retorno, este precisa ser sinalizado; para tanto, utiliza-se o comando `return`. Este é usado junto com qualquer expressão válida em C++, e converte a expressão dada no tipo da função, caso seja possível. Caso não seja, o compilador sinaliza com uma mensagem de erro.É importante mencionar que o emprego do `return` **termina a execução da função**, de forma que o controle passa para a instrução seguinte à chamada da função executada.

Existem duas formas de declaração de uma função:

- **Forma 1**: A declaração e a definição da função é feita ao mesmo tempo, **antes** da função `main`.

```
float funcao(float c){
    ...
}

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

- **Forma 2**: A declaração da função é feita inicialmente, com a sua assinatura seguida por um ponto e vírgula. Depois, é realizada a definição da função, geralmente **após** a função `main`.

```
float funcao(float c);

int main(){
    ...
}

float funcao(float c){
    ...
}
```

O emprego da segunda forma é desejado, sendo considerada uma boa prática de programação, já que permite a listagem dos comandos ao usuário. Para mostrar o uso das duas formas, considere o exemplo de um programa que converte temperatura c em graus Celsius para Fahrenheit, valendo-se da equação 5.1, para tal. Além disto, há o emprego da função implementada em um programa onde o usuário deve informar o valor da temperatura em graus Celsius, e o programa deve imprimir o valor em graus Fareinheit.

$$f(c) = 1,8c + 32 \tag{5.1}$$

Utilizando a primeira forma de definição, tem-se:

```
#include <iostream>

using namespace std;

float converte_temp(float c){
    return 1.8*c + 32;
}

int main() {
        float celsius, farenheit;
        cout << "Digite a temperatura em graus Celsius: " <<
           endl;
        cin >> celsius;
        fareinheit = converte_temp(celsius);
        cout << "Temperatura em graus Farenheit: ";
        cout << fareinheit << endl;
        return 0;
}
```

Pela segunda forma de definição:

```
#include <iostream>

using namespace std;

float converte_temp(float c);

int main() {
        float celsius, farenheit;
        cout << "Digite a temperatura em graus Celsius: " <<
           endl;
        cin >> celsius;
        fareinheit = converte_temp(celsius);
        cout << "Temperatura em graus Farenheit: ";
        cout << fareinheit << endl;
        return 0;
}

float converte_temp(float c){
    return 1.8*c + 32;
}
```

Neste ponto da disciplina, você deve ser capaz de entender porque todo programa deve ter `int main()` e `return 0` no seu corpo. Uma limitação do emprego de funções com o `return` é que não há suporte a mais de uma saída cos seja necessário. Neste caso, outra alternativa precisará ser empregada.

5.5 Escopo de Variáveis

Dentro das funções, pode haver a necessidade de se criar variáveis, como acontece em outras estruturas. Neste caso, há uma questão importante a ser tratada, que é o limite de existência de uma variável em um programa. Isto porque as variáveis podem estas definidas de duas formas:

- Globalmente: As variáveis globais não são definidas dentro de estruturas: geralmente são definidas no mesmo nível das diretivas de pré-processamento ou `using namespace`. Por causa disto, são vistas e podem ser utilizadas em qualquer momento do programa, por qualquer função ou estrutura de fluxo;
- Localmente: As variáveis locais, por outro lado, são definidas dentro de alguma estrutura da linguagem (estrutura de fluxo ou função). Por causa disto, são vistas e pode ser utilizadas apenas em algum momento do programa.

Denomina-se de **escopo** de uma variável a região do programa onde ela pode ser vista e, consequentemente, pode ser utilizada. Em C++, o escopo de uma variável é delimitado por chaves (`{ ... }`).

No caso específico de variáveis locais em funções, elas não são visíveis fora desta. Isto possui algumas implicações práticas:

- Usar o mesmo nome de uma variável que existe fora da função não implica em uma referência à esta última. Isto porque, caso haja a definição de uma variável local na função com o mesmo nome, será dada referência a esta última. Um exemplo disto pode ser visto em um programa que calcula a força peso sobre um corpo cuja massa é informada pelo usuário.

```
1
2  #include <iostream>
3
4  using namespace std;
5
6  float g = 9.81;
7
8  double peso(double massa) {
9          float g = 9;
10         return massa*g
11 }
12
13 int main() {
14         double m;
15         cout << "Digite o valor da massa:" << endl;
16         cin >> m;
17         cout << "A forca peso associada vale " << peso(m)
               << endl;
18         return 0;
19 }
```

  Caso o usuário digite `1.0`, o resultado da execução deste programa seria `9`, ao invés de `9.81`. Isto acontece porque o valor de `g` utilizado é aquele declarado internamente na função `peso`, e não o valor da variável global;
- Uma variável definida em uma função não pode ser utilizada em outra função ou na função `main`, pois são vistas apenas dentro da definição da função que ele é declarada. Como exemplo, considere uma modificação do programa anterior:

```
#include <iostream>

using namespace std;

float g = 9.81;

double peso(double massa) {
        float f = massa*g;
        return f;
}

int main() {
        double m;
        cout << "Digite o valor da massa:" << endl;
        cin >> m;
        cout << "A forca peso associada vale " << f <<
           endl; //Erro: 'f' nao existe para a main.
        return 0;
}
```

Neste caso, a variável `f` foi declarada dentro da função `peso`. Consequentemente, ela só existe durante a definição dessa. Portanto, não há definição da variável `f` na execução da função `main`, e um erro é sinalizado;

- Cada argumento da função também é uma variável local. Isto significa que é possível atribuir novos valores a um argumento durante a execução de um programa. Também, é possível passar o valor de um argumento com retorno da função. No entanto, **não é possível acessar um argumento fora da função que a define**. Um exemplo desta limitação pode ser visto em uma modificação do programa anterior.

```
#include <iostream>

using namespace std;

float g = 9.81;

double peso(double massa) {
        float f = massa*g;
        return f;
}

int main() {
        cout << "Digite o valor da massa:" << endl;
        cin >> massa;
        cout << "A forca peso associada vale " << peso(
           massa) << endl;
        /* Erro: 'massa' nao existe para a main. */
        return 0;
}
```

Este é um tipo de erro muito comum entre os alunos. Portanto, é importante deixar claro que um argumento de uma função não pode ser empregado na definição de outras funções ou na

`main`, pois ele só existe no escopo da função que a criou;

- Uma variável passada como entrada na chamada da função pode ter o mesmo nome que um parâmetro da função. Porém são duas variáveis distintas: a primeira é definida fora do escopo da função que é chamada, a outra é definida dentro da função que a solicita. Como exemplo, observa a seguinte modificação no programa que calcula a força peso.

```
1  #include <iostream>
2
3  using namespace std;
4
5  float g = 9.81;
6
7  double peso(double massa) {
8          /*Aqui, 'massa' e um parametro da funcao 'peso'.
               */
9          float f = massa*g;
10         return f;
11 }
12
13 int main() {
14         /*Aqui, 'massa' e uma variavel da funcao 'main'.
               */
15         double massa;
16         cout << "Digite o valor da massa:" << endl;
17         cin >> massa;
18         cout << "A forca peso associada vale " << peso(
               massa) << endl;
19         return 0;
20 }
```

Aqui, a variável `massa` da função `main` não é a mesma variável `massa` da função `peso`.

5.6 Funções sem Retorno

Como dito anteriormente, é possível criar função que não apresentam retorno. Funções deste tipo, também conhecidas como procedimentos, geralmente são úteis para a modificação de parâmetros e a exibição de informações (funções que devem escrever na tela). Aqui, é importante alertar que **escrever na tela com comando `cout` não gera retorno de função**, pois o valor do retorno pode ser manipulado em outros momentos do programa, o que não ocorro com a impressão na tela. Neste caso, o comando `return` não é obrigatório; podendo ser utilizado sem nenhuma expressão para encerrar a execução do procedimento.

Como exemplo, considere uma função que **imprime na tela** o caractere correspondente a um número passado como parâmetro, de acordo com a tabela ASCII.

```
1  void imprime_carac(int n);
2
3  int main(){
4      int x;
5      cin >> x;
6      imprime_carac(x);
7      return 0;
8  }
9
10 void imprime_carac(int n){
11     cout << (char) n << endl;
12 }
```

Atenção à chamada da função, uma vez que a função não retorna nenhum valor. Neste caso, não há a necessidade de utilizar uma variável para armazenar o valor ou mesmo de utilizar o comando `cout` para imprimir o valor associado a chamada, pois ele não existe. O que a função faz é apenas imprimir na tela o valor numérico do caractere.

5.7 Exercícios

1. Implemente funções para calcular:
   - A área de um retângulo $A_{ret} = b \times h$, onde b é a sua base e h é a sua altura
   - A área de um triângulo $A_{tri} = (b \times h)/2$, onde b é a sua base e h é a sua altura
   - A área de um círculo $A_{circ} = \pi \times r^2$, onde r é o raio

   Dica: pense em termos de entrada e saída para descobrir os parâmetros de cada uma destas funções. Em seguida, implemente a função `main` de forma que o usuário entre com os dados necessários para calcular cada uma das áreas utilizando chamadas às funções implementadas.
2. Implemente uma função que receba um número inteiro e retorne o valor absoluto do número. Implemente também a função `main`, de modo que o usuário possa digitar um número inteiro qualquer e visualizar o valor absoluto do número digitado na tela utilizando a função implementada. Exemplos:

   ```
   -- Exemplo 1:
   Insira um numero inteiro:
   -157
   Modulo do numero:
   157
   -- Exemplo 2:
   Insira um numero inteiro:
   1021
   Modulo do numero:
   1021
   ```

3. Implemente uma função que receba três letras minúsculas como parâmetros. A sua função deve computar qual a maior letra, de acordo com a ordem alfabética. Implemente também a função `main`, de modo que o usuário possa digitar as letras e visualizar na tela qual a maior delas de acordo com a chamada à função.

   ```
   -- Exemplo 1:
   Insira tres letras:
   p a k
   ```

```
Maior letra:
p
-- Exemplo 2:
Insira tres letras:
a b c
Maior letra:
c
```

4. Implemente uma função chamada `muda_tamanho`, que deve receber como parâmetro um caractere. Caso o caractere seja uma letra minúscula, a função deve retornar a sua versão em maiúscula. Caso o caractere seja uma letra maiúscula, a função deve retornar a sua versão em minúscula. Implemente a função `main` de forma que o usuário possa inserir uma letra e visualizar na tela o resultado computado pela função. Exemplos:

```
-- Exemplo 1:
Informe uma letra:
Q
Conversao:
q
-- Exemplo 2:
Informe uma letra:
e
Conversao:
E
```

5. Implemente uma função para cada uma das funções matemáticas a seguir. Implemente também uma função `main` que receba como entrada um número real `x` e um número inteiro `n` e imprima o resultado de cada uma das funções. **Não utilize a biblioteca de funções matemáticas**.
   (a) x^n
   (b) $x! = \prod_{i=1}^{x} i$
   (c) $\sum_{i=1}^{n} 2x$
6. Implemente uma função chamada `eh_primo`, que retorna verdadeiro caso um número inteiro passado como parâmetro seja primo e falso caso contrário. Implemente também a função `main`, que deve ler do usuário vários números inteiros positivos e imprimir na tela uma mensagem informando se cada um deles é primo ou não, de acordo com uma chamada à função. O programa deve se encerrar quando o usuário inserir um número menor ou igual a 0. Exemplo:

```
Informe um numero positivo (negativo para encerrar):
3
3 eh primo
Informe um numero positivo (negativo para encerrar):
4
4 nao eh primo
Informe um numero positivo (negativo para encerrar):
2
2 eh primo
Informe um numero positivo (negativo para encerrar):
```

```
-1
```

7. Implemente uma função chamada `conta_digitos`, que deve receber como parâmetro um número inteiro maior ou igual a zero e computar a quantidade de dígitos do número. Implemente também a função `main`, de modo que o usuário possa digitar um número inteiro maior ou igual a zero e, utilizando a função implementada, visualizar na tela quantidade de dígitos do número de acordo com o resultado da função. Exemplos:

```
-- Exemplo 1:
Informe um numero:
1461
O numero tem 4 digitos
-- Exemplo 2:
Informe um numero:
0
O numero tem 1 digitos
```

8. A sequência de Fibonacci é dada por

$$1, 1, 2, 3, 5, 8, 13, 21, ...,$$

isto é, ela é uma sequência infinita de números inteiros e positivos onde cada termo é formado pela soma dos dois anteriores, sendo os dois primeiros termos iguais a 1. Implemente uma função que compute o termo `k` da sequência, onde `k` é um parâmetro dado por um número inteiro maior ou igual a 0. Implemente também a função `main`, de modo que o usuário possa digitar um número inteiro `n` correspondente a quantidade de termos da sequência e visualizar na tela todos os `n` primeiros termos da sequência. Exemplos:

```
-- Exemplo 1:
Informe a quantidade de termos:
1
Sequencia de Fibonacci:
1
-- Exemplo 2:
Informe a quantidade de termos:
9
Sequencia de Fibonacci:
1 1 2 3 5 8 13 21 34
```

6. Funções: Passagem por Referência

Funções que produzem mais de um resultado possível são comuns em linguagens de programação. Também, funções que são utilizada apenas para modificar uma variável também aparecem em programação. Para dar suporte a estes mecanismos, geralmente utiliza-se o conceito de passagem de parâmetro por referência, que é a passagem do próprio espaço de memória associado a variável para função.

6.1 Introdução

Até agora, o modelo de passagem de parâmetros não permite a alteração do valor do parâmetro após a execução da função. Isto acontece porque é passada uma cópia do valor da variável é passado e não a variável em si. Este tipo de definição para a entrada de uma função é denominada **passagem de parâmetro por valor**, sendo o padrão habitual de entrada em C++. Este mecanismo possui a seguinte característica: por se passar a cópia da variável passada como parâmetro, e não a própria variável passada, caso haja alguma alteração no valor do parâmetro, este refletirá na cópia da variável, e não na própria. Isto permite preservar o valor da variável após a execução do programa, o que é desejável em alguns casos.

Como alternativa para a passagem por valor, C++ disponibiliza um mecanismo de **passagem de parâmetro por referência**, onde a variável em si é passada como parâmetro. Por não ser o padrão na linguagem, a passagem por referência precisa ser explicitamente indicada pelo programador. Para isto, utiliza-se o operador de referência `&`. Por passar a variável em si, qualquer alteração feita no parâmetro será passada a variável em si, de forma a alterá-la. Isto é útil quando existe a possibilidade da função modificar os parâmetros.

Para ilustrar na prática estas diferenças, considere um programa que realiza a troca de valor entre duas variáveis. Até este momento, a implementação deste problema poderia ser feita da seguinte forma:

```
void troca(int a, int b);

int main(){
    int n1, n2;
    cin >> n1 >> n2;
    troca(n1, n2);
    cout << n1 << " " << n2 << endl;
    return 0;
}

void troca(int a, int b){
    int aux = a;
    a = b;
    b = aux;
}
```

Neste caso, a implementação utiliza passagem por valor, que não altera os valores de `n1` e `n2`. Logo, ele não se comporta como desejado. A solução correta para este problema é feita com o uso de passagem de parâmetro por referência. Uma implementação possível, e que funciona adequadamente, é vista a seguir:

```
void troca(int& a, int& b);

int main(){
    int n1, n2;
    cin >> n1 >> n2;
    troca(n1, n2);
    cout << n1 << " " << n2 << endl;
    return 0;
}

void troca(int& a, int& b){
    int aux = a;
    a = b;
    b = aux;
}
```

Com a passagem por referência, também é possível enviar valores com saída para funções. Para isto, basta estabelecer parâmetros que sejam passados pro referência. É possível haver os dois tipos de parâmetros em uma função: por valor e por referência. Neste caso, é importante mencionar que **a ordem estabelecida para os parâmetros na assinatura da função também vale para o tipo de passagem de parâmetro**).

Como exemplo, considere uma função que calcula as raízes de uma equação do segundo grau, bem como um programa que recebe os coeficientes do usuário e imprime os valores das raízes.

```
#include <iostream>
#include <cmath>

using namespace std;

void raizes2grau (float a, float b, float c, float &x1, float
    &x2) {
        float delta = b*b - 4*a*c;
        if(delta < 0) {
                x1 = -1;
                x2 = -1;
        }
        else {
                x1 = (-b + sqrt(delta))/(2*a);
                x2 = (-b - sqrt(delta))/(2*a);
        }
}

int main() {
        float a = 0, b = 0, c = 0, r1 = 0, r2 = 0;
        cout << "Digite os coeficientes a, b e c:" << endl;
        cin >> a >> b >> c;
        float res = raizes2grau(a, b, c, r1, r2);
        if(res == 0) {
                cout << Nao ha raizes reais << endl;
        }
        else {
                cout << "Raizes: " << r1 << " " << r2 << endl;
        }
        return 0;
}
```

É importante notar que os parâmetros `a`, `b` e `c` da funçao `funcao2grau` são parâmetros passados por valor, enquanto que `x1` e `x2` são parâmetros passados por referência. No caso dos dois últimos, isto acontece porque o valor valor das raízes, que o retorno na função, será armazenado nestas variáveis. Além disto, na função `main`, duas variáveis `x1` e `x2` são definidas justamente para receber os valores retornados via passagem por referência. Isto deve ser feito sempre quando se faz uma chamada a funções que retornam valores por referência: declarar variáveis para armazenar o valor do retorno (lembre-se que parâmetros de função não podem ser utilizados em outras funções).

Mesmo quando há passagem de parâmetros por referência, é possível utilizar o comando returno, e passar valores com retorno da função. Isto é uma flexibilidade de C++ que permite, por exemplo, enviar informações sobre a execução de uma função como retorno, e dados de interesse como parâmetros por referência. Como exemplo, veja uma modificação do código anterior, onde o número de raízes é passado como retorno da função, enquanto que as raízes propriamente ditas são obtidas por parâmetros passados por referência.

```
#include <iostream>
#include <cmath>

using namespace std;

int raizes2grau (float a, float b, float c, float &x1, float &
    x2) {
        float delta = b*b - 4*a*c;
        if(delta < 0) {
                x1 = -1;
                x2 = -1;
                return 0;
        }
        else if(delta == 0) {
                x1 = x2 = (-b)/(2*a);
                return 1;
        }
        else {
                x1 = (-b + sqrt(delta))/(2*a);
                x2 = (-b - sqrt(delta))/(2*a);
                return 2;
        }
}

int main() {
        float a = 0, b = 0, c = 0, x1 = 0, x2 = 0;
        cout << "Digite os coeficientes a, b e c:" << endl;
        cin >> a >> b >> c;
        float res = raizes2grau(a, b, c, x1, x2);
        if(res == 0) {
                cout << Nao ha raizes reais << endl;
        }
        else {
                cout << "Raizes: " << x1 << " " << x2 << endl;
        }
        return 0;
}
```

Observe que, na `main`, três variáveis são utilizadas: Uma (`res`) para obter o retorno, e outras duas (`x1` e `x2`), para obter as raízes da equação. Isto permite por exemplo, saber se a raiz -1 é uma raiz inválida ou não, bastando apenas verificar que o valor de retorno é 0 ou diferente de 0.

Um resumo sobre passagem por referência:

- Uso de operador &
- Útil quando:
  - Uma função deve modificar os seus parâmetros;
  - **Uma função deve retornar mais de um valor**;
- Permite trabalhar com parâmetros:
  - De entrada: parâmetros necessários para computar a função;
  - **De entrada e saída**: parâmetros necessários para computar a função e usados para armazenar o resultado;
  - **De saída**: parâmetros usados apenas para armazenar resultados da função.

Exemplo

Implemente uma função que receba como parâmetro de entrada três números inteiros. A função a ser implementada deve armazenar os números em ordem crescente (no primeiro parâmetro está o menor, no segundo, está o segundo menor, etc.).

Solução:

```
void ordena_cresc(int& x, int& y, int& z){
    int a = x, b = y, c = z;
    if(x <= y && y <= z){
        x = a;
        y = b;
        z = c;
    }
    else if(x <= z && z <= y){
        x = a;
        y = c;
        z = b;
    }
    else if(y <= x && x <= z){
        x = b;
        y = a;
        z = c;
    }
    else if(y <= z && z <= x){
        x = b;
        y = c;
        z = a;
    }
    else if(z <= x && x <= y){
        x = c;
        y = a;
        z = b;
    }
    else if(z <= y && y <= x){
        x = c;
        y = b;
        z = a;
    }
}
```

6.2 Funções que chamam outras Funções

Até agora, foi visto implementações de funções que são utilizadas diretamente na função. Porém, foi visto também caso de funções que fazem chamadas à outras funções. Isto tem particular importância na **modularização de problemas**.

- Cada função realiza uma tarefa específica do programa todo;
- Cada vez que uma função precisa de fazer uma tarefa já implementada em outra função, ela realiza uma chamada a esta;

O modelo de chamada em outra é igual ao que foi visto até agora (inclui-se também as restrições).

Como exemplo, considere a implementação de uma função que receba como parâmetro de entrada dois números inteiros `x` e `y` e computar o MDC (Máximo Divisor Comum) e o MMC (Mínimo Múltiplo Comum) entre `x` e `y` sem modificar os parâmetros originais. Além disto, apresenta também a função `main`, para ler os dois números do usuário, chamar a função e exibir na tela o MDC e o MMC entre os números.

```
1  void computa_mdc_mmc(int x, int y, int& mdc, int& mmc);
2  int computa_mdc(int x, int y);
3  int computa_mmc(int x, int y);
4  int main(){
5      int a, b, mdc, mmc;
6      cin >> a >> b;
7      computa_mdc_mmc(a, b, mdc, mmc);
8      cout << "MDC: " << mdc << endl;
9      cout << "MMC: " << mmc << endl;
10
11     return 0;
12 }
13 void computa_mdc_mmc(int x, int y, int& mdc, int& mmc){
14     mdc = computa_mdc(x, y);
15     mmc = computa_mmc(x, y);
16 }
17
18 int computa_mdc(int x, int y) {
19     int mdc;
20     if(x < y){
21         mdc = x;
22     }
23     else{
24         mdc = y;
25     }
26     while(x % mdc != 0 || y % mdc != 0){
27         mdc--;
28     }
29     return mdc;
30 }
31
32 int computa_mmc(int x, int y) {
33     int mmc;
34     if(x < y){
35         mmc = y;
36     }
37     else{
38         mmc = x;
39     }
40     while(mmc % x != 0 || mmc % y != 0){
41         mmc++;
42     }
43     return mmc;
44 }
```

Observe que a solução proposta faz uso da função `computa_mdc_mmc`, que recebe quatro parâmetros: dois de entrada (`x` e `y`) e dois de saída (`mdc` e `mdc`). Por sua vez, cada parâmetro de saída recebe o valor da chamada a funções que calculam, separadamente, o MDC e MMC para os

dois valores de entrada `x` e `y`.

6.2.1 Modularização com arquivos `.h`

Um efeito desejado em programas que disponibilizam código-fonte ao usuário é o isolamento da assinatura das implementações das funções. Isto facilita a legibilidade por parte do usuário pois basta a este consultar apenas o arquivo com as assinaturas para especificar corretamente a chamada a função desejada em outros trechos de código. Este conceito é denominado **encapsulamento**, e costuma ser importante em projetos de sistemas.

No exemplo anterior, as assinaturas `void computa_mdc_mmc(int x, int y, int& mdc, int& mmc);`, `int computa_mdc(int x, int y);` e `int computa_mmc(int x, int y);` poderiam se disponibilizadas em um arquivo separadamente, enquanto que as implementações das funções estariam em outro arquivo. Nesta implementação, os dois arquivo precisariam ser referenciados no arquivo com a função `main`.

A linguagem C++ oferece um recurso para separar as *assinaturas* das *implementações*. Para tanto, além do arquivo `.cpp`, é necessário um arquivo de outra extensão, a `.h` . Esta extensão serve para armazenar as assinaturas das funções de maneira isolada, deixando o arquivo `.cpp` apenas para as implementações.

Utilizando o conceito de separação de arquivos, considere os arquivos para o programa:

- Arquivo `MDCMMC.h`:

```
1 #ifndef MMCMDC_H
2 #define MMCMDC_H
3
4 void computa_mdc_mmc(int x, int y, int& mdc, int& mmc);
5 int computa_mdc(int x, int y);
6 int computa_mmc(int x, int y);
7
8 #endif // MMC-MDC_H_INCLUDED
```

- Arquivo `MDCMMC.cpp`:

```cpp
#include "MMCMDC.h"

void computa_mdc_mmc(int x, int y, int& mdc, int& mmc){
    mdc = computa_mdc(x, y);
    mmc = computa_mmc(x, y);
}

int computa_mdc(int x, int y) {
    int mdc;
    if(x < y){
        mdc = x;
    }
    else{
        mdc = y;
    }
    while(x % mdc != 0 || y % mdc != 0){
        mdc--;
    }
    return mdc;
}

int computa_mmc(int x, int y) {
    int mmc;
    if(x < y){
        mmc = y;
    }
    else{
        mmc = x;
    }
    while(mmc % x != 0 || mmc % y != 0){
        mmc++;
    }
    return mmc;
}
```

- Arquivo `main.cpp`:

```cpp
#include <iostream>

#include "MMCMDC.h"

using namespace std;

int main()
{
    int x, y, mdc, mmc;
    cout << "Insira dois numeros:\n";
    cin >> x >> y;
    computa_mdc_mmc(x, y, mdc, mmc);
    cout << "MDC: " << mdc << endl;
    cout << "MMC: " << mmc << endl;
    return 0;
}
```

No arquivo `MMCMDC.h` se encontram as assinaturas das funções que serão implementadas em um arquivo `.cpp` separado. A primeira parte do código que chama atenção são as diretivas `#ifndef` e `#endif`. Elas existem juntas, e devem envolver toda a declaração das assinaturas no arquivo `.h`. São utilizadas para verificar se já existe uma definição anterior das funções. Caso não exista, a diretiva `#define` é executada, inserindo o arquivo `MMCMDC.h`. Caso já exista, a diretiva não é executada, e a declaração anterior prevalece. Isto evita erros de compilação por múltiplas definições do mesmo arquivo.

No arquivo `MMCMDC.cpp` se encontram as implementações das funções propriamente dito. Este arquivo deve ter mesmo nome do arquivo `.h` associado, para facilitar a associação pelo usuário. No início do arquivo há a chamada a diretiva `#include` para inserir o arquivo `MMCMDC.h`, entre aspas duplas: isto é necessário para permitir a integração da implementação com o as assinaturas no arquivo com a função principal.

O arquivo que apresenta da definição da função `main` é o arquivo homônimo `main.cpp`. Para que as implementações das funções sejam associadas as chamadas equivalentes no arquivo binário, também é necessário o uso da diretiva `#include "MMCMDC.h"`.

Exemplo

O problema básico de otimização consiste em encontrar o maior valor de uma função $f(x)$, quaisquer que seja ela. No caso numérico, esta busca pode ser feita dentro de um intervalo $[a,b]$, bastando definir o valor do passo h entre dois valores consecutivos. Quanto menor o valor de h, mais preciso será o valor, pois mais valores serão buscados. Crie uma função que retorne o máximo valor de uma função $y = f(x)$ qualquer (faça de conta que ela é uma função já implementada), dentro do intervalo $[a,b]$ com passo h. Também, deixe esta função com parâmetro de saída como sendo o valor de x que maximiza esta função. Teste esta função em um programa para verificar o máximo valor possível para a função $f(x) = -x^2 + 6$.

Solução:

```cpp
#include <iostream>
#include <cmath>

using namespace std;

float f(float x);
float findMax(float a, float b, float h, float &xm);

int main(){
        float a, b, h, xm;
        cout << "Digite os valores de a, b e h,
           respectivamente: ";
        cin >> a >> b >> h;
        float m = findMax(a, b, h, xm);
        cout << "O maior valor de f(x) no intervalo dado
           e " << m << endl;
        cout << "O valor maximo e obtido quando x = " <<
           xm << endl;
    return 0;
}

float findMax(float a, float b, float h, float &xm) {
        xm = a;
        float m = f(a);
        for(float i = a; i <= b; i += h) {
                if(f(i) > m) {
                        m = f(i);
                        xm = i;
                }
        }
        return m;
}

float f(float x) {
        return -x*x + 6;
}
```

6.3 Exercícios

1. Implemente uma função para imprimir na tela todos os dias da semana. A função `main` deve exibir na tela todos os dias da semana de acordo com uma chamada à função implementada.
2. Modifique a função anterior, de modo que agora seja impresso na tela o dia da semana correspondente a um inteiro passado como parâmetro de entrada: 0 imprime `domingo`, 1 imprime `segunda` e assim por diante, até 6, que imprime `sabado`. Para números fora do intervalo fechado de 0 a 6, a função deve imprimir `numero invalido`. A função `main` deve ler um número inteiro do usuario e exibir na tela o dia da semana correspondente ao número informado, de acordo com o resultado da função implementada.

3. Sabendo-se que a conversão de temperatura de Fahrenheit para Celsius é dada por

$$C = \frac{5}{9}(F - 32),$$

implemente uma função para realizar esta conversão. Faça isto de duas maneiras:
   (a) Com uma função que receba um número real como parâmetro de entrada e retorne um outro número real como o resultado da função.
   (b) Com uma função que receba um número real como parâmetro de entrada e um outro número real como parâmetro de saída. O resultado da função deve ser armazenado neste parâmetro de saída.

A função `main` deve ler um número real do usuário e exibir o resultado na tela de acordo com o resultado da função implementada. Exemplos:

```
-- Exemplo 1:
Informe a temperatura em Fahrenheit: 32
Temperatura em Celsius: 0
-- Exemplo 2:
Informe a temperatura em Fahrenheit: 80.0
Temperatura em Celsius: 26.6667
```

4. Implemente uma função que receba como parâmetro de entrada um número real e como parâmetros de saída, um número inteiro e um número real. A função a ser implementada deve armazenar a parte inteira e parte real do parâmetro de entrada nos parâmetros de saída adequados. A função `main` deve ler um número real do usuário, representando uma quantia em dinheiro, e exibir na tela uma mensagem informando quantos reais e quantos centavos compõem a quantia, de acordo com o resultado da função implementada. Exemplos:

```
-- Exemplo 1:
Informe uma quantia em dinheiro:
0.52
0 reais e 52 centavos
-- Exemplo 2:
Informe uma quantia em dinheiro:
15.30
15 reais e 30 centavos
```

5. Implemente uma função que receba um número inteiro como parâmetro de entrada e outros três números inteiros como parâmetros de saída. A função a ser implementada deve converter a quantidade de segundos do parâmetro de entrada em horas, minutos e segundos, armazenando estes valores em cada um dos parâmetros de saída. A função `main` deve ler a quantidade de segundos do usuário e exibir na tela a quantidade de horas, minutos e segundos no formato `HH:MM:SS` de acordo com o resultado da função implementada. Assuma que o usuário irá fornecer um número entre 0 e 86399. Exemplos:

```
-- Exemplo 1:
Informe a quantidade de segundos:
60
Hora:
00:01:00
-- Exemplo 2:
Informe a quantidade de segundos:
```

```
3702
Hora:
01:01:42
```

6. Implemente uma função que receba dois números inteiros como parâmetros de entrada e outros dois números inteiros como parâmetros de saída. A função a ser implementada deve armazenar o MDC (Máximo Divisor Comum) no primeiro parâmetro de saída e o MMC (Mínimo Múltiplo Comum) no segundo parâmetro de saída, ambos calculados entre os dois parâmetros de entrada. A função `main` deve ler dois números inteiros do usuário e exibir o MDC e MMC entre os dois números de acordo com o resultado da função implementada. Exemplos:

```
-- Exemplo 1:
Insira dois numeros:
6 8
O MDC entre 6 e 8 e 2
O MMC entre 6 e 8 e 24
-- Exemplo 2:
Insira dois numeros:
16 36
O MDC entre 16 e 36 e 4
O MMC entre 16 e 36 e 144
```

7. Implemente uma função que receba como parâmetros de entrada e saída três números inteiros. A função a ser implementada deve ordenar os parâmetros em ordem **decrescente**, ou seja, no primeiro número deve estar armazenado o maior, no segundo número o do meio e no terceiro, o menor. A função `main` deve ler três números inteiros do usuário e exibí-los na tela em ordem **crescente**. Exemplos:

```
-- Exemplo 1:
Insira tres numeros:
3 1 2
Numeros em ordem crescente:
1 2 3
-- Exemplo 2:
Insira tres numeros:
-1 9 4
Numeros em ordem crescente:
-1 4 9
```

8. Implemente uma função que receba como parâmetros de entrada dois números inteiros. A função a ser implementada deve retornar o valor do produto entre os dois números. **Não utilize o operador** `*`. A função `main` deve ler dois números inteiros do usuário e exibir na tela o produto entre os dois números de acordo com o resultado da função implementada.
9. O arco tangente de um número `x` pode ser aproximado pela seguinte série infinita:

$$arctan(x) = \sum_{i=0}^{\infty} \frac{(-1)^i x^{2i+1}}{2i+1} = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \frac{x^9}{9} - ...$$

Implemente uma função que receba como um número real como parâmetro de entrada e retorna o seu arco tangente. Considere encerrar o somatório quando algum termo da série atingir valor em módulo inferior a 0.00001. A função `main` deve ler um número real do

usuário e exibir o valor do arco tangente deste número de acordo com o resultado da função implementada. Além disto, a função `main` também deve checar se ele está no intervalo entre 0 e 1 e solicitar ao usuário que redigite o número se não for o caso. Exemplos:

```
-- Exemplo 1:
Insira um valor entre 0 e 1:
0.6
O arco tangente de 0.6 e 0.540422
-- Exemplo 2:
Insira um valor entre 0 e 1:
0.2
O arco tangente de 0.2 e 0.197396
```

7. Recursão

Matematicamente, é possível encontrar algumas definições de funções que fazem referência a si próprio. Para atender este tipo de definição, algumas linguagens implementam mecanismos de recursão de funções, que funcionam como uma alternativa aos comandos de repetição. Embora não sejam comuns, em alguns casos, elas são as únicas opções para descrever problemas recursivos.

7.1 Introdução

O conceito de recursividade em linguagens de programação, envolve o uso de funções que fazem chamadas a si próprio, direta ou indiretamente, dentro do seu bloco de comandos. O seu uso é voltado para problemas que podem ser desmembrados em instâncias menores do mesmo problema, sendo tratado então como uma estrutura recursiva.

Um caso clássico de função recursiva é a função fatorial, cuja definição matemática pode ser feita de duas formas:

1. A primeira delas, de maneira iterativa, considera o fatorial de um número n como o produto dos n primeiros termos:

$$n! = \begin{cases} 0, \text{ para } n = 0 \text{ ou } n = 1; \\ 1 \times 2 \times 3 \times ... \times (n-2) \times (n-1) \times n = \prod_{i=1}^{n} i, \text{ caso contrário.} \end{cases}$$

2. A segunda delas, dada de maneira recursiva, considera o fatorial de um número n como o produto de n com o fatorial de $n-1$:

$$n! = \begin{cases} 0, \text{ para } n = 0 \text{ ou } n = 1; \\ n * (n-1)!, \text{ caso contrário.} \end{cases}$$

É possível perceber que as duas formas são equivalentes, pois o valor de $(n-1)!$ é o produto dos $n-1$ primeiros termos, e o produto $n*(n-1)!$ corresponderia o produto dos n primeiros termos, que é a primeira definição de $n!$. Além disto, é importante notar que a definição do fatorial

de n é dada a partir do fatorial de $n-1$, sendo isto que caracteriza a recursividade para este tipo de função.

A implementação do fatorial pela primeira definição é dada como:

```
int fat(int n){
        int prod = 1;
    for(int i = 1; i <= n; i++){
        prod *= i;
    }
    return prod;
}
```

Já pela segunda definição, a implementação em C++ é dada como:

```
int fat(int n){
    if(n == 0  n == 1){
        return 1;
    }
    else{
        return n*fat(n-1);
    }
}
```

Perceba que, na chamada a função para um valor `n`, há uma chamada a própria função fatorial, mas para `n-1`. Isto não acontece por acaso: a ideia é realizar chamadas sucessivas a função fatorial enquanto o valor de `n` for diferente de 0 ou de 1. Em cada chamada, o valor do parâmetro é reduzido para que, em alguma chamada, alcance ou 0 ou 1. Estes últimos tem valor definido, e a partir dele, poderá se calcular o restante das chamadas, que dependerão das chamadas para parâmetros anteriores.

Para ilustrar este comportamento, uma simulação deste cálculo será mostrada a seguir, para `n = 6`.

1a. chamada: `fat(6)`
2a. chamada: $(6 * \texttt{fat(5)})$
3a. chamada: $(6 * (5 * \texttt{fat(4)}))$
4a. chamada: $(6 * (5 * (4 * \texttt{fat(3)})))$
5a. chamada: $(6 * (5 * (4 * (3 * \texttt{fat(2)}))))$
6a. chamada: $(6 * (5 * (4 * (3 * (2 * \texttt{fat(1)})))))$

A partir deste momento, já é possível calcular o valor da expressão formada na 6a chamada, pois o valor da chamada de `fat(1)` é conhecido: vale 1.

1o. retorno: $(6 * (5 * (4 * (3 * (2 * 1))))) \rightarrow$ `fat(1)` se tornou 1;
2o. retorno: $(6 * (5 * (4 * (3 * 2)))) \rightarrow$ `fat(2)` se tornou $2 * 1 = 2$;
3o. retorno: $(6 * (5 * (4 * 6))) \rightarrow$ `fat(3)` se tornou $3 * 2 = 6$;
4o. retorno: $(6 * (5 * 24)) \rightarrow$ `fat(4)` se tornou $4 * 6 = 24$;
5o. retorno: $(6 * 120) \rightarrow$ `fat(5)` se tornou $5 * 24 = 120$;
6o. retorno: $720 \rightarrow$ `fat(6)` se tornou $6 * 120 = 720$.

Portanto, o resultado de `fat(6)` será 720.

Toda função recursiva constrói uma sequência de operações a partir das chamadas a própria função, de maneira dinâmica. Uma função recursiva possui duas partes bem definidas:

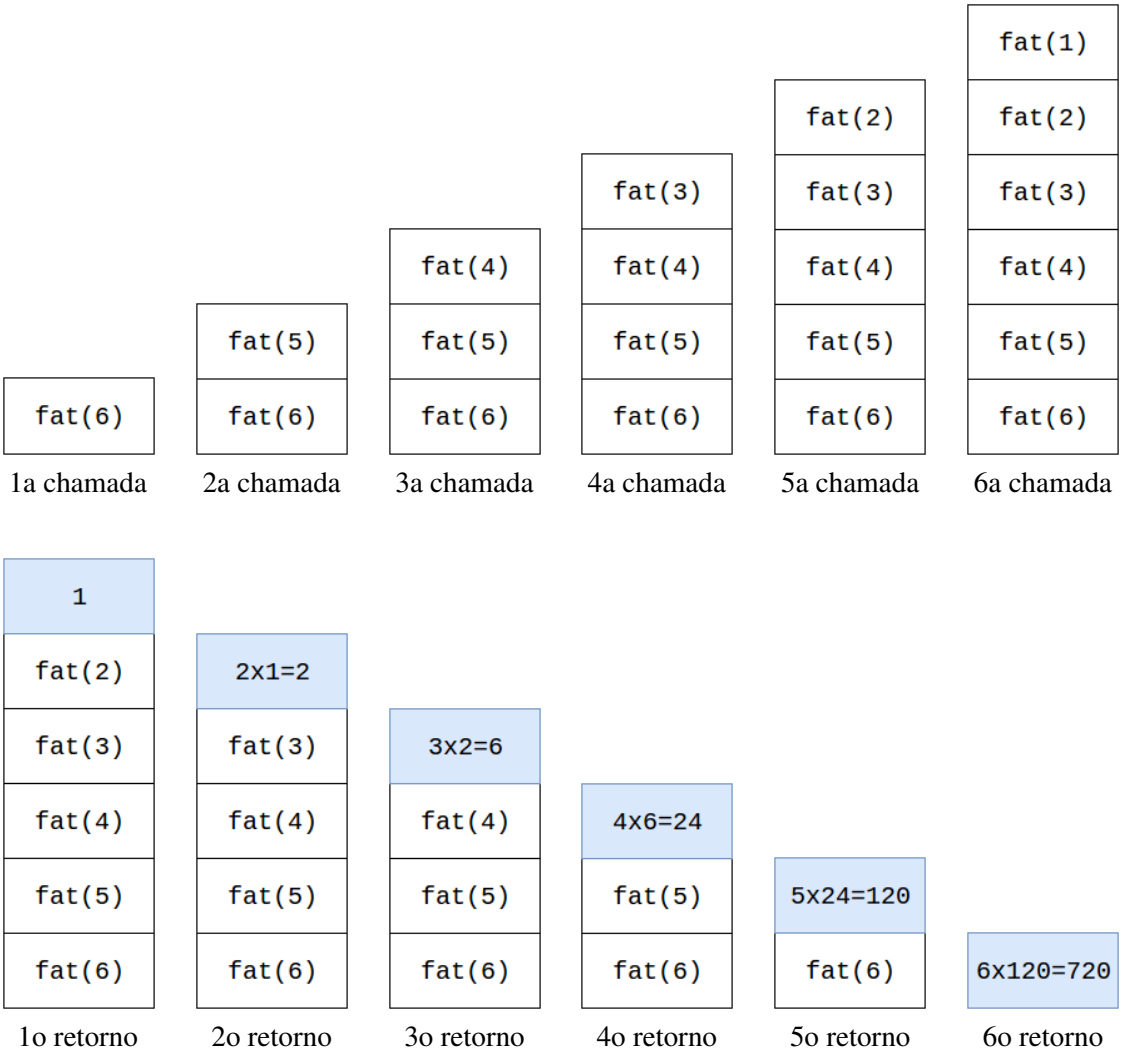
- Um (ou mais) **passo base**: Chamada cujo resultado é conhecido sem chamadas à própria função. No exemplo, `fat(0)` é um passo base, pois é igual a 1;

- Um (ou mais) **passo recursivo**: Chamada que envolve novas chamadas à própria função, variando-se os parâmetros **em direção a algum valor do passo base**. É neste passo em que se tenta resolver um sub-problema do problema inicial. No exemplo, qualquer valor de `n` maior que 1 gera um passo recursivo, pois `fat(n)` é igual a `n*fat(n-1)`

Definir funções de forma recursiva se resume a encontrar os dois passos. Quando definidos, a tradução para linguagem de programação é trivial: basta uma sequência adequada de comandos `if-else`. Veja que não é necessário o emprego de estruturas de repetição para lidar os passos base e recursivo, pois a linguagem de programação utiliza um mecanismo próprio para gerenciar chamadas de funções recursivas.

1. Internamente, o computador gerencia uma estrutura em forma de pilha, com todas as chamadas a funções de um programa;
2. Ao ser chamada, diz-se que uma função foi empilhada;
3. Em funções recursivas, primeiramente ocorre o empilhamento de várias chamadas a uma mesma função;
4. Tão logo um passo base é alcançado, a chamada correspondente é desempilhada;
5. Isto provoca o desempilhamento das chamadas subsequentes.

Como exemplo, acompanhe a sequência de operações para o cálculo da chamada `fat(6)`.



As funções recursivas estabelecem uma forma diferente de iteração. Elas, então, servem de alternativa as estruturas de repetição. Como vantagem, o emprego de recursão facilita a

implementação, pois não exige estruturas mais elaboradas do que um comando `if-else`. No entanto, apresentam como principal desvantagem uma execução mais lenta em relação à versão iterativa na maioria das linguagens de programação. Por causa disto, formas recursivas não costumam ser empregadas em programas de alto desempenho.

Exemplo

Crie uma função recursiva que calcula o valor da seguinte equação:

$$s(n) = \begin{cases} 1, & \text{se } n = 1 \\ n + s(n-1), & \text{se } n > 1 \end{cases}$$

Solução:

Neste caso, os passos da recursão já estão bem definidos:

- O passo base ocorre quando $n = 1$, onde $s(n) = s(1) = 1$;
- O passo recursivo ocorre quando o valor de n for maior do que 1, sendo dado por $s(n) = n + s(n-1)$.

A função recursiva que define a equação dada em C++ é dado por:

```
1 int soma_n(int n){
2     if(n == 1){
3         return 1;
4     }
5     else{
6         return n+soma_n(n-1);
7     }
8 }
```

Exemplo

Cria uma função recursiva em C++ que calcula a seguinte fórmula matemática a partir do valor de n:

$$f(n) = \sum_{i=1}^{n} 3i^2 = 3 + 12 + 27 + 48 + ... + 3(n-1)^2 + 3n^2$$

Solução:

Para trabalhar com expressões matemáticas baseadas em somatório e/ou produtório, é preciso reescrever a definição para que ela passe a ser recursiva.

- O passo base é dado para $n = 1$, quando o somatório $f(n)$ tem apenas um termo, definido pelo o limite inferior $i = 1$. Neste caso, o valor é igual a $3 \times (1)^2 = 3$;
- O passo recursivo, dado para um n qualquer, pode ser definido pela expansão do

somatório. Neste caso:

$$\begin{aligned} f(n) &= \sum_{i=1}^{n} 3i^2 \\ &= 3+12+27+48+\ldots+3(n-1)^2+3n^2 \\ &= (3+12+27+48+\ldots+3(n-1)^2)+3n^2 \\ &= \sum_{i=1}^{n-1} 3i^2+3n^2 \\ &= f(n-1)+3n^2 \end{aligned}$$

Logo, $f(n) = f(n-1)+3n^2$.
Isto permite reescrever o somatório da seguinte maneira:

$$f(n) = \begin{cases} 3, & \text{se } n = 1 \\ 3*n*n+f(n-1), & \text{se } n > 1 \end{cases}$$

Cuja função em C++ é dada por:

```
int formula(int n){
    if(n == 1){
        return 3;
    }
    else{
        return 3*n*n + formula(n-1);
    }
}
```

Exemplo

A Sequência de Fibonacci:

$$1,1,2,3,5,8,13,21,34,55,\cdots$$

Comumente é obtida a partir da seguinte definição matemática:

$$f(n) = \begin{cases} 1, & \text{se } n = 0 \\ 1, & \text{se } n = 1 \\ f(n-1)+f(n-2), & \text{se } n > 1 \end{cases}$$

Escreva uma função que calcula o `n`-ésimo termo desta sequência.

Solução:

Este é um típico caso de sequência recursiva, onde o valor do termo atual desta sequência é definido em termos dos valores anteriores. Neste caso, as etapas envolvidas na recursão já estão estabelecidas:

- Há dois passos base, para $n = 0$ e para $n = 1$. Em ambos, $f(n) = 1$;
- O termo corrente n na sequência de Fibonacci é a soma dos dois termos imediatamente anteriores $n-1$ e $n-2$ da sequência. Isto implica em duas chamadas a própria função no passo recursivo, de forma que $f(n) = f(n-1) + f(n-2)$

A função recursiva correspondente em C++ é dada por:

```
int fib(int n){
    if(n <= 1){
        return 1;
    }
    else{
        return fib(n-1)+fib(n-2);
    }
}
```

Exemplo

Implemente uma função recursiva que calcula o quociente entre dois números inteiros. **Não utilize o operador de divisão** `/`. Implemente também a função `main`, de modo que o usuário do seu programa possa informar dois números inteiros e visualizar o quociente computado na tela.

Solução:

Neste problema, a ideia para deduzir o passo base e o passo recursivo é proceder com sucessivas subtrações entre o dividendo e divisor.

- O passo baso, neste caso, é dado quando o valor do dividendo é maior que o do divisor, pois não é possível realizar mais subtrações. Aqui, o quociente (valor da divisão) será igual a zero;
- No passo recursivo, o seu programa deve incrementar em uma unidade o quociente a cada subtração realizada (chamada da função). Este processo é realizando até encontrar o passo base.

Como exemplo, para o cálculo de 11 dividido por 3:

- 3 deve ser decrementado de 11, restando 8; outra chamada à função deve ser realizada e somada com uma unidade.
- 3 deve ser decrementado de 8, restando 5; outra chamada à função deve ser realizada e somada com uma unidade.
- 3 deve ser decrementado de 5, restando 2; outra chamada à função deve ser realizada e somada com uma unidade.
- 2 é menor do que 3; a função deve retornar 0.

Em C++, o programa pode ser definido como:

```
int div(int a, int b){
    if(a < b){
        return 0;
    }
    else{
        return 1 + div(a-b,b);
    }
}
```

Exemplo

Implemente uma função recursiva que imprima os dígitos de um número inteiro separadamente. Implemente também a função `main`, de modo que o usuário do seu programa possa informar dois números inteiros e visualizar o quociente computado na tela. Como exemplo, ao executar a função ao valor `7331`, ele deve retornar:

```
1
3
3
7
```

Solução:

Para este problema, a extração do último dígito de um número (o dígito menos significativo) do resto no número pode ser feita por meio da divisão inteira por 10, já que a representação de um número inteiro é decimal:

- $7331\%10 = 1$;
- $7331/10 = 733$;

A cada nova chamada da função, o seu programa deve imprimir o resto da divisão inteira da entrada por 10, e chamando a função com o valor do quociente da divisão inteira da entrada por 10. Este último será o novo valor de entrada na próxima chamada. Este será o mecanismo do passo recursivo. Este procedimento deve ser realizado até encontrar o passo base, que é quando a entrada não tem dígito a ser extraído. Isto ocorre, por exemplo, quando a entrada é nula; Por exemplo, para extrair o valor de 7331:

1. Imprime-se $7331\%10 = 1$ e realiza-se outra chamada à função deve ser realizada passando com entrada o valor de $7331/10 = 733$;
2. Imprime-se $733\%10 = 3$ e realiza-se outra chamada à função deve ser realizada passando com entrada o valor de $733/10 = 73$;
3. Imprime-se $73\%10 = 3$ e realiza-se outra chamada à função deve ser realizada passando com entrada o valor de $73/10 = 7$;
4. Imprime-se $7\%10 = 7$ e realiza-se outra chamada à função deve ser realizada passando com entrada o valor de $7/10 = 0$;
5. Como o valor de entrada é 0, nada é feito. Encerra-se aqui a sequência de chamadas de funções.

Em C++, a função recursiva correspondente é dada por:

```
void extracao(int n){
    if(n > 0){
        cout << n%10 << endl;
        extracao(n/10);
    }
}
```

Exemplo

Implemente uma função recursiva em C++ que implemente a seguinte expressão, dados dois valores naturais (inteiro positivos) m e n:

$$A(m,n) = \begin{cases} n+1, & \text{if } m = 0; \\ A(m-1,1), & \text{if } m > 0 \text{ e } n = 0; \\ A(m-1,A(m,n-1)), & \text{if } m > 0 \text{ e } n > 0. \end{cases}$$

A função $A(m,n)$ é uma representação possível para a **função de Ackermann**.

Solução:

Neste caso, os passos da recursão já estão bem definidos. Porém, aqui é possível verificar a recursão desta função envolve dois parâmetros. Além disto, ela apresenta dois passos recursivos possíveis:

- O passo base ocorre quando $m = 0$, onde $A(m,n) = A(0,n) = n+1$;
- Um passo recursivo ocorre quando o valor de $m > 0$ e $n = 0$. Neste caso, o valor da função será dado por $A(m,n) = A(m,0) = A(m-1,1)$;
- O outro passo recursivo ocorre quando o valor de $m > 0$ e $n > 0$. Neste caso, o valor da função será dado por $A(m,n) = A(m-1,A(m,n-1))$.

A função recursiva em C++ correspondente é dada a seguir:

```
int A(int m, int n){
    if(m == 0) {
        return n+1;
    }
    else if(n == 0) {
        return A(m-1, 1);
    }
    else {
        return A(m-1, A(m, n-1));
    }
}
```

Observação: A função de Ackermann é uma alternativa para representar números gigantes. Veja o comportamento de $A(m,n)$ na figura a seguir.

Valores de $A(m, n)$

$m \backslash n$	0	1	2	3	4	n
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$n+2 = 2+(n+3)-3$
2	3	5	7	9	11	$2n+3 = 2\cdot(n+3)-3$
3	5	13	29	61	125	$2^{(n+3)}-3$
4	13 $=2^{2^2}-3$	65533 $=2^{2^{2^2}}-3$	$2^{65536}-3$ $=2^{2^{2^{2^2}}}-3$	$2^{2^{65536}}-3$ $=2^{2^{2^{2^{2^2}}}}-3$	$2^{2^{2^{65536}}}-3$ $=2^{2^{2^{2^{2^{2^2}}}}}-3$	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3}-3$

7.2 Recursão indireta

Pelos exemplos anteriores, verificou-se que uma função recursiva apresenta uma chamada a si própria dentro do seu bloco de instruções. No entanto, é possível definir funções que fazem chamadas a funções recursivas propriamente dita. Neste caso, fala-se em **recursão indireta**, e o seu principal emprego está na simplificação das chamadas.

Como exemplo, suponha uma função que imprima um quadrado numérico com n linhas, cada uma composta com elementos de 1 até n, de maneira recursiva. Neste caso considere como argumento apenas o valor de n. Para ilustrar, considere o resultado para $n = 5$:

```
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
```

Para resolver este problema, pode-se pensar inicialmente como seria a solução iterativa. Nela, seria necessário um conjunto de laços aninhados: um para percorrer as linhas, e outro para imprimir cada um dos elementos da linha. Como é passado o número n como argumento, uma possível construção é dada a seguir:

```
void quadrado(int n) {
        for(int i = 1; i <= 5; i++) {
                for(int j = 1; j <= 5; j++) {
                        cout << j << " ";
                }
                cout << endl;
        }
}
```

A solução recursiva terá comportamento simular ao caso iterativo. Para tanto será necessário estabelecer, além do valor de n, um argumento para iterar sobre a linha i, e outro para iterar sobre a coluna j do quadrado. Isto implica em uma recursão com dois termos. Além disto, o processo de ser feito de maneira que, quando uma linha estiver preenchida, uma quebra de linha é acionada, e a iteração sobre a próxima linha é iniciada. Tem-se então, as seguintes considerações para a chamada recursiva:

- Se o valor da coluna j for menor que n, significa que irá seguir na linha. Para tanto, imprime-se o valor de j na mesma linha, e anda para o próximo elemento da linha;
- Se o valor da coluna j for igual a n, significa que a linha foi esgotada. Então, imprime-se o valor de j e passa-se para a próxima linha, incrementando o valor de i em uma unidade e fazendo o valor j ser igual a 1, o primeiro valor da linha;
- A execução termina quando o valor da linha i for maior do que o limite máximo n. Este será o passo base.

A função que realiza este processo recursivo é dada a seguir:

```
void quadrado(int i, int j, int n) {
        if(i <= n) {
         cout << j;
         if(j < n) {
                cout << " ";
                quadrado(i, j+1, n);
         }
         else {
                cout << endl;
                quadrado(i+1, 1, n);
         }
    }
}
```

Perceba que, para este problema, o caso base não é explicitado. Isto não acontece em todos os casos de recursão indireta.

Além disto, e o mais importante, veja quanto parâmetros foram necessários para construir a função recursiva. Para o usuário, o mais natural - e o solicitado pelo problema - é que houvesse apenas um argumento:o valor de n. Os outros dois parâmetros servem para controle de fluxo da impressão, mas estão relacionados a um momento da recursão, e não ao caso completo. A impressão completa do quadrado ocorre apenas quando i e j possuem valor 1; para outros valores, parte do quadrado é impressa. Uma forma de resolver esta situação é definir outra função, com apenas um parâmetro - o valor de n, que chama a função recursiva com os valores adequados de i, j e n. Isto permite abstrair do usuário as informações de controle do fluxo de impressão, já que a chamada da função empregada pelo usuário irá necessitar apenas do valor de n.

O programa completo em C++ é dado a seguir:

```
#include <iostream>

using namespace std;

void quadrado(int n);

int main() {
        int n = 0;
        cout << "Digite o numero de linhas do quadrado" <<
           endl;
        cin >> n;
        quadrado(n);
        return 0;
}

void quadrado(int i, int j , int n){
    if(i <= n) {
         cout << j;
         if(j < n) {
                cout << " ";
                quadrado(i, j+1, n);
         }
         else {
                cout << endl;
                quadrado(i+1, 1, n);
         }
    }
}

void quadrado(int n) {
        quadrado(1, 1, n);
}
```

Nesta implementação, apenas a assinatura da função "quadrado" com um parâmetro é deixada acima. A função "quadrado" com três parâmetros não é mostrada para indicar que é interna ao programa. Também, foram utilizados os mesmos nomes para a função com um e para a com três parâmetros: isto é uma forma de garantir polimorfismo, ou nomear chamadas semelhantes com o mesmo nome. Porém, isto não é obrigatório: a função recursiva pode ter nome diferente da função que a chama indiretamente.

7.3 Exercícios

1. Sendo a definição matemática para a soma dos números de 0 a n dada por

$$f(n) = \begin{cases} 0, & \text{se } n = 0 \\ n + f(n-1), & \text{se } n > 0, \end{cases}$$

implemente esta função de forma recursiva. Implemente também a função `main`, de modo que o usuário do seu programa possa informar um número inteiro n e visualizar a soma de todos os números de 0 a n na tela.

2. Sendo a definição matemática para o fatorial de um número n dada por

$$f(n) = \begin{cases} 1, & \text{se } n = 0, \text{ ou } n = 1 \\ n * f(n-1), & \text{se } n > 0 \end{cases}$$

   implemente esta função de forma recursiva. Implemente também a função `main`, de modo que o usuário do seu programa possa informar um número inteiro n e visualizar o fatorial de N na tela.
3. Uma função matemática $f(n)$ é dada por

$$f(n) = \sum_{i=1}^{n} \frac{2i}{i+1}.$$

   Implemente uma função recursiva que calcula o valor do somatório para um dado valor de n. Utilizando a função implementada, faça um programa que receba como entrada um número inteiro n e imprima na tela o resultado de $f(n)$.
4. Uma função matemática $f(n)$ é dada por

$$f(n) = \prod_{i=1}^{n} 2i.$$

   Implemente uma função recursiva que calcula o valor do produtório para um dado valor de n. Utilizando a função implementada, faça um programa que receba como entrada um número inteiro n e imprima na tela o resultado de $f(n)$.
5. Uma sequência de números $a(0)$, $a(1)$, $a(2)$,...,$a(n)$,... é definida matematicamente por

$$a(n) = \begin{cases} a(0) = 1 \\ a(1) = 2 \\ a(n) = 2a(n-2) - a(n-1). \end{cases}$$

   Implemente uma função recursiva que computa o enésimo termo (termo $a(n)$) da sequência. Utilizando a função implementada, faça um programa que leia um número inteiro n e imprima na tela todos os $a(0)$, $a(1)$, $a(2)$,...,$a(n)$ termos da sequência.
6. Implemente uma função recursiva que calcula o produto entre dois números inteiros. **Não utilize o operador** $*$. Implemente também a função `main`, de modo que o usuário do seu programa possa informar dois números inteiros e visualizar o produto computado na tela. **Dica:** para achar o passo base e passo recursivo, leve em consideração que, por exemplo, $2*3 = 2+2*2 = 2+2+2*1$... e que $2*0 = 0$.
7. Implemente uma função recursiva que calcula a potenciação entre dois números inteiros. Implemente também a função `main`, de modo que o usuário do seu programa possa informar dois números inteiros e visualizar a potência computada na tela. **Dica:** para achar o passo base e passo recursivo, leve em consideração que, por exemplo, $2^3 = 2*2^2 = 2*2*2^1$... e que $2^0 = 1$.
8. Implemente uma função recursiva que calcula a soma dos dígitos de um número inteiro. Por exemplo, para um número `n = 2394`, a função deve computar `2+3+9+4=18`. Implemente também a função `main`, de modo que o usuário do seu programa possa informar um número inteiro e visualizar na tela a soma dos seus dígitos.
9. Implemente uma função recursiva que computa a quantidade de divisores de um número. A função `main` deve utilizar a função implementada para exibir uma mensagem na tela informando se um dado número lido do teclado é primo ou não.

III Parte 3: Blocos de Memória

8 Vetores 121
8.1 Introdução
8.2 Funções com Vetores
8.3 Ordenação
8.4 Exercícios

9 Matrizes 137
9.1 Introdução
9.2 Definição e Indexação de Matrizes
9.3 Inicialização de Matrizes
9.4 Funções com Matrizes
9.5 Exercícios

10 Strings 147
10.1 Introdução
10.2 Inicialização de uma *string*
10.3 Impressão e Leitura de *strings*
10.4 Uso de *strings* em funções
10.5 Tratamento de Texto com *strings*
10.6 Exercícios

8. Vetores

Par lidar com grandes quantidades de dados, as linguagens de programação disponibilizam estruturas com vários espaços de memória disponíveis. Elas precisam apresentar um mecanismo de acesso a cada um destes espaços, de maneira a permitir a manipulação individual destes dados. Geralmente, esta associação é feita de forma a se assemelhar com o modelo matemático de vetores. Isto acontece porque há uma teoria matemática bem fundamentada para estas estruturas. Inclusive, o emprego do nome também é adotado como referência a estes, embora permita o uso de dados simbólicos, lógicos ou mesmo definidos pelo usuário.

8.1 Introdução

Vetores são estruturas disponibilizado pela linguagem C++ que agrupam dados de um mesmo tipo em uma mesmo bloco de memória. Eles podem ser vistos, a grosso modo, como "uma estrutura composta de várias variáveis". Cada um destes "variáveis" ou dados, pode ser acessado individualmente por meio de um mecanismo de indexação do vetor, mas todo o conjunto de dados é tratado como uma única entidade. Esta característica a torna adequada para a manipulação de uma quantidade significativa de informações, cujo o custo ao se manipular variáveis individualmente é custoso (um vetor de 100 valores inteiros, por exemplo, substitui a necessidade declarar e manipular 100 variáveis inteiras) ou inviável (quando o usuário irá informar a quantidade de dados durante a execução do programa). Além disto, o emprego de vetores disponibiliza novos tipos de dados para a linguagem: vetor de `char`, vetor de `int`, vetor de `float`, vetor de `bool`, etc; de maneira que os vetores são tratados como **tipos compostos**.

Para efeito de comparação entre uma variável e um vetor, considere a figura 8.1, onde há a representação de uma variável e um vetor em memória.

A partir dela, é possível afirmar que enquanto uma variável identifica uma posição na memória para guardar um valor, o vetor identifica várias posições na memória para guardar vários valores. Assim, todas as posições são acessadas com um mesmo nome. No vetor, a quantidade de posições é indicada pelo **tamanho do vetor**, sendo especificado pelo programador durante a declaração deste.

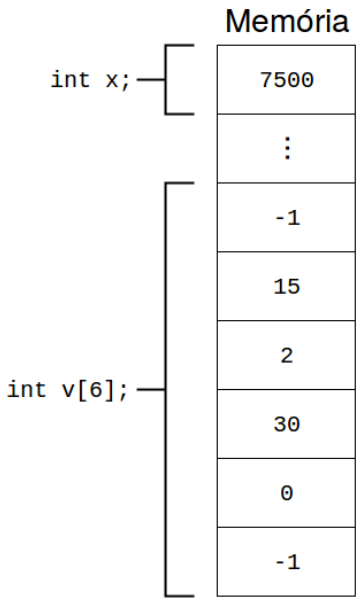


Figura 8.1: Representação simplificada de uma variável e um vetor em memória.

A sintaxe para a declaração de um vetor é dada por:

```
<tipo_do_vetor> <nome_do_vetor>[<tamanho_do_vetor>];
```

Onde:

- `<tipo_do_vetor>` são os tipos de variáveis conhecidos: `int`, `char`, `float`, `double`, `bool`;
- `<nome_do_vetor>` é o identificador associado ao bloco de memória, e segue as mesmas regras que são utilizadas para variáveis;
- O `<tamanho_do_vetor>` é uma expressão cujo tipo deve ser `int`.

Exemplo

Escreva a declaração para os seguintes vetores:

1. Do tipo inteiro, com 30 posições, chamado `idades`
2. Do tipo caractere, com 5 posições, chamado `vogais`
3. Para armazenar `n` valores do tipo `float`

Solução:

1. `int idades[30];`
2. `char vogais[5];`
3. `float v[n];`

8.1.1 Indexação de Vetores

Apesar de ser uma única estrutura, um vetor `v` não pode ser usado diretamente em expressões, pois o identificado está associado ao bloco inteiro de memória especificado para o vetor. Neste caso, cada elemento do vetor é que deve ser utilizado. Em C++, ao contrário de outras linguagens de programação, **um vetor com `n` posições tem posições válidas que vão de `0` até `n-1`**. Em outras palavras, a primeira posição de memória corresponde ao elemento zero do vetor, a segunda

posição corresponde ao elemento um do vetor, e assim sucessivamente, até o última posição que corresponde ao elemento $n-1$.

Em C++, a indexação de uma posição de um vetor é feita utilizando os colchetes como operadores. Isto é, em um vetor `v`, as posições de memória existentes para um vetor são `v[0]`, `v[1]`, `v[2]`, ..., `v[n-1]`. Como exemplo, considere a figura 8.2. É importante mencionar que **C++ não verifica se o acesso a uma posição do vetor é válido ou não: isto deve ser feito manualmente pelo programador, caso seja necessário**. Caso seja utilizado um índice inválido para acessar a posição de um vetor, o programa terá um comportamento inesperado e pode ser abortado de maneira abrupta.

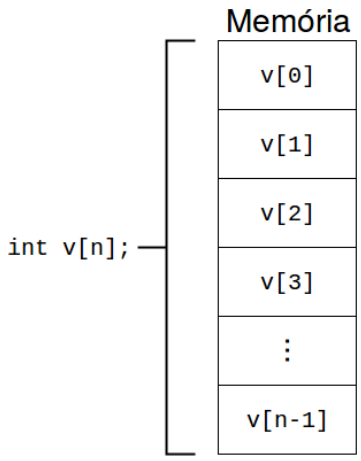
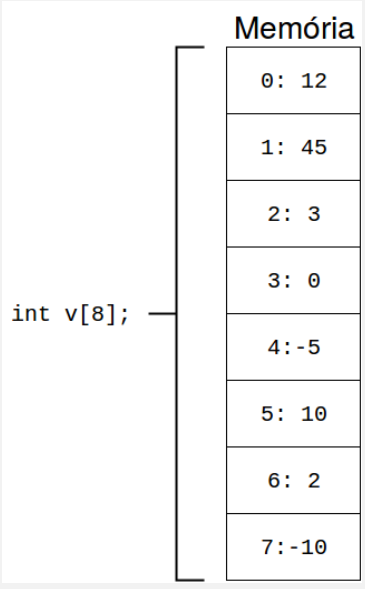


Figura 8.2: Posições possíveis de um vetor de `n` elementos.

Exemplo

Considere a seguinte disposição para um vetor:



Qual o valor de cada expressão a seguir, sendo `x = 2`?

1. `v[6]`
2. `v[0]+v[7]`

3. `v[x]`
4. `v[2*x + 3]`
5. `v[v[x]]`
6. `v[10 - v[-v[4]]]`
7. `v[-1]`
8. `v[8]`
9. `v[1000]`

Solução:

1. `v[6] = 2;`
2. `v[0] + v[7] = 12 + (-10) = 2;`
3. `v[x] = v[2] = 3;`
4. `v[2*x + 3] = v[2*2 + 3] = v[7] = -10;`
5. `v[v[x]] = v[v[2]] = v[3] = 0;`
6. `v[10 - v[-v[4]]] = v[10 - v[-(-5)]] = v[10 - v[5]] = v[10 - 10] = v[0] = 12;`
7. `v[-1]:` posição inválida.
8. `v[8]:` posição inválida.
9. `v[1000]:` posição inválida.

Exemplo

Implemente um programa em C++ que leia um número `n` do usuário e em seguida, armazene `n` notas em um vetor. O seu programa deve imprimir a maior das notas armazenadas.

Solução:

```
int main(){
    int n, i;
    cout << "Insira a quantidade de notas:\n";
    cin >> n;
    float notas[n], maior;
    for(i = 0; i < n; i++){
        cin >> notas[i];
        if(i == 0){
            maior = notas[i];
        }
        else{
            if(notas[i] > maior){
                maior = notas[i];
            }
        }
    }
    cout << "Maior nota: " << maior << endl;
    return 0;
}
```

Exemplo

Implemente um programa em C++ que leia um número `n` do usuário e em seguida, armazene `n` notas em um vetor. O seu programa deve imprimir a posição da maior nota armazenada.

Solução:

```
int main(){
    int n, i, maiorpos;
    cout << "Insira a quantidade de notas:\n";
    cin >> n;
    float notas[n], maior;
    for(i = 0; i < n; i++){
        cin >> notas[i];
        if(i == 0){
            maiorpos = i;
        }
        else{
            if(notas[i] > notas[maiorpos]){
                maiorpos = i;
            }
        }
    }
    cout << "Pos. da maior nota: " << maiorpos << endl;
    return 0;
}
```

8.1.2 Inicialização de Vetores

Assim como variáveis, é uma boa prática atribuir valores iniciais aos vetores. A simples declaração de um vetor, `int v[5]`, inicializa cada valor do vetor com lixo de memória. No entanto, ao contrário dos primeiros, vetores lidam com várias posições de memória ao mesmo tempo. Isto significa que é preciso atribuir valores para todos estes espaços. Algumas possibilidade de inicialização de vetores são mostradas a seguir. Em todas elas, a indicação dos valores do vetor é feita através de chaves ({}), sendo obrigatório o seu emprego nestes casos.

- Especificar o tamanho e listar cada elemento do vetor;

```
int v[5] = {10, 20, 30, 40, 50};
```

- Apenas listar cada elemento do vetor. Neste caso, não é necessário informar o tamanho do vetor entre colchetes, pois este será determinado pela quantidade de elementos na inicialização;

```
int v[] = {10, 20, 30, 40, 50};
```

- Especificar o tamanho e atribuir valor 0 a todas as posições do vetor;

```
int v[5] = {};
```

```
int v[5] = {0};
```

- Especificar o tamanho e listar apenas os elementos das primeiras posições. Neste caso, as outras posições serão inicializadas com 0.

```
int v[5] = {5, 10};
```

É importante mencionar que a listagem dos elementos de um vetor é feita apenas na inicialização. Após esta, a modificação de um valor é feita de maneira individual. Em outras palavras, operações com a seguir não são suportadas em C++:

```
int v[5] = {10, 20, 30, 40, 50};
v = {50, 40, 30, 20, 10}; //Operacao indevida.
```

8.2 Funções com Vetores

É possível passar vetores como parâmetro de funções, assim como acontece com variáveis. Isto é importante porque existem algumas operações com vetores que são feitas de maneira repetitiva diretamente na `main`, por exemplo. Em C++, ao contrário de algumas linguagens (Java, por exemplo), não há como verificar diretamente pelo vetor a quantidade de elementos deste (é possível utilizando o operador `sizeof`; para tanto, é necessário saber o tipo associado ao vetor). Neste caso, uma estrutura auxiliar é utilizada para armazenar este informação: geralmente, uma variável inteira. Como consequência disto, no uso de vetores como parâmetros de funções, além da estrutura em si, comumente um parâmetro com o tamanho do vetor também é necessário.

A definição de uma função com apenas um vetor pode ser feita da seguinte forma:

```
tipo_func nome_func(tipo_vetor nome_vetor[], int tam_vetor) {
        corpo_da_funcao
}
```

Onde:

- `tipo_vetor` é o tipo do vetor passado como parâmetro, como `int`, `char`, `float`, `bool`;
- `nome_vetor` é o nome do vetor passado como parâmetro. Observe que o `[]` está em branco: isto indica que um vetor de qualquer tamanho pode ser passado como parâmetro. Caso seja especificado um valor entre colchetes, apenas vetores com aquele tamanho podem ser empregados. Como exemplo, uma declaração do tipo:

  ```
  int func(int v[5], int tam) {
          ...
  }
  ```

  Apenas vetores de tamanho 5 serão permitidos. Caso sejam empregados vetores com outros valores de tamanho, um erro de compilação será emitido;
- `tam_vetor` é o tamanho do vetor passado como parâmetro.

Caso haja mais de um vetor, é necessário informar a estrutura e o tamanho do vetor para cada um destes. Um exemplo declaração, para quatro vetores, é dado a seguir:

```
int func(int v1[], int n1,
         int v2[], int n2,
         int v3[], int n3,
         int v4[10], int n4) {
        ...
}
```

Perceba que o último vetor tem tamanho pré-fixado. Isto significa que apenas vetores de tamanho 10 podem ser passados como parâmetro. Uma questão que pode ser levantada aqui é sobre a necessidade de uma variável para o tamanho do vetor quando este já possuir um quantidade fixa de posições. Isto pode ocorrer em problemas cujo valor da quantidade de elementos sofre variação ao longo da execução do programa. Neste caso, nem todos as posições existentes terão elementos válidos. Isto torna ainda mais importante o emprego da variável auxiliar.

Exemplo

Escreva assinaturas para as seguintes funções:

1. Função que imprime um vetor de inteiros na tela
2. Função que recebe um vetor de caracteres e retorna quantos dos elementos no vetor são iguais a um caractere passado como parâmetro;
3. Função que recebe dois vetores de inteiros e retorna o maior valor dentre todos, seja ele do primeiro ou do segundo vetor.

Solução:

1. `void imprime_vetor(int v[], int tam);`
2. `int conta_ocorrencias(char v[], int tam, char c);`
3. `int computa_maior(int v1[], int tam1, int v2[], int tam2);`

Nas chamadas às funções, variáveis do tipo vetor são passadas como parâmetros utilizando apenas o seu nome. Como exemplo, considere o emprego de uma função que imprime o valor de um vetor, para visualizar um vetor formado pelos n primeiros números ímpares.

```
1  void imprime_vet(int vet[], int n);
2  int main(){
3      int n;
4      cin >> n;
5      int v[n], i;
6      for(i = 0; i < n; i++){
7          v[i] = 2*n+i;
8      }
9      imprime_vet(v, n);
10     return 0;
11 }
```

Percebe-se que não há necessidade dos colchetes no emprego de um vetor em uma chamada, ao contrário do que acontece na inicialização deste. Uma informação relevante no emprego de vetores em funções é que estes são passados **por referência** por padrão. Isto significa dizer que as alterações realizadas nos vetores dentro da função são visíveis fora da função. De outra forma, todo vetor é um parâmetro de entrada e saída (ao mesmo tempo). Por causa disto, não é permitido o uso do operador de referência em vetores como argumentos.

Como exemplo, considere uma função que preenche com valores unitário um vetor de tamanho *n*,e um programa onde é especificado o valor de *n* e, em seguida, é exibido o vetor preenchido.

```
void func(int vet[], int n){
    int i;
    for(i = 0; i < n; i++){
        vet[i] = 1;
    }
}
int main(){
        int n;
        cin >> n;
        int v[n], i;
        func(v, n);
        for(i = 0; i < n; i++){
            cout << v[i] << " ";
        }
        cout << endl;
        return 0;
}
```

Veja que, mesmo que o problema requisite a modificação do vetor de entrada, não é necessário do emprego do operador de referência & no vetor.

8.3 Ordenação

Um problema comum em vetores envolve a ordenação de dados. É um tópico bastante estudado em algoritmos, porque ordenação costuma fazer partes de soluções para problemas maiores é pelo seu elevado custo computacional. Existem diversas soluções disponíveis na literatura para este tipo de problema. Neste material, será apresentada apenas uma delas: a ordenação por bolha ou *bubble sort*, como é mais conhecida.

Primeiramente, é preciso lembrar que a ordenação pode ser feita de duas formas: crescente ou decrescente. Na ordenação crescente, a sequência é posicionada do menor elemento, na primeira posição, até o maior elemento, na última posição. Já na ordenação decrescente, a sequência é invertida: do maior elemento, na primeira posição, para o menor, na última.

O *Bubble sort* se baseia no movimento das bolhas de um líquido, que saem de áreas de maior pressão, como o fundo do copo, e seguem até áreas de menor pressão, como a superfície do líquido. A ideia é reproduzir este movimento, neste caso levando o maior elemento do vetor a última posição; depois, o segundo maior valor para a penúltima posição; o terceiro maior para a antepenúltima; e assim sucessivamente até que todos os elementos estejam ordenados.

Para realizar este movimento de levar o maior até o final, é preciso percorrer cada elemento do vetor, e comparar o elemento atualmente referenciado, aqui denominado *elemento chave*, com o posterior. Na ordenação crescente, caso o posterior seja menor que o elemento chave, realiza-se uma troca entre os elementos das duas posições para garantir que o elemento na posição de maior índice seja sempre o maior. Este processo é executado até alcançar a penúltima posição do vetor (não a última, pois ela não possui posição posterior). Na prática, estabelece-se uma posição limite para as verificações em uma determinado momento pois é possível supor que as posições com índices maiores estão ordenadas.

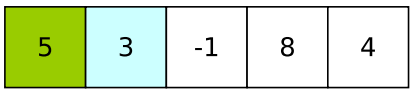
A técnica, neste caso, é interrompida quando a posição limite alcança a segunda posição do vetor, já que haverá a possibilidade de troca apenas entre a primeira e o segunda posição. Na prática, porém, é possível interromper a execução caso seja verificado que não houve troca de elementos do

vetor em algum momento da execução. Isto sinaliza que os elementos antes a posição limite já estão ordenados: caso não estejam, haverá troca em algum momento, e consequentemente, a necessidade de realizar uma nova varredura no vetor. A realização ou não de trocas pode ser sinalizado por uma *flag*, tipicamente uma variável do tipo `bool`.

Para compreender o processo de execução do algoritmo, considere a ordenação do seguinte vetor:

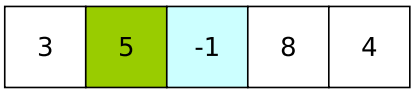
0	1	2	3	4
5	3	-1	8	4

O objetivo é ordenar vetor de números inteiros acima em ordem crescente. Na primeira iteração, é necessário varrer o vetor, deslocando o valor armazenado no elemento chave, o maior verificado, para a última posição (posição 5). Neste caso, o elemento chave começa na posição 0.



1a iteração:

- chave: `v[0] = 5`
- próximo: `v[1] = 3`
- `v[0] > v[1]` → verdadeiro. Trocar `v[0]` com `v[1]`
- Ir para o próximo elemento a ser comparado: v[1];



2a iteração:

- chave: `v[1] = 5`
- comparado: `v[2] = -1`
- `v[1] > v[2]` → verdadeiro. Trocar `v[1]` com `v[2]`
- Ir para o próximo elemento a ser comparado: v[2];

3a iteração:

- chave: `v[2] = 5`;
- comparado: `v[3] = 8`;
- `v[2] > v[3]` → falso. Não trocar;
- Ir para o próximo elemento a ser comparado: v[3];

4a iteração:

- chave: `v[3] = 8`
- comparado: `v[4] = 4`
- `v[3] > v[4]` → verdadeiro. Trocar `v[4]` com `v[3]`
- Ir para o próximo elemento a ser comparado

Ao final da primeira iteração do elemento chave, obtém-se o seguinte resultado:

FINAL DA PRIMEIRA ITERAÇÃO.

Na segunda iteração, varre-se o vetor para deslocando o valor armazenado no elemento chave para a penúltima posição (posição 4). Neste caso, o elemento chave volta para a posição 0.

1a iteração:

- chave: `v[0] = 3`;
- próximo: `v[1] = -1`;
- `v[0] > v[1]` → verdadeiro. Trocar `v[0]` com `v[1]`;
- Ir para o próximo elemento a ser comparado: v[1];

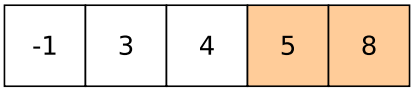
2a iteração:

- chave: `v[1] = 3`
- próximo: `v[2] = 5`
- `v[0] > v[1]` → falso. Não trocar;
- Ir para o próximo elemento a ser comparado: v[2];

3a iteração:

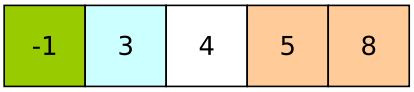
- chave: `v[2] = 5`
- próximo: `v[3] = 4`
- `v[0] > v[1]` → verdadeiro. Trocar `v[2]` com `v[3]`;
- Ir para o próximo elemento a ser comparado: v[3];

Ao final da segunda iteração do elemento chave, obtém-se o seguinte resultado:



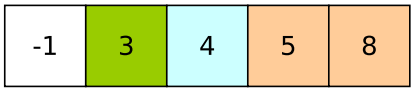
FINAL DA SEGUNDA ITERAÇÃO.

Na terceira iteração, varre-se o vetor para deslocando o valor armazenado no elemento chave para a antepenúltima posição (posição 3). Neste caso, o elemento chave volta para a posição 0.



1a iteração:

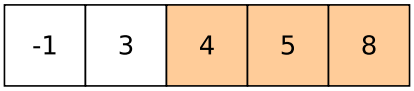
- chave: `v[0] = -1`;
- próximo: `v[1] = 3`;
- `v[0] > v[1]` → falso. Não trocar;
- Ir para o próximo elemento a ser comparado: v[1];



2a iteração:

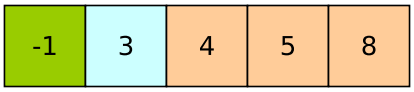
- chave: `v[1] = 3`;
- próximo: `v[2] = 4`;
- `v[0] > v[1]` → falso. Não trocar;
- Ir para o próximo elemento a ser comparado: v[3];

Ao final da terceira iteração do elemento chave, obtém-se o seguinte resultado:



FINAL DA TERCEIRA ITERAÇÃO

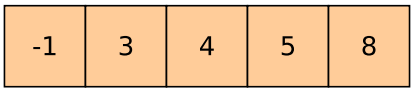
Na quarta e última iteração, varre-se o vetor para deslocando o valor armazenado no elemento chave para a antepenúltima posição (posição 2). Neste caso, o elemento chave volta para a posição 0.



1a iteração:

- chave: `v[0] = -1;`
- próximo: `v[1] = 0;`
- `v[0] > v[1]` → falso. Não trocar;
- Ir para o próximo elemento a ser comparado: v[1];

Ao final da última iteração do elemento chave, obtém-se o seguinte resultado:



FINAL DA ÚLTIMA ITERAÇÃO

Perceba que todas as comparações realizadas e o vetor está ordenado. Considerando que a busca é feita até a posição `i` do vetor, o elemento chave é `v[j]`, o elemento posterior é `v[j+1]`, para percorrer os elementos:

- É necessário um laço que percorra os índices `n-1,...,1`, para indicar onde se localizará a última posição:
  `for(i = n-1; i <= 1; i-)`
- É necessário um laço que passe por cada elemento para indicar o elemento chave. Este laço deve ir até `i-1`, pois este último será comparado com o elemento final `i` :
  `for(j = 0; j < i; j++)`
- Por fim, é necessária uma operação de comparação e troca entre os elemento chave `j` e o imediatamente posterior `j+1`.

O código em C++ pode ser visualizado no programa a seguir:

```
void ordena_crescente(int v[], int n){
    int i, j, aux;
    for(i = n-1; i >= 1; i--){
        for(j = 0; j < i; j++){
            if(v[j] > v[j+1]){
                aux = v[j+1];
                v[j+1] = v[j];
                v[j] = aux;
            }
        }
    }
}

int main(){
    int n, i;
    cin >> n;
    int v[n];
    for(i = 0; i < n; i++){
        cin >> v[i];
    }
    ordena_crescente(v, n);
    for(i = 0; i < n; i++){
        cout << v[i] << " ";
    }
    cout << endl;
    return 0;
}
```

8.4 Exercícios

1. Implemente uma função chamada `le_vet`, para ler os elementos de um vetor de números inteiros de tamanho `n`. Também, implemente uma função chamada de `imprime_vet`, para imprimir os elementos de um vetor de números inteiros em uma mesma linha da tela;
2. Implemente uma função que recebe como parâmetros de entrada um vetor de caracteres e um caractere. A função deve retornar quantos dos caracteres no vetor são iguais ao caractere passado como parâmetro;
3. Implemente uma função que recebe como parâmetros de entrada dois vetores de números inteiros e retorna o maior valor dentre todos, seja ele do primeiro ou do segundo vetor.
4. Implemente uma função que receba como parâmetro de entrada um vetor de números reais. A função a ser implementada deve retornar a média dos valores no vetor. A função `main` deve ler o tamanho `n` do vetor, cada um dos seus elementos e exibir na tela uma mensagem informando a média dos valores do vetor utilizando a função implementada. Exemplo:

   ```
   Informe o tamanho do vetor:
   5
   Informe os elementos do vetor:
   2.5 3.5 5.0 9.0 1.0
   Media do vetor: 4.2
   ```

5. Implemente uma função que receba como parâmetro de entrada um vetor de números reais. A função a ser implementada deve retornar a quantidade de valores que são menores do que

a média entre eles, utilizando para isto uma chamada à função da Questão 4. A função `main` deve ler o tamanho `n` do vetor, cada um dos seus elementos e exibir na tela uma mensagem informando quantos elementos do vetor são maiores do que a média utilizando a função implementada. Exemplo:

```
Informe o tamanho do vetor:
5
Informe os elementos do vetor:
2.5 3.5 5.0 9.0 1.0
Elementos menores do que a media: 3
```

6. Implemente uma função que receba como parâmetro de entrada um vetor de números inteiros. A função a ser implementada deve retornar o maior elemento do vetor. A função `main` deve ler o tamanho `n` do vetor, cada um dos seus elementos e exibir na tela uma mensagem informando todas as posições do vetor em que o maior elemento está presente utilizando a função implementada. Exemplo:

```
Informe o tamanho do vetor:
6
Informe os elementos do vetor:
-11 7 1 2 7 7
Maior elemento ocorre nas posicoes: 1 4 5
```

7. Implemente uma função que receba como parâmetros de entrada dois vetores de números reais. A função a ser implementada deve retornar o produto interno entre os dois vetores, ou seja,

$$\mathbf{u}^t.\mathbf{v} = \sum_{i=0}^{n-1} u_i.v_i = u_0.v_0 + u_1.v_1 + ... + u_{n-1}.v_{n-1},$$

com

$$\mathbf{u} = [u_0, u_1, ..., u_{n-1}]^t$$

e

$$\mathbf{v} = [v_0, v_1, ..., v_{n-1}]^t.$$

A função `main` deve ler o tamanho `n` dos vetores, cada um dos elementos de ambos os vetores e, utilizando chamadas à função implementada, exibir na tela mensagens informando:
   (a) O produto interno entre os dois vetores
   (b) A norma do primeiro vetor
   (c) A norma do segundo vetor

Observe que a norma $||\mathbf{u}||$ de um vetor qualquer pode ser obtida pela relação

$$||\mathbf{u}||^2 = \mathbf{u}^t.\mathbf{u}$$

Exemplo:

```
Informe o tamanho dos vetores:
3
Informe os elementos do vetor1:
1.0 0.0 2.0
```

```
Informe os elementos do vetor2:
0.0 1.0 1.0
Produto interno: 2
Norma do vetor1: 2.23607
Norma do vetor2: 1.41421
```

8. Implemente uma função que receba como parâmetros de entrada dois vetores de números inteiros e como parâmetro de saída um outro vetor de números inteiros. A função a ser implementada deve armazenar no vetor de saída a concatenação de todos os valores do primeiro vetor seguidos por todos os valores do segundo vetor, nesta ordem. A função `main` deve ler o tamanho de cada vetor (eles podem ter tamanhos diferentes), cada um dos elementos de ambos os vetores e, utilizando uma chamada à função implementada, exibir o vetor resultante. Exemplo:

```
Informe o tamanho do primeiro vetor:
4
Informe o tamanho do segundo vetor:
3
Informe os elementos do vetor1:
1 -5 8 7
Informe os elementos do vetor2:
1 5 3
Vetor resultante:
1 -5 8 7 1 5 3
```

9. Implemente uma função que receba como parâmetros de entrada dois vetores de números inteiros e como parâmetro de saída um outro vetor de números inteiros. A função a ser implementada deve armazenar no vetor de saída o entrelaçamento dos valores do primeiro vetor com os valores do segundo vetor (primeiro elemento do primeiro vetor, primeiro elemento do segundo vetor, segundo elemento do primeiro vetor, segundo elemento do segundo vetor, etc.). A função `main` deve ler o tamanho de cada vetor (eles podem ter tamanhos diferentes), cada um dos elementos de ambos os vetores e, utilizando uma chamada à função implementada, exibir o vetor resultante. Exemplo:

```
Informe o tamanho do primeiro vetor:
4
Informe o tamanho do segundo vetor:
3
Informe os elementos do vetor1:
1 -5 8 7
Informe os elementos do vetor2:
1 5 3
Vetor resultante:
1 1 -5 5 8 3 7
```

10. Implemente uma função que receba como parâmetro de entrada um vetor de inteiros e como parâmetros de saída outros dois vetores de inteiros. A função a ser implementada deve armazenar no primeiro vetor de saída todos os números pares e no segundo vetor de saída todos os números ímpares. Observe que esta função também deve computar o tamanho dos vetores de saída, ou seja, a quantidade de números pares e ímpares: para isto, faça com que o tamanho de cada vetor de saída seja também um parâmetro de saída. A função `main` deve ler o tamanho do vetor, cada um dos seus elementos e, utilizando uma chamada à função implementada, exibir os vetores resultantes. Exemplo:

```
Informe o tamanho do vetor:
7
Informe os elementos do vetor:
0 1 2 3 4 5 6
Elementos pares:
0 2 4 6
Elementos impares:
1 3 5
```

9. Matrizes

Tabelas costumam ser úteis na representação de informações em banco de dados. Porém, são mais empregadas em programas que lidam com dados matemáticos, pelo formalismo associado. São estruturas bidimensionais, portanto exigem, no mínimo, duas informações para serem bem definidas.

9.1 Introdução

Assim como vetores, matrizes, armazenam dados do mesmo tipo em um bloco contínuo de memória. No entanto, a disposição dos dados é feita em formato de tabela, de maneira que é considerada uma estrutura bidimensional. Por causa disto, esta estrutura exige dois subscritos para a indexação de um elemento: o primeiro sempre é associado as linhas da matriz, e o segundo às colunas. Além disto, são necessárias duas variáveis para armazenar o número de linhas e das colunas da estrutura.

- `nl` linhas
- `nc` colunas
- `nl` x `nc` elementos (de um mesmo tipo)

É comum o emprego de matrizes numéricas na matemática. No entanto, matrizes em C++ podem armazenar outros tipos de dados. Alguns exemplos são palavras de um texto, *pixels* de uma imagem, relações entre pessoas.

9.2 Definição e Indexação de Matrizes

A sintaxe para a definição de uma matriz segue regra semelhante para a declaração de vetores, porém, envolvendo uma dimensão a mais. Isto é feito de maneira que cada um das dimensões é delimitada com colchetes `[]`.

```
<tipo_da_matriz> <nome_da_matriz> [<n_linhas>][<n_colunas>];
```

Onde:

- `<tipo_da_matriz>` são os tipos de variáveis conhecidos: `int, char, float, bool`;
- `<nome_da_matriz>` segue as mesmas regras usadas para variáveis;
- `<n_linhas>` é uma expressão para o número de linhas da matriz cujo tipo deve ser `int`;
- `<n_colunas>` é uma expressão para o número de colunas da matriz cujo tipo deve ser `int`.

Exemplo

Escrever declaração de matriz:

1. Do tipo real e tamanho 2 x 2, chamada `A`;
2. Do tipo caractere e tamanho 10 x 10, chamada `img`;
3. Do tipo inteiro e tamanho `nl` x `nc`, chamada `mat`.

Solução:

1. `float A[2][2];`
2. `char img[10][10];`
3. `int mat[nl][nc];`

A disposição física de uma matriz é idêntica a de um vetor em C++: como um bloco de memória. No entanto, a representação lógica de uma matriz é diferente da representação de um vetor, podendo ser verificada na figura 9.1. Em uma matriz definida com `nl` linhas e `nc` colunas, os índices de linhas válidos vão de `0` até `nl-1`, e os índice de coluna válidos vão de `0` até `nc-1`.

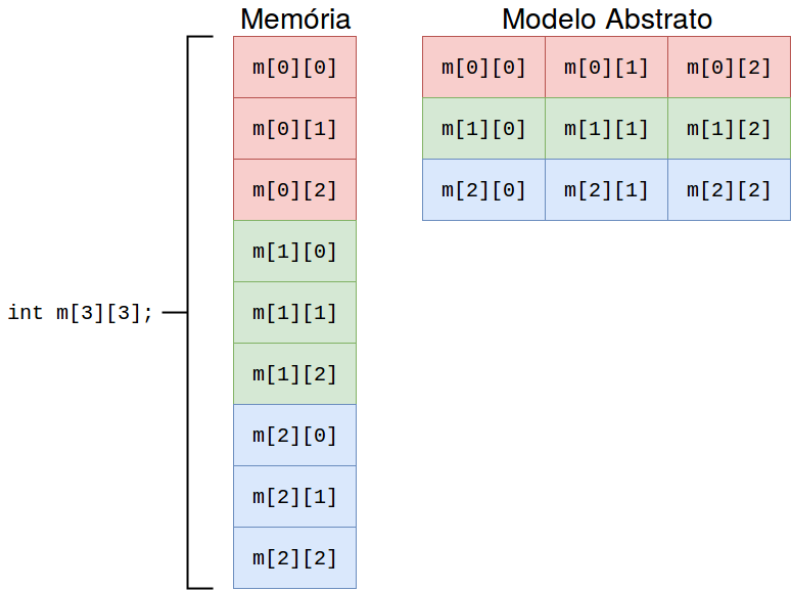
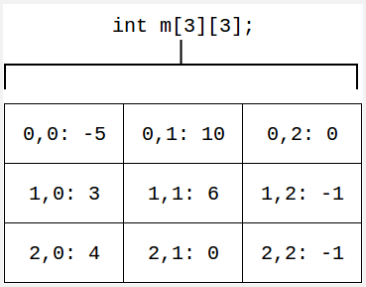


Figura 9.1: Representação física e lógica de uma matriz.

Assim como acontece me vetores, uma matriz `m` não pode ser usada diretamente em expressões; cada elemento `m[0][0]`,`m[0][1]`, ..., `m[nl-1][nc-1]` é que deve ser usado. Isto envolve a indexação adequada da matriz; para acessar cada elemento da matriz `m` é preciso lembrar que o primeiro índice acessa a linha da matriz e o segundo índice acessa a coluna da matriz. Como exemplo, a declaração `m[2][0]` acessa a terceira linha e primeira coluna da matriz `m`.

Exemplo

Dada a seguinte matriz:



Qual o valor de cada expressão a seguir, sendo `x = 1`?

1. `m[0][0] + m[0][1]`
2. `m[0][3*x-1]`
3. `m[m[2][1]][x+1]`
4. `m[0][-1]`
5. `m[3][3]`
6. `m[3][1000]`

Solução:

1. `m[0][0] + m[0][1] = (-5) + 10 = 5`
2. `m[0][3*x-1] = m[0][3*1-1] = m[0][2] = 0`
3. `m[m[2][1]][x+1] = m[0][x+1] = m[0][1+1] = m[0][2] = 0`
4. `m[0][-1]: posição inválida`
5. `m[3][3]: posição inválida`
6. `m[3][1000]: posição inválida`

Para acessar cada elemento de uma matriz de forma automática, é necessário iterar todas as colunas de todas as linhas da matriz. Isto significa um `for` para variar o índice da linha e um `for` para variar o índice da coluna, de maneira aninhados. A ordem dos laços pode ou não influenciar no resultado, dependendo do algoritmo.

Exemplo

Implemente um programa que leia do usuário as dimensões de uma matriz de inteiros e que, em seguida, armazene cada um de seus elementos.

Solução:

```
int main(){
    int nl, nc;
    cin >> nl >> nc;
    int m[nl][nc], i, j;
    for(i = 0; i < nl; i++){
        for(j = 0; j < nc; j++){
            cin >> m[i][j];
        }
    }
}
```

9.3 Inicialização de Matrizes

Assim com vetores, é possível atribuir valores iniciais a um matriz. Existem várias formas de realizar a inicialização de uma matriz.

- Forma 1: Informando o número de linhas e de colunas, e listando cada linha separadamente, junto com os seus elementos;

```
int m[3][4] = {{1,2,3,4},
               {5,6,7,8},
               {9,10,11,12}};
```

- Forma 2: Informando apenas o número de colunas, e listando cada linha separadamente, junto com os seus elementos. Neste caso, o número de linhas será obtido pelo número de elementos em cada linha;

```
int m[][4] = {{1,2,3,4},
              {5,6,7,8},
              {9,10,11,12}};
```

- Forma 3: Informando o número de linhas e de colunas, e apenas listando os elementos do vetor. Neste caso, a distribuição dos elementos na matriz será feito de forma automática, de acordo com a ordem dos elementos;

```
int m[3][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

- Forma 4: Informando apenas o número de colunas, e apenas listando os elementos do vetor. Neste caso, número de linhas será estabelecido de acordo com o número de elementos listados. Além disto, a distribuição dos elementos na matriz será feito de forma automática, de acordo com a ordem dos elementos.

```
int m[][4] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

É importante observar que o número de linhas pode ser omitido, mas o número de colunas precisa ser especificado durante a inicialização de uma matriz. Caso isto não aconteça, o compilador

não poderá determinar a distribuição dos dados no bloco de memória, e um erro de compilação será emitido.

Exemplo

Implemente um programa que leia dois números inteiros `M` e `N` e em seguida, leia as `N` notas de `M` turmas. O programa deve apresentar todas as turmas onde encontra-se a maior nota.

Solução:

```
int main(){
    int M, N, i, j, lm, cm;
    cout << "Informe a quantidade de turmas:\n";
    cin >> M;
    cout << "Informe a quantidade de alunos:\n";
    cin >> N;
    float notas[M][N], maior;
    cout << "Informe as notas:\n";
    for(i = 0; i < M; i++){
        for(j = 0; j < N; j++){
            cin >> notas[i][j];
            if(i == 0 && j == 0){
                maior = notas[i][j];
                lm = i;
                cm = j;
            }
            else if(notas[i][j] > maior){
                maior = notas[i][j];
                lm = i;
                cm = j;
            }
        }
    }
    for(i = 0; i < M; i++){
        for(j = 0; j < N; j++){
            if(notas[i][j] == maior){
                cout << "Turma " << i
                     << " tem a maior nota\n";
            }
        }
    }
    return 0;
}
```

C++ permite a definição de matrizes com mais do que 2 dimensões, como `int mat[3][3][3];`. O número máximo de dimensões depende do compilador.

9.4 Funções com Matrizes

É possível passar matrizes como parâmetro de funções. isto é um recurso útil para realizar blocos de código repetitivos ou para a construção, especialmente porque matrizes costumam ser utilizadas em operações complexas, com uma quantidade significativa de linha de código. Além da matriz a ser passada como parâmetro, é necessário passar outros dois parâmetros, relativos ao número de

linhas e colunas da matriz. No entanto, em relação aos vetores, a sintaxe para matrizes tem uma diferença sutil, porém importante.

Para passar matrizes como parâmetros de funções em C++, é necessário primeiramente a declaração de uma constante global com o número máximo de linhas/colunas de uma matriz (constante `MAX`, inicializada com um número grande). Todas as matrizes declaradas ou inicializadas durante o programa precisam ser declaradas com esta constante no número de linhas e colunas. Além disto,é preciso informar, no protótipo da função, que cada matriz passada como parâmetro tem `MAX` colunas, preenchendo o segundo par de colchetes com este valor. Isto acontece por causa da possibilidade de inicialização apenas com a listagem dos elementos (como um vetor), que exige que o número de colunas de uma matriz passada como parâmetro seja constante. Os programas implementados podem não usar todas as `MAX` linhas e `MAX` colunas das matrizes, mas precisam deixar explícito a sua capacidade máxima.

Como exemplo de aplicação, um programa declara matriz de `100 x 100`, mas usa `nl x nc` elementos.

```
const int MAX = 100;
int main(){
    int mat[MAX][MAX], nl, nc, i, j;
    cin >> nl >> nc;

    for(i = 0; i < nl; i++){
        for(j = 0; j < nc; j++){
            cin >> mat[i][j];
        }
    }

    return 0;
}
```

A linha `const int MAX = 100;`, como visto anteriormente, é a definição de uma constante, que pode ser feita com a diretiva `#define`: `#define MAX 100`. E importante lembrar que esta forma não tem ponto e vírgula.

A sintaxe de uma função com uma matriz passada como parâmetro é dada a seguir:

```
tipo_func nome_func(tipo_matriz nome_matriz[][MAX],
                    int n_linhas, int n_cols){
    corpo da funcao
}
```

Em relação às declarações de funções usadas anteriormente:

- `tipo_matriz` é o tipo da matriz passada como parâmetro: `int, char, float, bool`.
- `nome_matriz` é o nome da matriz passada como parâmetro. Notar que o primeiro `[]` é em branco e o segundo `[]` deve ter obrigatoriamente o número máximo de colunas (no caso, `MAX`);
- `n_linhas` é número de elementos válidos nas linhas da matriz;
- `n_cols` é número de elementos válidos nas colunas da matriz.

Caso haja mais de uma matriz, é necessário informar o número de linhas e colunas de cada uma das estruturas.

Exemplo

Escrever assinatura da função:

1. Que imprime uma matriz de inteiros na tela
2. Que recebe uma matriz de caracteres e retorna quantos dos elementos na matriz são iguais a um caractere passado como parâmetro
3. Que recebe duas matrizes de inteiros de mesmo tamanho e retorna o maior valor dentre todos, seja ele da primeira ou da segunda matriz

Solução:

```
void imprime_matriz(int mat[][MAX], int nl, int nc);
int conta_ocorrencias(char mat[][MAX], int nl, int nc,
                      char c);
int computa_maior(int mat1[][MAX], int mat2[][MAX],
                  int nl, int nc);
```

Nas chamadas às funções, matrizes são passadas como parâmetros utilizando apenas o seu nome (sem colchetes). Como exemplo, um programa que imprime uma matriz cujos elementos seguem a seguinte regra de formação: `m[i][j] = nc*i+j+1`.

```
const int MAX = 100;
void imprime_mat(int mat[][MAX], int nl, int nc);
int main(){
    int m[MAX][MAX], i, j, nl = 2, nc = 4;
    for(i = 0; i < nl; i++){
        for(j = 0; j < nc; j++){
        m[i][j] = nc*i+j+1;
        }
    }
    imprime_mat(m, nl, nc);
    return 0;
}
```

Assim como acontece com vetores, toda matriz passada para funções como parâmetro é passada por **referência**. Isto significa dizer que as alterações realizadas nas matrizes são visíveis fora do corpo da função. Também, o emprego do operador de referência (&) não é premitido para o caso de matrizes, ocasionando um erro de sintaxe.

9.5 Exercícios

1. Implemente uma função que recebe duas matrizes de inteiros de tamanhos iguais e computa a soma matricial. Implemente também a função `main`. A leitura de dados também deve ser implementada por uma função.
2. Implemente uma função que receba uma matriz de números inteiros quadrada de ordem `n` como parâmetro de saída. A função a ser implementada deve armazenar na matriz uma matriz identidade de tamanho `n` $\times$ `n`. A função `main` deve ler a ordem `n` da matriz quadrada e exibir a matriz resultante na tela utilizando a função implementada. Exemplo:

```
Informe o nr. de linhas/colunas:
4
```

```
Matriz resultante:
1 0 0 0
0 1 0 0
0 0 1 0
0 0 0 1
```

3. Implemente uma função que receba uma matriz de números inteiros quadrada de ordem `n` como parâmetro de saída. A função a ser implementada deve armazenar em sua diagonal secundária um elemento `x` passado como parâmetro. A função `main` deve ler a ordem `n` da matriz quadrada, o número inteiro `x` e exibir a matriz resultante na tela utilizando a função implementada. Exemplo:

```
Informe o nr. de linhas/colunas:
3
Informe o nr. a ser inserido na diagonal secundaria:
8
Matriz resultante:
0 0 8
0 8 0
8 0 0
```

4. Implemente uma função que receba uma matriz de números inteiros de tamanho `nl` $\times$ `nc` como parâmetro de saída. A função a ser implementada deve armazenar os `nl` $\times$ `nc` primeiros números ímpares nos seus elementos, seguindo a ordem de preencher cada coluna primeiro para então prosseguir para a próxima linha. A função `main` deve ler as dimensões da matriz `nl` e `nc` e exibir a matriz resultante na tela utilizando a função implementada. Exemplo:

```
Informe o nr. de linhas da matriz:
2
Informe o nr. de colunas da matriz:
3
Matriz resultante:
1 5 9
3 7 11
```

5. Implemente uma função que receba uma matriz de números inteiros quadrada de ordem `n` como parâmetro de entrada e saída. A função a ser implementada deve transformar a matriz passada como parâmetro em uma matriz triangular superior: matriz em que os elementos M_{ij} abaixo da diagonal principal são iguais a 0. A função `main` deve ler a ordem `n` da matriz quadrada, cada um dos elementos da matriz e exibir a matriz resultante na tela utilizando a função implementada. Exemplo:

```
Informe o nr. de linhas/colunas:
3
Informe os elementos da matriz:
1 -1 2
2 45 3
7 11 14
Matriz resultante:
1 -1 2
0 45 3
0 0 14
```

6. Implemente uma função que receba uma matriz de números inteiros de tamanho `nl` $\times$ `nc` como parâmetro de entrada e como parâmetro de saída três números inteiros. A função a ser implementada deve armazenar nos parâmetros de saída o maior elemento da matriz e também a linha e coluna onde este está localizado. A função `main` deve ler as dimensões da matriz `nl` e `nc`, cada um dos seus elementos e exibir na tela uma mensagem informando o maior elemento da matriz e a linha e coluna onde ele está localizado utilizando a função implementada. Exemplo:

```
Informe o nr. de linhas da matriz:
2
Informe o nr. de colunas da matriz:
4
Informe os elementos da matriz:
8 9 -1 1
2 3 5 1
Maior elemento e 9, esta na linha 0 e coluna 1
```

7. Implemente uma função que receba como parâmetro de entrada uma matriz de números inteiros de tamanho `nl` $\times$ `nc`. A função a ser implementada deve retornar verdadeiro caso a matriz seja uma matriz esparsa ou falso caso contrário. Considere uma matriz `nl` $\times$ `nc` como sendo esparsa se pelo menos 70% dos seus elementos forem iguais a zero. A função `main` deve ler as dimensões da matriz `nl` e `nc`, cada um dos seus elementos e exibir na tela uma mensagem informando se a matriz é esparsa ou não. Exemplo:

```
Informe o nr. de linhas da matriz:
2
Informe o nr. de colunas da matriz:
2
Informe os elementos da matriz:
0 1
0 0
A matriz e esparsa
```

8. Implemente uma função que receba uma matriz de números inteiros quadrada de ordem `n` como parâmetro de entrada. A função a ser implementada deve retornar verdadeiro caso a matriz seja uma matriz de permutação ou falso caso contrário. Uma matriz `n` $\times$ `n` é de permutação se ela for formada apenas por 0s e 1s e se cada uma de suas linhas e colunas possuir apenas um único elemento igual a 1. A função `main` deve ler a ordem `n` da matriz quadrada, cada um dos seus elementos e exibir na tela uma mensagem informando se a matriz é ou não de permutação utilizando a função implementada. Exemplo:

```
Informe o nr. de linhas/colunas:
4
Informe os elementos da matriz:
0 1 0 0
1 0 0 0
0 0 1 0
0 0 0 1
Matriz informada e de permutacao
```

9. Implemente uma função que receba como parâmetro de entrada uma matriz de números inteiros de tamanho `nl` $\times$ `nc` e também um vetor de números inteiros. A função a ser

implementada deve armazenar em cada posição do vetor o maior elemento de cada coluna da matriz. A função `main` deve ler as dimensões da matriz `nl` e `nc`, cada um dos seus elementos e exibir na tela o vetor resultante. Exemplo:

```
Informe o nr. de linhas da matriz:
3
Informe o nr. de colunas da matriz:
3
Informe os elementos da matriz:
5 1 3
4 8 9
2 6 3
Vetor resultante:
5 8 9
```

10. Strings

Textos geralmente são utilizados na exibição de informações. No entanto, existe a necessidade de processamento de textos em editores, por exemplo. Por apresentar uma complexidade maior de organização, a manipulação de textos envolve também operações mais elaboradas. Para a representação de textos, geralmente utilizam-se cadeias de caracteres especiais, as *strings*.

10.1 Introdução

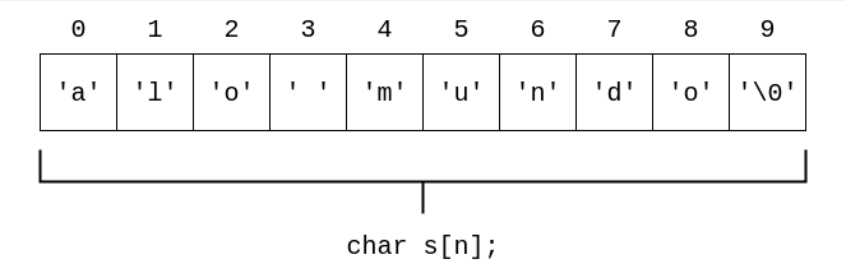
Durante o curso, foram vistas formas para lidar com caracteres individuais, por meio de variáveis, e de textos constantes, como ”`Digite um valor:`”. Neste último caso, o objetivo maior era apenas exibi-los na tela, sem nenhuma necessidade de processá-los antes. Por meio de vetores, foi visto como manipular sequências de valores, inclusive caracteres. Este será o foco deste capítulo, por meio de um subgrupo denominado *strings*.

Denomina-se de *string* uma cadeias ou sequência de caracteres finalizada com um caractere especial `'\0'` (já citado anteriormente). Ao se estabelecer um texto constante com aspas duplas (“ ”), o compilador geralmente a associa como uma string. Então, são exemplos de strings `"ect"` e `"linguagem de programacao"`. Existe um tipo `string` já definido pela linguagem; porém, ele não será visto: será trabalhada a ideia de `strings` como um vetor de caracteres especial.

Em C++, *strings* são vetores de `char` com um último caractere especial `'\0'`. É a presença do `\0` que caracteriza que aquele vetor de caracteres é uma *string*. Por isto, é importante afirmar que uma *string* é um vetor de caracteres, mas não o contrário: nem todo vetor de caracteres apresenta o `\0` e, portanto, não pode ser considerado *string*. Embora seja um caractere não-imprimível, o `\0` é importante nos algoritmos que manipulam *strings* justamente por ser o indicador de final do texto.

Exemplo

Dado o seguinte vetor de caracteres:



1. Este vetor pode ser considerado uma *string*? Por quê?
2. Qual o caractere na terceira posição?
3. Qual o caractere `s[5]`?
4. Como poderíamos mudar a string para `"ola mundo"`?
5. Qual o tamanho da string? O tamanho é a quantidade de todos os caracteres,menos o caractere especial.

Solução:

1. Como há a presença do caractere `\0`, então o vetor pode ser considerado uma *string*;
2. Terceira posição: `s[2] = 'o';`
3. `s[5] = 'u';`
4. A princípio, bastaria trocar as posições dos elementos 0 e 2 da *string*. Na prática, basta sobrepor o valor corrente com o novo valor, inserindo o caractere `'\0'` ao final;
5. Sem contar o `'\0'`, há 9 elementos. Portanto, o tamanho do vetor dado é de 9 unidades.

10.2 Inicialização de uma *string*

A inicialização de uma *string* pode ser feita de maneira semelhante a de um vetor convencional. A grande diferença é a necessidade de colocar o caractere `'\0'` ao final.

- Forma 1: Informando o número de espaços de memória e listando os elementos. Neste caso, é preciso definir espaço suficiente para os caracteres do texto e mais o `'\0'`;

```
char s[4] = {'e','c','t','\0'};
```

- Forma 2: Apenas listando os elementos. Neste caso, o próprio compilador de encarregará de alocar espaço suficiente para os caracteres do texto mais o `'\0'`;

```
char s[] = {'e','c','t','\0'};
```

- Forma 3: Escrevendo o texto com aspas duplas. Aqui, o caractere especial é inserido automaticamente;

```
char s[] = "ect";
```

Note a diferença para um vetor de caracteres (não delimitado com `'\0'`):

```
char s[] = {'e','c','t'};
```

Este tipo de inicialização não define uma *string*, apenas um vetor de caracteres.

10.3 Impressão e Leitura de *strings*

A impressão de uma string `s` pode ser feita utilizando o comando `cout`, diretamente na variável do tipo *string*. Isto significa que não é necessário utilizar um laço como no caso de vetores. Porém, isto só funciona se a string estiver terminada com o caractere delimitador; caso contrário, serão impressos vários caracteres a mais (lixo de memória).

```
int main(){
    char str[4] = {'e','c','t','\0'};
    cout << str << endl;
    return 0;
}
```

Para ler uma string `s` do usuário, pode-se usar o comando `cin`, diretamente na variável do tipo string, caso esta não possua espaços em branco. Isto dispensa o uso de um laço para ler cada caractere. Além disto, com o `cin`, o caractere especial `'\0'` é inserido automaticamente ao final do texto.

```
int main(){
    char str[10];
    cin >> str;
    cout << str << endl;
    return 0;
}
```

é importante reforçar que, embora a variável `str` tenha tamanho 10, o tamanho máximo do texto que cabe na string é de 9 caracteres. Isto acontece porque deve-se armazenar também o caractere especial `'\0'`. Logo, para ler uma *string* de tamanho n, é necessário um vetor de, no mínimo, $n+1$ espaços de memória.

No entanto, caso a string desejada possua espaços em branco (`' '`), ela deve ser lida com um outro comando. Isto porque, durante a leitra de dados com o `cin`, espaços em branco ou quebras de linha (tecla enter) informam o final da string digitada pelo usuário. Neste caso, a função `cin.getline` deve ser utilizada

```
cin.getline(<nome_da_string>, <capacidade_da_string>)
```

São parâmetros desta função:

1. `<nome_da_string>`: Vetor de caractere onde o texto deve ser armazenado;
2. `<capacidade_da_string>`: Número máximo de caracteres a serem lidos incluindo o `'\0'`

Nesta função, a entrada do teclado é lida até que uma quebra de linha (tecla enter) seja digitada. Como exemplo de aplicação, considere o exemplo a seguir:

```
int main(){
    char s[51];
    cin.getline(s, 51); //le 50 caracteres
    cout << s << endl;
    return 0;
}
```

Uma questão importante com o emprego do `cin.getline()` é que ele não funcionará corretamente em programas onde este comando comando é executado **após** a solicitação de um `cin`. Isto porque o compilador já considera a quebra de linha como um caractere válido para a leitura. Um exemplo deste tipo de problema pode ser visto no programa a seguir:

```
const int STRMAX = 31;
int main(){
    char s[STRMAX];
    int x;
    cout << "Informe um inteiro:\n";
    cin >> x;
    cout << "Informe uma string:\n";
    cin.getline(s, STRMAX);
    cout << "String informada: " << s << endl;
    return 0;
}
```

É importante perceber que antes da chamada ao comando `cin.getline()`, houve a solicitação de um `cin` para preencher o valor da variável `x`. Com isto, a entrada para a string é ignorada e o resultado é o seguinte:

```
Informe um inteiro:
5
Inteiro informado: 5
Informe uma string:
String:
```

Neste caso, é preciso indicar para o compilador que desconsidere a quebra de linha anterior como entrada do texto. Para isto, é preciso usar a função `cin.ignore()` antes da função `cin.getline()`. O exemplo corrigido é mostrado a seguir:

```
const int STRMAX = 31;
int main(){
    char s[STRMAX];
    int x;
    cout << "Informe um inteiro:\n";
    cin >> x;
    cin.ignore();
    cout << "Inteiro informado: " << x << endl;
    cout << "Informe uma string:\n";
    cin.getline(s, STRMAX);
    cout << "String informada: " << s << endl;
    return 0;
}
```

10.4 Uso de *strings* em funções

A princípio, o emprego de *string* em funções, basicamente segue a mesma regra do emprego de vetores em funções. No entanto, como este conjunto de vetores possui um caractere de delimitação

próprio, que sinaliza os caracteres válidos, não há a necessidade de informar o número de elementos do vetor. A definição de uma função com uma *string* pode ser vista a seguir:

```
<tipo_de_retorno> <nome_da_funcao> (char <nome_da_string>[])
```

Onde:

- `<tipo_de_retorno>` é o tipo associado a saída da função;
- `<nome_da_funcao>` é o identificado da função;
- `<nome_da_string>` é o nome associado ao vetor de caractere, a *string* propriamente dita;

Embora seja necessário um conjunto grande de funções para manipular textos, algumas funções úteis na manipulação de *strings* já são disponibilizadas em C++ pela biblioteca `cstring`. Algumas delas são mostradas a seguir:

- `strlen`: Computa o tamanho da string.

```
int strlen(char s[])
```

Ela retorna o tamanho da string `s`, usando o caractere delimitador para identificar o final da cadeia;

- `strcpy`: Computa uma cópia da string.

```
void strcpy(char dest[], char orig[])
```

Ela realiza a copia do conteúdo do vetor `orig` para o vetor `dest`;

- `strcat`: Concatena duas strings.

```
void strcat(char s1[], char s2[])
```

Ela concatena o conteúdo da cadeia `s2` na cadeia `s1`, após o final do conteúdo desta;

- `strcmp`: Compara duas strings.

```
int strcmp(char s1[], char s2[])
```

A função compara a cadeia `s1` com `s2`, possuindo três valores de retorno possíveis:

- Retorna 0 se elas forem iguais;
- Retorna um número negativo se `s1` for menor do que `s2`;
- Retorna um número positivo se `s1` for maior do que `s2`.

A condição de menor ou maior irá depende da posição lexicográfica (indicada em dicionário) das *strings* e não necessariamente dos tamanhos.

Exemplo

Implemente cada uma das funções da biblioteca `cstring`:

1. `int strlen(char s[]);`
2. `void strcpy(char dest[], char orig[]);`
3. `void strcat(char s1[], char s2[]);`
4. `int strcmp(char s1[], char s2[]);`

Solução:

1. Para a função `strlen`, basta contar quantos caracteres há antes do `\0`.

```
1
2  int strlen(char s[]) {
3          int c = 0;
4          while(s[c] != \0) {
5                  c++;
6          }
7          return c;
8  }
```

2. Para a função `strcpy`, basta varrer o vetor de entrada até o `'\0'` e atribuir a posição `i` do vetor `dest`, o elemento na posição `i` do vetor `orig`. Ao final, é necessário inserir o caractere `'\0'`.

```
1
2  void strcpy(char dest[], char orig[]) {
3          int i = 0;
4          while(orig[i] != '\0') {
5                  dest[i] = orig[i];
6                  i++;
7          }
8          dest[i] = '\0';
9  }
```

3. Para a função `strcat`, é preciso varrer o vetor `s1` até o final e, depois, começar a inserir os elementos do vetor `s2` a partir da posição final de `s1`. Ao final, é necessário inserir o caractere `'\0'`.

```
1
2  void strcat(char s1[], char s2[]) {
3          int i = 0;
4          while(s1[i] != '\0') {
5                  i++;
6          }
7          while(s2[i] != '\0') {
8                  s1[i] = s2[i];
9          }
10         s1[i] = '\0';
11 }
```

4. Para a função `strcmp`, é preciso varrer os vetores `s1` e `s2` até que seja encontrado um caso em que uma das *strings* foi esgotada ou foram encontrados dois caracteres distintos. Sabendo que o caractere `'\0'` vale zero na tabela ASCII, pode-se utilizar a diferença entre o valor dos caracteres indicados pelo índice de referência após a varredura para informar o resultado da comparação.

```
1
2  int strcmp(char s1[], char s2[]) {
3          int c = 0;
4          while(s1[i] != '\0' && s2 != '\0'
5                  && s1[i] == s2[i]) {
6                  i++;
7          }
8          return (int)s1[i] - (int)s2[i];
9  }
```

Exemplo

Implementar um programa utilizando funções da biblioteca `cstring` que leia uma string do usuário, copia a string lida para uma segunda string e verifica se a cópia da string é igual à palavra "`ect`", imprimindo uma mensagem conforme o caso.

Solução:

```
1  #include <cstring>
2
3  int main(){
4      char str[10], copia[10], palavra[10] = "ect";
5      cin >> str;
6      strcpy(copia, str);
7      if(strcmp(copia, palavra) == 0){
8          cout << "Iguais\n";
9      }
10     else{
11         cout << "Diferentes\n";
12     }
13     return 0;
14 }
```

10.5 Tratamento de Texto com *strings*

Em alguns casos, é necessário realizar a identificação e separação de palavras de um texto. O resultado final comumente é armazenado em um vetor de `strings`. Como cada `string`, por si só, envolve um vetor de caracteres, o resultado é expresso como uma matriz de caracteres. Portanto, vetor de `strings` e matriz de caracteres se referem a mesma estrutura. Neste caso, cada linha da matriz corresponde a uma palavra separada, de forma que, para uma matriz de caracteres `m`, o elemento `m[i]` refere-se a `i`-ésima palavra do texto.

Os algoritmos que realizam tais operações se baseiam no fato de que *cada palavra é separada por, no mínimo, um espaço em branco ou pontuação*. A ideia é que, caso um espaço em branca ou pontuação seja encontrado, uma palavra acabou de ser encerrada, e deve ser armazenada na matriz de caracteres. Além disto, uma vez que foi encontrado um espaço em branco ou pontuação, caso seja encontrado um caractere posterior que não seja deste grupo, uma nova palavra será inicializada.

Para ilustrar este problema, considere a construção de um algoritmo que realize a separação de um texto supondo que este não possui espaços em branco no início e no fim do texto, e que cada palavra é separada por apenas um espaço em branco. Isto significa que a frase ‘‘`A bola quica`’’ é tratada corretamente por este algoritmo, mas frases como ‘‘`A bola amarela.`’’, ‘‘`Zero, um, dois`’’ e ‘‘`-UFRN!!`’’ não são tratadas corretamente.

Neste caso, serão necessários três contadores para realizar a operação:

- Um contador `t` para varrer o texto de entrada do usuário;
- Um contador `p` para indicar a palavra corrente na matriz de caracteres. Este elemento estará associado a linha da matriz, e também ao número de palavras do texto. Isto acontece porque o número de elementos de um vetor também pode ser utilizado como índice para a próxima posição vaga do vetor;
- Um contador `m` para indicar a letra corrente da palavra indicada por `p`. Este elemento estará associado a coluna da matriz.

Ambos os contadores são inicializados em zero. Na execução do algoritmo, cria-se um laço para varrer o texto de entrada até o fim. Assumindo que este seja uma *string*, então este laço vai até que seja encontrado o caractere `\0`. A cada caractere do texto de entrada verifica-se se ele é um espaço em branco ou não.

- Caso não seja um espaço em branco, insere-se o caractere na posição p,m da matriz; ou seja, linha `p` e coluna `m`. Em seguida, incrementa-se o valor de `m`;
- Caso seja um espaço em branco, isto significa que a palavra indicada por `p` acabou, e outra irá começar. Então, as seguintes etapas são feitas:
  1. Insere-se o caractere `\0` na posição p,m da matriz;
  2. Incrementa-se o valor de `p`, indicando que uma nova palavra será inserida;
  3. Reduz-se o valor de `m` para zero, de maneira a apontar apar o inicio da nova palavra

Em todos os dois casos, ao final do processamento, é necessário incrementar o valor de `t` para que ele possa passar para o próximo caractere.

A princípio, ao final deste algoritmo, todas as palavras deveriam ser identificadas e separadas. NO entanto, como foi determinado que não houvesse espaços após o final do texto, a último palavra terminará justamente no `\0`. Isto significa que ela não receberá o caractere `\0` , pois isto só acontece quando um espaço é encontrado; consequentemente, não será identificada como uma `string`. Para contornar tal problema, basta que, após o laço de varredura do texto inicial, seja inserido o caractere `\0` na posição p,m da matriz, já que estes estarão apontando a próxima posição vazia da palavra. Além disto, incrementa-se também aqui o valor de `p`, indicando que uma nova palavra foi inserida. Isto é importante porque o valor de `p`, como citado anteriormente, também indica o número de palavras na matriz de caracteres/vetor de `strings`.

O código em C++ que implemente a operação dada é mostrado abaixo. Nele, a função que faz a separação das palavras recebe a `string` com o texto de entrada, a matriz de caracteres para armazenar as palavras e um parâmetro por referência para receber justamente a quantidade de palavras existente na matriz. O número de colunas não é citado, pois cada palavra equivale a uma linha da matriz, e são representadas como `strings`.

```
#define MAX 50

void palavras(char texto[], char palavras[MAX][MAX], int&
    nPalavras) {
     int t = 0; //Para indicar a posicao no texto;
     int p = 0; //Para indicar a palavra corrente;
     int m = 0; //Para indicar a proxima posicao livre na
        palavra;
     while (texto[t] != '\0') {
         //Se o caractere dado nao for um espaco em branco:
         if(texto[t] != ' ') {
             palavras[p][m] = texto[tp];
             m++;
         }
         //Caso contrario, e um espaco em branco;
         else {
                 palavras[p][m] = '\0';
                 p++;
                 m = 0;
         }
         t++;
     }
     //Tratamento para a ultima palavra:
     palavras[p][m] = '\0';
     p++;
     //O valor de 'p' informa o numero de palavras na matriz.
     nPalavras = p;
}
```

Existem vários tipos de problemas que envolvem o uso de vetores de *strings*. Um dos mais importante é a ordenação de palavras. Neste caso, leva-se em conta a posição lexicográfica (aquela determinada em dicionário) para realizar a ordenação cuja comparação pode feito com a função `strcmp`. O algoritmo para realizar a operação é semelhante ao que é empregado para tipos numéricos, como o *bubble sort*.

Quando se trata da posição lexicográfica, estabelece-se ordenação alfabética. Sabendo que a função `strcmp` recebe duas strings como parâmetros e realiza a comparação alfabética entre eles, basta estabelecê-la como condição de troca. Para isto, é preciso verificar se `strcmp(v[j], v[j+1])` retorna um número positivo se a *string* `v[j]` for alfabeticamente maior do que a *string* `v[j+1]` no cado da ordenação crescente. Caso deseja-se realizar a ordenação decrescente, é necessário verificar se textttstrcmp(v[j], v[j+1]) retorna um número negativo.

```
void ordenacao(char v[MAX][MAX], int n){
    int i, j;
    Aluno aux;
    for(i = n-1; i <= 1; i++){
        for(j = 0; j < i; j++){
            if(strcmp(v[j], v[j+1]) > 0){
                aux = v[i];
                v[i] = v[j];
                v[j] = aux;
            }
        }
    }
}
```

10.6 Exercícios

1. Implemente uma função que receba como parâmetro de entrada uma string e como parâmetro de saída uma outra string. A função a ser implementada deve armazenar na string de saída a string de entrada na ordem inversa. A função `main` deve ler uma string e exibir na tela a string computada pela função. Exemplo:

   ```
   Informe uma frase:
   Esta e uma frase
   Frase invertida:
   esarf amu e atsE
   ```

2. Implemente uma função que receba como parâmetro de entrada uma string e como parâmetro de saída uma outra string. A função a ser implementada deve armazenar na string de saída a string de entrada com todas as letras maiúsculas convertidas em minúsculas e vice-versa. A função `main` deve ler uma string e exibir na tela a string computada pela função. Exemplo:

   ```
   Informe uma frase:
   Estudos de linguagem de programacao
   String resultante:
   eSTUDOS DE LINGUAGEM DE PROGRAMACAO
   ```

3. Um palíndromo é uma palavra/frase que pode ser lida tanto da esquerda para a direita quanto da direita para a esquerda. Implemente uma função que receba como parâmetro de entrada uma string e retorne verdadeiro caso ela seja um palíndromo ou falso caso contrário. Considere as duas versões do problema:
   (a) Espaços em branco são considerados como parte da string:
   `osso`: é palíndromo
   `subi␣no␣onibus`: não é palíndromo
   (b) Espaços em branco não são considerados como parte da string:
   `subi␣no␣onibus`: é palíndromo
   `subi␣␣␣no␣␣␣onibus`: é palíndromo
   A função `main` deve ler uma string e exibir na tela uma mensagem informando se a string é um palíndromo ou não utilizando a função implementada.
4. Implemente uma função que receba como parâmetro de entrada uma string e como parâmetro de saída um vetor de inteiros de 26 posições. A função a ser implementada deve armazenar

no vetor a contagem de cada caractere minúsculo que aparece na string: na posição 0 deve ser armazenada a quantidade de `'a'`, na posição 1 a quantidade de `'b'` e assim por diante até a posição 25, que deve armazenar a quantidade de `'z'`. A função `main` deve ler uma string e exibir na tela quantas vezes aparece cada caractere na frase utilizando a função implementada. Exemplo:

```
Informe uma frase:
estudos de linguagem de programacao
Contagem de caracteres:
a: 4
c: 1
d: 3
e: 4
g: 3
i: 1
l: 1
m: 2
n: 1
o: 3
p: 1
r: 2
s: 2
t: 1
u: 2
```

5. Implemente uma função que receba como parâmetro de entrada uma string e como parâmetro de saída uma outra string. Assumindo que na string de entrada estará o nome completo de uma pessoa (separado por um único espaço entre cada nome e sem espaços antes do primeiro ou depois do último nome), a função a ser implementada deve armazenar na string de saída as iniciais do nome seguidas de ponto e espaço. A função `main` deve ler uma string contendo um nome completo e exibir na tela a string computada pela função. Exemplo:

```
Informe um nome:
Joao Francisco da Silva
String resultante:
J. F. d. S.
```

6. Uma string `s2` é considerada uma substring de uma string `s1` se `s2` fizer parte de `s1`. Implemente uma função que receba como parâmetros de entrada duas strings de entrada, `s1` e `s2`. A função a ser implementada deve retornar verdadeiro se `s2` for uma substring de `s1` e falso caso contrário. A função `main` deve ler duas strings e exibir na tela uma mensagem informando se a segunda string lida é uma substring da primeira utilizando a função implementada. Exemplos:

```
-- Exemplo 1:
Informe a primeira string:
Estudos de linguagem de programacao
Informe a segunda string:
ling
A segunda string faz parte da primeira
-- Exemplo 2:
Informe a primeira string:
```

```
Estudos de linguagem de programacao
Informe a segunda string:
ACAO
A segunda string nao faz parte da primeira
-- Exemplo 3:
Informe a primeira string:
Estudos de linguagem de programacao
Informe a segunda string:
std
A segunda string nao faz parte da primeira
```

7. Crie um programa que realiza a separação de *strings* considerando que pode haver mais de um espaço separando cada palavra, mais de um espaço antes da primeira palavra e mais de um espaço após a última palavra.
8. Exemplo:
   - Entrada:
     ␣␣Estudos␣de␣␣LIP␣␣␣
   - Saída:
     ```
     Estudos
     de
     LIP
     ```

IV Parte 4: Tipos Derivados

11 Tipos Estruturados 161
11.1 Introdução
11.2 Definição e uso de tipos estruturados
11.3 Funções com tipos estruturados
11.4 Exercícios

11. Tipos Estruturados

As linguagens de programação mais sofisticadas apresentam mecanismos para a definição de novos tipos a partir de tipos básicos e/ou outros tipos criados previamente. O conceito de criar tipos é a base para o paradigma de programação orientada a objetos, o mais utilizado na programação de sistemas atualmente.

11.1 Introdução

Tipo estruturado, também conhecido como registro ou estrutura, é um mecanismo disponibilizado pela linguagem C++ que permite ao programador definir um novo tipo de dado através do agrupamento de dados de **diferentes tipos** em memória. Par isto, cada dado diferente possui um campo que o referencia na região de memória definida para o tipo, e pode ser manipulado pelo programa. Os campos também são chamados de membros, atributos ou propriedades.

O emprego de tipos estruturados envolve o agrupamento de informações disponibilizadas de maneira independente, porém que estão relacionadas a mesma entidade. Como exemplo, considere o programa que lê do usuário o nome e a nota de 5 alunos e imprime o nome dos alunos com nota inferior à média de todas as notas.

```
const int STRMAX = 21;

int main(){
    int n = 5, i;
    char nomes[n][STRMAX];
    float notas[n], media = 0.0;
    for(i = 0; i < n; i++){
        cout << "Insira o nome do aluno: ";
        cin.getline(nomes[i], STRMAX);
        cout << "Insira a nota do aluno: ";
        cin >> notas[i];
        cin.ignore();
        media += notas[i];
    }
    media /= n;
    for(i = 0; i < n; i++){
        if(notas[i] < media){
            cout << nomes[i]
                 << " esta abaixo da media\n";
        }
    }
    return 0;
}
```

Na solução utilizada foram utilizados dois vetores independentes, um para os nomes e outro para as notas. Embora seja estruturas diferentes, um elemento de mesmo índice em cada uma das estruturas refere-se ao mesmo aluno. Isto significa que o acesso ao nome/nota de um aluno requer que um mesmo índice seja utilizado nos dois vetores, tornando o algoritmos sujeitos a falhas. A figura 11.1 mostra a disposição dos dados em memória.

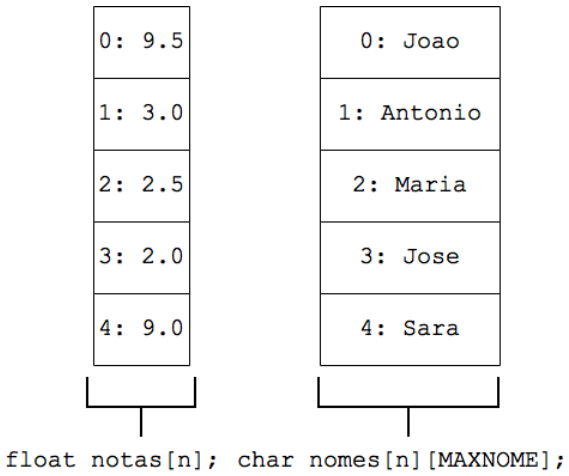


Figura 11.1: Disposição em memória dos vetores definidos para o problema.

Esta propensão a falhas se torna mais evidente quando há a necessidade de expansão das informações. Caso seja necessário mais dados para representar um aluno, mais vetores seriam necessários e, consequentemente, mais custoso se torna a manipulação destes dados. Isto é evitado com tipos estruturados pois, como os dados ficam agrupados, um único índice é utilizado para acessar todos os dados de um aluno, deixando o algoritmos menos sujeitos a falhas durante a

Figura 11.2: Disposição em memória de uma estrutura para o problema.

manipulação de mais informações. A figura 11.2 mostra como fica a disposição dos dados com o uso de tipos estruturados.

11.2 Definição e uso de tipos estruturados

A definição de tipos estruturados possui a seguinte sintaxe:

```
struct nome_tipo{
    tipo_campo_1 nome_campo_1;
    ...
    tipo_campo_N nome_campo_N;
};
```

Onde:

- `nome_tipo` é o nome do tipo estruturado. É uma boa prática definir um tipo com a primeira letra maiúscula;
- `tipo_campo` é o tipo (`char`, `int`, `float`...) do campo do tipo estruturado;
- `nome_campo` é o nome do campo do tipo estruturado;

É importante notar o emprego do ponto-e-vírgula a final da declaração, pois este é obrigatório. Além disto, as declarações de estruturas devem ser feita antes das funções.

Exemplo

Implementar cada um dos tipos estruturados a seguir:

1. Tipo estruturado para um ponto no plano 2D
2. Tipo estruturado para conta bancária, com número da agência, número da conta e nome do titular
3. Tipo estruturado para data, com dia, mês e ano
4. Tipo estruturado para funcionário, com nome, matrícula, salário, data de nascimento e data de admissão

Solução:

1. Tipo `ponto2D`:

```
struct Ponto2D{
    float x;
    float y;
};
```

2. Tipo `Conta`:

```
struct Conta{
    int agencia;
    int numero;
    char titular[STRMAX];
};
```

3. Tipo `Data`:

```
struct Data{
    int dia;
    int mes;
    int ano;
};
```

4. Tipo `Funcionaria`, assumindo que o tipo `Data` também foi definido:

```
struct Funcionario{
    char nome[STRMAX];
    int matricula;
    float salario;
    Data data_nascimento;
    Data data_admissao;
};
```

No último exemplo, verifica-se que é possível utilizar tipos estruturados como campos de outro tipo estruturado.

Uma vez definido o tipo estruturado, é possível:

- Declarar variáveis do tipo estruturado:
  `Ponto2D a1;`
  `Funcionario f2;`
- Declarar vetores de variáveis do tipo estruturado:
  `Data datas[10];`
- Utilizar o tipo estruturado em funções, tanto como parâmetros quanto como retorno.

O acesso aos campos de cada variável é feito com a indicação do campo após o nome da variável, separado por um ponto (.) (ponto).

```
struct Conta{
    int agencia;
    int numero;
    char titular[STRMAX];
};

Conta c1;
Conta contas[100];

cout <\,< ``Nome do titular: '' <\,< c1.titular <\,< endl;
cout <\,< ``Agencia do titular: '' <\,< c1.agencia <\,< endl;
cout <\,< ``Nome do titular: '' <\,< contas[0].nota <\,< endl;
```

Isto significa que o par `variável.nome_campo` é tratado como uma variável qualquer do tipo `tipo_campo`. Por isto, as regras que se aplicam aos tipos básicos valem para os campos dos tipos estruturados. Um alerta deve ser dado ao se inicializar uma variável de um tipo estruturado, pois cada campo deve ser inicializado individualmente, após a variável ter sido declarada.

```
struct Data{
    int dia;
    int mes;
    int ano;
};

Data d;
d.dia = 10;
d.mes = 5;
d.ano = 2017;
```

Como exemplo do emprego de estruturas, considere o problema anterior que lê do usuário o nome e a nota de 5 alunos e imprime o nome dos alunos com nota inferior à média de todas as notas. Uma possibilidade de programa com o uso de estruturas é dada a seguir:

```
const int STRMAX = 21;

struct Aluno{
    char nome[STRMAX];
    float nota;
};

int main(){
    int n = 5, i;
    Aluno alunos[n];
    float media = 0.0;
    for(i = 0; i < n; i++){
        cout << "Insira o nome do aluno: ";
        cin.getline(alunos[i].nome, STRMAX);
        cout << "Insira a nota do aluno: ";
        cin >> alunos[i].nota;
        cin.ignore();
        media += alunos[i].nota;
    }
    media /= n;

    for(i = 0; i < n; i++){
        if(alunos[i].nota < media){
            cout << alunos[i].nome
                 << " esta abaixo da media\n";
        }
    }
    return 0;
}
```

11.3 Funções com tipos estruturados

Variáveis de tipos estruturados podem ser utilizadas em funções como uma variável qualquer. Isto significa que elas podem ser utilizadas tanto como retorno quanto como parâmetro de uma função. Além disto, também ser passadas por valor ou por referência. Para isto, é importante lembrar que o tipo da variável ou do vetor associado é o nome dado a estrutura.

Como exemplo de aplicação, considere uma estrutura que representa um número complexo $z = a + bi$:

```
struct complex {
        float a;
        float b;
}
```

Uma possível implementação de uma função que calcula o produto de dois números complexos `produto` é dada a seguir:

```
complex produto(complex z1, complex z2) {
        complex p;
        p.a = z1.a*z2.a - z1.b*z2.b;
        p.b = z1.a*z2.b + z1.b*z2.a;
        return p;
}
```

É possível implementar a mesma função utilizando um parâmetro por referência como a saída de uma função.

```
void produto(complex z1, complex z2, complex &p) {
        p.a = z1.a*z2.a - z1.b*z2.b;
        p.b = z1.a*z2.b + z1.b*z2.a;
}
```

Neste caso, é importante mencionar que, para alterar o valor de um parâmetro por referência, é preciso alterar os campos associados. O emprego de uma variável como entrada por referência em uma chamada de função é semelhante ao que é feito com uma variável de tipo básico.

```
int main(){
        complex z1, z2, z3;
        z1.a = 3.5;
        z1.b = -2;
        z2.a = 4;
        z2.b = 0.5;
        produto(z1, z2, z3);
        cout << z3.a << " " << z3.b << endl;
}
```

É importante notar que a impressão, bem como a leitura de um tipo complexo também envolve a escrita/leitura de cada campo da estrutura.

Para funções com vetores e matrizes de tipos estruturados, a ideia é semelhante a empregada na definição destas estruturas de tipos básicos.

```
void copia(complex[] v1, int nv1, complex []v2, int &nv2) {
        nl2 = nl1;
        for(int i = 0; i < nv1; i++) {
                v1[i] = v2[i];
        }
}
```

```
void transposta(complex[MAX][MAX] m1, int nl1, int nc2,
        complex [MAX][MAX]m2, int &nl2, int &nc2) {
        nl2 = nc1;
        nc2 = nl1;
        for(int i = 0; i < nl2; i++) {
                for(int i = 0; i < nc2; j++ {
                        m2[i][j] = m1[j][i];
                }
        }
}
```

Existem duas operações que são comuns na manipulação de vetores com tipos estruturados:

- Busca de dados;
- Ordenação de dados.

Na busca de dados, geralmente é informado um valor específico de um campo e deseja-se obter o índice referente ao elemento que contém o valor desejado. Isto é comum para tipos estruturados que possuem identificadores, campos utilizados para distinguir elementos.

O processo de ordenação é semelhante ao que foi visto em vetores de tipos básicos e de `strings`. No entanto, pode haver vários tipos de ordenação para tipos estruturados, dependendo do campo escolhido para a ordenação.

11.4 Exercícios

1. Implemente uma função que receba como parâmetro um vetor com `n` pontos 2D. A função implementada deve computar e retornar o ponto 2D mais próximo da origem, $[x,y]^t = [0,0]^t$.

   Implemente também a função `main`, de modo que o usuário possa inserir a quantidade de pontos `n`, cada uma das `n` coordenadas dos pontos e visualizar na tela as coordenadas do ponto computado pela função.
2. Defina um tipo estruturado `Data`, com os campos `dia`, `mês` e `ano`. Em seguida, implemente uma função que receba como parâmetro duas variáveis do tipo `Data` e que calcule a data cronologicamente maior (12/12/2007 é maior do que 09/03/2000).

   Implemente também a função `main`, de modo que o usuário possa inserir o dia, mês e ano de cada data e visualizar na tela os dados da data cronologicamente maior no formato DD/MM/AAAA.
3. Defina um tipo estruturado `Aluno`, com os campos `nome`, `matricula` e `nota`. Em seguida, implemente uma função que receba como parâmetro um vetor com `n` alunos e que calcule o aluno com a menor e com a maior nota, armazenando cada um em um parâmetro de saída.

   Implemente também a função `main`, de modo que o usuário possa inserir a quantidade de alunos `n`, o nome, matrícula e a nota para cada um dos `n` alunos.

   A função implementada deve ser utilizada para visualizar na tela:
   - **Os dados de todos os alunos que obtiveram a menor/maior nota** (ou seja, considere o caso em que mais de um aluno tirou a menor/maior nota)
   - Os dados de forma agrupada: primeiro os alunos com a menor nota, seguido dos alunos com a maior nota
4. Defina um tipo estruturado `Aluno`, contendo os campos `nome`, `matricula`, e `nota`. Em seguida:

(a) Implemente uma função que receba como parâmetro de entrada um vetor com `n` alunos. A função deve computar a menor e maior notas dentre todos os alunos, armazenando o resultado em dois parâmetros de saída. A função `main` deve ler do usuário o número `n` de alunos, os dados de cada um deles e utilizar o resultado da função para exibir os dados de todos os alunos que obtiveram a menor nota seguidos pelos dados de todos os alunos que obtiveram a maior nota (considerando que mais de um aluno obteve a menor/maior nota).

(b) Implemente uma função que receba como parâmetro de entrada um vetor com `n` alunos. A função deve retornar a posição no vetor do aluno com o maior nome. A função `main` deve ler do usuário o número `n` de alunos, os dados de cada um deles e utilizar o resultado da função para exibir os dados do aluno com maior nome na tela.

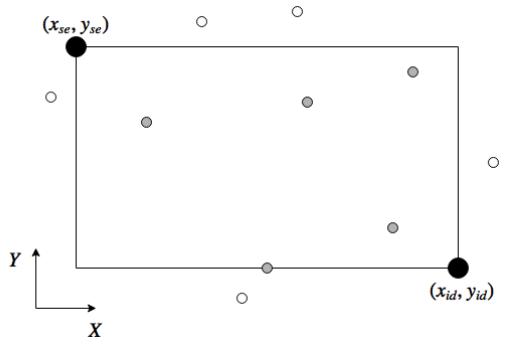
(c) Implemente uma função que receba como parâmetros de entrada um vetor com `n` alunos e uma string contendo um nome a ser buscado. A função deve retornar a posição no vetor do aluno cujo nome seja igual à string passada como parâmetro ou -1 caso um aluno com o nome passado como parâmetro não exista. Considere que todos os nomes serão compostos por letras minúsculas e apenas um espaço separando cada palavra. A função `main` deve ler do usuário o número `n` de alunos, os dados de cada um deles e um nome para busca. Por fim, ela deve exibir na tela todos os dados do aluno a ser buscado ou uma mensagem informando caso ele não tenha sido cadastrado de acordo com o resultado da chamada à função implementada.

5. Implemente uma função que receba como parâmetro de entrada duas variáveis do tipo estruturado `Data`, o qual deve ser composto pelos campos `dia`, `mes` e `ano`. A função a ser implementada deve retornar 1 caso a primeira data seja a mais antiga, 0 se as datas forem iguais e -1 caso a segunda data seja a mais antiga. A função `main` deve ler o dia, mês e ano de ambas as datas e exibir na tela uma mensagem informando qual das duas datas é mais antiga, de acordo com o valor retornado pela função.
6. Defina um tipo estruturado `Ponto2D` para representar pontos em um plano. Em seguida:

(a) Implemente uma função que receba como parâmetro de entrada um vetor com `n` pontos 2D. A função a ser implementada deve retornar o ponto 2D mais próximo da origem (ponto com coordenadas iguais a zero). A função `main` deve ler o número `n` de pontos, as coordenadas de cada ponto e exibir na tela as coordenadas do ponto mais próximo da origem de acordo com o resultado da função.

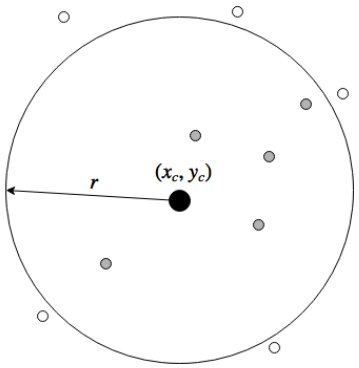
(b) Implemente uma função que receba como parâmetro de entrada um vetor com `n` pontos 2D. A função a ser implementada deve retornar o centróide dos pontos, isto é, um ponto que possui as coordenadas iguais à média de todos os pontos. A função `main` deve ler o número `n` de pontos, as coordenadas de cada ponto e exibir na tela as coordenadas do centroide de acordo com o resultado da função.

7. Um retângulo pode ser representado computacionalmente por um tipo estruturado contendo como campos as coordenadas (x_{se},y_{se}) do seu vértice superior esquerdo e as coordenadas (x_{id},y_{id}) do seu vértice inferior direito. Utilizando esta representação, implemente uma função que, recebendo como parâmetros um vetor com `n` pontos 2D e um retângulo, retorne a quantidade de pontos que estão dentro do retângulo. Considere a ilustração a seguir para desenvolver o seu raciocínio.



A função `main` deve ler a quantidade `n` de pontos, cada uma das `n` coordenadas dos pontos e as coordenadas dos vértices superior esquerdo e inferior direito do retângulo. Por fim, ela deve exibir na tela a quantidade de pontos dentro do retângulo utilizando uma chamada à função implementada.

8. Um círculo pode ser representado computacionalmente por um tipo estruturado contendo como campos as coordenadas do seu centro (x_c, y_c) e o seu raio r. Utilizando esta representação, implemente uma função que, recebendo como parâmetros de entrada um vetor com `n` pontos 2D e um círculo, retorne a quantidade de pontos que estão dentro do círculo. Considere a ilustração a seguir para desenvolver o seu raciocínio.



A função `main` deve ler a quantidade `n` de pontos, cada uma das `n` coordenadas dos pontos, as coordenadas do centro do círculo e o seu raio. Por fim, ela deve exibir na tela a quantidade de pontos que estão dentro do círculo utilizando uma chamada à função implementada.

9. Defina um tipo estruturado `Funcionario`, contendo os campos `nome`, `matricula`, e `salario`. Em seguida:
   (a) Implemente uma função que receba como parâmetro de entrada e saída um vetor com `n` funcionários. A função deve realizar a ordenação do vetor de funcionários em ordem decrescente de salário. A função `main` deve ler o número `n` de funcionários, os dados de cada um deles e exibir o vetor ordenado conforme o resultado da função.
   (b) Implemente uma função que receba como parâmetro de entrada e saída um vetor com `n` funcionários. A função deve realizar a ordenação do vetor de funcionários em ordem alfabética do nome, utilizando a função `strcmp`. A função `main` deve ler o número `n` de funcionários, os dados de cada um deles e exibir o vetor ordenado conforme o resultado da função.

V

Tópicos Especiais

12 Números Aleatórios 173
12.1 Introdução
12.2 Exercícios

13 Arquivos 177
13.1 Introdução
13.2 Operações com Arquivos
13.3 Exercícios

12. Números Aleatórios

Mecanismos de entrada de dados são comuns em programas que necessitam de interação com o usuário. Porém, em alguns casos, isto não é desejado. Para eles, qualquer mecanismo para gerar valores de entrada sem a intervenção do usuário é valido. Preferencialmente deseja-se que estas entradas sejam distintas e sem padrão aparente, para evitar a dedução do comportamento prévio do programa. Neste caso, é possível recorrer a mecanismos de geração numérica de números aleatórios como solução.

12.1 Introdução

Frequentemente, desejamos que o computador gere um número aleatório. Exemplos programas que utilizam esta noção: preencher um vetor/matriz com números aleatórios ou sortear um elemento de um vetor/matriz aleatoriamente. Esta necessidade é ainda maior com o uso de vetores e matrizes de tamanhos elevados, pois elimina a necessidade do usuário digitar vários números para testar os algoritmos implementados.

A linguagem C++ oferece funções para geração de números aleatórios disponíveis na biblioteca `cstdlib`. Como a geração de números aleatórios depende de um número chamado de *raíz*, será utilizada também a biblioteca `ctime`. Para se trabalhar com números aleatórios, são necessárias chamadas a duas funções:

- `void srand(int raiz); //biblioteca cstdlib`
- `int rand(); //biblioteca cstdlib`

A primeira função inicializa o gerador de números aleatórios com uma raiz.

```
srand(5);
```

A segunda função deve ser chamada cada vez que um número aleatório precisar ser gerado.

```
int n = rand();
```

Observe que, em relação à raiz do gerador de números aleatórios, a chamada à função `srand` não é obrigatória para o programa. Entretanto, com ela, é possível impedir que o programa gere a mesma sequência de úmeros em diferentes execuções do programa. Para isto basta utilizar a hora atual do sistema como raiz. Isto pode ser feito utilizando a função `time` da biblioteca `ctime`.

```
srand(time(0));
```

Exemplo

Implemente um programa gera `n` números inteiros aleatórios e os exibe na tela

Solução:

```
#include <cstdlib>
#include <ctime>

int main(){
    srand(time(0)); //inicializa raiz
    int i, n;
    cin >> n;
    for(i = 0; i < n; i++){
        cout << rand() << endl;
    }
    return 0;
}
```

A função `rand()` gera números aleatórios entre 0 e `RAND_MAX` (constante definida na biblioteca, valendo igual acima de 32767). Isto significa que os valores comumente extrapolam os limites desejados de valores. Para realizar a geração de números inteiros aleatórios em um intervalo de interesse, é necessário utilizar a função `rand`, usar o operador `%` para limitar a quantidade de valores gerados e somar o resultado com um valor constante para "deslocar" o intervalo de valores gerados. Isto pode ser resumido no seguinte mecanismo:

1. Geração de números inteiros aleatórios no intervalo $[0, \text{RAND\_MAX}]$:
   `int x = rand();`
2. Geração de números inteiros aleatórios no intervalo $[0, q-1]$:
   `int x = rand() % q;`
3. Geração de números inteiros aleatórios no intervalo $[p, q]$:
   `int x = (rand() % (q-p+1)) + p;`

Exemplo

Implemente uma função que gere números inteiros aleatórios em um intervalo qualquer delimitado por $[p, q]$

Solução:

```c
int gera_valor_inteiro(int p, int q){
    return (rand() % (q-p+1)) + p;
}
```

É importante mencionar que a função `rand` é a única disponível e retorna um inteiro. A geração de números reais aleatórios envolve utilizar a função `rand` dividida pelo valor `RAND_MAX`, multiplicar o resultado por um fator de escala e somar o resultado com um valor constante para "deslocar" o intervalo de valores gerados. Isto pode ser visto no seguinte mecanismo:

1. Geração de números reais aleatórios no intervalo $[0.0, 1.0]$:
   `float x = rand()/float(RAND_MAX);`
2. Geração de números reais aleatórios no intervalo $[0.0, q]$:
   `float x = q*rand()/float(RAND_MAX);`
3. Geração de números reais aleatórios no intervalo $[p, q]$:
   `float x = (q-p)*rand()/float(RAND_MAX) + p;`

Exemplo

Implemente uma função que gere números reais aleatórios em um intervalo qualquer delimitado por $[p, q]$.

Solução:

```c
float gera_valor_real(float p, float q){
    return (q-p)*rand()/float(RAND_MAX) + p;
}
```

No caso de caracteres, é preciso lembrar que um caractere também é um número inteiro. Portanto, basta gerar um número no intervalo $[0, 255]$ e convertê-lo com o operador de molde (*typecast*) para `char`. No entanto, vários destes caracteres não são visíveis.

Exemplo

1. Implemente uma função que preencha uma matriz de tamanho `n` $\times$ `n` com números inteiros aleatórios no intervalo $[-5, 5]$;
2. Implemente uma função que preencha um vetor de tamanho `n` com números reais aleatórios no intervalo $[1.0, 10.0]$.

Solução:

1.
```
void insere_aleatorios_matriz(int mat[][MAX], int n){
    int i, j;
    for(i = 0; i < n; i++){
        for(j = 0; j < n; j++){
            mat[i][j] = gera_valor_inteiro(-5,5);
        }
    }
}
```

2.
```
void insere_aleatorios_vetor(float v[], int n){
    int i;
    for(i = 0; i < n; i++){
        v[i] = gera_valor_real(1,10);
    }
}
```

12.2 Exercícios

1. Implemente uma função que receba como parâmetro de entrada uma matriz $2 \times$ `n`. Inicialmente, a sua função deve zerar a matriz e após isso, gerar `n` números inteiros aleatórios no intervalo $[1,6]$. A função deve inserir os números gerados que forem pares na primeira linha da matriz e os números ímpares na segunda. A função `main` do seu programa deve ler o número `n` e exibir na tela a matriz $2 \times$ `n` resultante utilizando a função implementada.

13. Arquivos

O armazenamento de dados para uso posterior é importante em sistemas de persistência. Isto garante a integridade dos dados manipulados, bem como evita a reinserção de informações, especialmente com dados grandes. Geralmente isto é feito por meio de arquivos. Estes são estruturas de armazenamento binário em disco rígido ou memória flash.

13.1 Introdução

A maior parte de programas de computador necessita de um mecanismo de armazenamento de dados. Até o momento, os programas implementados na disciplina não possuem este mecanismo Como exemplo, para um programa que lê uma matriz de 256 $\times$ 256, o usuário precisa digitar 256 $\times$ 256 números, pois o dado é perdido após o encerramento do programa. Consequentemente, a entrada de dados precisa ser repetida.

Uma solução para este tipo de problemas é o emprego de arquivos, que são recursos para armazenamento de dados que **dados da memória** em **disco rígido** de computador ou memórias **memórias flash** (*pendrives* e cartões de memória, por exemplo). A abertura de arquivos para leitura/escrita é um mecanismo disponibilizado pelas linguagens de programação para garantir que, em uma próxima execução do programa, este possa "continuar de onde parou". Costuma ser vistos como outra forma de entrada e saída em programas. Até agora, a forma usada para a recepção de exibição de dados até então era teclado/tela. É possível relacionar as operações entre a leitura e escrita padrão e de arquivos da seguinte forma:

- Ler do arquivo equivale a ler do teclado;
- Escrever no arquivo equivale a escrever na tela;

Neste material, arquivos são utilizados por meio das funções da biblioteca. Prioritariamente, a manipulação será feita apenas com arquivos de texto, que é o formato legível por humanos. Existem arquivos em formato binário, mas que não serão vistos. São exemplos de arquivos documentos de texto (ex.: Word), bancos de dados (ex.: SIGAA), imagens (ex.: Instagram), entre outros.

13.2 Operações com Arquivos

A manipulação de arquivos para leitura e escrita exige operações particulares. No entanto, existem 5 operações básicas com arquivos:

1. Abrir arquivo;
2. Verificar se o arquivo foi aberto corretamente;
3. Escrever/ler um arquivo;
4. Fechar arquivo.

Para abrir um arquivo, é preciso associar arquivo a uma variável. Inicialmente, é preciso criar variável do tipo `fstream`. Posteriormente, chamar a função `open`, que se encontra dentro da variável (deve ser usado o "." entre a variável e a função). ESta função recebe como parâmetro uma *string* com o nome do arquivo a ser aberto.

```
int main(){
    fstream arq;
    arq.open("arquivo.txt");
    ... //resto do programa
```

A função `open` assume que o arquivo está no mesmo diretório que o programa gerado. Porém, é possível passar também o caminho completo para o arquivo. Para tanto, as pastas precisam estar separadas por contra-barra ('\'), e deve-se evitar o emprego de espaços. Um exemplo de caminho completo é `"c:\documentos\arquivo.txt"`.

Após a abertura do arquivo, é preciso verificar se arquivo foi aberto corretamente. Caso contrário, qualquer tentativa de abrir o arquivo é inválida. Para tanto, utiliza-se a função `is_open`, que retorna verdadeiro caso o arquivo tenha sido aberto corretamente e falso caso contrário. Uma observação importante é que a função `open` do tipo estruturado `fstream` assume que o arquivo informado existe, ou seja, a função `is_open` retorna falso se o arquivo não existir.

```
int main(){
    fstream arq;
    arq.open("arquivo.txt");
    if(!arq.is_open()){
        cout << "Erro: arquivo nao encontrado\n";
        exit(0); //sai do programa (biblioteca cstdlib)
    }
    ... //resto do programa
```

No caso de escrita de arquivos, muitas vezes é desejável que o programa crie o arquivo de saída no disco. Como o emprego de variáveis do tipo `fstream` condiciona a função `is_open` retornar falso se o arquivo não existir. É preciso utilizar outros tipos de estrutura para abrir arquivos para escrita. Em C++, é possível utilizar variáveis do tipo estruturado `ofstream` (o "o" vem de *output*) para obter tal efeito. Ao usar `open` em variáveis deste tipo, o arquivo é criado no disco, e a função `is_open` somente retornará falso se não for possível criar o arquivo. Isto pode acontecer em casos de disco cheio ou falta de permissão para o diretório indicado, por exemplo.

```
int main(){
    ofstream arq;
    arq.open("arquivo.txt");
    if(!arq.is_open()){
        cout << "Erro: arquivo nao encontrado e nao criado\n";
        exit(0); //sai do programa (biblioteca cstdlib)
    }
    ... //resto do programa
```

Para escrever em um arquivo, a variável do tipo arquivo deve ser usada exatamente igual ao `cout`, com o operador de inserção < <. Esta operação só deve ser feita após o arquivo ter sido aberto corretamente.

```
int main(){
    //... abre arquivo corretamente
    char nome[] = "bruno";
    arq << "ola mundo" << endl;
    arq << "meu nome e " << nome << endl;
    ... //resto do programa
```

Para a leitura de um arquivo, a variável do tipo arquivo deve ser usada exatamente igual ao `cin` (com o operador de extração > >). Assim como ocorrena escrita em um arquivo, este operador deve ser usado após o arquivo ter sido aberto corretamente. Caso haja a necessidade de obter um texto com espaços, é possível utilizar comandos análogos ao `cin.getline` e `cin.ignore`, porém substituindo o `cin` pelo nome da variável do tipo arquivo. Para este comandos, as regras associadas ao `cin.getline` e `cin.ignore` também se aplicam. Como exemplo, considere a leitura de um arquivo formado por dois números inteiros seguidos de uma string de tamanho máximo igual a 50.

```
int main(){
    //abre arquivo corretamente
    int x, y;
    char nome[STRMAX];
    arq >> x >> y; //assume dois inteiros no arquivo...
    arq.ignore();
    arq.getline(nome, 51); //...seguidos por string
    ... //resto do programa
```

É importante notar que tanto na leitura quanto na escrita de um arquivo, é necessário conhecer a estrutura/especificação do arquivo antes. Isto acontece porque alguns arquivos possuem um formato próprio para os dados, de modo que, caso a disposição seja alterada, o arquivo é tratado como corrompido. Como exemplo, considere a leitura o arquivo contém o número de linhas e colunas de uma matriz de números. inteiros, seguidos por cada um de seus elementos, como especificado a seguir:

2 4
8 9 −1 1
2 3 5 1

```
int main(){
    //abre arquivo corretamente
    int nl = 0, nc = 0;
    int M[MAX][MAX];
    arq >> nl >> nc; //assume dois inteiros no arquivo...
    for(int i = 0; i < nl; i++)
        for(int j = 0; j < nc; j++) {
                arq >> M[i][j];
        }
    }... //resto do programa
```

Note que esta especificação é equivalente a ler do usuário as mesmas informações:

```
Informe o nr. de linhas da matriz:
2
Informe o nr. de colunas da matriz:
4
Informe os elementos da matriz:
8 9 -1 1
2 3 5 1
```

Para o exemplo anterior, considere a escrita do dado em um arquivo, de maneira a manter a mesma especificação.

```
int main(){
    //abre arquivo corretamente
    int nl = 0, nc = 0;
    int M[MAX][MAX];
    //leitura e manipulacao dos dados
    arq << nl << nc; //assume dois inteiros no arquivo...
    for(int i = 0; i < nl; i++)
        for(int j = 0; j < nc; j++) {
                arq << M[i][j] << " ";
        }
        arq << "\n";
    }... //resto do programa
```

Para fechar o arquivo, utiliza-se a função `close`, que também se encontra dentro da variável e por isso deve ser usado o "."). Ela deve ser chamada após a variável do tipo arquivo não ser mais necessária.

```
int main(){
    ... //resto do programa
    arq.close();
```

13.3 Exercícios

1. Defina um tipo estruturado `Aluno` para armazenar o nome e a nota tirada por cada um deles. Em seguida, implemente um programa que leia um arquivo com a seguinte estrutura:
   - Na primeira linha do arquivo terá o total de alunos (nr. inteiro)

- Nas linhas seguintes do arquivo, encontra-se o nome do aluno (podendo conter espaços), seguido por uma quebra de linha, seguidos pela sua nota

Exiba o nome do aluno com a maior nota.

VI

Bibliografia

Bibliografia

[1] Ana Fernanda Gomes Ascencio e Edilene Aparecida Veneruchi de Campos. *Fundamentos da Programação de Computadores*. 2ª ed. São Paulo: Pearson, 2008.

[2] H. M. Deitel e P. J. DEitel. *C++: Como Programar*. 3ª ed. Porto Alegre: Bookman, 2001.

[3] Victorine Viviane Mizrahi. *Treinamento em Linguagem C++ (Módulo 1)*. São Paulo: Makron Books do Brasil, 1995.

[4] Victorine Viviane Mizrahi. *Treinamento em Linguagem C++ (Módulo 1)*. 2ª ed. São Paulo: Pearson, 2008.

[5] Stephen Prata. *C++ Primer Plus*. Developer's Library. Pearson Education, 2011. ISBN: 9780321776402.