C(++) para não *Computeiros*: uma introdução ao raciocínio computacional

Lásaro Camargos Paulo R. Coelho Anilton Joaquim

25 de junho de 2014

Sumário

Introdução

A habilidade de olhar para um problema complexo e extrair do mesmo um conjunto de problemas mais simples e, portanto, mais facilmente resolvíveis, é essencial aos profissionais da computação. Isto por quê na automatização de tarefas por um computador, é necessário quebrá-las em unidades mínimas, que sejam compreensíveis pela máquina. Acontece que esta habilidade é também útil à profissionais das mais diversas áreas, da construção civil à de foguetes, da medicina à psicologia, e à todas as áreas da engenharia. A esta habilidade, chamamos de raciocínio computacional.

Este material é uma tentativa de organizar um curso de raciocínio computacional, via aprendizado de técnicas básicas de programação. Nosso público alvo são os estudantes cujo tópico principal de estudo não é a Ciência da Computação ou áreas afins, embora estes também possam usá-lo como uma introdução aos estudos mais aprofundados que farão.

Este tipo de curso é tradicional nos estudos de engenharia e mesmo outras áreas e, provavelmente por razões históricas, é dado usando-se como ferramenta a linguagem de programação C. Aqui usaremos C, mas não toda ela e não somente ela. Em vez disso, usaremos parte da linguagem C, extendida com partes da linguagem C++. A esta combinação particular, chamaremos C(++).

O material é dividido em capítulos correspondendo a aproximadamente uma aula de 100 minutos. São X lições, provavelmente mais do que comporiam um curso deste tipo dando, assim, opções para que os professores usando livro variem a estrutura de seus cursos.

total?

Sugere-se que a primeira parte do livro, mais básica, seja seguida na íntegra e na ordem em que é apresentada. A segunda parte contem tópicos mais (mas não completamente) independentes. A terceira parte apresenta tópicos mais avançados, que poderão ser usadas em turmas com melhor rendimento.

Cada capítulo é iniciado com uma motivação para o tópico apresentado, seguido de seções com a teoria e exemplos. O capítulo termina, então, com exercícios propostos e problemas para serem resolvidos em laboratório.

Parte I

Básico

Capítulo 1

O computador é uma máquina burra

Computadores são máquinas *burras* que não fazem nada além de seguir instruções. Sendo assim, eles precisam de instruções precisas e detalhadas sobre o que fazer. Agora que está ciente deste fato, você está pronto para entender que quando o programa não fizer o que você quer, é por que você lhe deu instrução errada.

Para que não cometamos erros (ou pelo menos para minimizá-los), ao montar a sequência de instruções que resolvem determinado problema precisamos, antes de qualquer outra coisa, entender o problema e sermos capazes de resolvê-lo "na mão". Uma vez que tenhamos uma solução teremos o que chamamos de um *algoritmo*[1].

1.1 Algoritmo

Um algoritmo nada mais é que um conjunto de instruções para se resolver um problema. Por exemplo, para se destrancar uma porta temos o seguinte algoritmo:

- Coloque a chave na fechadura
- Gire a chave

É claro que este algoritmo está em um nível de abstração muito alto e que poderia ser muito mais detalhado. Por exemplo, não há por quê destrancar a porta se ela já estiver destrancada e não há como destrancá-la se não estiver de posse da chave. Quanto mais detalhada sua sequência de passos, mais próximo de algo inteligível ao computador ela será. Para que isto ocorra, isto é, para que o computador entenda suas intruções, além de detalhadas, eles precisam ser escritas em uma linguagem de programação.

[1] `http://www.merriam-webster.com/dictionary/algorithm`

1.2 Linguagem de Programação

De acordo com fontes altamente confiáveis[2]

> Uma linguagem de programação é uma linguagem artificial projetada para comunicar instruções a uma máquina, particularmente computadores. Linguagens de programação podem ser usadas para criar programas que controlam o comportamento da máquina e expressar algoritmos de forma precisa (não ambígua).

De forma simplificada, uma liguagem de programação é um conjunto de palavras e regras sobre como usá-las na descrição de algoritmos interpretáveis por um computador.

Existem diversas [3] linguagens de programação, tendo cada uma seus pontos fortes e fracos; neste curso, usaremos as linguagens C e C++.

1.3 A linguagem C(++)

A primeira encarnação da linguagem de programação C foi desenvolvida no fim da década de 60, tendo sido extendida, melhorada e padronizada várias vezes depois. A linguagem C++, que extende a linguagem C com diversas funcionalidades como orientação a objetos, começou a ser desenvolvida na década de 70 e, como a linguagem C, também passou por várias reformas. Como resultado, hoje temos uma linguagem C++ padronizada que pode ser usada, mesmo que com certa dificuldade, para se programar em vários sistemas operacionais de forma portável. Além disso, a linguagem C++ é um super conjunto da linguagem C. Ou seja, todo e qualquer programa em C também é um programa em C++, mesmo que o oposto não seja verdade.

Como já mencionado, neste curso usaremos primariamente a linguagem C. Contudo, uma vez que o objetivo deste curso é introduzir o uso do raciocício computacional e não aprofundar no uso de determinada linguagem, estudaremos somente os aspectos da linguagem C. Além disso, usaremos partes da linguagem C++ para simplificar o desenvolvimento dos nossos programas. Assim, nos referiremos à linguagem usada como C(++), pois nem é toda a C, e nem é somente C, mas não chega a ser C++. Mas chega de ladainha; vamos ao nosso primeiro programa em linguagem C(++)!

1.3.1 Meu Primeiro Programa

O algoritmo em linguagem C(++) [4], abaixo, descreve para o computador os passos necessários para se escrever a mensagem "Olá Mundo!" na tela do computador. Não se preocupe com os detalhes de como isso é feito agora, mas foque-se nos seguintes aspectos:

[2] `http://en.wikipedia.org/wiki/Programming_language`

[3] Uma listagem não completa mas ainda assim impressionante pode ser encontrada em `http://en.wikipedia.org/wiki/Categorical_list_of_programming_languages`

[4] Usaremos também os termos "programa" e "código" para nos referirmos a tais algoritmos.

- existem várias formas de se dizer “Olá Mundo!”, por exemplo se pode saltar antes de fazê-lo, agaixar, ou fechar os olhos. Todas estas formas são corretas, embora algumas possam lhe causar certo constrangimento quando em público.
- um código de computador deve ser entendido, além de pelos computadores, também por humanos. Sendo assim, é imprescindível que você mantenha o código organizado.
- usar acentos em um programa é fonte de dores de cabeça; melhor simplesmente ignorá-los em nosso curso.

```
#include <iostream>

using namespace std;

int main()
{
    cout << "Ola Mundo!" << endl;
    return 0;
}
```

Código 1.1: Ola Mundo!

Analisando o Código 1.1, podemos facilmente identificar a linha que contém a frase “Ola Mundo!”. Esta linha é a que efetivamente *escreve* na tela do computador. Altere esta linha para que contenha, por exemplo, seu nome em vez da palavra *Mundo*. Digamos que seu nome seja *Feissibukson*. Este programa agora escreve na tela do computador os dizeres “Ola Feissibukson!”.

Para entendermos o que as demais linhas fazem, precisamos passar para o nosso próximo problema/programa.

1.3.2 Área de um Retângulo

A área de um retângulo pode ser facilmente calculada caso você saiba o comprimento de sua base e de sua altura. Matematicamente, seja b o comprimento da base e a a altura. A função f equivalente à área do retângulo pode ser definda como: $f(a, b) = a * b$. Isto é, a função f tem dois parâmetros (a altura a e base b do retângulo) e calcula a área como sendo a multiplicação de a e b.

Em linguagem C(++) a função f pode-se escrever esta função como mostrado no Código 1.2, que analisaremos em seguida.

```
int f(int a, int b)
{
    return a * b;
}
```

Código 1.2: Área de um retângulo

A linha 1 do código define a função `f` como uma função que aceita dois parâmetros `a` e `b`. Além disso, esta função também define que cada um destes parâmetros é um número inteiro ao preceder cada parâmetro pela palavra `int`. Finalmente, esta linha

também define que o resultado da função será um número inteiro ao preceder o nome da função (`f`) pela palavra `int`. Isto quer dizer que você não pode usar esta função para calcular a área de retângulos cujos lados não sejam inteiros. Mas não se preocupe, corrigiremos esta deficiência daqui a pouco.

As linhas 2 e 4 definem o *corpo da função*, isto é, quais outras linhas são partes da função `f`. Toda função na linguagem C precisa ter definido seu começo e fim usando `{` e `}`, respectivamente.

A linha 3 do código é onde o cálculo da área é efetivamente executado: `a*b`. Além disso, esta linha define também qual é o resultado da função ao preceder o resultado da multiplicação por `return`. Como a multiplicação de dois números inteiros só pode resultar em um número inteiro, o resultado da função também é inteiro, está justificado o tipo da função ser `int`.

1.3.3 Tipos Primitivos da Linguagem C

O Código 1.2, como mencionado, tem a limitação de só calcular a área de retângulos cujos lados tenham tamanhos inteiros. Para corrigir esta deficiência, vamos alterá-lo para que aceite números reais. Em computação, números reais são também chamados de números com *pontos flutuantes* e, em linguagem C, simplesmente de `float`. Sendo assim, podemos corrigir o programa simplesmente substituindo as ocorrências da palavra `int` por `float`, resultando no Código 1.3

```
float f(float a, float b)
{
    return a * b;
}
```

Código 1.3: Área de retângulo com dimensões reais.

Pronto, a função agora *recebe* dois parâmetros do tipo `float` e *retorna* um resultado também deste tipo. Juntamente com outros tipos que serão vistos adiante no curso, `int` e `float` são chamados de tipos de dados primitivos da linguagem. Isto sugere, obviamente, que há também tipos não primitivos, e nada poderia ser mais verdade. Estes tipos, contudo, só serão vistos bem mais adiante no curso, no Capítulo ?.

Adicionar referência

1.3.4 Organização do Código

É possível facilmente perceber um padrão nos exemplos de código apresentados até agora:

- A linha definindo a função é seguida por uma linha contendo apenas um `{` que é alinhado com o início da linha acima.
- A última linha da função contém apenas um `}`, alinhado com o `{` do início da função.
- Todas as linhas entre o `{` inicial e o `}` final estão alinhas e mais avançadas em relação às chaves.

Esta organização do código serve para facilitar a leitura do código, uma vez que torna extremamente óbvio onde a função começa e termina. Esta técnica é chamada *indentação*.

Algo que faltou nestes exemplos e que também serve ao propósito de facilitar o entendimento do código são os chamados comentários. O exemplo no Código 1.4 mostra como a função `f` poderia ser comentada.

```
/*
 * A funcao a seguir calcula a area de um retangulo de base
 * b e altura a. Os parametros e resultado da funcao sao do
 * tipo float.
 */
float f(float a, float b)
{
    //Calcula e retorna a area do retangulo.
    return a * b;
}
```

Código 1.4: Área do retângulo, com comentários.

Observe que há dois tipos de comentários no código. O primeiro começa com `\*` e termina com `*\` e o segundo começa com `\\` e termina no final da linha. Todos os comentários servem somente ao programador e são completamente ignorados pelo computador. Os comentários podem ser poderosos aliados na hora de procurar por erros no seu código, uma vez que permitem desabilitar trechos do mesmo.

Finalmente, é muito importante nomear suas funções e parâmetros com nomes intuitivos. Seguindo esta orientação, escreveremos a última versão de nossa função.

```
/*
 * A funcao a seguir calcula a area de um retangulo de base
 * base e altura altura. Os parametros e resultado da funcao
 * sao do tipo float.
 */
float area_retangulo(float altura, float base)
{
    //Calcula e retorna a area do retangulo.
    return altura * base;
}
```

Código 1.5: Área do retângulo, com comentários e nomes intuitivos.

1.4 Saída de Dados

Em computação, diz-se que um programa está executando a saída de dados quando envia para "fora" do programa tais dados. Exemplos comuns de saída de dados são a escrita em arquivo, o envio de mensagens na rede ou, o mais comum, a exibição de dados na tela.

Em nossos programas, a saída de dados efetuada mais comumente será para a tela do computador. Este tipo de saída, por ser tão comum, é chamada de a saída padrão do C(++), ou simplesmente *C out*. Para enviar dados para a saída do C(++), usamos a

expressão `cout <<`, seguido do dado a ser impresso na tela. Por exemplo, para imprimir a mensagem "Olá João", simplesmente adicionamos `cout << "Ola Joao"` ao código. Observe que o simbolos `<<` funciona como uma seta dizendo para onde os dados devem, neste caso, `cout`.

É possível enviar vários tipos de dados para a saída, como veremos no decorrer do curso. No caso da tela, os dados são convertidos para sua forma textual, para que possam ser lidos pelo usuário. O computador realiza a conversão de acordo com o tipo original do dado: se o dado já for um texto, ele é simplesmente copiado para a tela; se for um número, ele é convertido para um conjunto de dígitos que o represente. Por exemplo, o trecho de programa

```
cout << "numero ";
cout << 10;
```

gera a saída `numero 10` na tela.

Observe que a palavra `numero` no programa aparece entre aspas duplas e que o numero 10. Isto ocorre por quê `numero` é um texto, e `10` é um número; todos os textos devem ser colocados entre aspas duplas, para que o computador o identifique como tal, mas o mesmo não é necessário para números. Veja bem, o número `10` é bem difernte de `"10"` pois, por exemplo, `10` pode ser somado a outro número (`10 + 32`), mas `"10"` não.

Para simplificar a saída de dados, em C(++) é possível encadear várias saídas em uma só, assim

```
cout << "numero " << 10;
```

com o mesmo efeito do código anterior. Se houver necessidade de se iniciar nova linha na "impressão" na tela, basta enviar `endl` (contração de *end line*) para a saída. Assim, o código

```
cout << "numero " << 10 << endl << "texto " << endl;
```

imprime na tela o texto

```
numero 10
texto
```

Finalmente, quando uma "chamada" a uma função é enviada para o `cout`, o que é impresso é o resultado da função. Assim,

```
cout << "sen(1)" << endl << sen(1);
```

imprime na tela o texto

```
sen(1)
0
```

Você verá vários exemplos de usos do `cout` na próxima seção.

1.5 A Função `main`

Que o computador é uma máquina burra e executa somente o que você manda você já deve ter entendido, mas como mandá-lo executar algo? Em linguagem C(++), o computador *sempre* começa a execução de um código pela função `main`, a função principal. Sendo assim, se você quer dizer ao computador que calcule a área de um retângulo, então esta ordem dever partir, de alguma forma, da função `main`.

Para um exemplo,veja o seguinte código.

```
#include <iostream>

using namespace std;

/*
 * A funcao a seguir calcula a area de um retangulo de base
 * base e altura altura. Os parametros e resultado da funcao
 * sao do tipo float.
 */
float area_retangulo(float altura, float base)
{
    //Calcula e retorna a area do retangulo.
    return altura * base;
}

int main()
{
    //Calculemos a area de alguns retangulos.
    cout << area_retangulo(3.3, 2.0) << endl;
    cout << area_retangulo(2.0, 2.0) << endl;

    //Lembre-se, todo numero inteiro tambem e um numero real.
    cout << area_retangulo(4, 2) << endl;

    return 0;
}
```

Algumas observações importantes sobre a função `main`:

1. A função `main` tem sempre um resultado do tipo inteiro e seu resultado é sempre 0 (`return 0;`)[5]

2. Função `main` é como um *highlander*: só pode haver uma! Isto é, cada programa só pode conter a definição de uma função com este nome. Aliás, a regra vale para toda e qualquer função; se não fosse assim, o computador não saberia a qual função você está se referindo em seu código.

[5] Na verdade, nem nem sempre é 0, mas por enquanto definiremos sempre assim.

3. Finalmente, a função `area_retangulo` aparece antes da fução `main` no programa. Isto deve ser verdade para todas as funções do seu programa. Isto ocorre por quê, antes de executar a função `main`, o computador precisa aprender sobre a existência das outras funções.

O código como está, includindo as duas linhas iniciais que ainda não sabem para que servem, "pronto" para ser executado no seu computador. No próximo capítulo veremos exatamente como fazer isso. Até lá, executemos o código "na mão":

- O programa começa a executar pela função `main`; somente o que está no corpo desta função é executado.
- Na linha 18 não há o que executar, pois é só um comentário.
- Na linha 19, como vimos no Código 1.1, está se dizendo para o computador escrever algo na tela. Este algo é o resultado da aplicação da função `area_retangulo` aos parâmetros 3.3 e 2.0.[6]. Para que se conheça este resultado, o programa executa a função `area_retangulo`.
- A linha 13 calcula a área do retângulo, que é então retornado para a linha 19.
- Na linha 19, a *chamada* da função é "substituída" pelo resultado, e o número 6.6 é escrito na tela do computador. Na sequência, é escrito também `endl`, que faz com que o computador salte para a próxima linha da tela do computador.
- Na linha 20 o procedimento todo é repetido, agora escrevendo o valor 4.0 na tela.
- Na linha 23 também se repete o procedimento, mas agora passando como parâmetro para a função os valores inteiros 4 e 2. Como todo número inteiro também é um número real, a função é novamente executada e o valor 8.0 é impresso na tela.

1.6 Conclusão

Os códigos apresentados neste capítulo, apesar de simples, ilustraram vários pontos importantes da programação de computadores em geral e da linguagem C em específico. Estes pontos podem ser sumarizados assim:

- Em C(++) pode-se definir funções que executem computações bem definidas e específicas.
- C(++) tem vários *tipos* de dados, como `int` (números inteiros) e `float` (números reais).
- É importante manter o código organizado, comentado e indentado. Isso facilita seu entendimento e manutenção.

[6] Observe o uso de "." como separador de casas decimais.

1.7 Exercícios

Exercício 1.1 *Escreva um função que calcule a área de um círculo. Observe que a linguagem C é baseada na língua inglesa, na qual se separa casas decimais por* `.` *e não por* `,`*. Logo,* Π *é igual 3.14 e não 3,14.*

Exercício 1.2 *Escreva uma função que calcule a área de um triângulo.*

Capítulo 2

Compilação e Execução

Para colocarmos nossos algoritmos em execução, o primeiro passo é escrevê-los, usando um editor de textos qualquer que salve arquivos em texto puro, como o notepad, vim, gedit, etc. A este arquivo com o *código* chamaremos *código fonte* ou simplesmente *fonte*. Uma vez de posse do fonte, é preciso submetê-lo a um processos com vários passos que gera, ao final, um arquivo executável ou o que chamamos, comumente, de *programa*. O processo como um todo, descrito na seção seguinte, é conhecido como *compilação*, apesar de compilação ser apenas um dos passos do processo.

2.1 O processo de compilação

A sequência de passos que compõem a compilação é a seguinte:

Código Fonte → Pré-processador → Fonte Expandido → Compilador → Arquivo Objeto → Ligador → Executável

De forma simplificada, a pré-compilação é um passo que modifica o código fonte substituindo certas "palavras chave" encontradas ao longo do texto por suas definições. Por exemplo, pode-se definir que, no programa, toda vez que o pré-processador encontrar a palavra PI, ele a substituirá por 3.141649. A utilidade da pré-compilação ficará mais clara mais adiante no curso.

Uma vez terminada a pré-compilação, acontece a compilação do seu programa. A compilação traduz o código que você escreveu para uma linguagem inteligível ao computador, salvando-o em um arquivo chamado arquivo objeto. Por exemplo, a compilação transformaria o código "Olá Mundo!" escrito acima em algo como

```
...
CALL  write(0x1,0x400623,0xe)
GIO   fd 1 "Olá Mundo!"
RET
 ...
```

Após a compilação vem a *linkedição*, o passo que junta o seu arquivo objeto a outros arquivos objetos interessantes, como por exemplo um que contenha código de

funções matemáticas, manipulação de arquivos, ou interação gráfica com o usuário.[1]

2.2 A IDE Code::Blocks

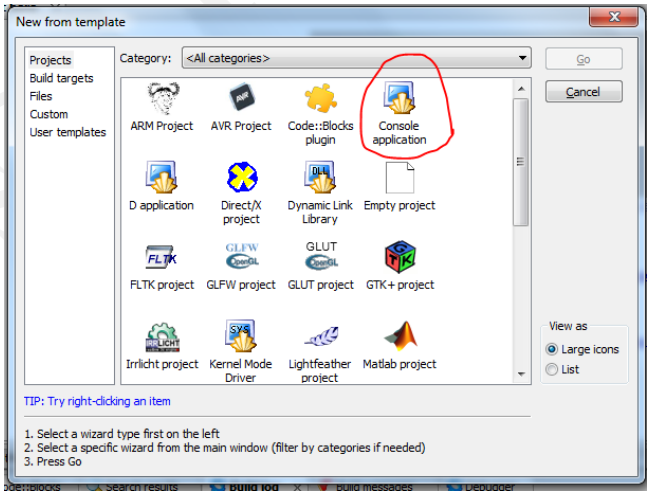
Embora a edição de um programa possa ser feita em praticamente qualquer editor de textos, há certos editores que são mais adequados a esta tarefa. Tais editores fazem, dentre outras, a colorização das palavras de seu código de forma a ajudá-lo a detectar erros e tentam alinhar automaticamente as linhas do seu código. A intenção destes editores é aumenentar sua produtividade como programador. Outros editores vão ainda mais longe e lhe permitem fazer todo o processo de compilação com um simples *click* do mouse ou apertar de uma tecla. Estes editores mais completos são conhecidos como *Integrated Development Environment*, ou simplesmente IDE.

No decorrer deste curso consideraremos que o aluno estará usando a IDE Code::Blocks, que é gratuita e com versões para Windows, Linux e OSX. Entretanto, qualquer outra IDE ou mesmo a compilação manual podem ser usados em substituição ao Code::Blocks.

2.2.1 Criando um Projeto

Para começar a programar no Code::Blocks, precisamos criar um *projeto*. Este projeto conterá seu código fonte e, no caso de uma programação mais avançada, arquivos de imagens, definições de personalização do processo de compilação, etc. Para criar um projeto no Code::Blocks, clique em **File** e, em seguida, **New**, **Project**.

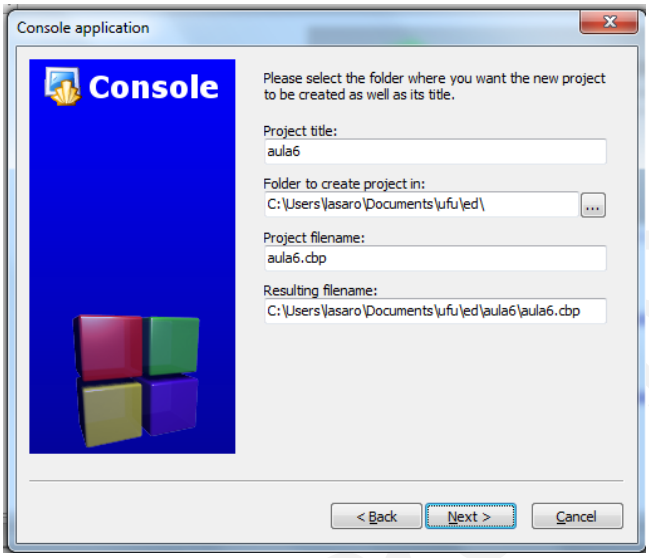
Na tela que se apresenta, você deve escolher o tipo de projeto a ser criado. Não se perca nos tipos; escolha **Console Application** e então clique em **Go**.



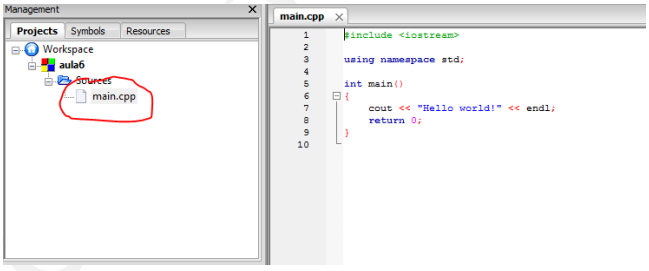
Na tela seguinte você deverá escolher a linguagem de programação usada; escolha C++ e clique em **Next** para passar para a tela onde deverá nomear o seu projeto. Em

[1] Embora a explicação dada aqui não seja estritamente correta, ela é próxima o suficiente da realidade para o escopo deste curso.

project title escreva algo como teste1; em **folder to create the project in**, clique no botao com ...e escolha uma pasta para salvar o projeto; esta pode ser, por exemplo, a pasta **Meus Documentos** ou uma pasta qualquer em um *pen drive*.[2]. Clique então **Next** e, na tela seguinte, clique em **Finish**.



Pronto, seu projeto foi criado. Agora abra o arquivo **main.cpp**, que está na pasta **sources**, dando um clique duplo no nome do arquivo. Observe que o Code::Blocks criou automaticamente um programa básico.



Finalmente, clique em **build** e então em **build and run**. Parabéns, você acaba de executar seu primeiro programa.

2.2.2 Depuração

Todo programa de tamanho considerável, e mesmo aqueles de tamanho diminuto, possuirão, ao menos em suas versões iniciais, erros. Por razões históricas, nos referimos a estes erros por *bugs*. Uma das formas de achar os bugs do seu programa é fazer

Referência para o uso de bug

[2] O importante aqui é salvar o arquivo em um lugar em que você possa voltar mais tarde para reler.

com que o computador execute seu programa passo a passo, isto é, linha a linha, e acompanhar esta execução verificando se o programa faz o que você espera.

Para experimentarmos a depuração, processo pelo qual removemos bugs, modifique a mensagem "Hello world!"do seu programa para "Olá <seu nome>!"e execute novamente o programa (**build and run**). Se o programa executou normalmente, você está no caminho certo. Agora, copie toda a linha contendo a mensagem e cole-a várias vezes, substituindo o nome em cada linha. Seu programa deve ficar como no Código 2.1

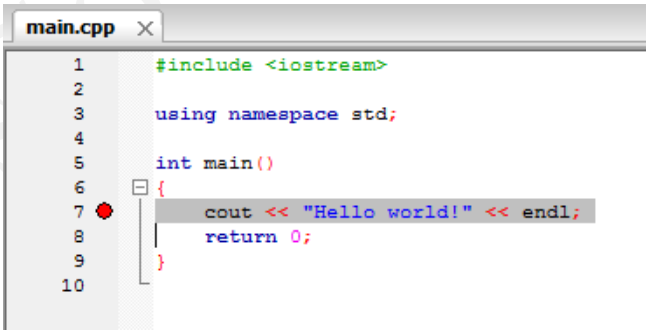
```
#include <iostream>

using namespace std;

int main()
{
    cout << "Hello Joao!" << endl;
    cout << "Hello Jose!" << endl;
    cout << "Hello Joaquim!" << endl;
    cout << "Hello Joselino!" << endl;
    cout << "Hello Asdrubal!" << endl;
    return 0;
}
```

Código 2.1: Programa com vários Hello's.

Mais uma vez, compile e execute seu programa. Se a execução foi bem sucedida, você está pronto para a depuração. Para depurar, clique ao lado direito do número 7 (sétima linha do programa), até que uma bolinha vermelha apareça, como na figura a seguir. A bolinha vermelha é, na verdade, um sinal de pare, e diz ao computador que deve, ao executar seu programa, parar ali.



Se você for adiante e executar o programa como fizemos até agora, verá que o pare não funcionou. Isto é por que o sinal é ignorado a não ser que você inicie a execução em modo de depuração. Para fazer isso, clique no menu **Debug** e então em **Start** ou, alternativamente, pressione a tecla F8. Observe que a execução parou onde você esperava. Agora, clique em **Debug** e **Next Line** ou aperte F7, no teclado, sucessivamente para ver o que acontece. Observe que cada linha é executada e então a execução pára novamente.

2.3 O Arquivo Executável

Agora que você já escreveu programas super interessantes, os compilou e executou, imagine como faria para enviar tais programas a um amigo que não tenha qualquer interesse ou aptidão em programação. A solução é simples: mandaremos a este amigo o arquivo executável do programa. Para fazê-lo, abra a pasta na qual salvou seu projeto Code::Blocks. Nesta pasta você encontrará um arquivo com extensão `.exe`; este é o arquivo executável que deveria enviar para seu amigo. Quando seu amigo executar este programa, verá exatamente a mesma coisa que você viu quando o fez. Além de ser muito útil, este procedimento é também uma ótima forma de se compartilhar vírus de computador.

2.4 Exercícios

Exercício 2.1 *Escreva o programa completo que calcula a área de vários retângulos do capítulo anterior e execute-o.*

Exercício 2.2 *Altere seu programa para usar, além da função de cálculo de área de um quadrado, as funções definidas nos exercícios do capítulo anterior.*

Capítulo 3

Variáveis, Entrada / Saída e Operadores

A todo momento precisamos representar informação do mundo a nossa volta em nossos programas. Essas informações, tais como nome, número de matrícula, nota final, temperatura, idade e outras tantas são armazenadas em entidades chamadas variáveis.

Uma variável nada mais é do que um pedaço de memória, no qual se pode ler ou escrever alguma informação. A estes pedaços de memória podemos dar nomes que nos ajude a lembrar o que exatamente está escrito ali. Por exemplo, se uma variável guarda a idade de alguém, um bom nome seria "idade", enquanto que "rataplam" ou "var13" provavelmente serão péssimas escolhas.

As alterações em uma variável resultam da interação com o usuário, isto é, quando o usuário informa valores para as mesmas em uma operação de leitura, ou da avaliação de expressões lógico-aritméticas (o tipo de cálculo nos quais o computador é especializado).

Neste capítulo veremos como criar nossas primeiras variáveis e como alterar seus valores por meio da leitura direta do teclado e da utilização de operadores.

3.1 Declaração de Variáveis

Na linguagem C, toda variável deve ser declarada (isto é, criada) no início do corpo da função que a contem. A declaração de uma variável tem pelo menos duas partes:

Nome: usado para referenciar a variável quando se precisa ler ou escrever a mesma;

Tipo: para que o computador saiba como tratar a informação, ele precisa saber de que tipo ela é, ou seja, se é um número, ou uma palavra, ou uma caractere, etc; e,

Algumas regras simples devem ser seguinda na hora de se nomear uma variável:

- o nome só pode conter os caracteres [a-z], [A-Z], [0-9] e o "_"; e,
- o nome não pode começar com números.

Quanto aos tipos usaremos, por enquanto, os seguintes:

int representando um número inteiro, como por exemplo 3, 4 e -78;

float: representando um número real, com casas decimais separadas por *ponto*, como por exemplo 3.1416 e -1.2; e

char: representando um caractere (letra, dígito, sinal de pontuação) como por exemplo 5, a, Z, ., e -.

São exemplos de declarações de variáveis válidas:

```
int nota1, nota2;
float media;
char _caractere;
```

São exemplos de declarações inválidas:

```
int 1nota, 2nota;
float #media;
char nome completo;
```

3.1.1 Atribuição e Uso

Como já dito, uma variável é um pedaço da memória do computador no qual se pode "escrever" e "ler" dados. Em vez de "escrever", contudo, no mundo da computação usamos a expressão *atribuir um valor a uma variável* para significar a mudança do valor da variável. Esta operação é executada pelo operador de atribuição =. Por exemplo, o seguinte código declara três variáveis numéricas, duas inteiras e uma real, e, em seguida, lhes atribui os valores `0`, `10` e `10.0`.

```
int inteiro1, inteiro2;
float real;

inteiro1 = 0;
inteiro2 = 10;
real = 10.0;
```

A memória do computador sempre tem algum dado, tenha ele sido colocado por você ou não, seja ele relevante ou não. Logo, para se usar o conteúdo de uma variável, é necessário ter certeza de que a mesma contém um valor que faça sentido. Isto é, algo que tenha sido atribuído pelo seu programa àquela variável, via uma operação de leitura, via uma computação qualquer, ou via uma atribuição como a do exemplo anterior.

Denominamos a primeira atribuição de um valor a uma variável de *iniciação* (ou *inicialização*). E já que qualquer variável só deve ser usada se tiver sido iniciada, o

C(++) permite que as variáveis sejam iniciadas já em sua declaração. Por exemplo, o código abaixo faz exatamente o que fazia o exemplo anterior, mas de forma mais compacta.

```
int inteiro1 = 0,
    inteiro2 = 10;
float real = 10.0;
```

Observe que se pode iniciar várias variáveis do mesmo tipo, declaradas na mesma linha, com valores distintos. Neste caso, note quebra de linha entre as declarações de `inteiro1` e `inteiro2`; ela é somente estética, mas ajuda a separar a declaração e iniciação das várias variáveis.

Agora que você viu como declarar e iniciar uma variável vem a parte fácil: usá-la. Veja como no seguinte exemplo.

```
#include <iostream>

using namespace std;

float area_circulo(float raio)
{
  float PI = 3.14,
        area;
  area = PI * raio * raio;
  return area;
}

char proxima_letra(char c1)
{
  char c2;
  c2 = c1 + 1;
  return c2;
}

int main() {
  int r1;
  float r2;
  char _c;
  _c = 'a';
  cout << "O proximo de " << _c << " eh " << proxima_letra(_c) << endl;
  r1 = 2;
  r2 = 9.7;
  cout << "r = " << r1 << ", area = " << area_circulo(r1) << endl;
  cout << "r = " << r2 << ", area = " << area_circulo(r2) << endl;
  r1 = 12;
  r2 = 0.4;
  cout << "r = " << r1 << ", area = " << area_circulo(r1) << endl;
  cout << "r = " << r2 << ", area = " << area_circulo(r2) << endl;
  return 0;
}
```

Código 3.1: cod:vars

É simples assim: para se usar uma variável, basta colocar seu nome na expressão a ser computada. Na linha 9, por exemplo, atribui-se à variável `area` o valor da multiplicação

do conteúdo da variável `PI` por `raio`, ao quadrado. Na linha 10, o resultado da função é o conteúdo da variável `area`.

3.1.2 Parâmetros são Variáveis

Nos exemplos de programas dos capítulos anteriores, você viu como o conteúdo de uma parâmetro é definido e usado. Por exemplo, os dois parâmetros da função `area_retangulo`, reproduzida abaixo, são declarados dizendo-se de que tipo eles são e quais são seus nomes. Em seguida, no corpo da função, os parâmetros são usados no cálculo da área simplesmente multiplicando-se "o nome de um pelo nome do outro"; os valores dos parâmetros são aqueles passados na chamada da função.

```
float area_retangulo(float altura, float base)
{
    //Calcula e retorna a area do retangulo.
    return altura * base;
}

int main()
{
    float area;
    area = area_retangulo(2.0, 2.0);
    cout << area;

    return 0;
}
```

Esta semelhança com a declaração e uso de variáveis não é coincidental: parâmetros não são mais do que variáveis declaradas e iniciadas de uma forma especial. Isto é, elas declaradas na definição da função e são iniciadas atribuindo-se os valores passados na invocação da função, na mesma órdem em que são passados. Isto é, se a função é invocada como `area_retangulo(1,2)`, então 1 é atribuído à variável/parâmetro `altura` e 2 à `base`. Se a função é invocada como `area_retangulo(X,y)`, então o valor da variável X, seja lá qual for é atribuído à variável/parâmetro `altura` e de y à `base`.

3.2 Entrada / Saída

Além da escrita ou impressão de dados na tela, vista no Capítulo1, uma das tarefas mais comuns em programação é a leitura de valores informados pelo usuário. A seguir veremos o comando que nos permitem executar tal tarefas.

3.2.1 Leitura

De forma semelhante ao `cout`, há um comando para leitura denominado `cin`. Este comando permite ler valores digitados pelo usuário atualizando a(s) variável(is) passada(s) para o `cin` por meio do conector >>.

A seguir temos um exemplo de entrada de dados:

```
char letra;
int idade;

cout << "Informe a letra inicial de seu nome e sua idade: ";
// a seguir eh feita a leitura
cin >> letra >> idade;
cout << "A letra eh " << letra;
cout << " e sua idade eh " << idade << endl;
```

3.2.2 Impressão

Complementando o que já vimos sobre o `cout`, vejamos como escrever o conteúdo de variáveis na tela:

- Variáveis e chamadas de funções aparecem diretamente também, e seus valores (e resultado) é que são colocados na saída.

A seguir, podemos ver alguns exemplos:

```
char letra = 'a';
int num = 2;
cout << "letra = " << letra << endl << "num = " << num << endl;
```

que gera a seguinte saída:

```
letra = a
num = 2
```

Agora que você consegue ler do teclado e escrever para a tela, veja como é fácil fazer um programa que calcule a área de retângulo cujos lados são digitados pelo usuário.

```
float area_retangulo(float altura, float base)
{
    //Calcula e retorna a area do retangulo.
    return altura * base;
}

int main()
{
    float area,
          b,
          a;
    cout << "Qual a altura do retangulo?" << endl;
    cin >> a;

    cout << "Qual a base do retangulo?" << endl;
    cin >> b;
```

```
18      area = area_retangulo(b, a);
        cout << area;
20
        return 0;
22  }
```

Formatação de Impressão

Em algumas ocasiões há necessidade de formatar a saída para, por exemplo, garantir que os dados fiquem alinhados, imprimir uma tabela, ou simplesmente por estética. A seguir veremos algumas maneiras de formatar, texto, números inteiros e reais. Para formatação de texto e números deve-se incluir a biblioteca `iomanip`.

A formatação de texto é obtida mediante definição da largura do conteúdo impresso e do alinhamento. O comando `setw(<valor>)`, define a largura do texto impresso para o valor informado como argumento, enquanto os comandos `right` e `left` definem o alinhamento para a direita e esquerda, respectivamente. O efeito do comando `setw` não é permamente. O código a seguir ilustra a utilização destes comandos:

```
#include <iostream>
2   #include <iomanip>

4   using namespace std;

6   float volume_cubo(float aresta)
    {
8       return aresta*aresta*aresta;
    }
10
    int main()
12  {
        float a, v;
14      cout << "Entre valor da aresta do cubo:" << endl;
        cin >> a;
16      v = volume_cubo(a);
        cout <<  setw(30) << left << "O volume do cubo eh: " << v << endl;
18      cout << setfill('-');
        cout <<  setw(30) << left << "O volume do cubo eh: " << v << endl;
20      cout <<  setw(30) << "O volume do cubo eh: " << setw(20) << v <<
            endl;
        cout <<  setw(30) << "O volume do cubo eh: " << setw(20) << right
            << v << endl;
22      cout <<  setw(30) << left << "O volume do cubo eh: " << v << endl;
        return 0;
24  }
```

A execução deste código produz a seguinte saída:

```
Entre valor da aresta do cubo:
2.5
```

```
O volume do cubo eh:          15.625
O volume do cubo eh: ---------15.625
O volume do cubo eh: ---------15.625--------------
O volume do cubo eh: ----------------------15.625
O volume do cubo eh: ---------15.625
```

O comando `setfill` permite definir o caractere que será usado para preencher os espaços restantes, de acordo com a largura definida com `setw`

Para formatação de números reais (`float` e `double`), o exemplo a seguir mostra alguns comandos para formatação:

```
#include <iostream>
#include <iomanip>

using namespace std;

float volume_cubo(float aresta)
{
    return aresta*aresta*aresta;
}

int main()
{
    float a, v;
    cout << "Entre valor da aresta do cubo:" << endl;
    cin >> a;
    v = volume_cubo(a);
    cout <<  "O volume do cubo eh: " << v << endl;
    cout << fixed << setprecision(2);
    cout <<  "O volume do cubo eh: " << v << endl;
    cout << fixed << setprecision(4);
    cout <<  "O volume do cubo eh: " << v << endl;
    return 0;
}
```

O comando `fixed` determina que o número de casas depois decimais será fixo, enquanto o comando `setprecision` define quantas casas decimais serão impressas. Desta maneira, para o exemplo anterior, teremos a seguinte saída:

```
Entre valor da aresta do cubo:
4
O volume do cubo eh: 64
O volume do cubo eh: 64.00
O volume do cubo eh: 64.0000
```

3.3 Operadores

Os operadores são os mecanismos por meio dos quais os computadores realizam os cálculos aritméticos e lógicos, atualizando valores das variáveis e executando as tarefas a que se destinam.

Os operadores matemáticos são os mais utilizados na maioria dos programas que serão desenvolvidos. Os principais operadores aritméticos são: $+, -, *, /$ e o $\%$, indicando, respectivamente, as operações de soma, subtração, multiplicação, divisão e resto da divisão.

Considere o exemplo a seguir:

```
#include<iostream>

using namespace std;

int main()
{
  int n, dobro_de_n;
  cout << "Entre um inteiro: ";
  cin >> n;
  dobro_de_n = 2*n;
  cout << "O dobro de " << n << " eh " << dobro_de_n << endl;
  return 0;
}
```

3.4 Exercícios

Exercício 3.1 *Escreva uma função em C que, dado uma temperatura em graus Célsius (do tipo float), retorne a temperatura equivalente em Farenheit. Escreva também a função* `main` *que leia a temperatura em Célsius do teclado, invoque a função de conversão, e imprima o resultado.*

Dado: $F = \frac{9C}{5} + 32$

Capítulo 4

Variáveis (II)

4.1 Escopo de Variáveis

No capítulo anterior estudamos como declarar e utilizar variáveis em nossos programas. Fizemos, por exemplo, um programa como o seguinte, que pede ao usuário que entre com as medidas da base e altura de um retângulo e então imprime na tela do computador a área deste retângulo.

```
1  float area_retangulo(float altura, float base)
   {
3      //Calcula e retorna a area do retangulo.
       return altura * base;
5  }

7  int main()
   {
9      float area,
             b,
11           a;
       cout << "Qual a altura do retangulo?" << endl;
13     cin >> a;

15     cout << "Qual a base do retangulo?" << endl;
       cin >> b;
17
       area = area_retangulo(b, a);
19     cout << "A area do retangulo de base " << b << " e altura "
             << a << " eh " << area << endl;
21
       return 0;
23 }
```

O que aconteceria se em vez de chamarmos as variáveis na função `main` de `a` e `b` as tivéssemos chamado de `base` e `altura`? Veja que estes são exatamente os nomes dos parâmetros da função `area_retangulo`. Melhor ainda, e se a função tivesse alterado os valores dos parâmetros?

Para descobrir as respostas a estas perguntas, faça o seguinte experimento:

- digite o programa tal qual acima em seu computador e *execute-o*.
- modifique somente a função `main` do seu programa para que fique assim

```
int main()
{
    float area,
          base,
          altura;
    cout << "Qual a altura do retangulo?" << endl;
    cin >> altura;

    cout << "Qual a base do retangulo?" << endl;
    cin >> base;

    area = area_retangulo(base, altura);
    cout << "A area do retangulo de base " << base << " e altura "
         << altura << " eh " << area << endl;

    return 0;
}
```

  e execute-o.

- Note quais as diferenças na execução.
- Finalmente, altere a função `area_retangulo` para que fique assim

```
float area_retangulo(float altura, float base)
{
    //Calcula e retorna a area do retangulo.
    altura *= base;
    return altura;
}
```

  e execute novamente o programa.

- Note se houve alguma alteração do valor da variável `altura`.

Como se pôde notar, estas mudanças não afetaram a execução do programa. Isto acontece por quê as variáveis tem escopos bem definidos em C. A variável `altura` da função `main` não é a mesma variável/parâmetro `altura` da função `area_retangulo`; cada uma só existe dentro do corpo da função em que foi declarada. Quando a função `area_retangulo` é invocada passando-se como parâmetro a variável `altura` da função `main`, o valor desta variável é *copiado* para o parâmetro `altura` da função invocada. Sendo assim, quaisquer alterações ao valor do parâmetro dentro da função afetam apenas a cópia, não o valor da variável de onde foi copiado.

A variáveis definidas até agora possuem o que chamamos *escopo local*. Isto é, elas são visiveis somente localmente à função em que foram definidas. Outro tipo de

escopo presente possível em C é o *escopo global*. Uma variável tem escopo global se for definida fora de qualquer função. Uma variáel com escopo global poderá ser acessada de (quase) qualquer parte do seu código. Para um exemplo de variável de escopo global, veja o código a seguir.

```
1  float PI = 3.1416;
2
3  float resposta = 0;
4
5  float area_retangulo(float altura, float base)
6  {
7      //Calcula e retorna a area do retangulo.
8      resposta = base * altura;
9      return resposta;
10 }
11
12 float area_circulo(float raio)
13 {
14     //Calcula e retorna a area do circulo.
15     resposta = PI * raio * raio;
16     return resposta;
17 }
18
19 int main()
20 {
21     float area,
22           base,
23           altura,
24           raio;
25
26     cout << "Qual a altura do retangulo?" << endl;
27     cin >> altura;
28
29     cout << "Qual a base do retangulo?" << endl;
30     cin >> base;
31
32     area = area_retangulo(base, altura);
33     cout << "A area do retangulo de base " << base << " e altura "
34           << altura << " eh " << area << endl;
35
36     cout << "Resposta da chamada de funcao " << resposta << endl;
37
38     cout << "Qual o raio do circulo?" << endl;
39     cin >> raio;
40
41     area = area_circlo(raio);
42     cout << "A area do circulo de raio " << raio
43     << " e PI arredondado para " << PI <<" eh " << area << endl;
44
45     cout << "Resposta da chamada de funcao " << resposta << endl;
46     return 0;
47 }
```

Observe a variável `PI`. Esta variável foi declarada fora de qualquer função e, sendo assim, é visível em qualquer delas, como demonstrado pelo seu uso na função `main` e

`area_circulo`.

Observe também que a mesma variável `area` foi utilizada mais de uma vez. Isto é comum em programação pois, com a quantidade limitada de recursos, pode não fazer sentido criar uma variável para cada novo uso. Observe que a variável `resposta` foi alterada dentro das duas funções de cálculo de área e que estas mudanças foram visíveis à função `main`.

Verifique de forma experimental (copiando e executando) que o programa acima funciona como esperado.

4.2 Faixas de Valores

Você já aprendeu que variáveis são espaços (células) da memória do computador para o qual damos nomes. Estes espaços, por serem limitados, podem armazenar uma quantidade limitada de valores. Pense, por exemplo, em quais os números, positivos e negativos, se pode representar com três dígitos: $-99, -98, \ldots, 0, 1, 2, \ldots, 998, 999$.

Tentemos descobrir qual a faixa de valores que "cabem" em uma variável `int`. Escreva um programa que declare uma variável do tipo `int`, inicie esta variável com um número (digamos, 10000), e imprima este número na tela do computador. Veja que o número é impresso na tela como deveria: `10000`.

Agora altere seu programa para que imprima 20000 e execute-o. Refaça este passo (adicionando 10000 a cada passo) até que a impressão fuja do que você esperava. Neste ponto, trabalhe com incrementos menores até determinar qual o maior número que é impresso como esperado. Fepita o processo para identificar qual o menor número que cabe em um `int`. Quais são estes valores?

Finalmente, tente identificar a faixa de valores que cabem em um `float`. Dica: os incrementos iniciais deveriam ser na faixa de milhões e não dezenas de milhares.

4.3 Exercícios

Colocar

4.4 Laboratório

Colocar

Capítulo 5

Seleção Simples

Nossos programas até agora foram extremamente simples, contendo apenas algumas pequenas funções além da `main`. Isto acontece em parte por quê nossos programas são apenas sequências diretas de comandos, sem execução condicional. Isto é, até agora não aprendemos a dizer para o computador "Se for assim, então faça assado! Senão, faça cozido!". Esta deficiência será corrigida neste capítulo.

Como exemplo de programação mais interessante, implementemos uma função que calcule as raizes de uma equação de segundo grau. Para fazê-lo, relembremos a fórmula de Bhaskara:

$x = \frac{-b \pm \sqrt{\Delta}}{2a}$, sendo $\Delta = b^2 - 4ac$.

Comecemos então definindo uma função para o cálculo do Δ.

```
1  float delta(float a, float b, float c)
   {
3      return b*b - 4*a*c;
   }
```

Para testar o cálculo do Δ precisamos da função `main`, juntamente com o restante do esqueleto de programa aprendido até agora.

```
   #include <iostream>
2
   using namespace std;
4
   float delta(float a, float b, float c)
6  {
       return b*b - 4*a*c;
8  }

10 int main()
   {
12     float a, b, c;

14     cout << "Equacao do segundo grau: axx + bx + c = 0" << endl;
       cout << "Digite o valor de a: ";
16     cin >> a;
```

```
    cout << "Digite o valor de b: ";
    cin >> b;
    cout << "Digite o valor de c: ";
    cin >> c;

    cout << "Delta: " << delta(a,b,c) << endl;
    return 0;
}
```

Agora, para cálculo das raízes! Comecemos por alterar o programa para imprimir *o número* de raízes da equação. O cálculo do número de raízes será feito na função `raizes`. A equação tem ou 0 raízes reais (se o $\Delta < 0$), ou duas raízes iguais (se $\Delta = 0$), ou duas raizes distintas (se $\Delta > 0$).

```
#include <iostream>

using namespace std;

float delta(float a, float b, float c)
{
    return b*b - 4*a*c;
}

int raizes(float a, float b, float c)
{
    float d = delta(a,b,c);
    int qtd;

    se d menor que 0
    {
        qtd = 0;
    }
    senao e se d igual a 0
    {
        qtd = 1;
    }
    senao
    {
        qtd = 2;
    }

    return qtd;
}


int main()
{
    float a, b, c;

    cout << "Equacao do segundo grau: axx + bx + c = 0" << endl;
    cout << "Digite o valor de a: ";
    cin >> a;
    cout << "Digite o valor de b: ";
    cin >> b;
    cout << "Digite o valor de c: ";
```

```
42      cin >> c;

44      cout << "Delta: " << delta(a,b,c) << endl;
        cout << "A equacao tem " << raizes(a,b,c) << " raizes.";
46      return 0;
}
```

Acontece que o computador não entende nem o `se` e nem o `menor que`. Mas então como faremos as verificações necessárias para determinar a quantidade raizes? A resposta tem duas partes.

5.1 Operadores Relacionais

As linguagens de programação provêem sempre formas de se comparar dados. Em C(++) os operadores relacionais, usados para comparar, são os seguintes:

==	igual a
!=	diferente de
>	maior que
<	menor que
>=	maior ou igual a
<=	menor ou igual a

Observe que o primeiro operador tem *dois* sinais de igual. Isto é para diferenciar este operador do operador de atribuição =, visto anteriormente. O segundo operador tem um sinal de exclamação (!) que, em linguagem C, significa *negação*. Logo, != significa não igual ou, simplesmente, diferente. Os demais operadores devem ter significados óbvios.

Usando estes operadores, podemos re-escrever a função raizes do nosso programa assim:

```
1  int raizes(float a, float b, float c)
   {
3      float d = delta(a,b,c);
       int qtd;
5
       se d < 0
7      {
           qtd = 0;
9      }
       senao e se d == 0
11     {
           qtd = 1;
13     }
       senao
15     {
           qtd = 2;
17     }

19     return qtd;
   }
```

Melhorou, mas ainda não pode ser executado pelo computador. Vamos então ao *se*.

5.2 `if-else`

Em C, testes simples (tudo o que você realmente precisa) podem ser feitos com a estrutura `if`, que tem uma das seguintes sintaxes:

- `if (` expressão lógica `)` bloco de comandos 1
- `if (` expressão lógica `)` bloco de comandos 1 `else` bloco de comandos 2

Uma *expressão lógica* é uma expressão cuja avaliação resulte em *verdadeiro* ou *falso* como, por exemplo, as expressões que usam os operadores relacionais apenas apresentados.

Um *bloco de comandos* é ou uma instrução ou um conjunto de instruções dentro de `{ }`.

Quando a expressão lógica é avaliada, se seu resultado for *verdadeiro*, então o bloco de comandos 1 será executado. Se o resultado for *falso*, o bloco de comandos 1 não será executado e o bloco 2, se existir, será executado em seu lugar.

Observe que o segundo bloco pode ser, por sua vez, outro `if`. Por exemplo, nosso programa pode ser reescrito assim:

```
#include <iostream>

using namespace std;

float delta(float a, float b, float c)
{
    return b*b - 4*a*c;
}

int raizes(float a, float b, float c)
{
    float d = delta(a,b,c);
    int qtd;

    if(d < 0)
    {
        qtd = 0;
    }
    else if(d == 0)
    {
        qtd = 1;
    }
    else
    {
        qtd = 2;
    }
```

```
28      return qtd;
}
30
int main()
32 {
    float a, b, c;
34
    cout << "Equacao do segundo grau: axx + bx + c = 0" << endl;
36  cout << "Digite o valor de a: ";
    cin >> a;
38  cout << "Digite o valor de b: ";
    cin >> b;
40  cout << "Digite o valor de c: ";
    cin >> c;
42


44  cout << "Delta: " << delta(a,b,c) << endl;
    cout << "A equacao tem " << raizes(a,b,c) << " raizes.";
46  return 0;
}
```

O último passo no desenvolvimento do nosso programa é imprimir na tela as raizes da equação, o que faremos em uma nova função: `imprime_raizes`.

```
1  #include <iostream>
   #include <math.h>
3
   using namespace std;
5
   float delta(float a, float b, float c)
7  {
       return pow(b,2) - 4*a*c;
9  }

11 int raizes(float a, float b, float c)
   {
13     float d = delta(a,b,c);
       int qtd;
15
       if(d < 0)
17     {
           qtd = 0;
19     }
       else if(d == 0)
21     {
           qtd = 1;
23     }
       else
25     {
           qtd = 2;
27     }

29     return qtd;
   }
31
   int imprime_raizes(float a, float b, float c)
```

```
33 {
       float d = delta(a,b,c);
35     int qtd;

37     if(d < 0)
       {
39         cout << "A equacao tem zero raizes reais." << endl;
       }
41     else if(d == 0)
       {
43         cout << "A equacao tem duas raizes iguais a " << -b/(2*a);
       }
45     else
       {
47         cout << "A equacao tem duas raizes iguais distintas " << endl
               <<
           "x' = " << (-b + sqrt(d))/(2*a) << endl <<
49         "x'' = " << (-b - sqrt(d))/(2*a) << endl;
       }
51
       return qtd;
53 }


55
int main()
57 {
       float a, b, c;
59
       cout << "Equacao do segundo grau: axx + bx + c = 0" << endl;
61     cout << "Digite o valor de a: ";
       cin >> a;
63     cout << "Digite o valor de b: ";
       cin >> b;
65     cout << "Digite o valor de c: ";
       cin >> c;
67


69     cout << "Delta: " << delta(a,b,c) << endl;
       cout << "A equacao tem " << raizes(a,b,c) << " raizes." << endl;
71     imprime_raizes(a,b,c);
       return 0;
73 }
```

Note que a nova função usa a função `sqrt` para calcular a raíz de Δ. Esta função é uma das muitas disponíveis na linguagem C. Para usar esta função é preciso dizer ao computador sua intenção. No nosso programa, isto é feito na linha 2, isto é,

`#include <math.h>`

em que dizemos ao computador que queremos usar as funções da biblioteca matemática da linguagem. Aproveitando a inclusão desta biblioteca também alteramos a linha 8 para usar a função `pow` para o cálculo do b^2. Várias outras funções estão disponíveis e podem ser consultadas em `http://www.cplusplus.com/reference/clibrary/cmath/`.

A seção seguinte deveria virar um capítulo e a parte if else vistos até agora se juntar ao próximo capítulo

5.3 Funções e Procedimentos

A função `imprime_raizes`, definida na seção anterior, tem por objetivo imprimir na tela as raizes da equação de segundo grau, se existirem. Esta função não tem, pela nossa definição, o objetivo de calcular a quantidade de raízes (que era o objetivo da função `raizes`). Em `imprime_raizes` não faz sentido, então, a função ter um *resultado*. Funções sem resultado são denominadas *procedimentos* e, em C, são declaradas como qualquer outra função, apenas com uma particularidade: o tipo do resultado é `void`. Antes de vermos alguns exemplos, precisamos ver a sintaxe de funções em geral, que estivemos usando nas seções e capítulos anteriores mas não havíamos definido formalmente.

tipo_resultado
identificador_função(tipo_parâmetro 1 identificador_do_parâmetro 1, . . .) bloco_de_comandos

- tipo_resultado – o tipo do valor que a função está calculando.
- identificador_função – o nome usado para invocar a função.
- tipo_parâmetro 1 – tipo do primeiro parâmetro da função.
- identificador_parâmetro 1 – identificador do primeiro parâmetro da função.
- . . . – tipo e identificador dos demais parâmetros.
- bloco_de_comandos – instruções que compõem o corpo da função.

Como mencionado, procedimentos são funções sem um resultado e, em C, são declarados como tendo resultado do tipo `void`. Em funções normais, o resultado é dado pela instrução `return`; em procedimentos, que não tem resultado, `return` não é utilizado. Além disso, o resultado de procedimentos não podem ser usados em atribuições.

Além de funções sem resultado, C permite a definição de funções sem parâmetros. Um exemplo deste tipo de função seria uma que lesse algum dado do usuário.

```
int ler_idade()
{
    int id;
    cout << "Qual sua idade? " <<endl;
    cin >> id;
    return id;
}
```

Finalmente, é importante relembrar que as funções precisam ser definidas *antes* de serem usadas. Sendo assim, você deve incluir a definição das funções antes da definição da função `main` em seu código. [1]. Ainda, relembrando a primeira aula, sobre a função `main`:

[1] É possível escrever o código de suas funções após a função `main` ou mesmo em outros arquivos. Fazer isso, contudo, requer conhecer um pouco mais do funcionamento dos compiladores do que o escopo deste livro.

1. A função `main` tem um resultado do tipo inteiro e seu resultado é sempre 0 (`return 0;`)[2].

2. Função `main` é como um *highlander*: só pode haver uma! Isto é, cada programa só pode conter a definição de uma função com este nome.

3. Finalmente, todas as funções devem ser declaradas antes da serem usadas, pois quando o computador tenta executá-la, já deve saber de sua existência.

5.4 O Tipo Primitivo `bool`

Como vimos neste capítulo, o `if` avalia uma expressão lógica para decidir-se por executar ou não um bloco de comandos. Expressões lógicas, como também já visto, são aquelas que são avaliadas em verdadeiro ou falso. Na linguagem C(++), quaisquer números inteiros podem também ser avaliados como verdadeiro ou falso, seguindo a seguinte regra:

- 0 corresponde a falso.
- qualquer outro número corresponde a verdadeiro.

Em C++, também é possível utilizar os valores `true` e `false`, que correspondem, respectivamente, a 1 e 0. Estes dois valores compõem o conjuntos dos booleanos, ou melhor, o tipo primitivo `bool`. Isto é, `true` e `false` estão para `bool` assim como -100, 10, 12, ... estão para `int`.

5.5 Exercícios

Exercício 5.1 *Muitas pessoas acreditam que um ano é bissexto se for múltiplo de 4. Contudo, a regra é um pouco mais complexa do que esta:*

- *Um ano é bissexto se for múltiplo de 4 mas não de 100, ou*
- *se for múltiplo de 100, então for múltiplo de 400.*

Escreva um programa que leia um ano, chame uma função para calcular se o ano é bissexto e imprima sim ou não de acordo.

```
#include <iostream>

using namespace std;

bool bissexto(int ano)
{
    if(ano % 4 == 0)
    {
        if(ano % 100 == 0)
        {
```

[2]Pelo menos nos programas simples que faremos.

```
            if (ano % 400 == 0)
            {
                return true;
            }
            else
            {
                return false;
            }
        }
        else
        {
            return true;
        }
    }
    else
    {
        return false;
    }
}

int main()
{
    int ano;

    cout << "Digite o ano que deseja verificar se e bissexto: ";
    cin >> ano;

    cout << "O ano " << ano;

    if(bissexto(ano))
        cout << " e bissexto" << endl;
    else
        cout << " nao e bissexto" << endl;

    return 0;
}
```

Exercício 5.2 *Este exercício é dividido em várias partes:*

1. *Escreva uma função que receba 3 números reais e retorne a média dos três números.*
2. *Escreva uma função que receba 3 números reais e retorne o menor dentre eles.*
3. *Escreva uma função que receba 3 números reais e retorne o maior dentre eles.*
4. *Escreva a função* `main` *de forma a ler três números reais, calcular a média dos mesmos e, caso a média seja menor que 0, imprima o menor dentre os três, ou, caso a média seja maior ou igual a zero, imprima o maior dentre os três. Sua função* `main` *deve usar as funções escritas nos itens anteriores para cálculo da média e impressão dos números.*

Exercício 5.3 *Escreva um programa que contenha*

1. *Uma função* `celsius_fahrenheit` *que receba uma temperatura em graus celsius e converta para fahrenheit.*

2. *Uma função* `fahrenheit_celsius` *que receba uma temperatura em fahrenheit e converta para graus celsius.*

3. *Função* `main` *que leia uma temperatura do teclado, pergunte ao usuário se a temperatura é em celsius ou fahrenheit, e imprima a temperatura convertida para a outra medida.*

Exercício 5.4 *Faça uma função denominada* ehPar *que receba um número inteiro como argumento e retorne verdadeiro se este número for par ou falso, caso contrário. A função* `main` *deve ler o número e imprimir o valor retornado pela função auxiliar.*

Exercício 5.5 *Elabore um programa com as seguinte descrição:*

- *Uma função que retorna verdadeiro se três números reais recebidos como argumentos formam um triângulo ou falso, caso contrário.*

- *Uma função que recebe três números reais como argumento representado os lados de um triângulo e retorna 0 caso os números formem um triângulo equilátero, 1 caso formem um triângulo isósceles, ou 2 caso sejam os lados de um triângulo escaleno.*

- *Por fim, a função* `main` *deve ler os 3 números que representam os lados, e caso formem um triângulo, imprimir se o triângulo formado é equilatero, isósceles ou escaleno.*

5.6 Laboratório

Refaça no computador os exercícios propostos acima.

Capítulo 6

Seleção Simples (II)

Uma vez apresentado a estrutura condicional *if-else*, veremos agora como realizar testes mais complexos utilizando operadores lógicos.

6.1 `if-else` e Operadores Lógicos

Até o ponto atual fizemos apenas testes simples dentro da condição dos nossos `if`, por exemplo:

```
...
/*
 * esta funcao retorna verdadeiro se a pessoa de sexo
 * (1=Masculino, 2=Feminino) e idade passados como argumentos
 * for maior que idade ou falso, caso contrario
 */
bool ehMaior(int sexo, int idade)
{
    if(sexo == 1)        // masculino
    {
        if(idade >= 18)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    else if(sexo == 2)  // feminino
    {
        if(idade >= 21)
        {
            return true;
        else
        {
            return false;
        }
    }
```

```
30      else            // sexo informado errado
        {
32          return false;
        }
34  }
```

Observe que na função `ehMaior` temos `if` *aninhados*, ou seja, `if` dentro de `if`. Isto porque um pessoa deve ser do sexo masculino *E* possuir idade maior ou igual a 18 anos para ser considerada maior de idade. *OU* ainda, ela pode ser do sexo feminino *E* possuir idade igual ou maior a 21 anos.

Quando esta situação ocorre, as condições podem ser combinadas em um único `if` utilizando-se operadores lógicos. Os operadores lógicos que usaremos são o E, o OU e a NÃO (negação).

Na linguagem C, eles são representados pelos símbolos a seguir:

Tabela 6.1: Operadores lógicos e seus símbolos na linguagem C.

Operador Lógico	Símbolo	Símbolo novo[1]
E	&&	and
OU	||	or
NÃO	!	not

Os operadores lógicos podem ser resumidos nas tabelas a seguir:

Tabela 6.2: NÃO lógico.

A	**!A**
V	F
F	V

Tabela 6.3: E lógico.

A	**B**	**A && B**
V	V	V
V	F	F
F	V	F
F	F	F

Voltando ao nosso exemplo, a função anterior pode ser reescrita da seguinte forma:

```
...
2 /*
   * esta funcao retorna verdadeiro se a pessoa de sexo
```

Tabela 6.4: OU lógico.

A	B	A || B
V	V	V
V	F	V
F	V	V
F	F	F

```
 * (1=Masculino, 2=Feminino) e idade passados como argumentos
 * for maior de idade ou falso, caso contrario
 */
bool ehMaior(int sexo, int idade)
{
    if((sexo == 1 && idade >=18) || (sexo == 2 && not(idade < 21))
    {
        return true;
    }
    else             // sexo errado ou idade errada
    {
        return false;
    }
}
```

Perceba que em apenas um `if` colocamos a condição completa, ou seja, "se sexo igual a 1 E idade maior ou igual a 18 OU sexo igual a 2 e idade NÃO menor do que 21 então é maior de idade".

6.2 Prioridade dos Operadores

Quando mais de um operador lógico aparece em uma expressão, a precedência pode ser expressa da seguinte maneira: primeiro o NÃO, depois o E, por último o OU. Quando houver parênteses, primeiro avalia-se o que estiver dentro dos mesmos.

Em diversas expressões e testes, diversos operadores dos vários tipos podem aparecer. A avaliação dessa expressão deve obedecer à seguinte ordem de prioridade em relação aos operadores:

1. Parênteses, incremento e decremento (`++`, `--`)
2. `not` (!)
3. Multiplicação, divisão e módulo (o que aparecer primeiro);
4. Soma e subtração;
5. Operadores relacionais (<, <=, >, >=)
6. Operadores relacionais (==, !=)

7. `and` (&&)
8. `or` (||)
9. Atribuição (=, +=, -=, *=, /=, %=)

Embora os operadores lógicos façam sentido somente para operandos `bool`, é importante relembrar que, para o computador, verdadeiro e falso são apenas formas de interpretar números na memória. Na linguagem C, qualquer número diferente de 0 é tratado como `true` e 0 é tratado como `false`. Sendo assim, é possível aplicar operadores lógicos também à números. Contudo, sempre que requisitado a representar o valor `true` como número, o computador usará o valor 1, o que faz com que nem todas as computações tenham resultados óbvios. Por exemplo, o código `cout<<(2||0);` imprime, na tela, o valor 1. Isto por quê 2 é tratado como verdadeiro e 0 como falso, e o resultado de verdadeiro OU falso é verdadeiro, que é então convertido para 1.

Da mesma forma, `!!3` é 1, pois `!3` é falso e sua negação é verdadeiro, que é 1.

6.3 Exercícios

Exercício 6.1 *Avalie o resultado de cada expressão a seguir (verdadeiro ou falso):*

- `2 < 5 && 15/3 == 5`
- `pow(3,2)- 5 > 0 && 5/2 == 3 - 4`
- `F || 20 == 18/3 != 21/3 / 2`
- `!V || 3*3/3 < 15 - 35%7`
- `!(5 != 10/2)|| V && 2 - 5 > 5 - 2 || V`
- `pow(2,4)!= 4 + 2 || 2 + 3 * 5/3%5 < 0`
- `!1+1`
- `!2+1`
- `!0+1`

Exercício 6.2 *Faça um programa que:*

1. *contenha uma função que retorna verdadeiro se um número for divisível por 3 ou 5, mas não simultaneamente pelos dois, e;*
2. *na função principal sejam lidos dois números inteiros, que deverão ser passados para a função criada, tendo seu resultado impresso.*

Exercício 6.3 *Faça uma função que receba 3 números reais (`float`) correspondentes aos lados de um triângulo e retorne `true` caso esse triângulo seja retângulo ou `false` caso não o seja. A função principal deve ler os valores dos lados do triângulo, verificar se realmente formam um triângulo e imprimir "sim"se é ou "não", caso não seja triângulo retângulo; se não formar um triângulo, imprimir "não forma".*

6.4 Laboratório

Laboratório 6.1 *Escreva um programa que implemente uma calculadora com as quatro operações +,-,* e /.*

*Sua calculadora deve ler um número real X, seguido de um inteiro representando um dos operadores definidos (1 para -, 2 para +, 3 para * e 4 para /), seguido de outro número real Y.*

Finalmente, seu programa deve escrever o resultado da operação desejada.

Revisado até aqui

Capítulo 7

Switch

Em diversas situações em programação, é necessário testar se uma determinada variável tem um dentre diversos possíveis valores. Nesta situação, embora seja possível usar vários `if`, outra solução nos é dada em linguagem C: o uso de `switch`.

7.1 `switch-case-default`

Considere o problema de transformar o mês de uma representação numérica de uma data em sua representação textual. Isto é, transformar, por exemplo, 25/12/2012 em 25 de Dezembro de 2012. Uma possível solução para este problema, em C(++), é o seguinte.

```
#include <iostream>

using namespace std;

int main()
{
    int dia, mes, ano;

    cout << "Dia? " <<endl;
    cin >> dia;

    cout << "Mes? " <<endl;
    cin >> mes;

    cout << "Ano? " <<endl;
    cin >> ano;

    cout << dia << " de ";

    if(mes == 1)
        cout << "Janeiro";
    else if(mes == 2)
        cout << "Fevereiro";
    else if(mes == 3)
        cout << "Marco";
```

```
26      else if(mes == 4)
            cout << "Abril";
28      else if(mes == 5)
            cout << "Maio";
30      else if(mes == 6)
            cout << "Junho";
32      else if(mes == 7)
            cout << "Julho";
34      else if(mes == 8)
            cout << "Agosto";
36      else if(mes == 9)
            cout << "Setembro";
38      else if(mes == 10)
            cout << "Outubro";
40      else if(mes == 11)
            cout << "Novembro";
42      else if(mes == 12)
            cout << "Dezembro";
44      else
            cout << "Hein?-zembro";
46      cout << " de " << ano << endl;
        return 0;
48  }
```

Em vez de usar vários `if` e `else-if`, uma solução melhor seria usar `switch`, criado exatamente para tratar estas situações. A sintaxe do uso do `switch` é a seguinte.
`switch` (identificador)
{
`case` valor1: bloco_comandos1
`case` valor2: bloco_comandos2
...
`case` valorN: bloco_comandosN
`default`: bloco_comandos_default
}

- identificador: Identificador da variável a ser testada
- valor1: primeiro caso a ser testado
- bloco_comandos1: bloco de comandos a ser executado caso a variável tenha valor igual a valor1
- valor2: segundo caso a ser testado
- bloco_comandos2: bloco de comandos a ser executado caso a variável tenha valor igual a valor2
- ... outros casos a serem testados
- valor n: último caso a ser testado

- bloco_comandosN: bloco de comandos a ser executado caso a variável tenha valor igual a valorN
- default: um valor especial, que sempre "casa" com o valor da variável
- bloco_comandos_default: bloco de comandos a ser executado caso a variável "case" com default.

Usando `switch-case-default`, o exemplo acima pode ser reescrito assim.

```
#include <iostream>

using namespace std;

int main()
{
    int dia, mes, ano;

    cout << "Dia? " <<endl;
    cin >> dia;

    cout << "Mes? " <<endl;
    cin >> mes;

    cout << "Ano? " <<endl;
    cin >> ano;

    cout << dia << " de ";

    switch(mes)
    {
    case 1:
        cout << "Janeiro";
    case 2:
        cout << "Fevereiro";
    case 3:
        cout << "Marco";
    case 4:
        cout << "Abril";
    case 5:
        cout << "Maio";
    case 6:
        cout << "Junho";
    case 7:
...
    case 11:
        cout << "Novembro";
    case 12:
        cout << "Dezembro";
    default:
        cout << "Hein?-zembro";
    }
    cout  << " de " << ano << endl;
    return 0;
}
```

Execute este código e digite, por exemplo, a data 1/1/2012 para ver que ele funciona "quase" corretamente. O problema, você deve ter observado, é que além de imprimir o nome do mês correto, o programa imprime também o nome de todos os meses subsequentes e o valor `default`. Isso ocorre por que, na verdade, o `switch` começa a executar o bloco correspondente ao `case` com o valor da variável mas, a partir daí, executa todos os blocos a não ser que seja instruído a fazer diferente, o que é feito via a instrução `break`.

7.2 `break`

A instrução `break` diz ao computador que pare de executar o `switch` no ponto em que é invocada.[1] Sendo assim, podemos reescrever o programa mais uma vez para obter exatamente o comportamento da versão usando `if`.

```
#include <iostream>

using namespace std;

int main()
{
    int dia, mes, ano;

    cout << "Dia? " <<endl;
    cin >> dia;

    cout << "Mes? " <<endl;
    cin >> mes;

    cout << "Ano? " <<endl;
    cin >> ano;

    cout << dia << " de ";

    switch(mes)
    {
    case 1:
        cout << "Janeiro";
        break;
    case 2:
        cout << "Fevereiro";
        break;
    case 3:
        cout << "Marco";
        break;
    case 4:
        cout << "Abril";
        break;
    case 5:
        cout << "Maio";
        break;
    case 6:
        cout << "Junho";
```

[1] Mais tarde veremos outros blocos no qual o `break` pode ser utilizado.

```
        break;
40    case 7:
        cout << "Julho";
42      break;
    case 8:
44      cout << "Agosto";
        break;
46    case 9:
        cout << "Setembro";
48      break;
    case 10:
50      cout << "Outubro";
        break;
52    case 11:
        cout << "Novembro";
54      break;
    case 12:
56      cout << "Dezembro";
        break;
58    default:
        cout << "Hein?-zembro";
60      break;
    }
62
    cout << " de " << ano << endl;
64  return 0;
}
```

7.3 Exercícios

Exercício 7.1 *Implemente uma função chamada* `menu` *que imprima o seguinte menu na tela:*

1. *Soma*
2. *Média*
3. *Menor*
4. *Maior*

Leia e que retorne o número da opção escolhida.

Implemente a função `main` *de forma a ler três números e, então, invocar a função definida acima para decidir o que fazer. O resultado da função deve ser armazenando em uma variável e seu conteúdo testado com* `switch`*. Cada opção deve invocar a função respectiva, que calculará e retornará o que se pede. A função* `main` *imprimirá então o resultado.*

```
#include <iostream>
2 #include <stdio.h>
#include <math.h>
```

```
4  using namespace std;
5
6  int menu()
7  {
8      int opcao;
9      cout << "1 Soma" << endl << "2 Media" << endl << "3 Menor" << endl
           << "4 Maior" << endl;
10     cout << "Qual sua opcao? ";
11
12     cin >> opcao;
13     return opcao;
14 }
15
16 int menor(int x, int y, int z)
17 {
18     if(x <= y && x <= z)
19         return x;
20
21     if(y <= x && y <= z)
22         return y;
23
24     return z;
25 }
26
27 int maior(int x, int y, int z)
28 {
29     if(x >= y && x >= z)
30         return x;
31     else if(y >= x && y<= z)
32         return y;
33     else
34         return z;
35 }
36
37 int soma(int x, int y, int z)
38 {
39     return x+y+z;
40 }
41
42 float media(int x, int y, int z)
43 {
44     float somatorio = soma(x,y,z);
45     return somatorio / 3.0;
46 }
47
48 int main()
49 {
50     int a, b, c;
51     int opcao;
52
53     cout << "digite tres numero inteiros" <<endl;
54
55     cin >> a >> b >> c;
56
57     opcao = menu();
58
59     switch(opcao)
```

```
60      {
            case 1:
62              cout << "A soma dos tres numeros eh " << soma(a,b,c);
                break;
64          case 2:
                cout << "A media dos tres numeros eh " << media(a,b,c);
66              break;
            case 3:
68              cout << "O menor dentre os tres numeros eh " << menor(a,b,c
                    );
                break;
70          case 4:
                cout << "O maior dentre os tres numeros eh " << maior(a,b,c
                    );
72              break;
            default:
74              cout << "Opcao invalida. Execute o programa novamente e
                    leia direito as opcoes.";
        }
76
        return 0;
78  }
```

Exercício 7.2 *Escreva um programa com uma função que receba um inteiro entre 1 e 7, inclusive, e escreva o dia correspondente da semana (1 para domingo e 7 para sábado).*

Exercício 7.3 *Escreva um programa com uma função que receba um inteiro de 1 a 12 e retorne a quantidade de dias no mês correspondente (assuma que o ano não é bisexto).*

Para este exercício, a solução mais simples envolve não colocar `break` *em alguns dos* `case`*.*

7.4 Laboratório

Implemente os exercícios de 7.1 a 7.3.

Capítulo 8

Repetição (I)

Em certas situações é necessária a repetição de um conjunto de comandos. Em situações como esta, temos duas opções: ou copiamos e colamos todo o trecho que desejamos repetir, fazendo os ajustes necessários; ou utilizamos uma saída mais inteligente por meio de comandos especiais que permitem automatizar a repetição. Neste capítulo veremos o comando de repetição `while` e alguns exemplos de seu uso.

8.1 Motivação

Suponha de você deseja fazer um programa para ler duas notas, calcular e imprimir a média de dez alunos da disciplina. A maneira menos prática de fazer isso seria:

```
...
float nota1, nota2, media;
cout << "Entre nota 1 e nota 2 do aluno 1: " << endl;
cin >> nota1 >> nota2;
media = (nota1 + nota2) / 2;
cout << "A media das notas eh " << media << endl;
cout << "Entre nota 1 e nota 2 do aluno 2: " << endl;
cin >> nota1 >> nota2;
media = (nota1 + nota2) / 2;
cout << "A media das notas eh " << media << endl;
cout << "Entre nota 1 e nota 2 do aluno 3: " << endl;
cin >> nota1 >> nota2;
media = (nota1 + nota2) / 2;
cout << "A media das notas eh " << media << endl;
cout << "Entre nota 1 e nota 2 do aluno 4: " << endl;
cin >> nota1 >> nota2;
media = (nota1 + nota2) / 2;
cout << "A media das notas eh " << media << endl;
cout << "Entre nota 1 e nota 2 do aluno 5: " << endl;
cin >> nota1 >> nota2;
media = (nota1 + nota2) / 2;
cout << "A media das notas eh " << media << endl;
cout << "Entre nota 1 e nota 2 do aluno 6: " << endl;
cin >> nota1 >> nota2;
media = (nota1 + nota2) / 2;
```

```
cout << "A media das notas eh " << media << endl;
cout << "Entre nota 1 e nota 2 do aluno 7: " << endl;
cin >> nota1 >> nota2;
media = (nota1 + nota2) / 2;
cout << "A media das notas eh " << media << endl;
cout << "Entre nota 1 e nota 2 do aluno 8: " << endl;
cin >> nota1 >> nota2;
media = (nota1 + nota2) / 2;
cout << "A media das notas eh " << media << endl;
cout << "Entre nota 1 e nota 2 do aluno 9: " << endl;
cin >> nota1 >> nota2;
media = (nota1 + nota2) / 2;
cout << "A media das notas eh " << media << endl;
cout << "Entre nota 1 e nota 2 do aluno 10: " << endl;
cin >> nota1 >> nota2;
media = (nota1 + nota2) / 2;
cout << "A media das notas eh " << media << endl;
...
```

Este código tem vários problemas. Primeiro, ele é mais propenso a erros; se, por exemplo, você resolve renomear a variável `nota2` para `notab`, você terá que fazê-lo em diversos pontos do programa, aumentando a possibilidade de esquecer algum. Em segundo lugar, o código exigiu grande retrabalho, isto é, a repetição de uma mesma tarefa por parte do programador. Finalmente, se você precisar aumentar a quantidade de repetições para, digamos, 100 alunos, terá que estender o código por páginas e páginas.

Para evitar tais problemas, para estas situações a linguagem C(++) fornece estruturas de repetições, as quais permitem repetir um determinado conjunto de comandos.

8.2 O comando `while`

Um destes comandos é o comando `while` (enquanto, em português). Sua forma geral é muito simples:

```
while (<condicao>)
{
    // bloco de comandos a ser repetido
}
```

O bloco de comandos entre as chaves será repetido **enquanto a condição dentro dos parênteses for verdadeira**.

Utilizando o `while`, o exemplo anterior pode ser reescrito de maneira bem mais prática:

```
...
float nota1, nota2, media;
int i = 1;  // valor inicial do identificador do aluno

while (i <= 10)
```

```
{
    cout << "Entre nota 1 e nota 2 do aluno: " << endl;
    cin << nota1 << nota2;
    media = (nota1 + nota2) / 2;
    cout << "A media das notas eh " << media << endl;
    i = i +1; // aumentamos o valor de i no final de cada calculo da
        media
}
...
```

Observe as seguintes modificações:

- Uma nova variável, `i`, foi criada para contabilizar o número de alunos.
- Esta variável é inicializada com o valor 1, representando o primeiro aluno.
- A condição dentro do comando de repetição será verdadeira enquanto o valor de `i` for menor ou igual a 10.
- Por este motivo, devemos incrementar o valor de `i` ao fim de cada ciclo.

Normalmente, a variável que conta a quantidade de iterações executadas, `i` no exemplo dado, é chamada de contadora. No exemplo, a variável contadora foi usada apenas para este fim, contar, e não aparece no bloco de comandos sendo repetido. Isto nem sempre é o caso, como veremos em outros exemplos. Antes, porém, vejamos uma variação do `while`.

8.3 O comando `do-while`

Se por acaso a condição verificada no `while` for inicialmente falsa, o bloco não será repetido nem mesmo uma vez. Para situações em que é preciso executar o bloco pelo menos uma vez, uma variação do comando `while` é fornecida pela linguagem C. Trata-se do comando `do-while` (faça-enquanto ou repita-enquanto, em português). Sua forma geral é dada por:

```
do
{
    \\ bloco de comandos
    \\ a ser repetido
}
while (<condicao>);
```

O mesmo exemplo anterior pode ser reescrito utilizando este comando:

```
do {
    cout << "Entre nota 1 e nota 2 do aluno : " << endl;
    cin >> nota1 >> nota2;
    i = 1+1; // aumentamos o valor de i no final de cada calculo da
        media
```

```
    media = (nota1 + nota2) / 2;
    cout << "A media das notas eh " << media << endl;
}
while (i <= 10);
```

Em comparação ao comando `while`, a única diferença existente é o fato do teste da condição ser feito após a execução do bloco de comandos que se deseja repetir. Uma implicação disto é que, em casos em que a condição é falsa logo no primeiro teste, o bloco de comandos é executado com `do-while`, mas não é executado com `while`. Isto aconteceria para a variável `i` com valor inicial de 11, por exemplo.

8.4 Mais exemplos

Considere que deseja-se somar todos os números pares entre 1 e 999. Ou fazemos uma soma com todos os valores em uma linha enorme, ou utlizamos o que apresendemos sobre comandos de repetição. Utilizando o `while`, teríamos:

```
...
int n = 2, // primeiro par maior do que 1
    soma = 0; // soma inicialmente zerada
while (n < 999)
{
    soma = soma + n;
    n = n + 2;
}
cout << "O valor da soma eh " << soma << endl;
...
```

Observe que a cada iteração o valor de soma é acrescido do próximo número par, o qual é obtido somando-se 2 ao valor de `n`.

Imagine que se deseja obter o maior entre 10 números inteiros lidos. Utilizando o `do-while`, uma possível solução seria:

```
...
int i = 0, // contador da qtde de numeros lidos
    maior,
    n;
do
{
    cout << "Entre um numero: ";
    cin >> n;
    if (i == 0) // se for o primeiro numero lido
    {           // ele sera o menor
        maior = n;
    }
    else        // a partir do segundo
    {
        if(n > maior) // atualizo o maior
        {
            maior = n;
```

```
        }
    }
    i = i + 1;
}
while (i < 10);
...
```

Neste exemplo temos uma situação especial em que, no primeiro caso (`i = 0`), o maior valor é o único valor lido. A partir do segundo número, se o número lido for maior do que o valor armazenado na variável `maior`, esta será atualizada.

Em outro exemplo, imagine que queira ler números até que leia um número maior que 100. Neste caso, o seguinte programa resolveria nosso problema.

```
...
int num;
do
{
    cout << "Entre um numero: ";
    cin >> num;
}
while (! num > 100);
...
```

Neste exemplo utilizamos `do-while` pois é necessário ler pelo menos um número. Reescreva o código utilizando `while` e veja como fica, necessariamente, mais complexo.

8.4.1 Operadores de incremento e outras construções especiais

Nos exemplos apresentados, a variável contadora foi manipulada em todas as repetições de uma forma bem comum, sendo incrementada de 1 em 1 ou de 2 em 2. Repetições tem esta característica, embora as operações aplicadas aos contadores não sejam sempre simples incrementos e decrementos. Com a finalidade de agilizar o desenvolvimento e simplificar algumas operações aritméticas mais comuns, a linguagem C(++) permite algumas construções especiais envolvendo operadores. Considere o seguinte trecho de código:

```
int a, b;
...
a = a + b;
b = b * 2;
a = a / 7;
```

Observe que nas três atribuições (indicadas pelo sinal de igualdade), as variáveis que são atualizadas também aparecem como primeiro elemento da operação aritmética à esquerda. Nestas situações, podemos reescrever as atribuições assim:

```
int a, b;
...
a += b;
b *= 2;
a /= 7;
```

As operações de incremento (aumento de uma unidade) e o decremento (diminuição de uma unidade) de uma variável são muito comuns em programação. Sendo assim, a linguagem C define dois operadores para as mesmas: ++ e −−, respectivamente. Veja o exemplo.

```
int a = 0;
a = a + 1;
cout << "a = " << a << endl;
a += 1;
cout << "a = " << a << endl;
a++;
cout << "a = " << a << endl;
a--;
cout << "a = " << a << endl;
```

O trecho acima deve imprimir os valores de `a`, ou seja, 1, 2, 3 e 2.

8.5 Exercícios

Exercício 8.1 *Diga o que será escrito na tela durante a execução do seguinte trecho de código:*

```
int a, b = 0, c = 0;
a = ++b + ++c;
cout << a << ", " << b << ", " << c << endl;
a = b++ + c++;
cout << a << ", " << b << ", " << c << endl;
a = ++b + c++;
cout << a << ", " << b << ", " << c << endl;
a = b-- + --c;
cout << a << ", " << b << ", " << c << endl;
```

Incluir exercícios mais simples como os da lista 1 do Prof. Anilton.

Faça os exercícios a seguir à mão.

Exercício 8.2 *Faça uma função que recebe um número inteiro positivo e retorna o fatorial deste número. A função principal deve ler o número do qual se deseja calcular o fatorial e imprimir o resultado.*

Exercício 8.3 *Faça uma função que recebe um número inteiro positivo e retorna* `true` *se o número for primo ou* `false`*, caso contrário. A função principal (*`main`*) deve ler o número e imprimir o resultado.*

Exercício 8.4 *Modifique o programa anterior para imprimir todos os números primos abaixo de dois milhões.*

Exercício 8.5 *Faça um programa que leia um número inteiro positivo e imprima esse número de trás pra frente. A impressão deve ser feita pela função auxiliar* `inverteNumero`*.*

8.6 Laboratório

Laboratório 8.1 *Implemente os exercícios acima no Code::Blocks.*

Laboratório 8.2 *Escreva uma função que receba dois parâmetros inteiros, x e y, $x < y$ e que imprima y pontos na tela e que a cada x pontos, imprima um acento circunflexo.*

Laboratório 8.3 *O exercício* **??** *pedia que você escrevesse uma função que gerasse um menu na tela. A função que você escreveu tinha um problema: ela aceitava qualquer valor, mesmo algum não correspondendo a nenhuma opção do menu.*

Usando `do-while`*, altere sua função para que continue exibindo o menu e lendo uma opção até que uma opção válida seja digitada pelo usuário.*

Laboratório 8.4 *Implemente uma função chamada* `pot` *que receba um número real e um inteiro, respectivamente* `base` *e* `expoente`*, como parâmetros e que calcule* $\texttt{base}^{\texttt{expoente}}$.

A função `pot`*, que você implementará, deve usar multiplicações sucessivas para calcular seu resultado (isto é, é proibido o uso da função* `pow`*).*

Laboratório 8.5 *Execute o seguinte programa em seu computador e observe o que acontece.*

```
...
int num = 10;
do
{
    cout << num;
    num *= 2;
}
while (1);
...
```

À propósito, para terminar um programa basta digitar a combinação de teclas `ctrl+c`*.*

Laboratório 8.6 *Escreva um programa que leia um numero inteiro positivo* `n` *e em seguida imprima* `n` *linhas do chamado Triângulo de Floyd:*

```
 1
 2  3
 4  5  6
 7  8  9 10
11 12 13 14 15
16 17 18 19 20 21
```

Laboratório 8.7 *Faça um programa que gera um número inteiro aleatório de 1 a 1000 em uma função denominada* `geraNumero`*. Na função principal, o usuário deve tentar acertar qual o número foi gerado. A cada tentativa o programa deverá informar se o chute é menor ou maior que o número gerado. O programa acaba quando o usuário acerta o número gerado. O programa deve informar em quantas tentativas o número foi descoberto.*

Laboratório 8.8 *Escreva um programa que receba como entrada o valor do saque realizado pelo cliente de um banco e retorne quantas notas de cada valor serão necessárias para atender ao saque com a menor quantidade de notas possível. Serão utilizadas notas de 100, 50, 20, 10, 5, 2 e 1 reais. O cálculo e impressão do resultado deve ser efetuado por uma função auxiliar denominada* `imprimeNotas`*.*

Capítulo 9

Repetição (II)

Você deve ter percebido que no uso de `do-while` e `while` quase sempre seguimos os mesmos passos:

- declarar uma variável que sirva de controle para a iteração;
- iniciar a variável de controle (e possivelmente outras);
- verificar a condição para iteração
- executar iteração
- executar incremento/decremento (mudança da variável de controle)

A linguagem C tem um comando iteração que agrega todos estes passos, chamado `for`.

9.1 `for`

Sua forma geral do comando `for` é a seguinte:

```
for(DI; C; I)
{
    \\bloco de comandos a ser repetido
}
```

O comando `for` tem três partes em sua declaração, além dos comandos a serem repetidos.

- DI – em DI variáveis podem ser **D**eclaradas e **I**niciadas. Variáveis já existentes também podem ter seus valores ajustados em DI;
- C – C define a **C**ondição necessária à execução do bloco de comandos. *Enquanto* a condição for verdadeira, o bloco será executado.

- I – comandos de modificação de variáveis, como **I**ncremento e decremento, são colocados diretamente na declaração do `for`. O comando é executado ao final de cada iteração.

A execução do `for` segue os seguintes passos:

1. Iniciação (execução de DI)
2. Avaliação (teste da condição em C)
3. Execução do bloco de comandos
4. Incremento
5. De volta ao passo 2

Considere o exemplo do capítulo anterior em que deseja-se somar todos os números pares entre 1 e 999. O código pode ser escrito, como vimos, usando `while`.

```
...
int n = 2, // primeiro par maior do que 1
    soma = 0; // soma inicialmente zerada
while (n < 999)
{
    soma = soma + n;
    n += 2;
}
cout << "O valor da soma eh " << soma << endl;
...
```

O código equivalente, usando `for` é essencialmente o mesmo.

```
...
int n, // primeiro par maior do que 1
    soma = 0; // soma inicialmente zerada
for (n = 2; n < 999; n += 2)
{
    soma = soma + n;
}
cout << "O valor da soma eh " << soma << endl;
...
```

O código, contudo, pode ser simplificado colocando-se a declaração da variável de controle no próprio `for`.

```
...
int soma = 0; // soma inicialmente zerada
for (int n = 2; n < 999; n += 2)
{
    soma = soma + n;
}
cout << "O valor da soma eh " << soma << endl;
...
```

É possível declarar e iniciar mais de uma variável no DI, mas não é possível definir novas variáveis e iniciar outras já definidas. No exemplo abaixo, errado, a variável soma sendo iniciada no `for` é diferente da variável definida antes do comando, apesar do nome ser igual.

```
...
int soma = 0; // soma inicialmente zerada
for (int n = 2, soma = 0; n < 999; n += 2)
{
    soma = soma + n;
}
cout << "O valor da soma eh " << soma << endl;
...
```

9.2 Mais Exemplos

Também do capítulo anterior, imagine o exemplo em que se deseja obter o maior entre 10 números inteiros lidos. Utilizando o `for`, uma possível solução seria:

```
...
int i, // contador da qtde de numeros lidos
    maior,
    n;

cout << "Entre um numero: ";
cin >> n;
maior = n;

for(i = 0; i < 9; i++)
{
    cout << "Entre um numero: ";
    cin >> n;
    if(n > maior) // atualizo o maior
    {
        maior = n;
    }
}
...
```

Observe que a primeira leitura aconteceu fora do `for`.

9.3 Declarações especiais

Em certas situações pode ser desejável omitir partes da declaração do `for`. Por exemplo, se as variáveis de controle já tiverem sido iniciadas ou, simplesmente, se não existirem, ou se não houver incremento a ser feito, então estas partes da declaração podem ser deixadas em branco. Por exemplo,

```
int menu()
{
    int opcao = 0;

    for( ; opcao < 1 || opcao > 4 ; )
    {
        cout << "1 Soma" << endl
             << "2 Media" << endl
             << "3 Menor" << endl
             << "4 Maior" << endl;

        cout << "Qual sua opcao? ";

        cin >> opcao;
    }
    return opcao;
}
```

Observe que embora neste exemplo tanto DI quanto I estão vazias, isso não é necessário. Isto é, qualquer das partes da declaração podem estar vazias independentemente, inclusive a segunda parte.

9.4 Alterando a repetição com o `break` e `continue`

Caso a segunda parte do comando esteja vazia, a repetição será executada *ad infinitum* ou até que seja interrompida. A interrupção de uma iteração pode ser feita usando-se o comando `break`. Veja o exemplo anterior reescrito para usar tal comando.

```
int menu()
{
    int opcao;

    for(;;)
    {
        cout << "1 Soma" << endl
             << "2 Media" << endl
             << "3 Menor" << endl
             << "4 Maior" << endl;

        cout << "Qual sua opcao? ";

        cin >> opcao;

        if(opcao > 0 && opcao < 5)
            break;
        else
            cout << "Opcao invalida" << endl << endl;
    }

    return opcao;
}
```

Outra forma de se alterar o fluxo é via o comando `continue`, que faz com que a o restante do bloco de comandos seja ignorado e, conseqüentemente, incremento e condição sejam reavaliados. Por exemplo, reescrevendo o código acima para o usar o `continue`.

```
int menu()
{
    int opcao;

    for(;;)
    {
        cout << "1 Soma" << endl
             << "2 Media" << endl
             << "3 Menor" << endl
             << "4 Maior" << endl;

        cout << "Qual sua opcao? ";

        cin >> opcao;

        if(opcao < 1 || opcao > 4)
            continue;

        cout << "A opcao escolhida foi " << opcao;
        break;
    }

    return opcao;
}
```

9.5 Exercícios

Exercício 9.1 *Refaça os execícios do capítulo anterior usando `for`.*

9.6 Laboratório

Laboratório 9.1 *Refaça o laboratório do capítulo anterior usando `for`.*

Laboratório 9.2 *Escreva uma função que receba dois parâmetros inteiros, X e Y, e imprima X linhas na tela, cada uma com Y ".". Por exemplo, se sua função for invocada com X igual 3 e Y igual 2, o resultado deveria ser o seguinte*

```
..
..
..
```

Laboratório 9.3 *Escreva uma função que receba dois parâmetros inteiros, X e Y, e imprima de forma decrescente os números de X*Y até 1, em X linhas de Y números.*

Por exemplo, se sua função for invocada com X igual 4 e Y igual 3, o resultado deveria ser o seguinte

```
12 11 10
9 8 7
6 5 4
3 2 1
```

Capítulo 10

Arranjos Unidimensionais

Tente resolver o seguinte problema:

1. ler um conjunto de 6 números inteiros
2. calcular sua média
3. imprimir todos os números maiores que a média na tela do computador

Fácil, certo? Basta declarar 6 variáveis do tipo `int`, ler seus valores, somar seus valores e dividir por 6, calculando a média. Finalmente, basta escolher aquelas variáveis com números maiores que a média e imprimí-las.

Mas e se alterássemos o problema para que, em vez de 6, precisasse ler 10 números, ou 100? Ainda assim se poderia usar o mesmo algoritmo, com 10 ou 100 variáveis em vez de 6. Mas ter muitas variáveis distintas com a mesma finalidade não é viável por duas razões:

- difícil de manter: se você precisar renomear uma variável, terá que fazê-lo em todas as variáveis;
  se precisar aumentar ou diminuir o número de variáveis, terá que apagar/copiar e renomear.
- evita reuso de código: se uma mesma operação precisa ser aplicada a cada variável, o mesmo código deve ser reescrito para cada variável, dificultado o reuso de código.

A solução para esta situação é o uso de vetores (ou variáveis indexadas, ou arranjo, ou *array*).

10.1 Vetores

Continuando com nossa analogia da memória do computador como uma planilha eletrônica, um vetor é uma variável que nomeia diversas células contíguas da memória do

computador. Isto é, de certa forma, enquanto uma variável `int` corresponde a uma área da memória que cabe 1 inteiro, um vetor de 10 `int` é uma variável que cabe 10 inteiros.

A sintaxe para a declaração estática de vetores é bem simples (em capítulos futuros veremos como declarar vetores dinâmicos, isto é, que podem variar seus tamanhos).

```
tipo identificador[tamanho];
```

Onde

- tipo – é o tipo do dado a ser armazenado em cada posição do vetor;
- identificador – é o nome do vetor;
- tamanho – é a quantidade de células no vetor;

O acesso a uma célula, para leitura ou atribuição, é feito usando-se o identificador seguido pela posição a ser acessada, entre colchetes (`[]`). Por exemplo, `x[3] = 0` atribui o valor 0 à posição de índice 3 do vetor `x`. Algo importante a ser observado aqui é que a primeira posição de um vetor de tamanho n tem índice 0 e a última tem índice $n - 1$.

O exemplo a seguir resolve o problema apresentado na seção anterior usando vetores.

```
#include<iostream>

using namespace std;

#define TAMANHO 10

int main()
{
  int num[TAMANHO];
  int soma = 0;
  int media;

  for(int i = 0; i < TAMANHO; i++)
  {
      cout << "Digite o " << i << "-esimo valor: ";
      cin >> num[i];
      soma += num[i];
  }

  media = soma/TAMANHO;

  cout << "Os valores acima da media " << media << " sao" << endl;
  for(int i = 0; i < TAMANHO; i++)
  {
     if(num[i] > media)
         cout << num[i] << endl;
  }

  return 0;
}
```

Observe a definição e o uso da palavra `TAMANHO` no programa. Uma vez definido que `TAMANHO` tem o valor 10, o computador substituirá toda ocorrência desta palavra no programa pelo valor correspondente, *antes* da compilação.

Uma variação interessante do problema calcula a média apenas de números positivos e a entrada de um número negativo serve para finalizar o fim da entrada. O código seguinte resolve o problema.

```
#include<iostream>

using namespace std;

#define TAMANHO 100

int main()
{
  int num[TAMANHO];
  int soma = 0;
  int media;
  int contador = 0;

  for(int i = 0; i < TAMANHO; i++)
  {
      cout << "Digite o " << i << "-esimo valor: ";
      cin >> num[i];

      if(num[i] >= 0)
      {
         soma += num[i];
         contador++;
      }
      else
      {
         break;
      }
  }

  media = soma/contador;

  cout << "Os valores acima da media " << media << " sao" << endl;
  for(int i = 0; i < contador; i++)
  {
     if(num[i] > media)
         cout << num[i] << endl;
  }

  return 0;
}
```

Observe como a variável `contador` é usada para contar a quantidade de números válidos lidos e como ela é usada como limitante da varredura do vetor no segundo `for`. Observe também como o `break` é utilizado para interromper o `for`.

10.2 Exercícios

Exercício 10.1 *Escreva um programa que leia 10 números e imprima o menor e o maior entre eles (não é necessário usar vetores aqui).*

Exercício 10.2 *Escreva um programa que leia 10 números e calcule e imprima a média e desvio padrão dos mesmos. Lembre-se que o desvio padrão é definido como a raiz quadrada dos quadrados das diferenças dos valores para a média dos valores.*

Exercício 10.3 *Escreva um programa que leia 11 números reais e imprima os valores dos 10 primeiros multiplicados pelo 11-ésimo. Por exemplo, se os valores digitados foram 1, 2, 3 ,4 ,5 ,6, 7, 8, 9, 10 e 1.1, então seu programa deve imprimir 1.1, 2.2, 3.3, 4.4, 5.5, 6.6, 7.7, 8.8, 9.9 e 11.0.*

10.3 Laboratório

Laboratório 10.1 *Escreva um programa que leia um número inteiro* `n` *e então leia* `n` *números reais. Em seguida, seu programa deve imprimir a média e desvio padrão dos números lidos. Seu programa deve* ***ignorar*** *números negativos.*

Laboratório 10.2 *Escreva um programa que leia* `n` *números reais e imprima seus valores na tela. Em seguida, leia mais um número real* `x` *e imprima o valor dos* `n` *números multiplicados* `x`*.*

Laboratório 10.3 *Escreva um programa que leia* `n` *booleanos e imprima seus valores na tela. Em seguida, imprima todos os booleanos invertidos, na linha seguinte.*

Capítulo 11

Caracteres, Vetores de Caracteres e Strings

11.1 Representação de caracteres

Além dos tipos de dados numéricos com os quais temos trabalhado até agora, outro tipo de dado é muito importante no desenvolvimento de programas de computador, o tipo caractere. Estes tipos são a base para representação de informação textual como, por exemplo, a frase "eu amo programar em C"?

Variáveis com caracteres, em C(++), são declarados com sendo do tipo `char`, e sua leitura e escrita ocorre como para qualquer outro tipo de dados. Por exemplo,

```
#include<iostream>

using namespace std;

int main()
{
  char letra;

  cout << "digite uma letra qualquer seguida de enter ";
  cin >> letra;
  cout << "voce digitou "<< letra << endl;

  return 0;
}
```

Se você pretende atribuir um caracter diretamente a uma variável, é importante se atentar à seguinte notação: caracteres são sempre escritos entre aspas simples. Por exemplo, `'a'`, `'3'` ou `'.'`.

Raramente você trabalhará com caracteres um a um, normalmente usando vetores para armazenar palavras e frases.

11.2 Vetores de Caracteres

Vetores podem conter dados de quaisquer tipos. Isto é, você pode declarar vetores de números reais ou inteiros, booleanos, e até tipos definidos por você, uma vez que aprenda como definir novos tipos. Um outro tipo interessante é o caractere, ou simplesmente `char`. Por exemplo, vamos definir um programa que leia um vetor de 10 caracteres e depois os escreva de volta à tela.

```
#include<iostream>

using namespace std;

#define TAMANHO 10

int main()
{
  char nome[TAMANHO];

  cout << "Digite " << TAMANHO << " caracteres: ";

  for(int i = 0; i < TAMANHO; i++)
  {
      cin >> nome[i];
  }

  cout << "Os caracteres digitados foram: ";
  for(int i = 0; i < TAMANHO; i++)
  {
         cout << nome[i];
  }

  cout << endl;

  return 0;
}
```

Agora, só para tornar as coisas mais interessantes, alteremos o programa para que leia *até* 100 caracteres, mas que pare de lê-los tão logo um "." seja digitado. Para representar um caractere em C(++), use aspas simples, isto é, `'.'`.

```
#include<iostream>

using namespace std;

#define TAMANHO 100

int main()
{
  char nome[TAMANHO];
  int i = 0;

  cout << "Digite ate " << TAMANHO << " caracteres. Para terminar antes
      , digite '.' ";
```

```
14    do
      {
16        cin >> nome[i];
          i++;
18    }while( i < TAMANHO && nome[i-1] != '.');

20    cout << "Os caracteres digitados foram: "
      for(int i = 0; i < TAMANHO && nome[i] != '.'; i++)
22    {
            cout << nome[i];
24    }

26    cout << endl;

28    return 0;
  }
```

Caracteres são, na verdade, números disfarçados e seguem uma codificação específica. Uma pessoa pode decidir que o 'a' será o 1, o 'b' será o 2 e assim por diante. Mas como outra pessoa que receber a informação saberá disso? Para evitar este problema a representação de caracteres como números foi padronizada. Os principais padrões existentes são: ASCII, EBCDIC e Unicode.

ASCII

ASCII, ou *American Standard Code for Information Interchange*, é o padrão mais utilizado, presente em todos os nossos computadores pessoais. Trata-se de uma codificação de caracteres de oito bits baseada no alfabeto inglês.

A codificação define 256 caracteres (2^8). Desses, 33 não são imprimíveis, como caracteres de controle atualmente não utilizáveis para edição de texto, porém amplamente utilizados em dispositivos de comunicação, que afetam o processamento do texto. Exceto pelo caractere de espaço, o restante é composto por caracteres imprimíveis.

A Figura 11.1 exibe a tabela ASCII.

A linguagem C provê um atalho para que você não tenha que recorrer à tabela ASCII sempre que precisar do valor de um caractere: para obter o valor de um caractere qualquer, basta colocá-lo entre aspas simples. Isto é, para verificar se um caracter `c` é uma letra maiúscula, por exemplo, basta efetuar o teste `if(c >= 'A'&& c <= 'Z')`.

Outras representações

As representações EBCDIC (*Extended Binary Coded Decimal Interchange Code*) e Unicode também mapeiam os caracteres em números de 8 e 16 bits, respectivamente.

EBCDIC é utilizado principalmente em *mainframes* IBM. O padrão Unicode foi criado para acomodar alfabetos com mais de 256 caracteres.

Binário	Decimal	Hexa	Glifo
0010 0000	32	20	
0010 0001	33	21	!
0010 0010	34	22	"
0010 0011	35	23	#
0010 0100	36	24	$
0010 0101	37	25	%
0010 0110	38	26	&
0010 0111	39	27	'
0010 1000	40	28	(
0010 1001	41	29	)
0010 1010	42	2A	*
0010 1011	43	2B	+
0010 1100	44	2C	,
0010 1101	45	2D	-
0010 1110	46	2E	.
0010 1111	47	2F	/
0011 0000	48	30	0
0011 0001	49	31	1
0011 0010	50	32	2
0011 0011	51	33	3
0011 0100	52	34	4
0011 0101	53	35	5
0011 0110	54	36	6
0011 0111	55	37	7
0011 1000	56	38	8
0011 1001	57	39	9
0011 1010	58	3A	:
0011 1011	59	3B	;
0011 1100	60	3C	<
0011 1101	61	3D	=
0011 1110	62	3E	>
0011 1111	63	3F	?

Binário	Decimal	Hexa	Glifo
0100 0000	64	40	@
0100 0001	65	41	A
0100 0010	66	42	B
0100 0011	67	43	C
0100 0100	68	44	D
0100 0101	69	45	E
0100 0110	70	46	F
0100 0111	71	47	G
0100 1000	72	48	H
0100 1001	73	49	I
0100 1010	74	4A	J
0100 1011	75	4B	K
0100 1100	76	4C	L
0100 1101	77	4D	M
0100 1110	78	4E	N
0100 1111	79	4F	O
0101 0000	80	50	P
0101 0001	81	51	Q
0101 0010	82	52	R
0101 0011	83	53	S
0101 0100	84	54	T
0101 0101	85	55	U
0101 0110	86	56	V
0101 0111	87	57	W
0101 1000	88	58	X
0101 1001	89	59	Y
0101 1010	90	5A	Z
0101 1011	91	5B	[
0101 1100	92	5C	\
0101 1101	93	5D	]
0101 1110	94	5E	^
0101 1111	95	5F	_

Binário	Decimal	Hexa	Glifo
0110 0000	96	60	`
0110 0001	97	61	a
0110 0010	98	62	b
0110 0011	99	63	c
0110 0100	100	64	d
0110 0101	101	65	e
0110 0110	102	66	f
0110 0111	103	67	g
0110 1000	104	68	h
0110 1001	105	69	i
0110 1010	106	6A	j
0110 1011	107	6B	k
0110 1100	108	6C	l
0110 1101	109	6D	m
0110 1110	110	6E	n
0110 1111	111	6F	o
0111 0000	112	70	p
0111 0001	113	71	q
0111 0010	114	72	r
0111 0011	115	73	s
0111 0100	116	74	t
0111 0101	117	75	u
0111 0110	118	76	v
0111 0111	119	77	w
0111 1000	120	78	x
0111 1001	121	79	y
0111 1010	122	7A	z
0111 1011	123	7B	{
0111 1100	124	7C	|
0111 1101	125	7D	}
0111 1110	126	7E	~

Figura 11.1: Tabela ASCII

11.3 Exercícios

Exercício 11.1 *Escreva um programa que leia 10 caracteres e os imprima na ordem inversa àquela em que foram digitados.*

Exercício 11.2 *Escreva um programa que leia 10 caracteres e os imprima na ordem inversa àquela em que foram digitados, trocando maiúsculas por minúsculas e vice-*

versa.

Exercício 11.3 *Escreva seu nome na codificação ASCII.*

11.4 Laboratório

Laboratório 11.1 *Escreva um programa que leia 100 caracteres ou até que* `'#'` *seja digitado e os imprima na ordem inversa àquela em que foram digitados.*

Laboratório 11.2 *Escreva um programa que leia e imprima strings até que o usuário digite a palavra 'fim'. Considere que cada string não possui espaços.*

Laboratório 11.3 *Escreva um programa que leia e imprima strings até que o usuário digite a palavra 'fim'. As strings podem conter espaços.*

Laboratório 11.4 *Acompanhe as atividades a seguir:*

Execute o seguinte programa e observe o que será impresso. Atente para as atribuições!

```
#include<iostream>

using namespace std;

int main()
{
  int a;
  char c;
  for(a = 65; a < 91; a++)
  {
    c = a;
    cout << c << endl;
  }
  return 0;
}
```

Você deve ter observado que os números não foram impressos, e sim as letras de 'A' a 'Z'. Olhe a tabela na Figura 11.1 e descubra o porquê.

Laboratório 11.5 *Modifique o programa a seguir para imprimir as letras de 'a' a 'z', ou seja, as letras minúsculas do alfabeto inglês.*

Laboratório 11.6 *Agora faça um programa que leia caracteres informados pelo usuário enquanto ele não pressionar a tecla ESC. Para cada caractere informado pelo usuário, escreva o número correspondente na tabela ASCII.*

Laboratório 11.7 *Modifique o programa anterior para que solicite que o usuário entre com letras minúsculas de 'a' a 'z' e imprima na tela, para cada letra, a maiúscula correspondente.*

11.5 Vetores de Caracteres como *Strings*

Um vetor de caracteres é essencialmente uma palavra ou uma frase. Assim, durante a leitura de um vetor de caracteres dificilmente sabe-se quantos caracteres serão digitados pelo usuário, por exemplo, quando deseja-se ler o nome de um indivíduo. Por este motivo, a maioria das linguagem de programação fornece métodos especiais de leitura, impressão e manipulação desses vetores de caracteres, ou como chamaremos agora, *strings*.

Na linguagem C++, os comandos `cout` e `cin` permitem imprimir e ler *strings* de maneira direta:

```
#include<iostream>

using namespace std;

#define TAMANHO 100

int main()
{
  char nome[TAMANHO];

  cout << "Digite ate " << TAMANHO << " caracteres. Para terminar
      pressione ENTER:";

  cin >> nome;
  cout << "Os caracteres digitados foram: " << nome << endl;

  return 0;
}
```

Entretanto, o comando `cin`, não permite, da forma como vimos até agora, a leitura de uma *string* que contenha espaços (uma frase, por exemplo). Para que isso seja possível precisamos utilizar a função `cin.getline()`. Esta função necessita de dois argumentos: o vetor de caracteres representando a *string* e o tamanho máximo de caracteres que será lido. A leitura é realizada até que seja atingido ou número máximo de caracteres ou o usuário pressione a tecla ENTER. A seguir uma modificação do exemplo anterior:

```
#include<iostream>

using namespace std;

#define TAMANHO 100

int main()
{
  char nome[TAMANHO];

  cout << "Digite seu nome completo. Para terminar pressione ENTER:";

  cin.getline(nome, TAMANHO);
  cout << "Seu nome eh: " << nome << endl;
```

```
16    return 0;
}
```

Por último, é importante entender que para marcar no vetor de caracteres até onde foi feita a leitura a linguagem C adiciona o caractere especial `'\0'` após o último caractere digitado pelo usuário. Desta forma, quando se deseja percorrer uma string, diferentemente de um vetor número, define-se como condição de parada a posição em que se encontra o `'\0'` e não o fim do vetor.

11.6 Laboratório

Laboratório 11.8 *Faça um programa que leia uma string e calcule e imprima o tamanho desta string em uma função auxiliar.*

Laboratório 11.9 *Faça um programa que leia uma string e verifique se a string lida é uma palíndrome. A verificação deve ser feita em uma função auxiliar, que deve retornar* `true` *em caso afirmativo ou* `false` *caso contrário.*

Laboratório 11.10 *Faça um programa que leia uma string e a imprima de trás para frente, trocando as vogais pelo caractere '*'.*

Laboratório 11.11 *Faça um programa que leia duas strings e concatene a segunda na primeira, separadas por um espaço em branco. A concatenação deve ser feita em uma função auxiliar.*

11.7 Funções para manipulação *Strings*

Quando trabalhamos com *strings* é muito comum a realização de algumas tarefas como descobrir o tamanho da palavra digitada pelo usuário, comparar duas palavras para saber a ordem, ou ainda, concatenar duas palavras em uma única.

Para isso, a biblioteca `string.h` fornece algumas funções prontas, como pode ser visto na tabela 11.1

Função	Descrição
strlen	retorna o tamanho (em caracteres) da palavra passada como argumento.
strcpy	copia o conteúdo da segunda *string* para a primeira.
strcat	concatena o texto da segunda *string* na primeira.
strcmp	compara duas *strings* (vide exemplo a seguir).
stricmp	compara duas *strings* sem diferenciar maiúsculas e minúsculas.
atoi	converte uma *string* para o inteiro correspondente.
atof	converte uma *string* para o número real correspondente.

Tabela 11.1: Algumas funções para trabalhar com *strings*.

O exemplo a seguir mostra a utilização destas funções:

```
#include<string.h>
#include<iostream>

using namespace std;

int main()
{
  char str1[50], str2[50];
  int i;
  float f;

  cout << ``Entre primeiro nome:'';
  cin >> str1;
  cout << ``Entre ultimo nome:'';
  cin >> str2;
  strcat(str1, `` ''); //junto espaco com str1
  strcat(str1, str2);
  cout << ``Seu nome completo eh '' << str1 << endl;
  cout << ``Ele possui '' << strlen(str1) << `` caracteres.'' << endl
      ;

  cout << ``Entre outro nome:''.
  cin >> str2;

  //comparacao de strings
  if(strcmp(str1, str2) == 0)
  {
    cout << ``os dois nomes sao iguais.'' << endl;
  }
  else if(strcmp(str1, str2) < 0)
  {
    cout << str1 << `` vem antes de'' << str2 << endl;
  }
  else
  {
    cout << str2 << `` vem antes de '' << str1 << endl;
  }
  return 0;
}
```

No uso destas funções, é importante manter-se em mente que o espaço adequado deve ser alocado para a string resultante das operações. Por exemplo, ao se concatenar duas strings de no máximo 100 caracteres, o resultado terá, no máximo, 200 caracteres.

11.8 Funções com vetores como parâmetros

As funções descritas acima recebem strings como parâmetros. Agora, veremos como definir suas funções próprias funções que recebem não somente strings, mas vetores de outros tipos de dados.

A declaração de uma variável do tipo vetor segue, como já visto, a seguinte forma.

```
tipo identificador[tamanho];
```

A declaração de parâmetros segue uma sintaxe parecida, no qual o tamanho do vetor não aparece

```
tipo identificador[]
```

Isto ocorre por que, para que a função seja genérica e opere em qualquer vetor passado, o parâmetro não pode ser especificado em código. Assim, a declaração de uma função que recebe um vetor como parâmetro segue a seguinte receita:

```
tipo identificadorDaFuncao(tipo identificador1[], int tamanho1, ...)
{
    Corpo da Funcao
}
```

Por exemplo, veja como definir uma função que transforme todos os caracteres de uma string para letras maíusculas.

```
#include<string.h>
#include<iostream>

using namespace std;

void maiusculas(char str[], int tam)
{
   for(int i = 0; i < tam && i < strlen(str); i++)
   if(str[i] >= 'a' && str[i] <= 'z')
      str[i] = str[i] - 'a' + 'A';
}

int main()
{
  char str[50];
  cout << ``Digite seu primeiro nome:'';
  cin >> str;

  maiusculas(str, 50)

  cout << str << endl;
  return 0;
}
```

Observe que a função **altera** a string passada como parâmetro, um comportamento diferente do que aconteceu quando uma variável que não é um vetor é passada. Este comportamento será estudado mais a fundo em capítulos posteriores.

Da mesma forma que uma string, um vetor de outro tipo de dados pode ser passado como parâmetro. A função a seguir, por exemplo, calcula qual o maior inteiro em um vetor de inteiros.

```
#include<iostream>

using namespace std;
```

```
5  int maior(int v[], int tam)
   {
7     int maior = -1;
      for(int i = 0; i < tam ; i++)
9        if(v[i] > maior)
             maior = v[i];
11
      return maior;
13 }
```

11.9 Laboratório

Laboratório 11.12 *Faça um programa que dado uma string, retorne 1 se ela for palíndromo e 0 se ela não for palíndromo. Lembrando que um palíndromo é uma palavra que tenha a propriedade de poder ser lida tanto da direita para a esquerda como da esquerda para a direita. Deve-se obrigatoriamente utilizar uma* `string` *auxiliar e a função* `strcmp` *para fazer a resolução.*

Ex: SUBI NO ONIBUS
ARARA
ANOTARAM A DATA DA MARATONA

Laboratório 11.13 *Faça um programa que troque todas as ocorrências de uma letra L1 pela letra L2 em uma string. A string e as letras L1 e L2 devem ser fornecidas pelo usuário.*

Laboratório 11.14 *Faça um programa que leia 3 strings e as imprima em ordem alfabética.*

Laboratório 11.15 *Faça um programa com uma função que receba um vetor de inteiros e um inteiro X, e que retorne como resultado o maior inteiro do vetor que é menor que o inteiro X.*

Laboratório 11.16 *Usando a resposta do problema anterior, faça um programa com uma função que receba dois vetores de inteiros, e que faça com que o segundo torne-se uma cópia do primeiro, ordenado de forma decrescente.*

Capítulo 12

Bits, bytes e bases numéricas

Até agora temos trabalhado essencialmente com os tipos numéricos, inteiros e reais, e booleanos. Para que possamos usar outros tipos, é essencial que antes entendamos como os dados são representados na memória do computador.

12.1 Bit & Byte

Como dito anteriormente, a memória do computador pode ser entendida como uma grande planilha eletrônica, podendo cada célula ser endereçada (atribuída, lida, nomeada) individualmente. Na memória do computador não existem números ou caracteres, mas várias "lampâdas" que podem estar ligadas ou desligadas. Nós humanos atribuímos significado a tais estados como sendo 0's e 1's. Cada posição da memória, que pode armazenar 0 ou 1, é denominado um *bit*. Por serem mínimos, normalmente trabalhamos com conjuntos de 8 bits por vez, o que denominamos *bytes*.

12.1.1 Base Binária

Os bits de um byte podem assumir todas as combinações de ligado e desligado (0 e 1). Dado um contexto, cada combinação corresponde a um certo valor. Por exemplo, se estiver usando um byte para representar números inteiros, então provavelmente a correspondência na Tabela 12.1 se aplica.

Talvez isso seja novidade para você, mas você representa números na base decimal usando 10 dígitos (de 0 a 9). Contudo, outras bases existem. Como os bits só possuem dois estados, é natural na computação usar a base binária, que usa 2 dígitos (0 e 1). Os valores à esquerda na tabela podem ser entendidos como números nesta base.[1]

Para conseguir converter números de uma base para a outra, basta entender o que exatamente significa a representação de um número em uma base genérica X. Nesta base, $ABCD_X$ significa $A*X^3+B*X^2+C*X^1+D*X^0$. O número 1234_{10}, por exemplo, significa $1*X^3+2*X^2+3*X^1+4*X^0 = 1*1000+2*100+3*10+4*1$ que

[1] sempre que não for colocada a base, considera-se a base 10, mais natural aos seres de vida baseada em carbono do nosso planeta, A.K.A., você.

Tabela 12.1: Exemplos de valores em binário e decimal correspondente.

bits	Número
00000000	0
00000001	1
00000010	2
00000011	3
00000100	4
00000101	5
00000110	6
00000111	7
00001000	8
00001001	9
00001010	10
...	
11111101	253
11111110	254
11111111	255
00000000	0
...	

é igual a, tá dá, 1234_{10}. Para um exemplo mais interessante, o número $10101010_2 = 1*2^7+0*2^6+1*2^5+0*2^4+1*2^3+0*2^2+1*2^1+0*2^0 = 128+0+32+8+2 = 170$.

Observe que, pela tabela, o maior número que se pode representar com 8 bits é $11111111_2 = 255_{10}$. Via de regra, com X bits pode se representar 2^X na base binária, de 0 a $2^X - 1$.

12.1.2 Base Hexadecimal

Embora se possa usar qualquer base numérica, as que fazem mais sentido no mundo computacional são as binária e hexadecimal. A base hexadecimal usa 16 dígitos em cada posição representando valores de 0 a 15. Para isto, usa os valores de 0 a 9 iguais aos da base decimal e também as letras de A a F representado os valores de 10 a 15. Por exemplo, o número $1A2F_{16}$ equivale a $1 * 16^3 + 10 * 16^2 + 2 * 16^1 + 15 * 16^0 = 4096 + 2560 + 32 + 15 = 6703$.

12.2 Conversão entre bases numéricas

Enquanto nós trabalhamos com base decimal, o computador armazena informação e realiza operações na base binária. Por isso, a todo momento ocorre a conversão entre um decimal que informamos como entrada em um programa, por exemplo, para a base binária (para efetuar o cálculo), bem como da base binária (resultado do cálculo) para a base decimal exibida na tela.

Conversão de Binário para Decimal

A conversão de binário (e qualquer outra base) para decimal pode ser feita facilmente usando a notação da base genérica apresentada acima. Isto é, para realizar a conversão basta multiplicar o valor de cada posição pelo seu peso (base elevada à posição). Por exemplo, o número 1111011_2 equivale a $1 * 2^6 + 1 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 1 * 2^0 = 64 + 32 + 16 + 8 + 0 + 2 + 1 = 123_{10}$.

Conversão de Decimal para Binário

A conversão de decimal para binário é feita realizando-se divisões sucessivas pela base de interesse (2, no nosso caso) até que o resultado seja zero. Os restos das divisões, na ordem inversa, correspondem ao número convertido.

Por exemplo, considere a conversão do número 169 para a base binária. O processo de conversão é o seguinte:
$169/2 = 84, Resto = 1$
$84/2 = 42, Resto = 0$
$42/2 = 21, Resto = 0$
$21/2 = 10, Resto = 1$
$10/2 = 5, Resto = 0$
$5/2 = 2, Resto = 1$
$2/2 = 1, Resto = 0$
$1/2 = 0, Resto = 1$
O número binário equivalente é, portanto, 10101001_2.

Conversão entre Binário e Hexadecimal

Sabendo-se que com 4 bits é possível representar até 16 números (de 0 a 15) e que a base hexadecimal tem exatamente 16 dígitos, concluímos que é possível representar cada dígito hexadecimal com 4 dígitos binários. Sendo assim, a conversão binário/hexadecimal pode ser feita facilmente substituindo conjuntos de 4 bits binários por um dígito hexadecimal e vice-versa.

Por exemplo, para convertermos 11101010_2 convertemos 1110_2 para E_{16} e 1010_2 para A_{16}, obtendo o número EA_{16}. Na direção inversa, para convertermos o número $7B_{16}$ para binário convertemos 7_{16} para 0111_2 e B_{16} para 1011_2, obtendo o número 01111011_2.

12.3 Tipos Numéricos Inteiros

Na linguagem C, os principais tipos, a quantidade de bits utilizada para sua representação e o intervalo de valores aceitos são resumidos na tabela a seguir: [2]

Observe que o tipo caractere também é armazenado internamente como um número inteiro de 8 bits. Com 8 bits podemos representar 2^8 números. Se considerarmos números sem sinal (*unsigned*) teremos de 0 a 255; se considerarmos números com

[2] A quantidade de bits pode variar de acordo com o compilador e a arquitetura do sistema

Tabela 12.2: Exemplos de valores em binário, decimal e hexadecimal correspondentes.

Binário	Decimal	Hexadecimal
00000000	0	0
00000001	1	1
00000010	2	2
00000011	3	3
00000100	4	4
00000101	5	5
00000110	6	6
00000111	7	7
00001000	8	8
00001001	9	9
00001010	10	A
00001011	11	B
00001100	12	C
00001101	13	D
00001110	14	E
00001111	15	F
00010000	16	10
...		
11111101	253	FD
11111110	254	FE
11111111	255	FF
00000000	0	0
...		

Tabela 12.3: Intervalo de representação dos tipos numéricos inteiros.

Tipo	Quantidade de Bits	Intervalo
char	8	-128 a 127
unsigned char	8	0 a 255
int	16	-32.768 a 32.767
unsigned int	16	0 a 65.535
long	32	-2.147.483.648 a 2.147.483.647
unsignet long	32	0 a 4.294.967.295

sinal, teremos metade para positivos e metade para negativos, ou seja, de -128 a 127. O mesmo raciocínio se aplica para os outros tipos de dados da tabela.

A partir desta tabela podemos observar a importância da escolha adequada do tipo correto de acordo com sua aplicação. Se utilizarmos um inteiro para somarmos 20.000 e 50.000, por exemplo, o resultado será inesperado, uma vez que o maior valor que um inteiro aceita é 32.767. Quando isto ocorre, dizemos que houve um *overflow*.

12.3.1 Números Binários Negativos

Os exemplos que vimos até agora de conversão não contemplam números negativos. Na base decimal, o sinal de menos (-) antes do valor do número tem a finalidade de representar números negativos.

Como a memória do computador armazena apenas 0's e 1's, uma possível estratégia é "desperdiçar" um dos bits do número para o sinal. Adota-se, por padrão, o zero para representar um número positivo e o um para negativo. O bit do sinal é armazenado na posição mais à esquerda.

Para representar o valor de um número negativo há duas principais abordagens: **sinal-magnitude** e **complemento de 2**.

Na representação **sinal-magnitude**, o número negativo tem seu valor absoluto (magnitude) representado da mesma forma que um número positivo e o bit mais significativo, que representa o sinal, será igual a um. A tabela a seguir mostra alguns exemplos de números na base decimal e na representação sinal-magnitude.

Tabela 12.4: Exemplos de valores na representação sinal-magnitude.

Decimal	Binário Sinal-Magnitude
+20	**0**0010100
-20	**1**0010100
+115	**0**1110011
-115	**1**1110011

Com 8 bits, temos, então números de 11111111_2 a 01111111_2 (-127 a +127). Um problema que aparece com esta representação é o zero. Qual a representação correta, 00000000 ou 10000000? E por que precisaríamos de duas representações?

A representação **complemento de 2** utiliza uma maneira um pouco diferente de representar números negativos. Para expressá-los devemos inverter cada bit do número

positivo e somar 1 ao resultado. Por exemplo, o número -15 é representado em complemento de 2 com 5 bits como 10001, ou seja, calcula-se o valor de +15 = 01111_2, e, em seguida invertemos e somamos 1: $10000_2 + 1_2 = 10001_2$. Se a representação usar 8 bits, então $15_{10} = 00001111_2$ e $-15_{10} = 11110001_2$.

12.4 Aritmética Inteira Binária

Aritmética binária é tão simples como $1 + 1 = 10$. Sério, é assim que se faz: $0 + 0 = 0, 0 + 1 = 1 + 0 = 1, 1 + 1 = 10_2$. Logo, em sistemas computacionais as somas e subtrações geralmente são realizadas aos pares, da mesma forma que na base binária.

12.4.1 Números positivos

A **adição** de números positivos é muito simples. Para fazê-la, basta somar os valores na mesma posição, do menos significativo para o mais significativo. O "vai-um" funciona da mesma forma como fazemos na base decimal. Por exemplo, para somarmos 5 e 9, fazemos:

```
  1
0101 +
1001
----
1110
```

Como a quantidade de bits para determinado tipo de dados é limitado, durante uma soma pode ocorrer de o resultado ultrapassar o maior valor permitido para aquela quantidade de bits, quando isto ocorre, temos um *overflow* (derramamento). Considere a soma de 7 e 9, ambos com 4 bits.

```
1 111
  0111 +
  1001
  ----
1 0000  <-- overflow!
```

Houve um "vai-um" além do bit mais significativo, caracterizando um *overflow*. Neste caso, o resultado da operação está errado.

12.4.2 Números em Complemento de 2

Como vimos anteriormente, o computador usa duas principais representações para números negativos. A representação complemento de 2 é a mais utilizada porque permite que a soma de números negativos seja feita da mesma maneira que a soma de números positivos. Por exemplo considere as soma de -2 + 5 e -1+(-1), em complemento de 2:

```
    1 1
(-2)  1110 +
```

```
(+5)  0101
      ----
    1 0011  <-- Não é overflow! Esse "vai-um" deve ser desprezado!

    1 111
(-1)  1111 +
(-1)  1111
      ----
    1 1110  <-- Não é overflow! Esse "vai-um" deve ser desprezado!
```

Observe que nestes exemplos tivemos um "vai-um" além do bit mais significativo. Em soma de números em complemento de 2 isto não é suficiente para caracterizar um *overflow*. Um *overflow* apenas ocorrerá quando, ao somarmos dois números de mesmo sinal, obtivermos um número de sinal diferente. Desta maneira, no exemplo apresentado, este "vai-um" a mais deve simplesmente ser ignorado.

Já os exemplos a seguir mostram operações problemáticas.

```
      111
(+7)  0111 +
(+7)  0111
      ----
      1110  <-- O resultado começa com 1, logo é negativo. Absurdo!

    1
(-3)  1101 +
(-6)  1010
      ----
    1 0111  <-- O resultado começa com 0, logo é positivo. Absurdo!
```

12.4.3 E a subtração?

Na verdade, a maioria dos computadores não sabe fazer subtração. O que eles fazem é utilizar um truque para transformar uma subtração em uma soma.

Por exemplo, a operação $7 - 5$ pode ser reescrita como $7 + (-5)$, ou seja, para realizar a subtração basta inverter o segundo número (em complemento de 2) e somá-lo ao primeiro da maneira usual.

12.5 Tipos Numéricos Reais

Para representarmos números reais no computador, definimos que uma quantidade de bits do número será usada para representar a mantissa e o restante o expoente do número. Por exemplo, se dispusermos de 8 bits, podemos definir que os quatro primeiros bits serão a mantissa e os quatro últimos serão o expoente, em notação científica. Assim, $10101010_2 = 1,0 * 10^{11}$ e $10111010_2 = 1,1 * 10^{11}$.

Antes que você saia dizendo por aí que o ideal é usar tais números pois podemos representar números tão grandes como $11111111_2 = 15 * 10^{15}$, que é muito mais que os $2^{16} - 1$ da notação binária convencional, pense em como representaria o número 161_{10}, já que a maior mantissa representável é 1,5. Além disso, como representar 1,1111, já que só dispomos de 4 dígitos na mantissa?

Além disso, se representações de números reais pecam em sua precisão, também pecam na velocidade de processamento. A aritmética de números binários é muito mais simples que a de números reais (ou de ponto flutuante, como costumamos dizer na computação). Para manipular números reais, computadores normalmente precisam de componentes dedicados a este fim e que tem alto tempo de execução.

Para saber mais sobre representações de números em pontos flutuantes visite a URL `http://en.wikipedia.org/wiki/Floating_point`

12.6 Exercícios e laboratório

12.6.1

Quantos números se pode representar, na base binária, com 1, 8, 16, 32 e 64 bits?

12.6.2

Escreva os seguintes números nas bases binária e hexadecimal:

1. 16
2. 45
3. 129
4. 23
5. 1290

12.6.3

Converta os números a seguir para a base decimal:

1. 16_{16}
2. $D5_{16}$
3. 1100101101_2
4. $2C04_{16}$
5. 11101_2

12.6.4

Escreva os números a seguir nas representações sinal-magnitude e complemento de 2 com 8 bits:

1. -19
2. +47
3. -29
4. -37
5. -105

12.6.5

Realize as seguintes operações aritméticas em complemento de 2 com números de 8 bits:

1. 16 - 9
2. -45 - 7
3. -12 + 12

12.6.6

Laboratório 12.1 *Faça um programa para descobrir quantos bits tem uma variável do tipo* `int`*? E um* `unsigned int`*?*

Laboratório 12.2 *Escreva um programa que leia um número de 2 dígitos na base decimal e imprima o mesmo número na base binária.*

Laboratório 12.3 *Escreva um programa que leia um número binário de até 10 bits e imprima o mesmo número na base decimal.*

Capítulo 13

Funções Úteis I

13.1 Funções Matemáticas

A biblioteca `math.h` fornece algumas funções aritméticas muito úteis no desenvolvimento de programas. A Tabela 13.1 apresenta algumas destas funções.

Função	Descrição/Exemplo
abs	valor absoluto do argumento. Ex: `int x = abs(-9);`
sin	seno do argumento (em radianos). Ex: `double x = sin(3.14159);`
cos	cosseno do argumento (em radianos). Ex: `double x = cos(3.14159);`
tan	tangente do argumento (em radianos). Ex: `double x = tan(3.14159);`
asin	arco cujo seno é passado como argumento. Ex: `double x = asin(1);`
acos	arco cujo cosseno é passado como argumento. Ex: `double x = acos(1);`
atan	arco cuja tangente é passada como argumento. Ex: `double x = atan(sqrt(2)/2);`
floor	piso do valor passado como argumento. Ex: `double x = floor(3.2); //=3`
ceil	teto do valor passado como argumento. Ex: `double x = floor(3.2); //=4`
round	arredonda o argumento para o inteiro mais próximo. Ex: `double x = round(9.9); //=10`
pow	eleva o primeiro argumento ao expoente no segundo. Ex: `double x = pow(2,3); //=8`
sqrt	retorna a raiz quadrada do argumento. Ex: `double x = sqrt(169); //=13`
log	retorna logaritmo natural do argumento.
log10	retorna log. do argumento na base 10.
log2	retornar log do argumento na base 2

Tabela 13.1: Algumas funções aritméticas.

Além das funções em `math.h`, duas outras funções, da biblioteca `stdlib.h`, são particularmente interessantes na manipulação de números. Estas funções são apresentadas na Tabela 13.2.

A função `rand()` é utiliza para gerar números aleatórios. Um número aleatório é gerado internamente pelo computador aplicando-se operações aritméticas que o usuário desconhece a partir de um valor inicial chamado de *semente*. O valor dessa semente é definido com a função `srand()`. O exemplo a seguir imprime na tela 30 números

Função	Descrição/Exemplo
rand	retorna um número aleatório (biblioteca `stdlib.h`)
srand	define a semente para a geração de números aleatórios por `rand` (biblioteca `stdlib.h`)

Tabela 13.2: Funções para geração de números aleatórios.

"aleatórios".

```
#include<stdlib.h>
#include<time.h>
  . . .
int main()
{
  for (int i = 0; i < 10; i++)
  {
    cout << rand() << endl;
  }

  for (int i = 0; i < 10; i++)
  {
    cout << rand() << endl;
  }

  //Configuracao da semente
  //com valor que depende da hora atual.
  //Isto garante maior 'aleatoriedade'
  srand(time(NULL));
  for (int i = 0; i < 10; i++)
  {
    cout << rand() << endl;
  }
  return 0;
}
```

O codigo a seguir é para um jogo de adivinhação. O programa gera um número aleatório entre 1 e 10, e o usuário deve descobrir qual é este número[1].

```
/* rand example: guess the number */
#include <iostream>
#include <stdlib.h>
#include <time.h>

using namespace std;

int main ()
{
  int iSecret, iGuess;

  /* initialize random seed: */
  srand ( time(NULL) );

  /* generate secret number: */
  iSecret = rand() % 10 + 1;

  do {
    printf ("Guess the number (1 to 10): ");
    cin >> iGuess;
    if (iSecret<iGuess) puts ("The secret number is lower");
    else if (iSecret>iGuess) puts ("The secret number is higher");
  } while (iSecret!=iGuess);
```

[1] http://www.cplusplus.com/reference/clibrary/cstdlib/rand/

```
25    puts ("Congratulations!");
      return 0;
27  }
```

13.2 Laboratório

Laboratório 13.1 *Faça um programa que leia os catetos de um triângulo e imprima o valor de sua hipotenusa. Utiliza as funções aritméticas.*

Laboratório 13.2 *O valor de π pode ser dado pela série:*
$\pi = \sum_{n=0}^{\infty}(-1)^n \frac{4}{2n+1}$
Faça uma função chamada `pi` *que recebe o valor de* `n` *e retorna o valor calculado de acordo com a função informada. A função principal deve ler o valor de* `n`*, invocar a função* `pi` *e imprimir o resultado.*

Laboratório 13.3 *Faça um programa que leia dois números x e y e calcule $log_y x$. O cálculo deve ser feito em uma função auxiliar.*

Laboratório 13.4 *Faça uma função que receba como parâmetro o valor de um ângulo em graus e o número de iterações (n) e calcule o valor do cosseno hiperbólico desse ângulo usando sua respectiva série de Taylor:*
$cosh(x) = \sum_{n=1}^{\infty} \frac{x^{2n}}{(2n)!}$
, onde x é o valor do ângulo em radianos*. Considerar pi = 3.141593.*

Capítulo 14

Arranjos Multidimensionais

Embora os arranjos unidimensionais sejam úteis em várias situações, eles não são suficientes para resolver todos os problemas relacionados a arranjos. Em certas situações, várias dimensões precisam ser representadas. Um exemplo simples é o problema de multiplicação de matrizes.[1]

14.1 Declaração e Iniciação

Como já visto, a declaração de arranjos tem a seguinte sintaxe,

```
tipo identificador[tamanho];
```

A declaração de uma variável do tipo matriz tem uma forma semelhante, apenas aumentando uma segunda dimensão ao arranjo, como a seguir.

```
tipo identificador[tamanho1][tamanho2];
```

Assim, uma declaração como

```
1 int matriz[10][20];
```

declara uma matriz de 10 linhas por 10 colunas, cujas células são números inteiros.

A matriz pode ser iniciada como em arranjos unidimensionais, colocando-se os valores do elemento dentro de chaves após a declaração da matriz. Os valores para cada linha devem ficar dentro de chaves próprias e são separados por vírgula:

```
1 {int matriz[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

A matriz criada pode ser visualizada da seguinte maneira:

$$\begin{vmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{vmatrix}$$

[1] A bem da verdade, é possível representar qualquer matriz finita como um arranjo unidimensional, mas a complexidade é muito maior em relação ao uso de arranjos multidimensionais.

14.1.1 Acesso aos elementos

O acesso às células é feito também como em arranjos unidimensionais, exceto que ambas as coordenadas devem ser especificadas. O exemplo a seguir mostra a criação, iniciação, e impressão dos elementos de uma matriz bidimensional.

```
#include<iostream>

#define M 5
#define N 10

using namespace std;

int main()
{
   int m[M][N];

   cout << "Digite o valor dos elementos de uma matriz " << M << " por
       " << N;
   for(int i = 0; i < M; i++)
      for(int j = 0; j < N; j++)
      {
         cout << i << "," << j;
         cin >> m[i][j];
      }

   cout << "A matriz lida foi a seguinte "<< endl;

   for(int i = 0; i < M; i++)
   {
      for(int j = 0; j < N; j++)
      {
         cout << m[i][j] << " ";
      }

      cout << endl;
   }
}
```

Observe que tanto na leitura como impressão da matriz, o laço mais externo percorre as linhas. Para cada valor da variável `i`, a variável `j`, dentro do laço interno percorre os valores de 0 a $N - 1$. Desta maneira, a matriz é lida e impressa linha a linha.

Observe a impressão. Para que serve o comando `cout<<endl;`?

14.2 Mais Dimensões

Agora você já deve ter deduzido que para adicionar mais dimensões aos seus arranjos, basta colocar esta dimensão na declaração dos mesmos. Por exemplo, o seguinte trecho de código declara um arranjo com três dimensões, em que cada célula é um inteiro.

```
char matriz[100][100][100];
```

Outra forma de ver tal arranjo é como um livro com várias “páginas”, em que cada uma é uma matriz bidimensional.

14.3 Multiplicação de Matrizes

Agora que você viu alguns exemplos de declarações de arranjos multidimensionais, vejamos alguns usos reais, como a já mencionada multiplicação de matrizes.

Sejam duas matrizes A e B, tal que A tem dimensões $m \times n$ e $B, n \times o$; então, a matriz AB tem dimensões $m \times o$, e $C_i, j = \sum_{k=1}^{m} A_{m,n} B_{n,o}$.

Observe que na matemática os índices começam, normalmente, em 1. Na computação, contudo, os arranjos tem seu menor índice igual a zero. Logo, o código para multiplicar as duas matrizes fica assim.

```
int main()
{
   int soma = 0, m, n, o;
   int a[100][100], b[100][100], ab[100][100]; //m,n e 0 precisam ser
       menores que 100.

   //ler as dimensoes
   cout << "Quais as dimensoes das matrizes?";
   cin >> m >> n >> o;

   for(int i = 0; i < m; i++)
      for(int j = 0; j < n; j++)
         cin >> a[i][j];


   for(int j = 0; j < m; j++)
      for(int k = 0; k < n; k++)
         cin >> a[j][k];



   for(int i = 0 ; i < m ; i++ )
   {
      for(int j = 0 ; j < o ; j++ )
      {
         for(int k = 0 ; k < n ; k++ )
         {
            sum = sum + a[i][k] * b[k][j];
         }

         ab[i][j] = sum;
         sum = 0;
      }
    }
}
```

Observe que fizemos nosso código de manipulação da matriz dentro da função `main` . Fizemos isso por que usar matrizes como parâmetro é bem mais complicado do que vetores unidimensionais. Contudo, não se aflija, pois veremos isso logo a seguir.

14.4 Passagem de matriz como parâmetro em funções

Vimos que, ao passar vetores para uma função, não era necessário especificar o número de elementos no vetor. Em matrizes bidimensionais, não é necessário especificar o número de linhas na matriz, apenas o número de colunas. O programa a seguir usa a função `exibeMatriz` para exibir o conteúdo de matrizes bidimensionais:

```
#include <iostream>
#define ncol 10

using namespace std;

void exibeMatriz(int matriz[][ncol], int linhas)
{
  int i, j;
  for (i = 0; i < linhas; i++)
  {
    for (j = 0; j < 10; j++)
    {
      cout << matriz[i][j] << ''\t'';
    }
    cout << endl;
  }
}

int main()
{
  int a[1][10] = {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}};

  int b[2][10] = {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
                  {11, 12, 13, 14, 15, 16, 17, 18, 19,20}};

  int c[3][10] = {{1, 2, 3, 4, 5, 6, 7, 8, 9, 10},
        {11, 12, 13, 14, 15, 16, 17, 18, 19, 20},
        {21, 22, 23, 24, 25, 26, 27, 28, 29, 30}};

  exibeMatriz(a, 1);
  exibeMatriz(b, 2);
  exibeMatriz(c, 3);
  return 0;

}
```

Via de regra, não é necessário especificar a primeira dimensão da matriz. Isto é, em vetores, você não passava qualquer dimensão. Em matrizes bidimensionais, não precisa especificar a quantidade de linhas. Em uma matriz tridimensional, não precisaria especificar a quantidade de “páginas”, e assim por diante.

Lembre-se, semelhante ao que acontece com vetores, matrizes alteradas em funções auxiliares implicam em alteração na matriz original.

14.5 Matrizes de Caracteres

Da mesma forma que vetores de caracteres podem ser manipuladas de forma especial no C(++), também podem as matrizes de caractere. Na verdade, se um vetor de caracteres é o que chama-se de uma string, então uma matriz bidimensional de caracteres é, na verdade, um vetor de strings. Por exemplo,

```
1 char nomes[10][20];
```

pode ser visto como um vetor de 10 strings, cada uma com 20 caracteres, e cada string pode ser manipulada como tal, como no exemplo a seguir.

```
1  #include<iostream>
2
3  #define M 10
4  #define N 11
5
6  using namespace std;
7
8  int main()
9  {
10    char m[M][N];
11
12    cout << "Digite " << M << " palavras de no maximo " << N-1 << "
       caracteres" << endl;
13    for(int i = 0; i < M; i++)
14    {
15          cout << i << ": ";
16          cin >> m[i];
17    }
18
19    cout << "As palavras lidas foram as seguintes "<< endl;
20
21    for(int i = 0; i < M; i++)
22    {
23       cout << m[i] << "  (com tamanho = " << strlen(m[i]) << ")" <<
          endl;
24    }
25 }
```

14.6 Exercicios

Laboratório 14.1 *Escreva um programa com uma função que receba como parâmetro uma matriz bidimensional e suas dimensões, e que incremente cada elemento da matriz em 10%.*

Laboratório 14.2 *Escreva um programa que leia e some duas matrizes, imprimindo o resultado. Utilize funções auxiliares para leitura, impressão e soma.*

Laboratório 14.3 *Escreva um programa que leia uma matriz e diga se ela é a matriz identidade ou não.*

Laboratório 14.4 *Escreva um programa que leia uma matriz e imprima sua transposta.*

Capítulo 15

Ordenação de Arranjos

15.1 Introdução

Uma das aplicações mais estudadas e realizadas sobre arranjos é a **ordenação**. Ordenar um arranjo significa permutar seus elementos de tal forma que eles fiquem em ordem crescente, ou seja, $v[0] <= v[1] <= v[2] <= \ldots <= v[n-1]$. Por exemplo, suponha o vetor
$v = 5, 6, -9, 9, 0, 4$.
Uma ordenação desse vetor resultaria em um rearranjo de seus elementos:
$v = -9, 0, 4, 5, 6, 9$.

Exitem diversos algoritmos de ordenação para vetores. Eles variam em relação à dificuldade de implementação e desempenho. Usualmente algoritmos mais fáceis de serem implementados apresentam desempenho inferior. Veremos 3 algoritmos diferentes de ordenação:

1. Algoritmo de Inserção (*Insertion Sort*);
2. Algoritmo de Seleção (*Selection Sort*); e
3. Algoritmo de Ordenação por Troca (*Bubble Sort*).

15.2 Algoritmos de Ordenação

15.2.1 Algoritmo de Inserção (*Insertion Sort*)

Trata-se de um dos algoritmos de implementação mais simples. Seu método de ordenação semelhante ao que usamos para ordenar as cartas de um baralho. A idéia básica do algoritmo é descrita a seguir:

- Compare a chave (`x`) com os elementos à sua esquerda, deslocando para direita cada elemento maior do que a chave;
- Insira a chave na posição correta à sua esquerda, onde os elementos já estão ordenados;

- Repita os passos anteriores atualizando a chave para a próxima posição à direita até o fim do vetor.

A figura 15.1 apresenta um exemplo de uma etapa da execução do algoritmo.

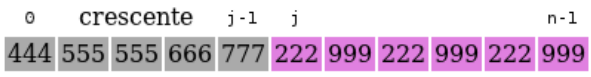


Figura 15.1: Exemplo do algoritmo *Insertion Sort*.

O código a seguir implementa o algoritmo em C, considerando um vetor `v` de tamanho `n`.

```
void insertionSort(int v[], int n)
{
  int i, j, x;
  for(i = 1; i < n; i++)
  {
    x = v[i];
    j = i - 1;
    while(j >= 0 && v[j] > x)
    {
      v[j+1] = v[j];
      j--;
    }
    v[j+1] = x;
  }
}
```

15.2.2 Algoritmo de Seleção (*Selection Sort*)

A implementação deste método de ordenação é muito simples. A idéia básica é descrita a seguir:

- Selecione o menor elemento do vetor de tamanho `n`;
- Troque esse elemento com o elemento da primeira posição do vetor;
- Repita as duas operações anteriores considerando apenas os `n-1` elementos restantes, em seguida repita com os `n-2` elementos restantes; e assim sucessivamente até que reste apenas um elemento no vetor a ser considerado.

A figura 15.2 apresenta um exemplo da execução do algoritmo.

O código a seguir implementa o algoritmo em C, considerando um vetor `v` de tamanho `n`.

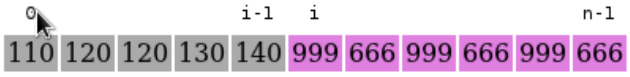


Figura 15.2: Exemplo do algoritmo *Selection Sort*.

```
void selectionSort(int v[], int n)
{
  int i, j, aux, min;
  for(i = 0; i < n-1; i++)
  {
    min = i;
    for(j = i+1; j < n; j++)
    {
      if(v[j] < v[min])
      {
    min = j;
      }
    }
    aux = v[i]; v[i] = v[min]; v[min] = aux; //troca
  }
}
```

15.2.3 Algoritmo de Ordenação por Troca (*Bubble Sort*)

Outro algoritmo simples, muito útil para ordenação de vetores pequenos, mas não indicado para vetores maiores devido ao seu baixo desempenho computacional. Sua déia básica é apresentada a seguir:

- Compare o primeiro elemento com o segundo. Se estiverem desordenados, então efetue a troca de posição. Compare o segundo elemento com o terceiro e efetue a troca de posição, se necessário;
- Repita a operação anterior até que o penúltimo elemento seja comparado com o último. Ao final desta repetição o elemento de maior valor estará em sua posição correta, a n-ésima posição do vetor;
- Continue a ordenação posicionando o segundo maior elemento, o terceiro,..., até que todo o vetor esteja ordenado.

A figura 15.3 apresenta um exemplo de um vetor sendo ordenado pelo algoritmo.

O código a seguir implementa o algoritmo em C, considerando um vetor `v` de tamanho `n`.

```
void bubbleSort(int v[], int n)
{
  int i, j, aux;
  for(i = n-1; i > 0; i--)
```

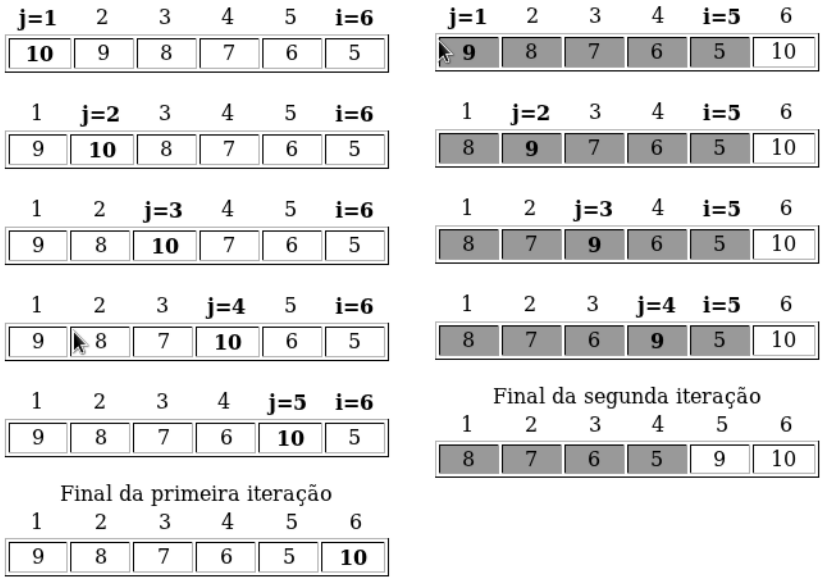


Figura 15.3: Exemplo de ordenação usando o algoritmo *Bubble Sort*.

```
  {
6   for(j = 0; j < i; j++)
    {
8     if(v[j] > v[j+1])
      {
10  aux = v[j]; v[j] = v[j+1]; v[j+1] = aux; //troca
      }
12    }
  }
14}
```

15.3 Exercícios

15.3.1

Implemente na linguagem C o algoritmo de ordenação *insertion sort*. Utilize funções auxiliares para implementar a ordenação, a leitura do vetor desordenado e a impressão do vetor ordenado.

15.3.2

Implemente na linguagem C o algoritmo de ordenação *selection sort*. Utilize funções auxiliares para implementar a ordenação, a leitura do vetor desordenado e a impressão do vetor ordenado.

15.3.3

Implemente na linguagem C o algoritmo de ordenação *bubble sort*. Utilize funções auxiliares para implementar a ordenação, a leitura do vetor desordenado e a impressão do vetor ordenado.

Parte II

Intermediário

Capítulo 16

Estruturas Não-Homogêneas

16.1 Introdução

Imagine que você deseje armazenar informações sobre um funcionário de uma empresa. Entre estas informações podemos ter, por exemplo, nome, endereço, telefone, sexo, estado civil, cargo, setor e salário. Em um programa, isto seria representado por oito variáveis apenas para um funcionário. Se você desejar incluir mais informações, isto implicará em mais variáveis, aumentando a complexidade do programa.

Por esta razão, em muitas aplicações, é importante a capacidade de se tratar todas as informações de uma entidade (uma pessoa, por exemplo), como sendo uma única unidade de armazenamento, ou seja, uma única variável. Por outro lado, deve ser possível, que esta única variável permita acesso a cada informação em separado. Outros exemplos de informação com esta característica é o endereço, que pode ser decomposto em: logradouro, número, complemento, CEP, cidade, estado, país.

Neste tipo de informação é possível observar que, diferentemente de variáveis indexadas (vetores e matrizes), não há homogeneidade quanto ao tipo de dados tratado. Por isso, deve haver um mecanismo para trabalhar com variáveis estruturadas heterogêneas, também conhecidas simplesmente como *estruturas* ou *structs*.

16.2 Declaração

Na linguagem C, a palavra reservada `struct` é destinada à declaração de variáveis não-homogêneas, e seu uso segue a seguinte sintaxe:

```
struct identificador{
tipo_campo_1 nome_campo_1;
...
tipo_campo_n nome_campo_n;
};
```

Em primeiro lugar, cada estrututura tem um identificador, usado na declaração de variáveis do tipo desta estrutura. Em seguida, são declarados os campos da estrutura,

especificadas entre as chaves. Os campos podem ter qualquer tipo, inclusive ser outra estrutura. O exemplo a seguir declara uma estrutura para o armazenamento de endereços e várias variáveis deste tipo.

```
struct endereco {
  char logradouro[15];
  int numero;
  char complemento[6], bairro[10];
  char cidade[10], estado[3], pais[10];
};
...
struct endereco e1, e2, e3;
```

O uso da palavra reservada `struct` é obrigatório na declaração de variáveis, pois ela faz parte do tipo da variável. O uso repetido desta palavra reservada pode "poluir" o código, e por essa razão recomenda-se a definição de novos tipos baseados na estrutura.

16.2.1 typedef

A palavra chave `typedef` permite que se defina novos tipos de dados a partir de tipos já existentes. Por exemplo, é possível definir um tipo "numero_real" ou um tipo "caractere". A sintaxe para o uso `typedef` é a seguinte:

```
typedef tipo_antigo tipo_novo;
```

Os exemplos apenas dados são implementados assim:

```
typedef float numero_real;
typedef char caractere;
```

Mais importante, é possível definir um tipo basedo na estrutura. Neste caso, há duas formas de fazê-lo. O exemplo a seguir mostra a mais verborrágica.

```
struct end{
  char logradouro[15];
  int numero;
  char complemento[6], bairro[10];
  char cidade[10], estado[3], pais[10];
};
...
typedef end endereco;

endereco e1, e2, e3;
```

Embora mais explícita, esta forma não é usada normalmente, em função forma mais compacta a seguir.

```
typedef struct{
  char logradouro[15];
  int numero;
  char complemento[6], bairro[10];
  char cidade[10], estado[3], pais[10];
} endereco;
...
endereco e1, e2, e3;
```

16.3 Acesso aos Campos de Uma Estrutura

Embora as estruturas agreguem vários campos, com raras exceções, o manuseio da mesma deve ser feito campo a campo, como variáveis normais. Para acessar um dos campos de uma estrutura, usa-se o operador '.' (ponto). Diferentemente de variáveis indexadas, variáveis do tipo estrutura podem receber o valor de outra do mesmo tipo (desde que nenhum dos campos seja vetor). Por exemplo:

```
typedef struct st1 st1;
struct st1 {
  char l;
  int i;
  float b;
};
...
st1 s1 = {'c', -9, 4.76},       //Atribuicao na iniciacao.
    s1Copia, s2;
s2.l = 'z';                     //Atribuicao...
s2.i = -4;                      //... campo a ...
s2.b = 0.89;                    //... campo.
s1Copia = s1;                   //Copia de valores de todos os campos.
...
```

16.4 Exemplo

Considere a estrutura `complexo`, a qual representa um número imaginário na forma $a + b \times i$, em que o valor de a é armazenado na variável `real` e o valor de b é armazenado na variável `imag`. O exemplo a seguir apresenta uma matriz de 3×3 em que cada elemento é do tipo `complexo`.

```
#include <iostream>

using namespace std;

typedef struct{
  float real, imag;
} complexo;
```

```
8
int main() {
10   int i, j;
   complexo A[3][3] = {
12                       { {1.0, -0.1}, {2.0, -0.2}, {2.0, -0.2} }, //real
                         { {4.0, -3.4}, {5.0, 4.1}, {6.0, -2.6} }   //imag
14                      };
   for ( i= 0; i < 3; i++) {
16     for (j = 0; j < 3; j++)
         cout << A[i][j].real << " + " A[i][j].imag << "*i" << \t;
18     cout << endl;
   }
20   return 0;
}
```

16.5 Exercícios

16.5.1

Faça um programa que leia informações sobre 15 pessoas. Essa informação deve ficar em um vetor de variáveis do tipo estruturado `pessoa`, o qual deve conter as seguintes informações:

- Nome: string de tamanho 30;
- Sexo: tipo enumerativo com os valores `masc`, `fem`;
- Idade: valor inteiro;
- Estado Civil: tipo enumerativo com os valores `solteiro`, `casado`, `separado`, `viúvo`.
- Salário: valor real.

Em seguida, imprima o número de homens, número de mulheres e informações da pessoa com maior salário.

16.5.2

Faça um programa que leia o nome, duas notas e número de faltas de 10 alunos. As informações desses alunos devem ser armazenadas em um vetor de variáveis do tipo estruturado `aluno`, o qual deve conter as seguintes informações de cada aluno:

- Nome: string de tamanho 30;
- Média: número real resultado da média das duas notas lidas;
- Situação: caractere representando situação, isto é, 'A' (Aprovado), se média maior ou igual a 6 e número de faltas menor que 10, e 'R' (Reprovado), caso contrário.

- Faltas: número de faltas (valor inteiro).

Por fim, devem ser impressas as informações de cada aluno.

16.6 Laboratório

Primeiramente, implemente sua solução para o exercício anterior. Em seguida, implemente duas funções, que ordenem os dados das pessoas por Nome e Idade, e imprimam todos os dados na tela. Finalmente, demonstre o uso destas funções.

16.7 Estruturas e funções

Estruturas são utilizadas em funções da mesma forma que os tipos básicos que vimos (`int`, `float`, `char`, ...). Veja o exemplo a seguir:

```
#include<iostream>

using namespace std;

typedef struct pessoa pessoa;
struct pessoa
{
  char nome[40];
  char sobrenome[40];
  int idade;
    float salario;
};
pessoa maisVelho(pessoa p1, pessoa p2)
{
  return p1.idade > p2.idade ? p1 : p2;
}

int main()
{
  pessoa pes1, pes2, mv;
  cout << ``entre nome, sobrenome, idade e salario da primeira pessoa:
      '';
  cin >> pes1.nome >> pes1.sobrenome >> pes1.idade >> pes1.salario;
  cout << ``entre nome, sobrenome, idade e salario da segunda pessoa: '
      ';
  cin >> pes2.nome >> pes2.sobrenome >> pes2.idade >> pes2.salario;
  mv = maisVelho(pes1, pes2);
  cout << ``O mais velho eh '' << mv.nome << `` '' << mv.sobrenome <<
      ``, com '';
  cout << mv.idade << `` anos e salario de '' << mv.salario << endl;
  return 0;
}
```

De maneira análoga, vetores de estruturas devem ser utilizados da mesma forma que vetores dos tipos básicos:

```
#include<iostream>
#define T 5
using namespace std;

typedef struct pessoa pessoa;
struct pessoa
{
  char nome[40];
  char sobrenome[40];
  int idade;
  float salario;
};

void lePessoas(pessoa p[])
{
  for(int i = 0; i < T; i++)
  {
    cout << ``entre nome, sobrenome, idade e salario da pessoa '' << (i
        +1) << ``:'' << endl;
    cin >> p[i].nome >> p[i].sobrenome >> p[i].idade >> p[i].salario;
  }
}

int maisVelho(pessoa p[])
{
  int idade = p[0].idade, posicao = 0;
  for(int i = 1; i < T; i++)
  {
    if(p[i].idade > idade)
    {
      idade = p[i].idade;
      posicao = i;
    }
  }
  return posicao;
}

int main()
{
  pessoa p[T];
  int pos;
  lePessoas(p);
  pos = maisVelho(p);
  cout << ``O mais velho eh '' << p[pos].nome << `` '' << p[pos].
      sobrenome << ``, com '';
  cout << p[pos].idade << `` anos e salario de '' << p[pos].salario <<
      endl;
  return 0;
}
```

16.8 Laboratório

Laboratório 16.1 *Considerando a estrutura:*

```
struct Ponto {
  int x;
  int y;
};
```

para representar um ponto em uma grade 2D, implemente uma função que indique se um ponto `p` *está localizado dentro ou fora de um retângulo. O retângulo é definido por seus vértices inferior esquerdo* `v1` *e superior direito* `v2`*. A função deve retornar 1 caso o ponto esteja localizado dentro do retângulo e 0 caso contrário.*

Laboratório 16.2 *Faça um função que recebe um valor em segundos e retorna o valor equivalente para o tipo horário, o qual é composto por hora, minuto e segundos. Em seguida imprima os campos da estrutura retornada.*

Laboratório 16.3 *Crie a estrutura baralho, baseado em um "baralho tradicional" (cada carta tem seu naipe e seu valor). Implemente uma função que faça parte de distribuição (sorteio) de cartas para 2 jogadores, considerando que cada jogador irá receber 5 cartas. Exiba na tela as cartas que cada um dos jogadores recebeu.*

Laboratório 16.4 *Faça um programa que controla o consumo de energia dos eletrodomésticos de uma casa. Leia um inteiro* `n` *e, utilizando funções auxiliares:*

- *Crie e leia* `n` *eletrodomésticos que contém nome (máximo 15 letras), potencia (real, em kW) e tempo ativo por dia (real, em horas).*
- *Leia um tempo* `t` *(em dias), calcule e mostre o consumo total na casa e o consumo relativo de cada eletrodoméstico (consumo/consumo total) nesse período de tempo. Apresente este último dado em porcentagem.*

Capítulo 17

Referências

O uso da analogia da memória do computador com uma planilha eletrônica nos permitiu entender como variáveis simples e também arranjos são alocados em nossos programas. Agora que já conhecemos vetores, usaremos uma analogia mais precisa para entendermos referências, ou como são mais popularmente conhecidas, os famigerados ponteiros.

17.1 A memória é um grande vetor

A memória de um computador pode ser vista como um grande vetor de bytes. O primeiro byte é denominado byte 0, o segundo, byte 1 e assim por diante até o tamanho da memória disponível ao seu programa.

Quando em seu programa você, por exemplo, declara uma variável do tipo caractere (`char`) com o nome `c`, o compilador separa um byte da memória, digamos o byte 7542, e passa a chamá-lo de `c`. Assim, toda vez que se referir à variável `c` em seu programa, o computador sabe que está se referindo ao byte 7542. Neste caso, dizemos que o endereço da variável `c` é 7542.

Similarmente, quando você aloca uma variável do tipo inteiro, o compilador reserva quatro bytes para você, digamos, os bytes 8012, 8013, 8014 e 8015. Tal variável tem o endereço 8012 (como os tipos tem sempre tamanho fixo, o endereço do primeiro byte mais a informação sobre o tipo da variável é suficiente para o computador determinar de onde até onde na memória cada variável fica armazenada).

Finalmente, quando você aloca um vetor de tamanho n de algum tipo qualquer, o compilador reserva um espaço de tamanho $n \times t$, onde t é o tamanho do tipo em questão. Se são 10 inteiros, por exemplo, então o vetor terá tamanho 40 bytes. O endereço desta variável é o endereço do primeiro byte da primeira célula. Assim, quando você, por exemplo, acessa a posição 5 de um vetor de inteiros cujo endereço é 1000, o computador faz o seguinte cálculo para determinar a posição de memória que você está acessando, sabendo que cada inteiro ocupa 4 bytes: $1000 + 5 * 4$.

17.2 Variáveis do Tipo Referência

Em certas situações, o poder de se referir a posições de memória pelo seus endereços em vez de por um nome de variável é muito útil. Por exemplo, imagine que precisemos calcular uma série de estatísticas sobre os dados em um vetor. Neste caso, gostaríamos de definir uma função que analisasse o vetor e desse como resultados as várias estatísticas, como média, desvio padrão, variância, intervalo de confiança etc. Entretanto, como bem sabemos, funções em C(++) retornam um único resultado. Se, contudo, pudéssemos passar como parâmetro para a função as referências de onde gostaríamos que os resultados fossem colocados, então a função poderia colocá-los diretamente nestas posições de memória. Esse mecanismo é o que chamamos de passagem de parâmetro por referência.

Antes de se poder exercitar a passagem de parâmetros por referência, contudo, precisamos aprender a declará-las, a obter referências para posições de memória, e a usar tais referências. Estas operações são feitas pelos operadores `*` e `&`. A declaração de variáveis segue a seguinte sintaxe:

```
tipo  *identificador;
```

Assim, para se declarar variáveis de referência para uma posições de memória que armazenam um inteiro, um caractere, e um float, você usaria as seguintes declarações:

```
int *ref_int;
char *ref_char;
float *zabumba;
```

Observe que o `*` é sempre considerado como ligado ao identificador e não ao tipo da variável. Assim, a linha de código

```
int *a, b, *c;
```

declara duas referências para inteiro (`a` e `c`), e um inteiro (`b`). Para se evitar confusões, sugere-se que não se declare variáveis do tipo referência e não referência em conjunto. Ou seja, a código acima deveria ser reescrito como

```
int *a, *c;
int b;
```

O operador `&` permite que se extraia o endereço (ou uma referência) de uma variável. No código,

```
int *a;
int b;

a = &b;
```

a recebe o endereço da variável b. Logo, se b está nos bytes 2012, 2013, 2014 e 2015, a variável a recebe o valor 2012.

Aqui, gostaríamos de chamar a atenção para o seguinte fato: *em raríssimas situações é necessário saber o endereço de uma variável. Na maior parte dos casos, basta saber que uma variável tem tal endereço.*.

Finalmente, o operador * também nos permite acessar o conteúdo de uma variável referenciada, em do valor da referência. No código,

```
int *a;
int b;
int c;

a = &b;
*a = 20;
c = *a;
c++;
```

a variável b termina com o valor 20, enquanto que a variável c termina com o valor 21.

17.3 Passagem de Referências como Parâmetros

Variáveis do tipo referência, como outros tipos de variáveis, podem ser passadas como parâmetro em chamadas de função. A grande vantagem desta técnica, também conhecida como passagem de parâmetros por referência, é a possibilidade de modificação da memória para qual a referência foi passada. Por exemplo, analise o seguinte código.

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

using namespace std;

void sen_cos_tan(float val, float *seno, float *cosseno, float *
    tangente)
{
    *seno = sin(val);
    *cosseno = cos(val);
    *tangente = tan(val);
}

int main()
{
    float v = 0, s, c, t;

    sen_cos_tan(v, &s, &c, &t);

    cout << "seno: " << s << " cosseno: " << c << " tangente: "<< t;

    return 0;
}
```

Esse programa calcula em uma única chamada, os valores do seno, cosseno e tangente da variável `v`. Enquanto o feito não parece muito impressionante, pense no caso em que você precisa calcular diversas estatísticas de um vetor de números. Neste caso, devido à forte dependência do cálculo de uma estatística no cálculo de outras, esta função poderia simplificar em muito o seu código.

17.4 Laboratório

Laboratório 17.1 *Escreva um programa que contenha funções que calculam a média, variância e desvio padrão de um vetor de números reais (uma função por dado estatístico).*

Laboratório 17.2 *Altere o programa do exercício anterior para que calcule as três estatísticas em uma função apenas, reaproveitado os cálculos uns dos outros.*

Capítulo 18

Referências II

Continuando com o estudo de ponteiros, vamos ver como utilizá-los para referenciar estruturas e arranjos.

18.1 Ponteiros para Structs

Se você recordar-se da anologia apresentada na seção 17.1, em que a memória é comparada a um grande vetor e cada tipo de dados tem um tamanho pré-definido, é fácil entender como uma estrutura é representada na memória.

Considere a seguinte estrutura e responda: qual o tamanho desta estrutura na memória do computador?

```
typedef struct
{
  int i;
  char c;
  float j;
} exemplo1;
```

Para responder a esta pergunta precisamos entender que uma estrutura ocupa espaço equivalente à soma do espaço ocupado por cada um de seus campos. Desta forma, o espaço total ocupado por uma variável do tipo `exemplo1` será a soma dos tamanhos de 1 `int`, 1 `char` e 1 `float`, ou seja, $4 + 1 + 4 = 9$ bytes.

Assim, e você cria uma variável `exemplo1 v[10]`, o compilador deve reservar um espaço de $10 \times 9 = 90$ bytes.

Assim como nos referimos a variáveis dos tipos primitivos (inteiro, real, caractere) utilizando ponteiros, podemos também utilizar ponteiros para acessar uma variável do tipo `struct` e seus campos. Veja o exemplo para a estrutura declarada anteriormente.

```
exemplo1 a = {-5, 'z', 0.89}, b;
exemplo1 *p;
p = &a;
cout << (*p).i << endl;
```

```
p = &b;
cin >> (*p).c;
```

Este trecho mostra na linha 1 a criação de duas variáveis do tipo `exemplo1` e na linha 2 a criação de uma variável do tipo ponteiro para `exemplo1`. As linhas 3 e 5 atualizando o ponteiro, fazendo com que referencie as variáveis `a` e `b`, respectivamente. Desta maneira, a linha 4 irá imprimir o valor de `a.i`, enquanto a linha 6 irá ler a partir do teclado um valor para `b.c`.

Quando usamos ponteiros e estruturas, aparece o incoveniente de termos que digitar toda vez entre parênteses a variável do tipo ponteiro (`(*p).c`). Para simplificar a utilização de ponteiros no acesso a campos de uma estrutura, a linguagem C permite escrever esta mesma instrução de uma maneira mais fácil de digitar: `p->c`. O código anterior pode ser reescrito como:

```
exemplo1 a = {-5, 'z', 0.89}, b;
exemplo1 *p;
p = &a;
cout << p->i << endl;
p = &b;
cin >> p->c;
```

Por fim, a passagem de estruturas por referência em funções auxiliares é feita da mesma forma que vimos no capítulo anterior:

```
  void inverte_structs(exemplo1 *s1, exemplo2 *s2) {
    exemplo1 aux
    aux.i = s2->i;
    s2->i = s1->i;
    s1->i = aux.i;
    aux.c = s2->c;
    s2->c = s1->c;
    s1->c = aux.c;
    aux.j = s2->j;
    s2->j = s1->j;
    s1->j = aux.j;
  }

  int main()
  {
    exemplo1 a = {1, 'a', 1.0}, b = {2, 'b', 2.0};
    inverte_structs(&a, &b);
    cout << a.i << ", " << a.c << "," << a.j << endl;
    cout << b.i << ", " << b.c << "," << b.j << endl;
    return 0;
  }
```

A função auxiliar troca os valores de duas variáveis do tipo `exemplo1` passadas por referência. Após a invocação da função auxiliar a partir da `main` os valores iniciais das variáveis `a` e `b` serão trocados.

18.2 Arranjos e Ponteiros

Até agora vimos vetores sem nenhuma relação com ponteiros. No fundo, quando criamos um vetor com a instrução `int a[10]`, estamos dizendo ao compilador para reservar na memória espaço para 10 inteiros (40 bytes) e armazenar o endereço em que começa o vetor na variável `a`. Em outras palavras, a variável `a` é um *ponteiro* para o primeiro elemento do vetor.

Por esta razão, a relação entre vetores e ponteiros é mais direta e mais fácil de utilizar. Uma vez que a variável que usamos contém o endereço do começo do vetor, um ponteiro pode receber diretamente seu valor, sem necessidade do operador &. Considere o exemplo a seguir:

```
float a[10];
float *p;
p = a;
cout << "Digite 10 numeros reais: ";
for(int i = 0; i < 10; i++)
  cin >> a[i];
cout << "Num. digitados: ";
for(int i = 0; i < 10; i++)
  cout << p[i];
```

Observe a linha 3 e perceba como a atualização da referência do ponteiro `p` para o vetor `a` não utilizou o &. Isto porque `a` também é um ponteiro. A diferença entre `p` e `a` é que a primeira pode ter sua referência atualizada, enquanto a segunda não. A partir da linha 3, o vetor pode ser acessado tanto pela variável `a` (linha 6) quanto pela variável `p` (linha 9).

É por este motivo que vetores sempre são passados por referência em funções auxiliares, sem a necessidade do & e do *.

18.2.1 Percorrendo vetores com ponteiros

Já sabemos a relação entre ponteiros e vetores. Veremos a seguir algumas maneiras diferentes de acessar os elementos de um vetor utilizando ponteiros. Considere o trecho de um programa a seguir:

```
char str[100];
char *s;
int totalletras = 0;
s = str;
cout << "entre uma frase com letras minusculas" << endl;
cin.getline(s,100);
for(int i = 0; i < strlen(s); i++)
{
  if(s[i] >= 'a' && s[i] <= 'z')
  {
    totalletras++;
  }
}
cout << "O total de letras da frase eh " << totalletras;
```

Neste trecho utilizamos o ponteiro `s` como o vetor `str`, pois a partir da linha 3 ele "aponta" para o vetor. Este mesmo trecho pode ser reescrito:

```
1  char str[100];
2  char *s;
3  int totalletras = 0;
4  s = str;
5  cout << "entre uma frase com letras minusculas" << endl;
6  cin.getline(s,100);
7  for(int i = 0; i < strlen(s); i++)
8  {
9    if( *(s+i) >= 'a' && *(s+i) <= 'z' )
10   {
11     totalletras++;
12   }
13 }
14 cout << "O total de letras da frase eh " << totalletras;
```

Observe a linha 9 de cada um dos dois trechos anteriores e veja a diferença na notação. Quando utilizamos ponteiro, podemos acessar um vetor pelo índice (colchetes) ou por um deslocamento sobre o valor do endereço inicial do vetor armazenado em `s`, ou seja, a notação `*(s+i)`, significa o caractere localizado `i` caracteres a partir do caractere na primeira posição do vetor.

O mesmo trecho pode ser reescrito ainda de uma terceira forma:

```
1  char str[100];
2  char *s;
3  int totalletras = 0;
4  s = str;
5  cout << "entre uma frase com letras minusculas" << endl;
6  cin.getline(s,100);
7  for(int i = 0; i < strlen(str); i++) //Observe que aqui usamos str e
       nao s. Por que?
8  {
9    if( *s >= 'a' && *s <= 'z' )
10   {
11     totalletras++;
12   }
13   s++;
14 }
15 cout << "O total de letras da frase eh " << totalletras;
```

Qual a diferença entre esta versão e as versões anteriores? Qual a diferença entre `s++` e `(*s)++`? Qual o valor de `s[0]` em cada versão? E de `*s`? E de `str[0]`?

18.3 Laboratório

Laboratório 18.1 *Crie uma estrutura com os campos nome, idade e salario dos tipos string, inteiro e real, respectivamente.*

Em seguida, crie uma função que receba referências para duas variáveis de tal tipo e uma variável correspondente a uma porcentagem de aumento que deve ser aplicada sobre o campo salário de cada estrutura. A função principal deve ler cada campo das duas variáveis e imprimir o novo salário.

Finalmente, crie uma função que recebe duas variáveis de tal tipo e que troque os valores de todos os campos das duas estruturas. A função principal deve agora imprimir as estruturas antes e depois da troca.

Laboratório 18.2 *Execute as três versões do programa da seção 18.2.1 e veja se há diferença no resultado. Responda às perguntas no final da seção.*

Capítulo 19

Alocação Dinâmica

Por mais interessante que seja poder criar ponteiros para variáveis já existentes e passar parâmetros por referência, o poder dos ponteiros só fica mesmo evidente quando se trabalha com alocação dinâmica de memória.

Alocação dinâmica é a habilidade de se criar novas variáveis durante a execução do programa, sem que elas tenham sido declaradas antes. Justamente por não terem sido declaradas antes da compilação, estas variáveis alocadas dinamicamente não têm nomes e, assim, só podem ser acessadas por meio de referências.

19.1 Alocação Dinâmica de Tipos Simples

Existem várias formas de se alocar memória dinamicamente, todas elas via funções que retornam como resultado uma referência para o começo da memória alocada, na forma de uma referência genérica.

A função mais simples para alocação de memória é a `malloc`. Esta função recebe como único parâmetro o tamanho da memória a ser alocada, em bytes. Observe o seguinte código para alguns exemplos.

```
char *s1;
int *i1;
float *f1;

s1 = (char *) malloc(1);
i1 = (int *) malloc(4);
f1 = (float *) malloc(4);
```

No exemplo acima você percebe que antes de cada `malloc` há, entre parênteses, o tipo da variável que guardará a referência. Isso é o que chamamos de *casting*, e serve para transformar a referência genérica retornada pela função para o tipo de referência da variável. **Este *casting* é obrigatório em seu código**.

Você também percebe que cada alocação especifica exatamente o tamanho do tipo para o qual estamos alocando a memória. Isto é, 1 para caracteres e 4 para inteiros

e números reais. Estes tamanhos, contudo, podem variar de uma máquina/sistema operacional para outra, o que tornaria complicado especificar tais tamanhos em cada invocação à `malloc`. É por isso que a linguagem C(++) especifica o construto `sizeof`, que "descobre" o tamanho do tipo para você, conforme o seguinte exemplo.

```
char *s1;
int *i1;
float *f1;
double *d1;

s1 = (char *) malloc(sizeof(char));
i1 = (int *) malloc(sizeof(int));
f1 = (float *) malloc(sizeof(float));
d1 = (double *) malloc(sizeof(double));
```

19.2 Alocação Dinâmica de Vetores

Como visto nos capítulos anteriores, a diferença entre referências para tipos simples e vetores é... **nenhuma**! Isso quer dizer que podemos alocar vetores usando um código similar ao anterior, alterando apenas a quantidade de memória alocada, como no seguinte exemplo.

```
char *s2;
int *i2;
float *f2;
double *d2;

s2 = (char *) malloc(sizeof(char) * 100);
i2 = (int *) malloc(sizeof(int) * 10);
f2 = (float *) malloc(sizeof(float) * 6);
d2 = (double *) malloc(sizeof(double) * 8);

cin.getline(s2,100);
cout << s2;

for(int i = 0; i < 10; i++)
  cin >> i2[i];

for(int i = 0; i < 10; i++)
  cout << i2[i] << endl;


for(int i = 0; i < 6; i++)
  cin >> f2[i];

for(int i = 0; i < 6; i++)
  cout << f2[i] << endl;


for(int i = 0; i < 8; i++)
  cin >> d2[i];
```

```
for(int i = 0; i < 8; i++)
  cout << d2[i] << endl;
```

19.3 Liberação de memória

Todos os exemplos mostrados até agora neste capítulo são incompletos. Isso por quê nenhum programa que faça alocação dinâmica em C(++) pode ser completo sem que se faça a liberação da memória alocada. Liberar ou "desalocar" a memória consiste simplesmente na invocação da função `free`, passando-se como parâmetro a referência a ser desalocada. O uso de `free` será exemplificado na próxima seção.

19.4 Alocação Dinâmica de Estruturas

Assim como declarar uma referência para uma estrutura é tão simples quanto declarar uma referência para um tipo primitivo(simples), alocar um vetor de estruturas dinamicamente é tão simples quanto alocar um vetor de tipos primitivos dinamicamente. O exemplo a seguir mostra exatamente como alocar um vetor com 10 estruturas do tipo definido.

```
#define NUM_PESSOAS 10
typedef struct
{
  char prim_nome[20],
  char ult_nome[20],
  int idade,
  float salario,
  char regiao,
  char sexo
} pessoa_t;

int main()
{
  pessoa_t *pessoas;

  pessoas = (pessoa_t *) malloc(sizeof(pessoa) * NUM_PESSOAS);

  for(int i = 0; i < NUM_PESSOAS; i++)
  {
      cin >> pessoas[i].prim_nome;
      cin >> pessoas[i].ult_nome;
      cin >> pessoas[i].idade;
      cin >> pessoas[i].salario;
      cin >> pessoas[i].sexo;
      cin >> pessoas[i].regiao;
  }

  //Uso dos dados
  ...

```

```
  //Liberacao da memoria
32  free(pessoas);

34  return 0;
}
```

19.5 Exercícios

19.5.1

Escreva um programa que leia um numero inteiro N, aloque dinamicamente um vetor de N inteiros, leia cada um dos N inteiros, e imprima os N inteiros na tela.

19.5.2

Escreva um programa que repita o seguinte procedimento X vezes: leia um numero inteiro N, aloque dinamicamente um vetor de N carateres, leia uma palavra de N caracteres, transforme todas as maiúsculas em minúsculas e vice-versa na palara e imprima o resultado na tela. X deve ser lido no início da execução do programa.

19.6 Laboratório

19.6.1

Implemente os exercícios acima.

19.6.2

Escreva um programa que aloque um vetor de 100000 float e imprima os 10 primeiros e os 10 últimos (o lixo que estiver na memória). [1]

[1] Se tudo funcionar, aumente o vetor para 1000000 e assim por diante. O objetivo neste laboratório é mostrar que a quantidade de memória disponível para seu programa é limitada, e que se extrapolada, seu programa incorrerá em um erro.

Capítulo 20

Arquivos

Muitas vezes desejamos guardar o resultado de alguma computação para consulta posterior. Imagine, por exemplo, que você fez um programa que calcula a média final e situação de toda a turma. Se o resultado deste processamento não puder ser armazenado, toda vez que for necessário consultar a situação de algum aluno, o programa deverá ser executado e todos os dados inseridos novamente. Para solucionar este problema, o usuário pode salvar resultados de computação em uma estrutura persistente. Esta estrutura de dados manipulada fora do ambiente do programa (memória principal) é conhecida como **arquivo**.

Um arquivo é armazenado em um dispositivo de memória secundária (CD, DVD, disco rígido, pendrive) e pode ser lido ou escrito por um programa. Em C, um arquivo pode representar diversas coisas, como documentos, uma impressora, um teclado, ou qualquer dispositivo de entrada ou saída. Consideraremos apenas dados em disco, iniciando por dados na forma textual.

20.1 Arquivos de texto

Nesta seção estudaremos apenas arquivos texto, ou seja, arquivos que contêm apenas caracteres e podem ser visualizados em editores de textos como Notepad, Gedit, Vim, etc.

A linguagem C++ dá suporte à utilização de arquivos por meio da biblioteca `fstream` . Esta biblioteca fornece várias funções para manipulação de arquivos e define alguns tipos de dados para serem usados especificamente com arquivos. O principal tipo definido nessa biblioteca que será usado é o tipo `fstream`. Um variável do tipo `fstream` é capaz de identificar um arquivo no disco, direcionando-lhe todas as operações. Essas variáveis são declaradas da seguinte maneira:

```
fstream arq;
```

20.1.1 Abertura e Fechamendo de Arquivos

Antes que o arquivo seja manipulado, é preciso abrí-lo, o que é feito via "função" `open` do arquivo. Uma vez aberto, o arquivo funciona como o `cin` e `cout`, com os quais você já está acostumado a usar `<<` e `>>`, como no exemplo a seguir. Ao terminar o uso do arquivo, é importante fechá-lo, para garantir que tudo que, em teoria, está escrito no arquivo, realmente foi colocado no disco.

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  fstream arquivo;
  arquivo.open ("arquivo.txt");
  arquivo << "O que eu deveria escrever neste arquivo?" << endl;
  arquivo.close();
  return 0;
}
```

Nem sempre é possível abrir o arquivo como no exemplo, o que será explicado adiante. Para testar se o arquivo foi aberto ou não, use a função `is_open()`. É importante que você teste se o arquivo realmente está aberto, antes de tentar usá-lo.

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  fstream arquivo;
  arquivo.open("arquivo.txt");

  if(arquivo.is_open())
  {
     arquivo << "O que eu deveria escrever neste arquivo?" << endl;
     arquivo.close();
     cout << "tudo escrito";
  }
  else
     cout << "falhou";

  return 0;
}
```

Se você tentou executar o código acima, percebeu que a mensagem "falhou"foi escrita na tela. O problema é que o código abre um arquivo e escreve no mesmo, mas somente se ele já existir, o que não é o caso. Por enquanto, vá até a pasta onde está o seu projeto e crie o arquivo manualmente. Na seção seguinte veremos como criar um arquivo "de dentro" do programa. Já no código a seguir, veja como os dados do arquivo podem ser lidos.

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  fstream arquivo;
  arquivo.open ("arquivo.txt");

  if(arquivo.is_open())
  {
    cout << "yeah";
    arquivo << "Sera que isso foi escrito?" << endl;
    arquivo.close();

    arquivo.open ("arquivo.txt");
    char str[1000];
    arquivo.getline(str,1000);
    arquivo.close();

    cout << str;
  }
  else
    cout << "O arquivo nao foi aberto";

  return 0;
}
```

20.1.2 Criação de Arquivos

Para abrir um arquivo que não existe, ou seja, criar o arquivo, é necessário passar um conjunto especial de instruções para a função de abertura. Infelizmente, o único jeito de passar estas instruções é bem mais complicado do que o que vimos até agora. Veja o seguinte código, em especial a linha que abre o arquivo.

```
#include <iostream>
#include <fstream>
using namespace std;

int main () {
  fstream arquivo;
  arquivo.open ("arquivo.txt", ios::out | ios::in | ios::trunc);

  if(arquivo.is_open())
    cout << "yeah";
  else
    cout << "nope";

  arquivo << "O que eu deveria escrever neste arquivo?" << endl;
  arquivo.close();

  arquivo.open ("arquivo.txt");
  char str[1000];
  arquivo.getline(str,1000);
```

```
20   arquivo.close();

22   cout << str;

24   return 0;
}
```

Na frente do parâmetro com o nome do arquivo, na função `open`, foram passadas três opções para o modo de abertura do arquivo, que especifica o tipo de operações que se intende fazer no arquivo (o caractere `|` combina as opções passadas). Algumas das opções possíveis são:

- ios::in – Operações de escrita.
- ios::out – Operações de leitura.
- ios::app – Posiciona no fim do arquivo. Não pode ser reposicionado. Não pode ser usado com ios::in.
- ios::trunc – Posiciona no início do arquivo. Se o arquivo já existe, então seu conteúdo anterior é perdido.

As opções para leitura e escrita do arquivo, e somente estas, são implícitas, quando nenhuma opção é especificada. Se você quiser, pode usar apenas uma destas por questões de segurança (o arquivo pode ser lido mas não escrito, ou o arquivo pode ser aumentado mas não lido). Usando somente as opções de leitura e escrita, contudo, o arquivo não é criado caso não exista. Para que seja criado, é necessário que se especifique ou `ios::trunc` ou `ios::app`, que farão com que o arquivo seja ou truncado ou que toda modificação seja adicionada ao fim do arquivo (*append*).

20.1.3 Cuidados com a Formatação dos Dados

Quando se escreve um dado em um arquivo de texto, precisa-se entender que o dado ocupa somente o espaço que você especificar. Isto é, se você escrever a sequência de caracteres “C(++) é bom!!!”, 14 caracteres serão escritos, e pronto. Se escrever algo na linha seguinte depois resolver mudar a string acima para “C(++) não é bom!!!”, então a divisão entre as linhas (o `endl`) terá sido sobrescrito e as duas linhas se tornarão uma. Estes cuidados não precisam ser tomados agora, com os exemplos simples com os quais estamos lidando, mas nos próximos capitulos você terá que tomá-los. Assim, melhor que você já saiba sobre estes problemas desde agora.

```
#include <iostream>
2 #include <fstream>
using namespace std;
4
int main () {
6   fstream arquivo;
   arquivo.open ("arquivo.txt", ios::out | ios::in | ios::trunc);
8   char str[1000];
```

```
  if(arquivo.is_open())
  {
        cout << "O arquivo foi aberto!";
        arquivo << "Escrevendo um numero pequeno:" << endl;
        arquivo << 10 <<endl;
        arquivo << "Numero pequeno escrito." << endl;

        cout << "Fechando o arquivo.";
        arquivo.close();

        cout << "Reabrindo o arquivo.";

        arquivo.open ("arquivo.txt");

        arquivo << "Escrevendo um numero pequeno:" << endl;
        arquivo << 10000 <<endl;

        cout << "Fechando o arquivo.";
        arquivo.close();

        cout << "Reabrindo o arquivo.";
        arquivo.open ("arquivo.txt");

        arquivo.getline(str,1000);
        cout << str;
        arquivo.getline(str,1000);
        cout << str;
        arquivo.getline(str,1000);
        cout << str;

        arquivo.close();
  }

  else
    cout << "nope";

  return 0;
}
```

20.2 Laboratório

Laboratório 20.1 *Faça um programa que leia o nome e sobrenome de 30 alunos e armazene em um arquivo, de tal forma que o arquivo tenha um aluno por linha. Abra o arquivo usando um editor de textos qualquer, como o Notepad.*

Laboratório 20.2 *Faça um programa que tente abrir um arquivo e, caso não consiga, tente criá-lo e abrí-lo. Com arquivo aberto, leia um vetor A de inteiros de tamanho 20 e guarde seus valores em um arquivo, um por linha. Em seguida, reabra o arquivo e leia os elementos para o vetor B, verificando se os valores foram gravados corretamente.*

Laboratório 20.3 *Faça um programa em Linguagem C que receba do usuário um arquivo, e mostre na tela quantas linhas esse arquivo possui.*

Laboratório 20.4 *Faça um programa que receba dois arquivos do usuário, e crie um terceiro arquivo com o conteúdo dos dois primeiros juntos (o conteúdo do primeiro seguido do conteúdo do segundo).*

Laboratório 20.5 *Desenvolver um programa em C que lê o conteúdo de um arquivo e cria um arquivo com o mesmo conteúdo, mas com todas as letras minúsculas convertidas para maiúsculas. Os nomes dos arquivos serão fornecidos, via teclado, pelo usuário. A função que converte minúscula para maiúscula é o toupper(). Ela é aplicada em cada caractere da string.*

Laboratório 20.6 *Faça um programa no qual o usuário informa o nome do arquivo, e uma palavra, e retorne o número de vezes que aquela palavra aparece no arquivo.*

Capítulo 21

Navegação dentro do arquivo

Embora seja possível trabalhar com arquivos usando apenas as ferramentas apresentadas até agora, alguns problemas são de difícil resolução, a não ser que extendamos nosso conhecimento.

Suponha por exemplo que você tenha o seguinte arquivo:

```
O rato roeu a roupa do
rei de Roma e a rainha
Ruinha resolveu rir-se
pois o rei que remente
suas roídas roupagens.
```

Observe que todas as linhas tem o mesmo tamanho (22 caracteres) e que a palavra "remende" está escrita de forma incorreta. Para corrigir esta palavra, podemos reescrever toda a frase, ou posicionar o cursor do arquivo exatamente sobre o caractere errado e corrigi-lo. Há várias formas de se fazer isso, como veremos agora.

21.1 Posicionamento no arquivo

Como visto no capítulo anterior, operações de leitura e escrita no arquivo vão mudando a posição em que as próximas operações serão executadas. Isto é, se considerarmos que há um cursor dentro do arquivo, na posição em escreveremos, caso uma escrita seja feita, ou leremos, caso uma leitura seja executada, então se lermos ou escrevermos 3 caracteres, então o cursor "andará" 3 caracteres. Se quisermos colocar o cursor no fim do arquivo, podemos reabrí-lo usando `ios::app`. Caso queiramos colocar o cursor no começo do arquivo, podemos simplesmente reabri-lo sem especificar qualquer opção ou usando `ios::trunc` (e com isso também zerarmos nosso arquivo).

Caso precisemos posicionar o cursor no meio do arquivo, podemos usar a função `seekg`, que pode ser usada duas formas:

```
seekg(nova posição [em bytes])

seekg(diferença [em bytes] em relação ao marco, marco)
```

Na primeira forma você pode especificar onde o cursor deve ser posicionado em relação ao início do arquivo, em número de bytes. Por exemplo, `arq.seekg(10)` posiciona o cursor no décimo byte do arquivo `arq`. Já pela segunda forma, você especifica onde posicionar o cursor em relação a um marco no arquivo, que pode ser seu início (`ios_base::beg`), fim (`ios_base::end`), ou posição atual (`ios_base::cur`). A diferença em relação ao marco é especificada como um inteiro com sinal. Por exemplo, `arq.seekg(-10,ios_base::end)` posiciona o cursor no décimo caractere *antes do fim* do arquivo `arq`.

Para resolver o problema apresentado acima, podemos então usar as seguintes soluções.

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    fstream arq;
    arq.open("arquivo.txt"); //Posicionando o cursor no inicio.
    arq.seekg(23*3+20);
    arq << 'd';
    arq.close();


    arq.open("arquivo.txt"); //Buscando a partir do fim.
    arq.seekg(-23-3, ios_base::end);
    arq << 'd';
    arq.close();

    arq.open("arquivo.txt", ios::out | ios::trunc); //Recriando o
        arquivo.
    arq << "O rato roeu a roupa do" << endl
        << "rei de Roma e a rainha" << endl
        << "Ruinha resolveu rir-se" << endl
        << "pois o rei que remende" << endl
        << "suas roidas roupagens.";
    arq.close();

    return 0;

}
"
```

21.2 Arquivos formatados

Para que se possa movimentar o cursor dentro de um arquivo, é importante que se conheça a estrutura do mesmo, principalmente se pretender sobrescrevê-lo sem destruí-lo. Por exemplo, considere o seguinte problema.

Escrever um programa que leia matrícula e nome de alunos do curso de Engenharia Química e armazene estas informações em um arquivo. Em seguida, seu programa deve escrever todas as informações na tela, precedidas por um número identificando cada aluno. Finalmente, o programa deve ler números correspondentes a alunos e sobrescrever os dados do aluno com novas informações.

```
#include <iostream>
#include <fstream>

#define NOME_ARQ "arquivo.txt"

using namespace std;


bool mais_info()
{
    char resp = 'x';
    cout << "Entrar com mais informacoes? (s/n)";

    while(true)
    {
        cin >> resp;

        switch(resp)
        {
        case 's':
        case 'S':
            return true;
        case 'n':
        case 'N':
            return false;
        }
    }
}

void ler_aluno(char mat[], int tmat, char nome[], int tnome)
{
    cin.ignore(1000,'\n');
    cin.clear();
    cout << "Digite a matricula (max " << tmat-1 << " caracteres:)";
    cin.getline(mat,tmat);
    cout << "Digite o nome (max " << tnome-1 << " caracteres";
    cin.getline(nome,tnome);
}

int main()
{
    fstream arq;

    int totalcount = 0;
    int count;
    char linha[1000];

    char mat[15];
    char nome[100];

    arq.open(NOME_ARQ);
```

```
    if(! arq.is_open())
53      arq.open(NOME_ARQ, ios::in | ios::out | ios::trunc);

55

    while(mais_info())
57  {
        ler_aluno(mat, 15, nome, 100);
59      arq << mat << " " << nome << endl;
    }
61
    count = 1;
63  arq.clear();
    arq.seekg(0);
65
    cout << "Dados atuais" << endl;
67  while(arq.getline(linha,1000))
    {
69
        cout << count++ << " - "<< linha << endl;
71      totalcount++;
    }
73
    while(mais_info())
75  {
        cout << "Sobrescrever qual linha?";
77      cin >> count;
        if(count > totalcount)
79      {
            cout << "Linha nao existente";
81      }
        else
83      {
            ler_aluno(mat, 15, nome, 100);
85
            arq.clear();
87          arq.seekg(0);
            while(count > 1)
89          {
                arq.getline(linha,1000);
91              count--;
            }
93          arq << mat << " " << nome << endl;
        }
95  }

97


99

    arq.clear();
101 count = 1;
    arq.seekg(0, arq.beg);
103 cout << "Dados atuais" << endl;
    while(arq.getline(linha,1000))
105 {
        cout << count++ << " - "<< linha << endl;
107 }
```

```
109     return 0;
}
```

Este código possui duas particularidades. Primeiro, ele usa a função `arq.clear()`, ainda não mencionada e, segundo, ele está errado.

Toda vez que o arquivo é lido até o seu fim, seu estado interno é mudado para identificar o fato. Isto é verificável pela função `arq.eof()`, que testa se o arquivo chegou ao *end of file*. Quando isso é verdade, as funções de reposicionamento não funcionarão, a não ser que este status seja reiniciado, o que é feito com a função `clear()`. No código simplesmente usamos esta função sempre que formos fazer um *seek*, por simplicidade.

Sobre o código não funcionar como esperado, execute-o e insira um aluno apenas, com matrícula "aaaaaaaaaa"e nome "bbbbbbbbbb". Em seguida, altere os dados da linha 1 e insira matrícula "cc"e nome "dd". Ao final, os dados impressos serão os seguintes:

```
Dados atuais
1 - cc dd
2 - aaaa bbbbbbbbbb
```

21.3 Exercícios

21.3.1

Corrija o programa exemplo mostrado acima para que escreva sempre linhas de um mesmo tamanho e que, portanto, não tenha o problema de fragmentação de registros.