UENP - UNIVERSIDADE ESTADUAL DO NORTE DO PARANÁ
***CAMPUS* LUIZ MENEGHEL**
Centro de Ciências Tecnológicas

Programação I

Notas de Aula

Prof. José Reinaldo Merlin
Prof. Estevan Braz Brandt Costa
Prof. Wellington Della Mura

Bandeirantes – 2013

Sumário

Capítulo 0

Introdução à Programação

O objetivo desta disciplina é introduzir o estudo de algoritmos e lógica de programação.

0.0 Ementa

Resolução de problemas e desenvolvimento de algoritmos. Análise do problema. Estratégias de solução. Representação e documentação. Programação de algoritmos usando uma linguagem de programação. Estruturação de programas. Tipos e estruturas elementares de dados. Conceito de recursão e sua aplicação. Manipulação de Arquivos.

0.1 Livros recomendados

Estes são alguns dos livros úteis para a disciplina.

DEITEL, H. M. *C How to Program*. Prentice Hall International Inc., 2006.

FORBELLONE, A. L. V.; EBERPACHER, H. F. *Lógica de Programação: a construção de algoritmos e estrutura de dados*. São Paulo: Pearson Education do Brasil, 2000.

LOPES, G. e GARCIA, G. *Introdução à Programação*. Rio de Janeiro: Campus, 2002.

MIZRAHI, V. V. *Treinamento em Linguagem C*. São Paulo: Pearson, 2005.

SCHILDT, H. *C, Completo e Total*. São Paulo: Pearson Education, 1997.

0.2 Lógica de Programação

Geralmente, inicia–se o estudo de programação com o conceito de "lógica de programação". Mas o que é lógica? Usamos o termo *lógica* cotidianamente, por exemplo, quando dizemos a um amigo: *"seu raciocínio não tem lógica"*. Uma das definições do iDicionário Aulete é "modo coerente pelo qual coisas ou acontecimentos se encadeiam" [1]. Vamos imaginar que precisemos pegar o dinheiro que está no cofre fechado. "Pela lógica", precisamos antes abrir o cofre para pegar o dinheiro.

Programas de computadores são criados (ainda) por seres humanos, que utilizam a capacidade de raciocínio para gerar soluções para os problemas propostos. No entanto, o raciocínio é algo intangível. Neste sentido, a lógica de programação é uma ferramenta para transformar esse raciocínio abstrato em algo que possa ser transformado em programa de computador.

[1] aulete.uol.com.br.

Um computador só entende o que deve fazer se receber *instruções* claras, bem definidas e em ordem correta. Computadores (ainda) não são programáveis em linguagem natural, pois esta é sujeita a ambiguidades e imprecisões. A lógica de programação tem o objetivo de estabelecer uma sequência lógica de passos que devem ser executados por um programa de computador.

Essa sequência costuma ser chamada de algoritmo.

0.2.0 Algoritmos

Na definição de Cormen et al. [2], um algoritmo é um procedimento computacional bem definido que toma algum valor ou conjunto de valores como entrada e produz algum valor ou conjunto de valores como saída. Dito de outra forma, "um algoritmo é uma sequência de passos computacionais que transformam a entrada na saída".

Geralmente computadores são utilizados para resolver problemas. Um algoritmo, neste sentido, é um conjunto de passos para a solução de um problema. Um exemplo de problema computacional é como ordenar um conjunto de valores em ordem crescente.

O algoritmo deve especificar ações claras e precisas que, a partir de um estado inicial, após um período de tempo finito, produzem um estado final previsível e bem definido [3].

Um exemplo clássico de algoritmo é o processo de troca de uma lâmpada queimada, cujos passos seriam:

0. pegar a escada;
1. posicionar a escada embaixo da lâmpada;
2. buscar a lâmpada nova;
3. subir na escada;
4. retirar a lâmpada velha;
5. colocar a lâmpada nova.

0.2.1 Passos para construção de um algoritmo

Para construir um algoritmo, o problema deve ser dividido em três partes fundamentais:

$$\boxed{\text{entrada}} \rightarrow \boxed{\text{processamento}} \rightarrow \boxed{\text{saída.}}$$

onde:

Entrada: são os dados necessários para se resolver o problema
Processamento: é a manipulação dos dados de entrada
Saída: é a resposta para o problema

Considere o problema de se calcular a média final de três provas de um determinado aluno. Analisando em termos das três partes fundamentais de um algoritmo, tem-se:

Entrada:as notas das três provas
Processamento: o cálculo da média
Saída: a média final

[2] Cormen, T. H.; Leiserson, C. E.; Rivest, R. L.; Stein, C. *Algoritmos Teoria e Prática*. São Paulo: Elsevier, 2002.
[3] Forbellone, A. L. V; Eberspächer, H. E. *Lógica de Programação*. São Paulo: Pearson, 2005.

Representando como uma sequência de passos:

0. Receba a nota da primeira prova
1. Receba a nota da segunda prova
2. Receba a nota da terceira prova
3. Some as três notas
4. Divida a soma por três
5. Mostra o valor da divisão

0.3 Computadores: Hardware e Software

Um computador é um dispositivo que processa dados sob o controle de conjunto de instruções chamado programa. Os programas orientam o computador a executar um conjunto de ações especificado pelo programador. Apesar das diferenças na aparência física, todo computador (com raras exceções) é constituído por seis unidades lógicas:

Unidade de entrada: é responsável por receber dados dos dispositivos de entrada (teclado, mouse, microfone) e colocá-los à disposição de outras unidades;

Unidade de saída: é o componente que "pega" os dados processados e disponibiliza através dos dispositivos de saída (alto-falante, impressora, tela);

Unidade de memória: atua como "depósito" guardando informações que chegaram pela unidade de entrada, tornando-as acessíveis ao processador. Também retém informações processadas até que sejam colocadas em dispositivos de saída;

Unidade lógico-aritmética (ULA): unidade que realiza cálculos e executa as tomadas de decisões, tais como comparar dois números e decidir qual deles é o maior;

Unidade central de processamento (CPU): é a seção "administrativa" que coordena e supervisiona as outras unidades. Determina quando a unidade de entrada vai ler informação para a memória, diz para a ULA quando a informação da memória deve ser usada para um cálculo e diz para a unidade de saída quando enviar informação da memória para algum dispositivo de saída; e

Unidade de memória secundária: "depósito" de informação de alta capacidade e longa duração. Informação guardada na memória secundária é chamada de persistente, pois é preservada mesmo quando o computador é desligado.

Quando um programador escreve um programa, ele está escrevendo instruções para utilização destas unidades.

0.4 Tipos de Linguagens de Programação

Instruções ou programas são escritos utilizando-se de linguagens de programação, que são um conjunto de vocábulos e regras para sua utilização. Existem centenas de linguagens sendo utilizadas atualmente. No entanto, elas podem ser agrupadas em três tipos genéricos:

Linguagens de máquina: as instruções são escritas com cadeias de números que instruem o computador a executar as ações. Cada processador possui sua própria linguagem de máquina, o que significa que a linguagem de um tipo de processador não é entendida por outro processador. As instruções abaixo ilustram um programa que adiciona o valor de horas-extras ao salário base e armazena o resultado em salário bruto:

```
+1300042774
+1400593419
+1200274027
```

Linguagens *assembly*: escrever programas em linguagem de máquina é pouco produtivo e muito sujeito a erros. As linguagens *assembly* utilizam abreviações de termos em inglês para representar as operações elementares. Um programa tradutor (*assembler*) precisa ser usado para converter estas "palavras" em linguagem de máquina. O código a seguir executa a mesma ação descrita no item anterior:

```
load salarioBase
add horasExtras
store salarioBruto
```

Linguagens de alto nível: as linguagens *assembly* tornaram o processo de escrever programas menos complexo. No entanto, os programadores ainda tinham que escrever várias instruções para executar uma única ação. O código a seguir executa a mesma operação em linguagem de alto nível:

```
salarioBruto = salarioBase + horasExtras;
```

Programas especiais chamados compiladores convertem a linguagem de alto nível em linguagem de máquina. Programas compilados são muitas vezes chamados de código objeto. Um programa compilado adicionado de funções de bibliotecas produz um programa executável, que pode ser executado pelo processador.

O processo de compilação é mostrado simplificadamente na Figura 0[4].

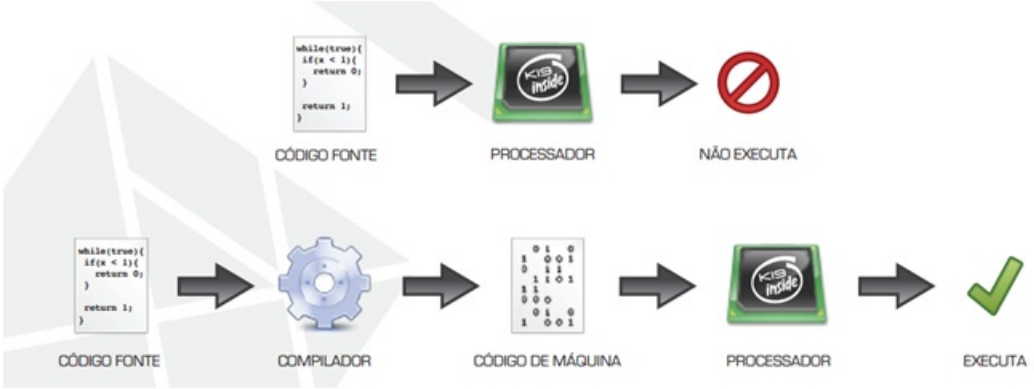


Figura 0: Processo de compilação.

[4]K19 Treinamentos. www.k19.com.br. Uso autorizado.

Capítulo 1

Introdução à Linguagem C

A Linguagem C foi criada na década de 1970 por Dennis Ritchie. Tornou-se muito conhecida como linguagem de desenvolvimento para o sistema operacional UNIX. Com a popularização dos microcomputadores, diversas implementações de C foram desenvolvidas, geralmente compatíveis entre si. Isto tornou C uma linguagem muito popular. Porém, apesar de geralmente compatíveis, havia discrepâncias entre as diversas implementações. Para remediar esta situação, o ANSI (*American National Standards Institute*) estabeleceu um padrão conhecido como C-ANSI. Os compiladores desenvolvidos por qualquer fabricante devem implementar no mínimo o padrão ANSI, apesar de poder adicionar outras funções não definidas neste padrão. Apesar de, atualmente, C não ser muito utilizada para o desenvolvimento de aplicações comerciais, praticamente todos os sistemas operacionais são escritos em C.

C é uma linguagem para programadores profissionais. Isto aumenta a responsabilidade do programador quanto à verificação de determinadas situações que podem gerar problemas, como a mistura de tipos.

A linguagem C++, por sua vez, foi desenvolvida por Bjarne Stroustrup, que adaptou a linguagem para o paradigma orientado a objetos. De maneira geral, tudo que é possível fazer em C é possível fazer em C++, no entanto, o inverso não é verdadeiro.

Neste curso, será adotada a linguagem C com o paradigma de programação estruturada.

1.0 Processo de desenvolvimento de programas em C/C++

De maneira simplificada, desenvolver um programa em C é executar as seguintes fases:

0. Escrever o programa: o programa precisa ser escrito com o auxílio de um editor de texto;

1. Compilar o programa: um compilador traduz as instruções em C para linguagem de máquina ou código-objeto. Nesta fase são feitas verificações quanto a erros, como por exemplo, erros de grafia e de sintaxe; e

2. "Linkar": o código escrito pelo programador precisa ser complementado por funções de bibliotecas e funções escritas por terceiros[1].

[1] Funções e bibliotecas serão estudadas posteriormente.

Após essa última fase, um programa executável gerado é gravado no disco. Para ser executado, ele precisa ser lido para a memória. Durante a execução, o processador busca as instruções do programa na memória e as executa.

1.1 Estrutura básica de um programa em C

Um programa em C é composto por uma ou mais unidades (módulos) chamadas de *funções*. Todo programa deve ter uma função chamada *main*, que é por onde o programa começa a ser executado. Por ora, serão mostrados programas com uma única função.

Nos exemplos mostrados no restante do texto sempre que aparecerem colchetes angulares < > significa que o que estiver entre eles deve ser substituído por expressões reais.

```
int main(){
    <instrução 1>
    <instrução 2>
    <...>
    <instrução n>
    return 0;
}
```

1.1.0 Primeiro programa

O programa a seguir exibe na tela a mensagem "Programando em C".

```
#include <iostream>
using namespace std;

int main(){
    cout << "Programando em C";
    return 0;
}
```

A primeira linha do programa possui o que é chamado de “diretiva de pré–processamento” e é caracterizado pelo caractere # no início da linha. A diretiva *#include <iostream>* indica ao compilador que o arquivo *iostream* será necessário ao programa. Considere que todo programa que realiza entrada e/ou saída deve ter essa diretiva. A expressão *using namespace std* também é necessária. Não se preocupe em entender o significado desses elementos, basta colocá–los no seu programa que mais adiante eles serão estudados aos poucos.

O principal no programa anterior vem após esses elementos, que é o módulo principal. Por ser uma função, ele deve ter um nome, que obrigatoriamente é *main( )*. As funções sempre vem acompanhadas de parênteses. A palavra *int* antes do nome da função indica que, quando ela terminar, vai ser devolvido um valor numérico do tipo inteiro. Novamente, não se preocupe em entender, apenas lembre que a palavra *main()* deve ser precedida por *int*. Seguindo no código do programa, a instrução *cout* (lê–se *c aut*) acompanhada do operador *«* faz com que o programa escreva na tela. No caso, o que vai ser escrito é a mensagem “Programando em C”. As aspas são necessárias para exibir textos e não aparecem na tela.

O operador « deve sempre acompanhar o *cout*. Por último, a instrução *return 0;* "devolve" o valor zero e finaliza o programa. Zero significa que o programa terminou sem erros. A chave de abertura ({) indica o início do bloco e a correspondente chave de fechamento (})indica o fim do mesmo bloco.

Outro detalhe muito importante é que a linguagem C é *case sensitive*, o que significa que diferencia maiúsculas e minúsculas. Assim, a linguagem determina que a palavra *main*, por exemplo, deve ser em letras minúsculas. Escrever *Main* ou *MAIN* acarreta um erro.

Atente, também, para o ponto e vírgula (;) ao final das instruções. Este sinal indica que a instrução termina ali.

1.1.1 Blocos e indentação

Programas em C são compostos por vários blocos. Um bloco é um conjunto de instruções com função definida. Blocos são delimitados por chaves ({ e }). Indentação é o processo de deixar espaços em branco antes da linha, geralmente 4 ou 5 espaços, de forma que fique claro quais linhas fazem parte do bloco. Na Figura1.0 é mostrado um exemplo de bloco indentado, destacado pela linha vertical. A indentação de blocos, em linguagem C, não faz nenhuma diferença para o compilador, no entanto, torna o código mais legível para o programador[2].

```
1  #include <iostream>
2
3  using namespace std;
4
5  int main(){
6      cout << "Programando em C" << endl;
7      return 0;
8  }
9
10
```

Figura 1.0: Exemplo de indentação.

✓IMPORTANTE: escreva o código pensando na pessoa que vai ler o programa.

1.2 Variáveis, constantes e tipos de dados

Basicamente, o que todo programa faz é manipular dados. Esses dados são armazenados em *variáveis* na memória. Variáveis são elementos fundamentais em qualquer programa. Uma variável representa um espaço reservado na memória que é capaz de armazenar um dado. Exemplos de dados que podem ser manipulados por um programa são o salário de uma pessoa, a nota de um aluno, a medida de um lado de um quadrado.

Uma variável, como já mencionado, está ligada a um espaço na memória e tem um nome para referenciar o seu conteúdo. Além disso, está ligada a um determinado tipo de dado.

1.2.0 Tipos de dados

Os principais tipos de dados em C são:

[2]Em algumas linguagens, tais como Python, isso é diferente.

Tipo	Faixa de valores
char	-127 a 127
int	-2.147.483.648 a 2.147.483.647
float	-3.4E+38 a 3.4E+38 ($3.14 * 10^{38}$)
double	-1.7E+308 a 1.7E+308
bool	true / false

O tipo *char*, embora guarde um número, é utilizado para armazenar caracteres. Isso acontece porque cada caractere está relacionado a um código numérico de acordo com um convenção conhecida como Tabela ASC. Só como exemplo, o código 41 equivale ao caractere 'A', o 42 equivale ao 'B'.

1.2.1 Declaração de variáveis

Declarar uma variável é reservar um determinado espaço na memória e relacioná-lo a um identificador (nome). A quantidade de memória destinada à variável depende do seu tipo. Por exemplo, uma variável do tipo *char* ocupa um *byte* de memória enquanto que uma variável do tipo *double* ocupa 8 *bytes*.

A declaração de variáveis é feita da seguinte maneira:

```
<tipo> <lista de variáveis>;
```

Exemplo:

```
int x, y, z; //declara 3 variáveis do tipo inteiro
char sexo;
```

1.2.2 Operador de atribuição

Operador de atribuição é o elemento que permite atribuir (armazenar) um valor na variável, o que corresponde a armazenar o valor na posição da memória referenciada pela variável.

Em C, o operador de atribuição é o sinal "=". Não confundir com o operador de igualdade, que em C é "==". Para atribuir valor a uma variável deve–se escrever a variável, o operador de atribuição e o valor que se quer atribuir à variável.

```
<variável> = <valor>;
```

Exemplos:

```
idade = 20;
sexo = 'm';
```

Lê–se *idade recebe 20*, *sexo recebe m*. Observe que variáveis do tipo *char* armazenam um único caractere, que deve estar entre aspas simples.

1.2.3 Inicialização de variáveis

Quando se declara uma variável apenas se reserva uma quantidade de memória que antes estava livre. A porção da memória agora reservada geralmente tem "lixo", por isso as variáveis em C devem ser inicializadas. Inicializar significa atribuir um valor válido à variável. Geralmente variáveis inteiras são inicializadas com 0 (zero) e variáveis do tipo *char* são inicializadas com o caractere vazio. Exemplos:

```
int x;
x = 0;
```

É o mesmo que

```
int x = 0;
```

1.2.4 Nomes de identificadores

Identificadores são os nomes de variáveis, funções e outros elementos definidos pelo programador. Um identificador pode ter um ou diversos caracteres, não podendo haver espaços entre eles. O primeiro caractere deve ser uma letra ou o caractere de sublinhado "_" e os caracteres subsequentes devem ser letras, números ou sublinhados. Exemplos:

Correto	**Incorreto**
i	$
a1	1a
cont	1cont
saldo_anterior	saldo...anterior
nomeAluno	nome aluno
nomeDoAluno	nome.aluno

C é uma linguagem *case-sensitive*. Isto significa que um identificador `nome` é diferente de `Nome` e `NOME`.

Os identificadores não podem ser iguais às palavras reservadas da linguagem. Por exemplo, não se pode declarar uma variável chamada *main*.

1.2.5 Comentários em C

Comentários são observações feitas pelo programador para explicar algum trecho de código. Comentários de uma linha são precedidos de "//". Comentários que se estendem por mais de uma linha são iniciados com "/*" e terminados por "*/". Exemplo de comentário de uma linha:

```
int nascimento; //ano de nascimento da pessoa
```

Exemplo de comentário de várias linhas:

```
/* Este programa transforma uma imagem colorida em uma imagem
em tons de cinza.  */
```

Comentários servem apenas para esclarecer algo para o programador, sendo ignorados durante a compilação do programa.

1.2.6 Constantes

Constantes são valores fixos que o programa não pode alterar. O modificador `const` na declaração de variável impede que seu valor seja alterado. Exemplo:

```
const int i = 10;
```

1.2.7 Constantes simbólicas

Muitas vezes o programa manipula um determinado valor fixo, por exemplo, o número PI. Uma constante simbólica é um nome simbólico para este valor constante. A definição de constantes simbólicas é feita pela diretiva de compilação `#define`. Exemplo:

```
#define PI 3.14159
```

Depois, a constante pode ser utilizada no programa:

```
x = PI * 5;
```

1.3 Operadores e expressões

Sempre que houver uma expressão do lado direito do operador de atribuição, o valor da expressão é calculado e depois atribuído. Por exemplo, na instrução:

```
x = (2+3) * 5;
```

primeiro serão efetuadas as operações do lado direito do operador de atribuição e depois o resultado atribuído à variável x.

Além do operador de atribuição há operadores aritméticos, relacionais, de igualdade e lógicos.

1.3.0 Operadores aritméticos

Os operadores aritméticos são:

Operador	Significado
-	subtração
+	adição
*	multiplicação
/	divisão
%	resto da divisão
– –	decremento unitário
++	incremento unitário

Os operadores aritméticos podem ser aplicados a qualquer tipo de dado, porém em alguns casos ocorre truncamento. Por exemplo, o resultado da divisão (/) atribuído a um inteiro acarreta a perda de um eventual resto. No caso seguinte, x terá valor 2:

```
int x;
x = 5/2;
```

Incremento e decremento

C possui dois operadores "concisos" para incremento e decremento unitário de variável. O operador "++" incrementa (soma 1) ao seu operando e o operador "– –" subtrai 1 de seu operando. Assim:

```
x++;
```

equivale a:

```
x = x + 1;
```

Os operadores podem ser utilizados como prefixo ou sufixo. A expressão `x++` incrementa `x` em uma unidade, da mesma forma que `++x`. No entanto, em uma expressão do tipo:

```
y = x++;
```

primeiro o valor de `x` é atribuído a `y` e depois a variável `x` é incrementada.

C reduzido

Algumas construções válidas em C:

```
x +=y; // equivale a x = x + y;
x *= y; //equivale a x = x * y;
```

Precedência dos operadores aritméticos

A regra de precedência dos operadores aritméticos é:

0. Operações de expressões contidas em parênteses são avaliadas primeiro;
1. Multiplicação, divisão e resto são avaliadas em seguida, da esquerda para direita; e
2. Adição e subtração são avaliadas por último, da esquerda para direita.

Exemplo:

z	=	p	*	r	%	q	+	w	/	x	-	y	;
	6		1		2		4		3		5		

1.3.1 Operadores relacionais e de igualdade

Operadores relacionais comparam dois valores ou variáveis, resultando em verdadeiro ou falso. Os operadores relacionais e de igualdade são:

Operador	Significado
>	maior que
<	menor que
>=	maior que ou igual
<=	menor que ou igual
==	igual
!=	diferente

1.3.2 Operadores lógicos

Os operadores lógicos são utilizados para formar condições complexas combinando condições simples. Por exemplo, se desejarmos comparar se o valor de uma variável está entre dois limites, precisamos usar um operador lógico:

```
se (idade >=18) e (idade <= 70) ...
```

Os operadores lógicos são:

Operador	Significado
&&	E
||	OU
!	NÃO

Operadores lógicos serão detalhados no Capítulo 2.

1.3.3 *Casts*

Para forçar um resultado ser de um determinado tipo, pode-se usar um *cast*. Por exemplo, a divisão de dois inteiros resulta em inteiro. Para forçar o resultado como ponto flutuante, usa-se:

```
int x = 5;
float f;
f = (float) x/2;  // f vale 2.5
```

Sem o *cast f* estaria com o valor 2.

1.4 Entrada e Saída

Grande parte dos programas envolve algum tipo de entrada de dados e algum tipo de apresentação de resultados. Esses tipos de operações são chamadas de operações de *entrada* e *saída*. Nas operações de entrada, os *bytes* recebidos de algum dispositivo (teclado, *pen drive*, internet...) para a memória principal. Nas operações de saída, os dados são enviados da memória principal para algum dispositivo de saída (tela, impressora, disco). Em C, por padrão, o dispositivo de entrada é o teclado e o de saída é o terminal de vídeo.

A operação de entrada de dados é geralmente chamada de "leitura" e a operação de saída, de " escrita". Para entrada de dados, em C++, é utilizado o comando `cin` seguido do operador de deslocamento de *bits* "*»*". Por exemplo, para ler o valor para variável *i*, as instruções são:

```
int i=0;
cin » i;
```

Para saída de dados, em C++, é utilizado o comando *cout* seguido do operador de deslocamento de *bits* "*«*". No exemplo a seguir o valor de *i* é exibido na tela, precedido de uma mensagem.

```
cout « "O valor de i é:  "  « i;
```

Observe que é possível exibir vários dados utilizando um único *cout*. Para isso, coloca–se o operador « entre os valores/expressões que se quer concatenar.

1.4.0 Armazenando e recuperando dados da memória

Quando atribuímos um valor para uma variável, seja por meio do operador de atribuição, seja por meio de uma leitura pelo teclado, realizamos uma escrita para memória. Essa operação é chamada de "destrutiva", pois o conteúdo que havia naquela posição é perdido e substituído pelo novo valor.

Por outro lado, quando recuperamos o valor de uma variável, seja utilizando em uma operação ou exibindo na tela, o valor daquela variável se mantém preservado. Esta operação é chamada de "não–destrutiva".

1.4.1 Quebra de linha

Quando se vai escrever na tela pode ser necessário mudar de linha. Isso pode ser conseguido com o comando *endl* ou com a expressão "\ n". Por exemplo, considere as instruções a seguir:

```
cout « "Programar é fácil";
cout « "Muito fácil";
```

embora escritas em duas linhas geram a seguinte saída:

```
Programar é fácil.Muito fácil.
```

No entanto:

```
cout « "Programar é fácil" « endl;
cout « "Muito fácil";
```

resultam na seguinte saída:

```
Programar é fácil.
Muito fácil.
```

1.5 Considerações Finais

Uma linguagem de programação é constituída de um conjunto de comandos e funções e de regras para se utilizar esses elementos. Embora uma linguagem seja diferente de outra, alguns conceitos são universais. Por exemplo, C++ utiliza o comando *cin* para entrada de dados, Pascal utiliza o comando *read*, mas ambos representam a mesma coisa, que é a entrada de um dado e armazenamento em uma variável. O próximo capítulo apresenta as estruturas básicas de seleção da linguagem C/C++.

1.6 Exercícios

0. Todo programa em C começa a executar pela função _____.
1. A _____inicia o corpo de cada função e a _____finaliza o corpo de cada função.
2. Toda instrução termina com _____.
3. Se um novo valor é atribuído a uma posição da memória, este valor sobrescreve o antigo valor armazenado naquela posição. Este processo é chamado de _____.
4. Se um valor é lido de uma posição da memória, o valor naquela posição é preservado. Este processo é chamado de _____.
5. _____são utilizados para tornar os programas mais legíveis.
6. Explique o que são variáveis e dê exemplos dos principais tipos de variáveis. Explique o que é declarar uma variável e o que é inicializar uma variável.
7. Construa um programa que leia dois número inteiros. Exiba a soma dos dois números.

8. Faça um programa que tenha duas entradas: o preço da mercadoria e a quantidade comprada. A saída deve ser o valor a pagar.

9. A medida de temperatura pode ser feita com base em diversas escalas, tais como graus Célsius ou graus Farenheit. Faça um programa que leia uma temperatura em graus Farenheit (F) e converta para Célsius (C), utilizando a fórmula:

   C = 5/9 (F - 32)

10. Alguns amigos fizeram um jogo de loteria e combinaram dividir o prêmio se ganhassem. Para a sorte deles, foram premiados. Faça um programa que leia o valor do prêmio e a quantidade de amigos. Exiba o quanto cada um deles vai receber.

11. Um grupo de amigos vai a uma pizzaria e comem uma quantidade de pizzas e bebem algumas cervejas (sem álcool). Sabendo–se que cada pizza custa R$ 23,00 e cada cerveja custa R$ 4,00, faça um programa que receba como entrada a quantidade de pizzas degustadas e a quantidade de cervejas tomadas. Calcule o total da conta. Leia a quantidade de pessoas e calcule e exiba o quanto cada um tem de pagar.

12. Faça um programa que leia os lados de um retângulo e mostre *a)*sua área e *b)*seu perímetro.

13. Alonso tem duas opções para ir para o trabalho. Ele pode ir de ônibus, pagando R$ 5,00 por viagem ou ir de carro, gastando R$ 20,00. Faça um programa que receba como entrada a quantidade de dias úteis do mês e exiba o quanto Alonso gastaria indo de ônibus e o quanto gastaria indo de carro.

14. Crie uma variação do programa anterior utilizando constantes simbólicas para os dois custos.

15. Faça um programa cuja entrada seja um número do tipo ponto flutuante. O programa deve "arredondar" o número para cima. Exiba o número arredondado. (Não utilize a função *ceil()*).

Capítulo 2

Estruturas de Decisão

Estruturas de controle permitem controlar a sequência das ações lógicas de um programa. Basicamente, existem dois tipos de estruturas de controle: estruturas de repetição e estruturas de decisão (ou seleção). A estrutura de repetição permite que um bloco de instruções seja executado uma quantidade controlada de vezes. A estrutura de decisão permite executar um entre dois ou mais blocos de instruções. A seguir são mostradas as estruturas de C/C++ que permitem implementar essas estruturas.

2.0 Estrutura de seleção *if/if-else*

Muitas estruturas em C/C++ fazem teste condicional que determina o curso de uma ação. Uma expressão condicional é uma expressão que, ao ser avaliada, resulta em um valor verdadeiro ou falso.

A estrutura de seleção *if* realiza um teste lógico em uma *condição*. Se o teste lógico resultar em "verdadeiro", o bloco de comandos (corpo do *if*) é executado; se resultar em "falso", a execução continua a partir da primeira instrução após o bloco. A forma geral da estrutura *if* é:

```
if (<condição>) {
    <comandos>;
}
```

Na Figura 2.0 é ilustrada, por meio de fluxograma, a estrutura de seleção.

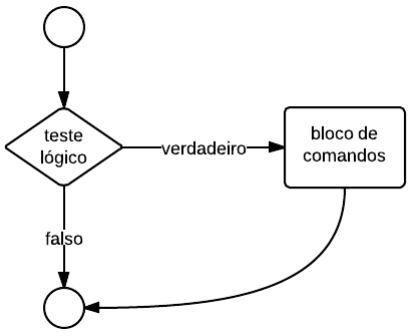


Figura 2.0: Representação da estrutura *se*.

Exemplo em C:

```
if (idade >= 16) {
    cout << "Você tem idade para votar";
} //fim do bloco
```

A estrutura de seleção *if/else* realiza um teste lógico na *condição*. Se o teste lógico resultar em "verdadeiro", o bloco de comandos correspondente ao *if* é executado; se resultar em "falso", o bloco correspondente ao *else* é executado. A forma geral da estrutura *if/else* é:

```
if (<condição>) {
    <comandos>;
}
else {
    <comandos>;
}
```

Na Figura 2.1 a estrutura de seleção composta é ilustrada, por meio de fluxograma.

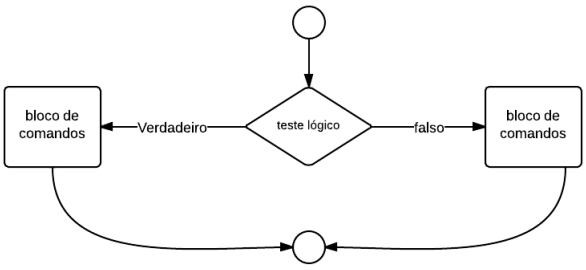


Figura 2.1: Representação da estrutura *se/senão*.

Exemplo em C:

```
if (idade >= 16) {
    cout << "Você tem idade para votar";
}
else {
    cout << "Você é muito novo para votar";
}
```

2.0.0 Operador ternário

O operador ternário "?" substitui sentenças do tipo *if/else*. Exemplo:

```
a > b ?  (a = b) :  (b = a);
```

equivale a:

```
    if (a > b)
        a = b;
    else
        b = a;
```

2.0.1 Um programa de exemplo

O programa a seguir lê um número e diz se ele é par ou ímpar. Este programa corresponde ao fluxograma da Figura 2.1.

```
#include <iostream>

using namespace std;

int main(){
    int num=0;
    cout << "Digite um número: ";
    cin >> num;
    if (num % 2 == 0){
        cout << "É par" << endl;
    }
    else{
        cout << "É ímpar" << endl;
    }
    return 0;
}
```

2.0.2 Estruturas *if/else* aninhadas

Em algumas situações, é necessário tomar decisões dentro do corpo de *if* ou do *else*. Nestes casos, deve–se aninhar a estrutura desejada na outra estrutura. Considere a seguinte situação: um aluno é considerado aprovado se obtiver média final igual ou maior que 7. Caso contrário, se a média for maior ou igual a 4, ele terá direito a realizar o exame final. Caso a média seja inferior a 4, ele está reprovado sem direito a exame.

```
...
int main(){
    float media=0;
    cout << "Digite a média: ";
    cin >> media;
    if (media >= 7){
        cout << "Aprovado" << endl;
    }
    else {
        if (media >= 4){
            cout << "Exame" << endl;
        }
        else {
            cout << "Reprovado" << endl;
        }
    }
    return 0;
}
```

2.1 Operadores lógicos

Embora nos exemplos anteriores tenham sido utilizadas condições simples, tais como $x < 10$, é possível ter múltiplas condições a serem avaliadas. Para isto, utiliza–se os operadores lógicos && (E lógico), || (OU lógico) e ! (NÃO lógico).

Suponha que duas condições devam ser verdadeiras para se escolher entre dois caminhos de execução. Neste caso, utiliza–se o operador &&.

Por exemplo, considere um programa que leia o sexo (m/f) e a idade de uma pessoa. Caso seja do sexo masculino e maior de idade, deve ser solicitado que informe o número do certificado de reservista. O código seria:

```
if (sexo == 'm' && idade >= 18){
    cout << "Informe certificado de reservista: ";
    cin >> certificado;
}
```

Neste caso, as duas condições tem que serem verdadeiras para que o corpo do *if* seja executado.

Em outros casos, basta uma condição ser verdadeira para que a expressão seja considerada verdadeira. Por exemplo, um aluno não tem direito a exame final se tiver frequência abaixo de 75% *ou* se tiver média inferior a 4. Basta uma destas condições ser verdadeira para que ele não tenha direito. O código correspondente é:

```
if (nota < 4 || frequencia < 75){
    cout << "Reprovado direto";
}
```

A análise dos operadores E e OU geralmente é feita por meio de uma estrutura conhecida como *tabela verdade*, que é uma tabela que mostra o resultado de todas as combinações possíveis entre valores

A	B	Resultado
V	V	verdadeiro
V	F	falso
F	V	falso
F	F	falso

Tabela 2.0: Tabela Verdade do operador E.

A	B	Resultado
V	V	verdadeiro
V	F	verdadeiro
F	V	verdadeiro
F	F	falso

Tabela 2.1: Tabela Verdade do operador OU.

lógicos. Considere duas expressões lógicas A e B: na Tabela 2.0 são mostrados os resultados relativos ao operador E e na Tabela 2.1, os resultados para o operador OU.

Em resumo, o operador *E* resulta em verdadeiro se *todas* as condições forem verdadeiras e o operador OU resulta em verdadeiro se *pelo menos uma* condição for verdadeira.

2.2 Estrutura de seleção *switch*

A estrutura *switch* permite a seleção de um caminho dentre vários, testando sucessivamente o valor de uma *expressão* contra uma lista de inteiros ou caracteres. Quando o valor coincide, os comandos associados àquela expressão são executados. Esta estrutura possui a seguinte sintaxe:

```
switch(<expressão>){
    case constante_1:
       <sequência de comandos>;
       break;
    case constante_2:
       <sequência de comandos>;
       break;
    //...
    case constante_n:
       <sequência de comandos>;
       break;
    default:
       <sequência de comandos>;
       break;
}
```

A cláusula *default* é opcional e indica o que deve ser executado se nenhuma das opções for escolhida. O comando *break* transfere a execução para a instrução seguinte após o fim de *switch*. Dentro de cada *case* pode haver quantos comandos forem necessários, no entanto, sempre deve ser finalizado pelo *break*.

✓Deixar o caso mais frequente em primeiro lugar melhora a eficiência do programa.

✓Esquecer a cláusula *break* é um erro grave de lógica que causa efeitos imprevisíveis durante a execução.

Exemplo:

```
...
printf("Digite código do voto: ");
cin >> voto;
switch(voto){
    case 1:
        candidato1++;
        break;
    case 2:
        candidato2++;
        break;
    case 3:
        brancos++;
        break;
    case 4:
        nulos++;
        break;
    default:
        cout << "Código inválido";
        break;
}  //fim do switch
```

2.3 Exercícios

0. A estrutura de seleção _____é utilizada para executar uma ação quando a condição é verdadeira e outra ação quando a condição é falsa.

1. Diversas instruções agrupadas dentro de chaves ({ e }) são chamadas de _____.

2. Escreva quatro formas diferentes de incrementar 1 a uma variável *x*.

3. Assinale V (verdadeiro) ou F (falso):

   (a) O caso *default* é obrigatório nas estruturas *switch*.

   (b) Cada *caso* nas estruturas *switch* só pode ter uma linha de instrução.

   (c) O comando *break* é obrigatório no caso default das estruturas *switch*.

4. Escreva as instruções em C para executar cada uma das seguintes tarefas:

   (a) Declare as variáveis *x* e *soma* do tipo inteiro.

   (b) Inicialize *x* com 1.

   (c) Inicialize *soma* com 0.

   (d) Some a variável *x* à variável *soma* e atribua o resultado à variável *soma*.

   (e) Exiba na tela a mensagem “A soma é: ” seguida do valor da variável *soma*.

5. O valor da conta de água é calculado progressivamente conforme a faixa de consumo (valores no quadro abaixo). Faça um programa que leia o quanto o consumidor gastou em m^3 e mostre o valor a ser pago.

Consumo em m^3	**Tarifa por m^3**
menor ou igual a 10	2.55
acima de 10 e menor ou igual a 20	3.44
maior que 20	5.25

6. Um automóvel percorre 10 km com um litro de álcool e 15 km com um litro de gasolina. Faça um programa que leia o preço do álcool, o preço da gasolina e a distância da viagem. Calcule e exiba o quanto o motorista gastaria se abastecer com álcool e o quanto gastaria se abastecer com gasolina. Exiba ao final se compensa usar álcool ou gasolina.

7. A expressão `if (10 == 10 == 10)` resulta em verdadeiro ou falso? Por que?

8. A expressão `if (a * b < c)` acarreta erro? Como corrigi-la?

9. Analise este programa. O teste de condição está escrito corretamente? Explique.

```
//...
int a, b;
cout << "Digite dois números: ";
cin >> a;
cin >> b;
if (b){
    cout << (float) a/b;
}
else{
    cout << "não pode dividir por zero";
}
```

10. Uma loja vende 5 produtos, cujos códigos e preços estão na tabela. Faça um programa que leia o código do produto, a quantidade comprada pelo cliente e diga o quanto ele tem que pagar, utilizando a estrutura *if/else*.

Código	**Preço**
1	5.78
2	9.99
3	5.25
4	18.90
5	22.60

11. Reescreva o programa do exercício anterior utilizando a estrutura *switch*.

12. Faça um programa para calcular o volume de uma esfera de raio R, sendo R um dado fornecido pelo usuário. O volume de uma esfera é dado por $V = \frac{4}{3}\pi R^3$.

13. Escreva um programa que leia três valores inteiros e diferentes e mostre-os em ordem decrescente. Utilize uma seleção encadeada.

14. Elabore um programa que leia a altura (h) e o sexo (m ou f) de uma pessoa e calcule seu peso ideal, utilizando as seguintes fórmulas:
    - para homens: (72.7 * h) - 58;
    - para mulheres: (62.1 * h) - 44.7

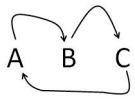
15. Sabendo-se que uma aula tem 50 minutos e que a carga-horária das disciplinas é definida em horas, escreva um programa que leia a carga-horária de uma disciplina, calcule e mostre a quantidade de aulas da disciplina.

16. Sabendo-se que a frequência mínima exigida é de 75%, escreva um programa que leia a quantidade total de aulas de uma disciplina e calcule o máximo de faltas que um aluno pode ter. Apresente o resultado como número inteiro.

17. Escreva um programa que leia um código numérico inteiro correspondente à região e escreva o nome da região por extenso, utilizando uma estrutura do tipo *switch*. As regiões estão assim classificadas:

Código	**Descrição**
1	Sul
2	Sudeste
3	Centro-Oeste
4	Nordeste
5	Norte

18. Escreva um programa que leia três valores inteiros: A, B e C. A seguir, substitua o valor de B pelo valor de A, o valor de C pelo valor de B e o valor de A pelo valor de C.



19. Elabore um programa que leia o preço de venda e calcule o valor a ser pago por um produto, de acordo com a condição de pagamento. Utilize os códigos e condições:

Código	**Condição**
1	à vista, em dinheiro: 10% de desconto
2	à vista, cartão de crédito: 5% de desconto
3	cheque para 30 dias: preço normal

20. Elabore um programa que leia a idade de um nadador. A seguir, classifique em uma das seguintes categorias:

Idade	Categoria
5 até 7	Infantil A
8 até 10	Infantil B
11 até 13	Juvenil A
14 até 17	Juvenil B
maiores de 18	adulto

21. O IMC - Índice de Massa Corporal - é o critério da Organização Mundial de Saúde para dar uma indicação sobre a condição de peso de uma pessoa adulta. A fórmula para cálculo é: $IMC = peso/altura^2$. Escreva um programa que leia o peso e a altura de um adulto e mostre sua condição, de acordo com a tabela:

IMC	Condição
abaixo de 18.5	Abaixo do peso
de 18.5 e abaixo de 25	Peso normal
de 25 e abaixo de 30	Acima do peso
30 ou acima	Obeso

22. Crie um algoritmo em que sejam lidas 2 notas, seja calculada a média e seja mostrado "Aprovado", caso a média seja maior ou igual a 7, e "Reprovado" caso seja menor.

23. Refaça o exercício anterior, adicionando o conceito de sexo do aluno (variável tipo *char*), alterando o resultado final para "Aprovada" e "Reprovada" quando a aluna for do sexo feminino.

24. Faça um programa que leia um caractere e diga se a letra digitada é uma vogal ou não.

25. João Papo-de-Pescador, homem de bem, comprou um microcomputador para controlar o rendimento diário de seu trabalho. Toda vez que ele traz um peso de peixes maior que o estabelecido pelo regulamento de pesca do estado de São Paulo (50 quilos) deve pagar um multa de R$ 4,00 por quilo excedente. João precisa que você faça um algoritmo que leia a variável *peso* (peso de peixes) e verifique se há excesso. Se houver, calcular e mostrar o valor da multa.

26. Um caixa eletrônico trabalha com cédulas de R$ 50.00, R$ 20.00, R$ 10.00 e R$ 5.00. Faça um programa que tenha como entrada o valor que o cliente quer retirar e mostre quantas cédulas de cada valor vão ser entregues. A máquina sempre paga com as cédulas de maior valor. Exemplo:
    Valor do saque: R$75.00
    Serão entregues: 1 nota de R$ 50.00, 1 nota de R$ 20.00 e uma nota de R$ 5.00

    Caso o cliente queira sacar um valor que não seja múltiplo de 5, não é possível completar o saque.

Capítulo 3

Estruturas de Repetição

Muitos programas envolvem repetição (*loop*). Um *loop* é um conjunto de um ou mais ações que são executadas repetidamente enquanto a *condição de continuação* permanecer verdadeira. Repetições podem ser controladas por *contador* ou por *sentinela*.

Em repetições controladas por contador sabe-se exatamente quantas vezes o *loop* vai ser executado. Uma variável de controle, ou contador, é utilizada. Esta variável é incrementada (ou decrementada) cada vez que o bloco é executado. Quando o valor da variável de controle atinge o número definido de repetições, o *loop* termina e a execução prossegue na primeira instrução depois do bloco.

✓Não inicializar a variável de controle é um erro grave de lógica e causa efeitos não previsíveis.

✓Não incrementar a variável de controle é um erro grave de lógica e causa repetição infinita do *loop*.

Repetições controladas por sentinela são utilizadas quando *a)* não se sabe qual o número exato de repetições e *b)* o corpo do *loop* inclui leitura de dados que, dependendo do valor, podem provocar o fim da repetição.

3.0 Repetição com teste no início (*while*)

A estrutura de repetição *while* faz com que a condição de continuação seja avaliada antes de se iniciar o bloco. Se a condição é verdadeira o bloco é executado uma vez e a condição é avaliada novamente. Caso a condição seja falsa a repetição é terminada. Se a condição for inicialmente falsa, o bloco não vai ser executado nem uma vez. A sintaxe desta estrutura é:

```
while (<condição>){
    <bloco instruções>;
}
```

Na Figura 3.0 a estrutura *while* é representada graficamente.

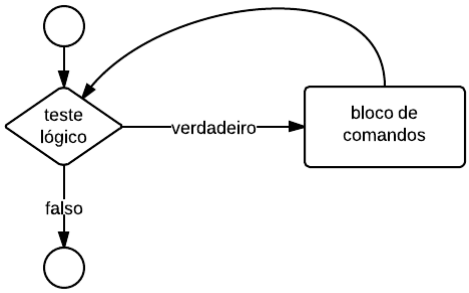


Figura 3.0: Repetição com teste no início.

3.0.0 *while* controlado por contador

Repetições controladas por contadores requerem:

0. A variável de controle;
1. O valor inicial para a variável de controle;
2. O incremento (ou decremento) que faz com que a variável de controle seja modificada a cada execução do bloco; e
3. A condição que testa o valor da variável de controle para determinar a continuação ou término do *loop*.

O exemplo a seguir imprime os números de 0 a 10 usando a estrutura *while* controlada por contador.

```
int contador = 0;
while (contador <= 10){
    cout << contador;
    contador++;
} //fim do loop
```

✓Utilize sempre variáveis do tipo inteiro para controlar repetições.

3.0.1 *while* controlado por sentinela

Uma estrutura de repetição controlada por sentinela é uma estrutura em que não se sabe com antecedência quantas vezes o bloco vai ser executado. Ele será executado até que o valor da variável de controle seja igual ao valor determinado para o fim do *loop*. O valor da sentinela indica o "fim dos dados" e deve ser um valor distinto dos dados válidos.

O exemplo a seguir executa a leitura de diversos valores e encerra a entrada quando for informado um número negativo. Observe que não está determinado a quantidade de números que serão lidos. Pela lógica do programa, espera–se apenas números maiores ou iguais a zero. Por isso, utiliza–se como sentinela um valor negativo para variável de controle.

```
int main(){
    int num=0;
    cout << "Digite um número: ";
    cin >> num;
    while (num >=0){
        cout << "Digite um número: ";
        cin >> num;
    }
    return 0;
}
```

O bloco de instruções contido na estrutura de repetição será executado até que um número menor que zero seja digitado.

3.1 Repetição com teste no final (*do...while*)

A estrutura `do...while` é uma estrutura de repetição que executa o teste de parada no final. Isto significa que o corpo do *loop* é executado ao menos uma vez. A sintaxe da estrutura é:

```
do{
    <bloco instruções>;
}while(<condição>);
```

em que <*condição*> é uma expressão lógica e <*bloco instruções*> é um conjunto de uma ou mais instruções. Da mesma forma que a estrutura *while*, *do...while* também pode ser controlada por contador ou por sentinela.

Na Figura 3.1 a estrutura *do...while* é representada graficamente.

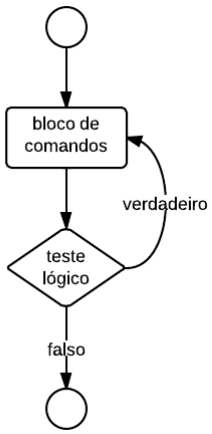


Figura 3.1: Repetição com teste no final.

3.1.0 *do...while* controlado por contador

De forma muito similar ao *while*, na estrutura *do...while* uma variável pode ser utilizada como contador para determinar o fim da repetição. No exemplo a seguir, uma estrutura *do...while* é utilizada para ler a

nota de 10 alunos.

```
int main(){
    float nota=0;
    int cont=0;
    do{
        cout << "Digite a nota: ";
        cin >> nota;
        cont++;
    }while(cont < 10);
    return 0;
}
```

Observe que, a cada nota lida, a variável de controle é incrementada. Ao final do bloco (}) a variável é testada para se verificar se a repetição para ou continua.

3.1.1 *do...while* controlado por sentinela

Estruturas *do...while* controladas por sentinela utilizam o valor de uma variável como condição de término do *loop*. No exemplo a seguir, são lidas notas de uma quantidade de alunos não conhecida previamente. A sentinela é uma nota negativa. Observe que o valor da sentinela deve ser um valor não válido, no caso como não existem notas negativas, o valor negativo para uma nota pode ser usado como sentinela.

```
int main(){
    float nota=0;
    do{
        cout << "Digite a nota: ";
        cin >> nota;
    }while(nota >= 0);
    return 0;
}
```

O laço se repete enquanto a condição for verdadeira; quando um número negativo for digitado, o *loop* termina.

3.2 Estrutura de repetição *for*

A estrutura de repetição *for* utiliza todos os elementos das estruturas controladas por contador (variável de controle, valor inicial da variável, incremento ou decremento e condição).

A sintaxe da estrutura é:

```
for (<inicialização da variável>; <condição>; <incremento>){
    <comandos>;
}
```

O código a seguir imprime os números de 0 a 10.

```
int cont = 0;
for (cont = 0; cont <= 10; cont++){
    cout << cont;
} //fim do loop
```

Quando a execução da estrutura começa, o valor da variável de controle é inicializado com 0. Então a condição de controle *cont <= 10* é verificada. Se a condição é verdadeira, o conjunto de comandos do bloco é executado e a a variável é incrementada. A repetição continua até que a condição seja avaliada como falsa (*cont = 11*). Neste caso, a execução continua a partir do primeiro comando depois do fim do bloco.

✓Substituir <= por < ou vice-versa é um erro comum de programação.

A estrutura *for* é especialmente indicada quando se sabe exatamente quantas vezes o programa deve executar o bloco.

Embora no exemplo o incremente seja unitário (*cont++*), é possível incrementos (ou decrementos) em outras quantidades, bem como iniciar a variável de controle com outros valores além de 0, como por exemplo:

```
int cont = 0;
for (cont = 7; cont <= 77; cont = cont + 7){
    ...
} //fim do loop
```

em que a variável será incrementada em 7 a cada iteração. Também é possível utilizar expressões aritméticas, como por exemplo:

```
int cont = 0;
for (cont = 0; cont + 2 < 100; cont++){
    ...
} //fim do loop
```

Em muitos casos, deseja–se que o incremento seja negativo (decremento), como por exemplo:

```
int cont = 0;
for (cont = 10; cont >= 0; cont--){
    ...
} //fim do loop
```

Em geral, a estrutura *for* equivale à estrutura *while* controlada por contador.

3.3 Operadores lógicos

Embora nos exemplos anteriores tenham sido utilizadas condições simples, tais como $x < 10$, é possível ter múltiplas condições a serem avaliadas. Para isto, utiliza–se os operadores lógicos && (E lógico), || (OU lógico) e ! (NÃO lógico).

Por exemplo, considere um programa que realize leituras de números enquanto não for informado um número negativo *e* enquanto a soma dos números lidos não atinja 100:

```
int num=0, soma=0;
cin >> num;
while (num >=0 && soma < 100){
    soma = soma + num;
    cin >> num;
}
```

Neste caso as duas condições devem ser verdadeiras, tanto *num >=0* e *soma < 100*. Basta uma das condições ser falsa para encerrar o *loop*.

O mesmo raciocínio vale para mais de duas condições bem como para o operador OU.

3.4 Exercícios

0. Elabore um programa que leia 10 números inteiros. Ao final, mostre a soma de todos os números lidos.

1. Faça um programa que some os números ímpares entre 1 e 99 utilizando uma estrutura *for*.

2. Faça um programa em que sejam lidos números inteiros maiores que zero (encerre a digitação quando for informado o número zero). Ao final, mostre a média dos números lidos.

3. Por meio de teste de mesa, determine o que vai ser impresso pelo código a seguir:

```
int cont = 0;
for (cont = 44; cont >= 0; cont-=11){
    cout << cont;
}
```

4. Faça um programa que leia as notas de vários alunos em uma prova de Programação I (encerre a digitação quando for informada uma nota menor que zero). Ao final, mostre:

   (a) A média da sala

   (b) A maior nota

   (c) A menor nota

   (d) Quantas notas foram menores que 7

5. Utilizando a estrutura *for*, escreva os comandos para gerar as seguintes sequências de valores:

   (a) 3, 8, 13, 18, 23

   (b) 20, 14, 8, 2, -4, -10

   (c) 19, 27, 35, 43, 51

6. Repita o exercício anterior, utilizando a estrutura *while*.

7. Repita o exercício anterior, utilizando a estrutura *do/while*.

8. Escreva um programa que some uma sequência de inteiros. Assuma que o primeiro número informado especifique a quantidade de números que devam ser lidos. Exemplo:
   5 40 80 97 23 44
   o primeiro número lido (5) indica que devem ser lidos mais 5 números. Ao final, exiba a soma.

9. Escreva um programa que imprima os triângulos da figura usando *for*. Imprima separadamente, um após o outro. Dica: os dois últimos exigem que cada linha comece com uma quantidade apropriada de espaços.

```
(A)          (B)          (C)          (D)
*            **********   **********             *
**           *********     *********            **
***          ********       ********           ***
****         *******         *******          ****
*****        ******           ******         *****
******       *****             *****        ******
*******      ****               ****       *******
********     ***                 ***      ********
*********    **                   **     *********
**********   *                     *    **********
```

10. Faça um programa em que seja lido um número inteiro. Gere a figura seguinte com base nesse número. Exemplo: 4. Será gerada a figura:

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

11. Faça um programa em que sejam lidos um número inteiro e um caractere. Gere a figura seguinte com base nos dados fornecidos. Exemplo: 3, #. Será gerada a figura:

```
###
###
###
```

12. Elabore um programa utilizando a estrutura `for` que calcule o quanto um investidor vai receber ao final de 10 anos de aplicação. Suponha que foram depositados R$ 1.000,00 e que a taxa de juros seja de 5% ao ano. A fórmula para cálculo de juros compostos é: $m = p(1 + t)^n$, onde m é o montante ao final do período, p é o valor inicial (principal), t é a taxa de juros e n é o número de anos.

13. Faça um programa em que o usuário digite um valor, e o programa gere a sequência de Fibonacci com uma quantidade de elementos igual ao fornecido pelo usuário.
    Exemplo : ENTRADA $\rightarrow$ 8
    SAIDA $\rightarrow$ 1, 1, 2, 3, 5, 8, 13, 21

14. Faça um programa para calcular o fatorial de um número informado pelo usuário.

15. Faça um programa que leia um número e diga se ele é primo ou não.

16. Elabore um programa que calcule o somatório:
    $soma = \frac{1}{1} + \frac{3}{2} + \frac{5}{3} + ... + \frac{99}{50}$

17. Elabore um programa que calcule o somatório:
    $soma = \frac{1}{1} - \frac{2}{4} + \frac{3}{9} - \frac{4}{16} + \frac{5}{25}... - \frac{10}{100}$

18. Faça um programa em que seja declarado um número inteiro entre 1 a 100 (utilize constante simbólica). O usuário (que não sabe qual o número) deverá tentar adivinhá–lo. Para cada tentativa, o programa deve dizer se o número digitado é maior ou menor que o número correto. Quando o usuário acertar, o programa deve dizer quantas tentativas foram feitas até a adivinhação.

19. Faça um programa que realize as operações básicas de uma calculadora. O programa deve fornecer um menu com as seguintes opções:
    1 - Somar
    2 - Subtrair
    3 - Multiplicar
    4 - Dividir
    5 - Sair
    O usuário deverá escolher qual a operação desejada, e somente depois disso o programa solicita os 2 números da operações. Depois de calculado e mostrado o resultado, o programa deve mostrar o menu novamente. O programa só deverá finalizar caso o usuário escolha a opção 5.

20. Faça um programa que mostre a tabuada de um número fornecido pelo usuário.

21. Faça um programa que mostre as tabuadas dos números de 1 a 10.

22. Faça um programa utilizando for para mostrar os caracteres de ‘a’ até ‘z’.

23. Faça um programa que mostre todas as placas de veículos possíveis, começando de AAA-0001 e terminando em ZZZ-9999.

24. ***(Pedágio)*** Em uma praça de pedágio passam diversos tipos de veículos: carros (c), motos (m) e caminhões (h). A tarifa para carros é de R$ 8,00, para motos R$ 4,00 e para caminhões R$ 20,00. Faça um programa, utilizando a estrutura de repetição *while* combinada com a estrutura de seleção *switch*, em que sejam inseridos os códigos dos veículos que passam pelo pedágio (c, m ou h). Não se sabe com antecedência quantos veículos vão passar, assim, encerre a entrada de dados quando for digitado um código igual a 'x'. Para cada veículo, mostre o quanto vai pagar. Ao final, o programa deve mostrar o total arrecadado pela benevolente concessionária.

25. ***(Quem será premiado)*** Um professor deseja premiar o aluno com a maior nota com um livro de Linguagem C. Como são muitos os alunos, faça um programa para automatizar a apuração. O programa deve ler a nota e o número de cada aluno. Ao final, deve mostrar o número do aluno a ser premiado e a nota que ele obteve. (Para simplificar, admita que nenhuma nota é igual à outra.)

26. ***(República)*** Alguns alunos de Sistemas de Informação e de Ciência da Computação resolveram montar uma república, a qual foi batizada de "Rep. Iostream". As despesas comuns, como de costume, são divididas entre os membros. Assim, eles fizeram um programa de computador em que são inseridos os valores das despesas (água, luz, telefone, etc). Não se sabe com antecedência quantos valores serão inseridos, então encerre a repetição quando for digitado o valor -1. Não é necessário especificar a despesa, o programa deve receber apenas o valor. Faça um programa que leia os valores das despesas e mostre o quanto cada um tem de pagar. Note que é necessário entrar com a quantidade de moradores. Detalhe: como os alunos são precavidos, eles acrescentam 10% como fundo de reserva para eventuais imprevistos.

27. ***(Jogo de dados)*** Elabore um programa que simule um jogo de dados. O dado deve ser jogado 787 vezes. Ao final, deve ser exibido a porcentagem que cada lado saiu. Faça um histograma para exibir a porcentagem.

28. ***(Adega)*** O dono de um restaurante deseja saber quantas garrafas de vinho de cada tipo há na adega. Escreva um algoritmo para contar as garrafas e classificar os vinhos pelos tipos 'T' para tinto, 'R' para rosê e 'B' para branco. O algoritmo deverá escrever no final:

    (a) o total de garrafas de vinho tinto

    (b) o total de garrafas de vinho rosê

    (c) o total de garrafas de vinho branco

    (d) o total geral de garrafas

    A quantidade de garrafas é desconhecida, por isso utilize uma estrutura de repetição com teste no final. O final da digitação ocorre quando é informado o tipo 'X'. O algoritmo deverá tratar respostas erradas (diferentes de 'T', 'R', 'B' ou 'X') mostrando uma mensagem "Tipo inválido".

29. ***Tanque de combustível*** A destilaria de álcool Destila S/A deseja controlar o carregamento de caminhões tanque. Considere que há uma fila de caminhões para serem carregados e que os caminhões podem ter capacidades diferentes. Faça um programa que, inicialmente, leia a quantidade

de álcool no tanque (existe um único tanque). A seguir, faça uma estrutura de repetição que leia a capacidade do caminhão e, para cada caminhão carregado, subtraia a quantidade carregada da quantidade existente no tanque. Encerre a leitura quando não houver mais caminhão para ser carregado (digite -1 para encerrar) ou quando não houver mais combustível no tanque. Ao final, exiba quanto de combustível restou no tanque.

Exemplos:

Entradas	**Saída**
100, 30, 30, 30, 10	Restou no tanque: 0
200, 100, 150	Restou no tanque: 0
200, 100, -1	Restou no tanque: 100
200, 100, 20, -1	Restou no tanque: 80

Capítulo 4

Estruturas de Dados Homogêneas (*arrays*)

Nos exemplos apresentados nos capítulos anteriores somente variáveis foram utilizadas. Uma variável, relembrando, é a representação de uma posição na memória que pode armazenar um único valor. Estruturas de dados são elementos que podem armazenar mais de um valor. As estruturas de dados podem ser homogêneas (todos os dados são do mesmo tipo) ou heterogêneas (os dados podem ser de tipos diferentes). Neste capítulo são apresentadas a estrutura homogênea *array*.

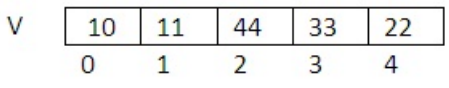
Um *array* é uma coleção de variáveis do mesmo tipo que é referenciada por um nome comum. Um elemento específico de um *array* é acessado por meio de um índice, que indica a posição do elemento. *Arrays* podem ter uma ou mais dimensões. *Arrays* de uma dimensão são comumente chamadas de "vetores", sendo o termo "matriz" é utilizado para as estruturas com mais de uma dimensão. Na literatura em inglês, o termo *array* é utilizado para este tipo de estrutura, independentemente da dimensão.

Arrays são muito utilizados nas operações de busca e ordenação.

4.0 *Arrays* de uma dimensão (vetores)

Um *array* de uma dimensão é um conjunto de elementos do mesmo tipo que ocupam espaços contíguos na memória. Os elementos podem ser referenciados individualmente, por meio do nome do vetor e do índice do elemento. Isto significa que, por exemplo, pode–se armazenar 5 valores do tipo inteiro em um vetor sem ter que declarar 5 diferentes variáveis, cada uma com um identificador diferente.

Um vetor pode ser visualizado como:



em que *v* é o identificador (nome) que representa um vetor de inteiros com 5 elementos. A sintaxe da declaração de um vetor em C é:

```
<tipo> <identificador>[<quantidade de elementos>];
```

Assim, a declaração do vetor da figura acima é:

```
int v[5];
```

que indica um vetor de 5 elementos do tipo inteiro. O primeiro elemento do vetor é o de índice 0 e o último é tamanho -1. Cada um dos elementos podem ser referenciados pelo nome do *array* seguido da posição do elemento entre colchetes ([ ]).Assim, para referenciar o primeiro elemento do *array*, a forma é *v[0]* (tratar o primeiro elemento como v[1] é um erro comum).

Deve–se ter um cuidado especial ao referenciar o último elemento do *array*. No exemplo anterior, em que o tamanho do *array* é 5, o último elemento é o *v[4]*. Tentar referenciar o último elemento como sendo *v[tamanho]* é um erro comum que causa efeitos imprevisíveis.

4.0.0 Inicialização de *arrays*

Arrays devem ser inicializados, assim como ocorre com variáveis. Para inicializar um *array* na declaração, a instrução é:

```
int v[5] = {10, 11, 44, 33, 22};
char vogais[5] = {'a', 'b', 'c', 'd', 'e'};
```

Para inicializar um vetor utilizando uma estrutura de repetição:

```
int i, v[5];
for (i=0; i < 5; i++){
    v[i] = 0;
}
```

4.0.1 Exemplo de leitura e escrita de *arrays*

Considere um *array* de 3 elementos. Os elementos podem ser lidos da seguinte maneira:

```
...
int v[3];
cin >> v[0];
cin >> v[1];
cin >> v[2];
...
```

No entanto, *arrays*, em geral, costumam ter muitos elementos. Por isso, são lidos utilizando–se estruturas de repetição. Considerando que a quantidade de elementos é conhecida, geralmente emprega–se *for* para manipulação de *arrays*. Por exemplo, para um *array* de 100 elementos, o código para leitura é:

```
int v[100]={0}; //declara e inicializa
int i=0;   //índice para percorrer o array
for (i=0; i < 100; i++){
    cin >> v[i];
}
```

Para exibir os elementos de um *array*, também são usadas estrutura de repetição:

```
for (i=0; i < 100; i++){
    cout << v[i];
}
```

4.1 Ordenação

Ordenar elementos seguindo algum critério, por exemplo, do menor para o maior, é uma operação executada por meio de *arrays*. Ordenação é um assunto muito discutido em computação, pois o processo pode consumir muitos recursos quando aplicado a uma quantidade muito grande de dados. Existem diversos algoritmos clássicos para executar a ordenação. No contexto desta disciplina, não cabe discuti-los, apenas vamos mostrar um deles.

4.1.0 Método da Bolha

Um dos algoritmos de ordenação mais simples e conhecido é chamado de "Método da Bolha". Para ordenar os elementos, eles devem estar em um *array*. O método consiste em realizar diversas "passadas" pelo *array*. A cada passada, pares de elementos sucessivos são comparados. Se o elemento comparado for maior que o seguinte, eles são trocados de lugar.

O código do algoritmo é:

```
for (passo = 1; passo < TAM; passo++){
    for (i = 0; i <  TAM - 1; i++){
        if (vet[i] > vet[i+1]){
                mao = vet[i];
                vet[i] = vet[i+1];
                vet[i+1] = mao;
            }
        }
}
```

em que vet é o *array* a ser ordenado e *TAM* é o tamanho deste *array*.

4.2 *Arrays* bidimensionais

Arrays podem ter mais de uma dimensão. Assim, pode–se ter *arrays* de duas, três, ou mais dimensões. Um *array* de duas dimensões geralmente é chamado de matriz e pode ser entendido como uma tabela. A sintaxe da declaração de um *array* de duas dimensões é:

```
<tipo> <identificador>[<linhas>][<colunas>];
```

No exemplo a seguir é declarada uma matriz com 10 linhas e 5 colunas:

```
float tabela[10][5];
```

4.2.0 Exemplo

Considere uma turma de 4 alunos que realizam 3 provas durante o ano. Pode–se utilizar uma tabela para armazenar essas notas, que poderia ser representada como:

3.0	4.5	9.7
8.0	6.0	7.5
3.4	8.8	9.1
6.8	9.9	1.4

Nesta tabela, as colunas representam as notas das provas e as linhas representam os alunos. Para manipular dados em matrizes de duas dimensões, é necessário fazer referência expressa à linha e coluna. Por exemplo, para mostrar o conteúdo da primeira linha e segunda coluna, a instrução é:

```
cout « tabela[0][1];
```

4.2.1 Leitura e exibição de matrizes

Assim como *arrays* de uma dimensão, *arrays* de duas dimensões são lidos por meio de estruturas de repetição. No entanto, como se tem linhas e, para cada linha, diversas colunas, emprega–se uma estrutura de repetição aninhada em outra. Novamente, como as dimensões são conhecidas, a estrutura mais indicada é *for*.

Um exemplo típico de leitura de *array* de duas dimensões é mostrado no trecho de código a seguir.

```
int main(){
    int i=0, j=0;
    int tabela[4][5];
    for (i=0; i < 4; i++){
        for (j=0; j < 5; j++){
            cin >> tabela[i][j];
        }
    }
    return 0;
}
```

Utilizou–se um *for* dentro de outro. Sempre que isto ocorre, para cada iteração do *for* externo, o *for* interno vai executar todas as repetições. Desta forma, para cada linha, serão lidas todas as colunas. É necessário haver uma variável para controlar o índice da linha e outra para controlar o índice da coluna.

✓Não utilize a variável que serve de índice para outros propósitos no corpo da repetição.

O trecho de programa a seguir exibe uma tabela na tela. Observe que depois de exibir uma linha há o comando *cout « endl*. Isto serve apenas para deixar os dados na forma de tabela, caso contrário, seriam exibidos em uma única linha.

```
int main(){
    ...
    for (i=0; i < 4; i++){
        for (j=0; j < 5; j++){
            cout << tabela[i][j] << "  ";
        }
        cout << endl;
    }
    return 0;
}
```

4.3 Exercícios

0. Elementos de um *array* possuem a característica de terem o mesmo _____e _____.

1. O número usado para referenciar um elemento específico de uma *array* é chamado de _____.

2. *Arrays* são empregados especialmente nas operações de _____e _____.

3. *Arrays* de duas dimensões utilizam dois _____para referenciar elementos.

4. Assinale V (verdadeiro) ou F (falso):

   (a) *Arrays* podem armazenar muitos tipos diferentes de valores.

   (b) Um índice de *array* pode ser do tipo *double*.

   (c) O primeiro elemento de um *array* é o elemento de índice zero.

5. ***(Invertendo)***Faça um programa em que o usuário insira 5 números e, em seguida, o programa mostre os 5 números na ordem inversa da qual foram inseridos.

6. ***(Elementos duplicados)*** Escreva um programa utilizando um *array* de uma dimensão para resolver o seguinte problema. Leia 20 números. Para cada número lido, armazene no *array* apenas se ele ainda não foi lido. Ao final, o *array* terá 20 números diferentes. Exiba o *array* completo.

7. ***(Provas)*** Faça um programa para correção de provas. O programa deve armazenar em um *array* chamado *gabarito* as alternativas corretas das questões. A seguir, armazene em outro *array* chamado *resp* as respostas do aluno. Ao final, mostre quantos pontos o aluno fez, considerando que cada questão vale um ponto.

8. ***(Compras)*** Faça um programa que peça ao usuário para inserir 5 itens comprados em uma loja. Para cada item, deve ser informado seu código, quantidade e seu valor unitário. Depois disso, montar na tela um "demonstrativo" contendo a informação do código, quantidade, valor unitário e valor total de cada item, e um valor total geral. Exemplo:

```
Código        Qtde       Val. Unit.     Val. Tot.
-------------------------------------------------
0001            10            0,50          5,00
0007             2            7,50         15,00
0158             1          100,00        100,00
0021             7            7,00         49,00
0008             2            1,00          2,00
-------------------------------------------------
Valor Final:                                171,00
```

Utilize uma estrutura de dados do tipo *array* de duas dimensões para armazenar a tabela.

9. ***(Vestibular)*** Faça um programa que gerencie o resultado do vestibular. Para sua felicidade, só existe um curso e o curso tem 10 vagas. O programa deve manter a lista dos 10 classificados em um *array*. O programa também mantém, em outro *array* (de 20 posições), a lista de espera, que contém os candidatos aprovados, mas excedentes ao número de vagas (caso algum candidato classificado desista, será chamado um da lista de espera). A lista de espera pode ter até 20 candidatos.

   O programa deve solicitar o número do candidato e dizer:

   (a) se ele foi classificado, ou

   (b) se ele está na lista de espera, neste caso, indicar em que lugar da fila de espera ele está, ou

   (c) se ele não foi aprovado.

   Exemplo. Sejam os classificados:

20	32	01	14	99	34	21	02	15	07

   Seja a lista de espera (valor -1 significa que não há candidato):

08	04	10	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

   Exemplos de consultas: Exemplo. Sejam os classificados:

Entrada	**Saída**
01	“classificado”
04	“número 2 na lista de espera
97	“não aprovado”

   Obrigatório usar *array* e estrutura de repetição.

10. Crie um programa que leia uma lista de nomes e depois exiba em ordem alfabética.

Capítulo 5

Caracteres e *arrays* de caracteres (*strings*)

C possui uma série de funções nas bibliotecas da própria linguagem que facilitam a manipulação de caracteres e cadeias de caracteres (*strings*). Essas funções permitem aos programas processarem caracteres, cadeias, linhas e textos completos.

5.0 Fundamentos

String pode ser entendida como um conjunto de caracteres tratados como uma unidade. Uma *string* pode incluir letras, dígitos e caracteres especiais tais como +, -, * e assim por diante.

Uma *string* em C é uma *array* de caracteres (*char*) terminando com o caractere *null* ('\0').

5.1 Declaração de *arrays* de *char*

Um *array* de caracteres pode ser declarado da mesma forma que *arrays* de outros tipos.

```
<tipo> <identificador>[<tamanho>];
```

A declaração:

```
char nome[30];
```

cria um *array* de 30 elementos do tipo *char*. Uma característica dos *arrays* de *char* é que nem sempre todas as posições são utilizadas. Observe que uma *array* de 30 posições deve armazenar no máximo 29 caracteres, pois é necessário terminar a *string* com o '\0'.

Em se tratando de *strings*, é preciso atentar para o fato de que tamanho do *array* é diferente de tamanho do conteúdo do *array*. O caso do *string* anterior é um exemplo típico. Definimos uma *string* para armazenar um nome. Nomes podem variar, como por exemplo "Ana Sá", que contém apenas 6 letras, enquanto que "Joaquim José da Silva Xavier" contém 28 letras. Não se pode esquecer que uma *string* de 6 letras ocupa 7 posições do *array*, pois o programa "automaticamente" coloca o '\0' no final para indicar que não há mais caracteres válidos após este ponto. Geralmente, o que se tem nas posições que sobram é "lixo".

Podemos representar graficamente este exemplo como:

A	n	a		S	á	\0																				

Strings podem ser inicializadas na declaração, das seguintes formas:

```
char estado[] = ``PR'';  //cria um array de 3 posições
char nome[30] = ``Mona Lisa'';
char texto[100] = ``''; //inicializa com vazio
```

✓Assegure–se de que o array tenha um tamanho suficiente para armazenar todos os caracteres necessários mais o '\0'.

✓Ao se armazenar mais caracteres do que a capacidade do *array*, os caracteres excedentes podem sobrescrever outros dados presentes na memória.

5.2 Entrada e saída de *strings*

Cadeias de caracteres (*strings*) não devem ser lidas utilizando o comando *cin*. Uma característica deste comando é ignorar qualquer coisa que venha depois de um espaço em branco. Se tentássemos ler o nome "Machado de Assis", por exemplo, apenas "Machado" seria armazenado. Ao invés de *cin*, para entrada de *strings*, deve-se utilizar a função *gets()*. Para utilizar esta função é necessário inserir a diretiva de compilação `#include <cstdio>`.

Exemplo:

```
char nome[50];
gets(nome);
```

Para exibição de *strings*, pode–se optar pelo comando *cout* ou pela função *puts()*.

5.3 Funções para manipulação de *strings*

A biblioteca <*cstring*> fornece muitas funções úteis para manipulação de cadeias de caracteres, tais como cópia, comparação, determinação do tamanho, entre outras. Algumas destas funções estão resumidas no quadro a seguir.

Função	**Descrição**
strcpy(s1, s2)	Copia a string s2 para string s1.
strcmp(s1, s2)	Compara se s1 é igual a s2. Retorna 0, se as duas são iguais, -1 se s1 for menor que s2 e 1 se s1 for maior que s2.
strcat(s1, s2)	Concatena s2 a s1.
strlen(s)	Retorna o tamanho (quantidade de caracteres) da string s. O '\0' não é considerado.

Estas são apenas algumas das funções para manipulação de *strings*. Consulte a documentação da linguagem[1] para outras funções, se forem necessárias.

[1] http://www.cplusplus.com/reference/cstring/

As funções são necessárias porque alguns operadores não podem ser utilizados com *arrays* de *char*, tais como o operador de igualdade (==), adição (+) e atribuição (=). Assim, para se comparar duas strings utiliza–se a função *strcmp()*, para se atribuir um valor, a função *strcpy()*.

Ne trecho de código a seguir é mostrado um exemplo de comparação de *strings*.

```
int main(){
    char s1[100]= "";
    char s2[100]="";
    gets(s1);
    gets(s2);
    int res = strcmp(s1,s2);
    if (res == 0){
        cout << "Strings iguais" << endl;
    }
    else{
        cout << "Strings diferentes" << endl;
    }
    return 0;
}
```

A operação de atribuição de valor a uma *string* pode ser feita da seguinte forma:

```
int main(){
    char texto[100]= "";
    //...
    strcpy(texto, "Este é o texto a ser atribuído");
    //...
    return 0;
}
```

5.4 Funções para manipulação de caracteres

Existem em C várias funções para manipulação de caracteres individualmente. Algumas destas funções, que exigem a diretiva `#include <ctype>`, são mostradas no quadro a seguir.

Função	**Descrição**
islower(c)	Retorna verdadeiro se *c* é uma letra minúscula.
isupper(c)	Retorna verdadeiro se *c* é uma letra maiúscula.
tolower(c)	Converte *c* para minúscula.
toupper(c)	Converte *c* para maiúscula.
isspace(c)	Retorna verdadeiro se *c* é um espaço.
isalpha(c)	Retorna verdadeiro se *c* é uma letra.

É importante notar que embora uma *string* possa ser tratada como uma unidade, permitindo, por exemplo, ler todos os caracteres de uma vez com a função *gets()*, seus caracteres podem ser manipulados individualmente. Considere a seguinte declaração:

```
char texto[100]= “texto simples”;
```

que cria uma array de 100 char. Nada impede que uma posição seja tratada individualmente. Por exemplo, a instrução:

```
texto[0]= toupper(texto[0]);
```

converte o primeiro caracter para letra maiúscula.

Com caracteres, é possível utilizar operadores de igualdade (==) e atribuição (=), o que não é possível com cadeias de caracteres. O trecho de código a seguir compara se os primeiros caracteres de duas *strings* são iguais:

```
if (s1[0] == s2[0]){
    cout << "São iguais" << endl;
}
else{
    cout << "São diferentes" << endl;
}
```

5.5 *Arrays* de *strings*

Suponha que seja necessário armazenar uma lista de nomes de 30 alunos de uma turma. Neste caso, pode–se pensar em uma lista de 30 linhas. Supondo que cada nome tenha até 50 caracteres, um *array* de duas dimensões (30 linhas e 50 colunas) resolveria o problema.

O trecho de código a seguir exemplifica como seria uma matriz de *strings*. Para simplificar, vamos criar uma tabela com 5 nomes, sendo que cada nome tem até 9 caracteres + '\0' (10 posições).

```
char nomes[5][10] = {"João", "Maria", "Antonio", "Zacarias",
                     "Carlos"};
```

Simbolicamente, esses nomes seriam representados na matriz da seguinte maneira:

J	o	ã	o	\0					
M	a	r	i	a	\0				
A	n	t	o	n	i	o	\0		
Z	a	c	a	r	i	a	s	\0	
C	a	r	l	o	s	\0			

Para se ler os elementos de uma matriz de *strings*, basta indicar a linha em que a *string* lida será armazenada. O programa se encarrega de distribuir os caracteres a partir da coluna 0 da linha indicada. Analise o código:

```
for (int i=0; i < 5; i++){
    gets(nomes[i]);
}
```

Para se exibir o conteúdo da "lista", o procedimento é similar:

```
for (int i=0; i < 5; i++){
    cout << nomes[i];
}
```

O que o comando *cout* faz é exibir os caracteres, começando pelo primeiro, até encontrar um '\0'.

5.6 Exercícios

0. Escreva um programa que leia um caractere e diga se ele é uma letra. Se for letra, diga se é minúscula ou maiúscula.

1. Faça um programa em que o usuário digite uma letra e um número inteiro. Caso a letra seja vogal e o número for par, ou a letra for consoante e o número ímpar, mostre *"BAZINGA!"*. Caso contrário, mostre *"SHAZAM!"*.

2. Escreva um programa que leia uma caractere e leia uma *string*. O programa deve mostrar se o caractere é igual à primeira letra da *string*.

3. Escreva um programa que leia uma caractere e leia uma *string*. O programa deve mostrar quantas vezes o caractere aparece na *string*.

4. Faça um programa que leia um texto. O programa deve contar e exibir quantas vogais há no texto.

5. Faça um programa que leia um texto. O programa deve contar e exibir quantas vogais há no texto, sem contar vogais repetidas. Exemplo:
   Entrada: "Este texto tem vogais"
   Saída: 4

6. ***(Vogais)*** Faça um programa que, dada uma frase, retire todas as vogais da frase. Assim, a frase:
   borboletas amarelas
   seria transformada em
   brblts mrls

   Importante: não é para apenas exibir a frase sem as vogais, é para modificar o *array* (retirar as vogais) ou criar um novo *array* sem elas.

7. Faça um programa que converta todas as letras de uma frase para maiúsculas. Exemplo:
   Entrada: esTA EH uMa frasE
   Saída: ESTA EH UMA FRASE

8. Faça um programa que converta a primeira letra de cada palavra para maiúscula e as demais para minúsculas. Exemplo:
   Entrada: esTA EH uMa frasE
   Saída: Esta Eh Uma Frase

9. ***(Telefone)*** Jordoel é muito paquerador, porém muito supersticioso. Quando ele conhece uma garota, ele pede o celular dela, mas só vai em frente se for um número de sorte. De acordo com seus cálculos numerológicos, um número traz sorte quando:

   (a) não possui o dígito 6

   (b) o último dígito é menor que 3

   (c) possui o dígito 2 ou 8

   Faça um programa que ajude ao Jordoel a determinar se um número é de sorte ou não.
   Entrada: um *array* de *char* com o número de telefone
   Saída: uma mensagem dizendo se é um número de sorte ou não.

   Exemplos:
   9745-8222 → "número de sorte"
   9745-6322 → "não é um número de sorte"
   9788-8227 → "não é um número de sorte"
   9834-9991 → "número de sorte"

10. ***(Dieta)*** Falafel está com problemas de excesso de peso, então ele foi ao médico e este lhe prescreveu uma dieta. A dieta é muito simples: dada uma lista de alimentos, ele só pode comer se o alimento estiver na lista e se ainda não foi consumido no dia. Faça um programa que controle a dieta de Falafel. Inicialmente, o programa deve pedir a dieta recomendada pelo médico, na forma de uma lista de até 26 caracteres. A seguir, cada vez que Falafel desejar comer, ele informa ao programa o alimento. O programa deve dizer *"pode comer"*, se o alimento estiver na lista e ainda não tiver sido comido, ou *''não pode comer"*, caso contrário. Encerre a digitação quando for digitado '#' para o código do alimento.
    Exemplos:
    Seja a dieta: ABFGHIK
    Casos de teste:
    B → "pode comer"
    F → "pode comer"
    I → "pode comer"
    R → "não pode comer"
    B → "não pode comer"
    #

11. ***(Invertendo)*** Crie um programa que leia um texto. Em seguida, exiba o texto "de trás para frente".

12. ***(Misturando)*** Escreva um programa que leia dois textos. O programa deve gerar um terceiro texto em que a primeira letra seja a primeira letra do primeiro texto. A segunda letra deve ser a primeira letra do segundo texto e assim por diante. Exiba o texto final.

Capítulo 6

Ponteiros

Ponteiros, ou apontadores (*pointers*), é um dos temas mais complexos da linguagem C. Embora ponteiros sejam pouco utilizados no dia a dia, é fundamental compreendê-los para entender outros conceitos fundamentais como estruturas de dados dinâmicas, por exemplo, que dependem de ponteiros. Ponteiros são variáveis que guardam endereços da memória. Em outras palavras, uma variável ponteiro guarda o endereço de outra variável[1].

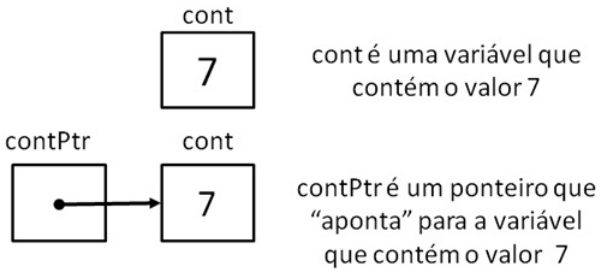
Para entender ponteiro, é preciso ter em mente o conceito de variável. Quando uma variável é declarada em um programa duas informações são fornecidas: o tipo e o nome da variável. Por exemplo:

```
int i;
```

O compilador utiliza essas informações para alocar um espaço na memória de acordo com o tipo da variável (geralmente 4 *bytes* para inteiros). Além disso, o compilador cria uma tabela de símbolos com os identificadores e respectivos endereços. Assim, ao encontrar uma referência a *i*, o programa procura na tabela qual o endereço que deve ser lido ou escrito.

6.0 Declaração e inicialização

No exemplo mostrado na figura a seguir, a variável *cont* é uma variável do tipo inteiro. O nome da variável *diretamente* referencia um valor. Já a variável *contPtr* é do tipo ponteiro e, *indiretamente*, referencia uma variável que contém um valor.



Ponteiros, como qualquer variável, devem ser declarados antes de serem usados. A declaração:

[1]Para conhecer mais sobre ponteiros: http://cesarakg.freeshell.org/pointers.html

```
int cont=0;
int *contPtr=NULL;
```

especifica que a variável *contPtr* é do tipo ponteiro, pois o identificador é precedido de *. Uma variável ponteiro deve ser do mesmo tipo da variável cujo endereço ela vai referenciar. No exemplo anterior, diz-se que *contPtr* aponta para um objeto do tipo inteiro. Incluir as letras *ptr* ajuda a identificar a variável como sendo um ponteiro.

Ponteiros podem ser inicializados na declaração ou por atribuição de valor posteriormente. Na declaração, um ponteiro pode ser inicializado com 0 (zero), *NULL* (*NULL* é preferível) ou um endereço de alguma variável.

6.1 Operações com ponteiros

O operador de endereço *&* é um operador unário que retorna o endereço de seu operando. Por exemplo, considerando a definição:

```
int cont = 5;
int *contPtr;
```

o comando:

```
contPtr = &cont;
```

atribui o endereço da variável *cont* à variável ponteiro *contPtr*. Observe que não é o fato de a variável ponteiro ter o mesmo nome da variável "apontada" acrescido de "Ptr" que associa o endereço ao ponteiro, e sim a atribuição do endereço ao ponteiro.

O operador unário * retorna o valor do objeto para o qual seu operando aponta. Por exemplo:

```
cout « *iPtr;
```

irá exibir o conteúdo do endereço apontado por *iPtr*. Por outro lado,

```
cout « iPtr;
```

irá exibir o conteúdo da variável *iPtr*.

Na Tabela 6.0 são mostradas algumas operações com ponteiros e endereços.

Operador	Significado
int i	declara uma variável do tipo inteiro
int *iPtr	declara um ponteiro para um inteiro
cout « &i	exibe o endereço de i
cout « iPtr	exibe o conteúdo de iPtr (um endereço)
cout « *iPtr	exibe o conteúdo do endereço contido em iPtr
*iPtr = 7	atribui 7 ao endereço apontado por iPtr

Tabela 6.0: Operações com ponteiros.

Para ilustrar as operações com ponteiros, acompanhe o programa a seguir:

```
int main()
{
    int x;       // x é do tipo inteiro
    int *xPtr;   // xPtr é um ponteiro para um inteiro
    x = 7;
    xPtr = &x;   // xPtr recebe o endereço de x
    cout << "Endereço de x: " << &x << endl;
    cout << "Valor de xPtr: " << xPtr << endl << endl;
    cout << "Valor de x: " << x << endl;
    cout << "Conteúdo da variável apontada por *xPtr: " << *xPtr;
    *xPtr = 9;
    cout << "Valor de x: " << x << endl;
    return 0;
}
```

A execução do programa resulta na seguinte saída[2]:

```
Endereço de x: 0x22ff08
Valor de xPtr: 0x22ff08

Valor de x: 7
Valor da variável apontada por *xPtr: 7
Valor de x: 9
```

6.2 *Arrays* e ponteiros

O nome de um *array* é um ponteiro, nada além disso[3]. A declaração *int k[10]* reserva 40 *bytes* contíguos, espaço necessário para armazenar 10 inteiros. No entanto, o programa não mantém 10 entradas na tabela de símbolos e sim apenas uma, o endereço da posição inicial do *array*. Considere o programa a seguir:

[2] O endereço *0x22ff08* foi alocado durante a execução e, naturalmente, varia de uma execução para outra.
[3] Perry, G. *C by Example*. Indianapolis: Que, 2000.

```
int k[10]={ 1, 2, 3, 4 ,5, 6, 7, 8, 9, 10};
cout << k << endl;
cout << &k << endl;
cout << &k[0] << endl;
```

As três instruções *cout* produzirão a mesma saída, que é o endereço inicial de k, pois um *array* é mantido pelo programa como um ponteiro para o endereço da posição inicial.

Analise o trecho seguinte:

```
int main(){
    int k[10]={ 1, 2, 3, 4 ,5, 6, 7, 8, 9, 10};
    cout << *k << endl;
    cout << k[0] << endl;
    cout << *(k+1) << endl;
    return 0;
}
```

O programa exibirá o número 1, que é o conteúdo da posição de memória apontado por *k. Em seguida, exibirá o número 1 novamente (k[0]). Por último, exibirá o número 2, que é o conteúdo da posição de memória apontado por k+1.

A aritmética de ponteiros é um pouco diferente. No exemplo anterior, quando fazemos k+1, na verdade estamos somando 4 *bytes*, pois um inteiro ocupa quatro *bytes*. Somar 1 significa somar a quantidade de *bytes* equivalentes ao tipo de variável armazenada no *array*. É como se o ponteiro se "movesse" 4 *bytes* para frente.

6.3 Exercícios

0. Na expressão *float *f* o que é do tipo *float*:
   (a) a variável f
   (b) o endereço de f
   (c) a variável apontada por f
   (d) nenhuma das alternativas anteriores

1. Crie um programa em que haja duas variáveis inteiras *x* e *y*. Crie dois ponteiros *xPtr* e *yPtr* e atribua o endereço de *x* a *xPtr* e o de *y* a *yPtr*. Troque os valores de *x* e *y* utilizando apenas as variáveis *xPtr* e *yPtr*.

2. Qual a saída do programa a seguir?

```
char texto[30] = "C eh legal";
char *p = texto;
while(*p != '\0'){
    cout << *p;
    p += 2;
}
```

3. Qual a saída do programa a seguir?

```
int i=99, j;
int *p;
p = &i;
j = *p + 100;
cout << j;
```

4. Qual a saída do programa a seguir?

```
int a=5, b=12;
int *p, *q;
p = &a;
q = &b;
int c = *p + *q;
cout << c;
```

5. Escreva um programa que exiba os elementos de um *array* de inteiros na ordem inversa (do último para o primeiro) utilizando ponteiro.

Capítulo 7

Funções

Subprogramas, que em Linguagem C são chamados de funções, são blocos de construção que permitem modularizar programas. Este capítulo traz os conceitos de subprogramas, passagem de parâmetros entre funções e introduz o conceito de função recursiva.

7.0 Introdução

Em Linguagem C, todas as instruções do programa devem estar dentro de um bloco de código (função). Todo programa em C deve ter pelo menos uma função, que é a função *main( )*. Para pequenos programas, é possível escrever todo o código dentro da função *main( )*. No entanto, para programas com muitas linhas de código, isto se torna impraticável. Assim, o comum é que um programa seja constituído de vários módulos, relativamente pequenos, de forma que cada bloco seja responsável por uma funcionalidade específica. Não importa quantas funções o programa tenha, a execução sempre se inicia pela função *main( )*. Há diversas razões para se utilizar funções, as principais são:

- Não é necessário repetir certos trechos de código;
- É mais fácil alterar funcionalidades;
- É mais prático para encontrar e corrigir erros; e
- Pode-se criar bibliotecas de funções reutilizáveis.

A própria linguagem já traz um conjunto grande de funções predefinidas e organizadas em bibliotecas. Assim, para calcular a raiz quadrada de um número, o programador não precisa escrever o código para isso, basta utilizar a função da linguagem (que em C é *sqrt()*). Além dessas funções, o programador escreve suas próprias funções.

7.1 Conceitos

Para ilustrar alguns conceito sobre funções, considere o programa a seguir, que calcula e exibe a raiz quadrada de um número utilizando a função *sqrt()*:

```
/*  1*/ #include <iostream>
/*  2*/ #include <cmath>
/*  3*/
/*  4*/ using namespace std;
/*  5*/
/*  6*/ int main()
/*  7*/ {
/*  8*/    double num=0, raiz=0;
/*  9*/    cout << "Digite o número: ";
/* 10*/    cin >> num;
/* 11*/    raiz = sqrt(num);
/* 12*/    cout << "\n\nA raiz é: " << raiz << endl;
/* 13*/    return 0;
/* 14*/ }
```

A função *sqrt()* é uma função da biblioteca *cmath*, própria da linguagem. Por isso, é necessário o `#include <cmath>` da linha 2. Na linha 11, *sqrt( )* é utilizada. Diz-se que a função *main( )* "chamou" a função *sqrt( )*; por seu lado, a função *sqrt( )* foi "chamada" pela função *main( )*. A variável entre parênteses é chamada de *parâmetro* e é "passada" para a função. A função *sqrt( )* calcula o valor da raiz e devolve para a função *main( )*, sendo que o valor é atribuído à variável *raiz*. Geralmente se diz que a função *retorna* um valor.

Duas características podem ser resaltadas dos conceitos anteriores:

- Toda unidade de programada chamadora é suspensa durante a execução do programa chamado, o que implica na existência de somente uma função em execução em determinado momento[1]; e
- O controle sempre retorna ao chamador quando a execução da função chamada termina.

7.2 Sintaxe

Uma função tem a forma:

```
<tipo de retorno> <identificador> (<parâmetros de chamada>){
    //corpo da função
}
```

O *tipo de retorno* é o tipo do dado (*int, float, char*) que a função retorna ao final da execução, como o resultado de um cálculo, por exemplo. Uma função retorna um e apenas um dado. O *identificador* é qualquer identificador válido. Por convenção, nomes de funções são escritos em minúsculas. Os *parâmetros de chamada* são valores que são recebidos pela função ao ser chamada. Podem variar de zero a 256 elementos. Funções podem:

- Não receber parâmetros quando chamadas e não devolver valor após serem executadas;
- Receber parâmetros mas não retornar valor;
- Não receber parâmetros, mas retornar valor; e
- Receber parâmetros e devolver valor.

[1] Em programas paralelos que executam em múltiplos processadores isto pode ser diferente

7.2.0 Funções do tipo void

O tipo *void* (vazio) é utilizado quando a função não retorna valor. Para exemplificar, observe o código abaixo cuja função é limpar a tela. Para isto, não é necessário passar nenhum parâmetro e não se deseja retornar nenhum valor após a função ser executada:

```
void limpaTela(void){
    system("cls");
    return;
}
```

Quando uma função não recebe valores, pode-se emitir o *void* entre parênteses. No entanto, nunca se pode omitir o tipo de retorno. O código abaixo é equivalente ao anterior:

```
void limpaTela(){
    system("cls");
    return;
}
```

7.2.1 Funções que recebem parâmetros

Em alguns casos, para que a função execute a computação requerida, é necessário que receba valores. O exemplo a seguir mostra um caso em que uma função recebe dois inteiros e imprime qual o maior deles.

```
    void acharMaior (int x, int y){
        if (x > y) {
            cout << "O maior é: "<< x;
        }
        else if ( y > x) {
                cout << "O maior é" << y;
            else {
                cout << "São iguais";
            }
        }
        return;
    }
```

Variáveis podem ser declaradas dentro dos parênteses da função. No entanto, neste caso, cada variável deve ser precedida do tipo. A declaração a seguir está errada:

```
    void acharMaior (int x, y){
        ...
```

7.2.2 Funções que retornam valor

Pode acontecer de uma determinada função, ao ser executada, devolver um dado à função chamadora sem que seja necessário receber parâmetros. A função a seguir, ao ser chamada, retorna o ano corrente com base na data atual do sistema.

```
int obterAnoCorrente ( ){
    int ano;
    time_t agora;
    time (&agora);
    struct tm *ptm= localtime(&agora);
    ano = (ptm->tm_year) + 1900;
    return (ano);
}
```

7.2.3 Funções que recebem parâmetros e retornam valor

O caso mais comum de funções são as do tipo que recebem parâmetros, executam alguma computação com esses valores e retornam um resultado. A função a seguir recebe o comprimento do raio de um círculo e devolve a área respectiva.

```
double calculaArea (double raio){
    double area = 0;
    area = M_PI * (raio * raio);
    return (area);
}
```

7.3 Integrando funções

Funções chamam funções. Como mencionado anteriormente, quando uma função chama outra, a execução da primeira é suspensa e a função chamada começa a ser executada. Considere o programa da Figura 7.0, que recebe um inteiro e retorna o cubo deste número.

Na linha 12 a função *calculaCubo( )* foi chamada e a variável *num* foi passada para a função. Neste momento, a função *main( )* é interrompida e a função *calculaCubo( )* inicia a execução. O valor da variável passada é recebido na variável *n* (linha 17). Na linha 18 é calculado o cubo de *n* e atribuído à variável *cb*. Na linha 19 o valor calculado é retornado ("enviado de volta") para a função chamadora. Como há um retorno de valor, é necessário que esse dado seja tratado na função chamadora. No caso, o dado é atribuído à variável *cubo*.

```
/*  1*/  #include <iostream>
/*  2*/
/*  3*/  using namespace std;
/*  4*/
/*  5*/  int calculaCubo(int);
/*  6*/
/*  7*/  int main()
/*  8*/  {
/*  9*/       int num=0, cubo=0;
/* 10*/       cout << "Digite um número: ";
/* 11*/       cin >> num;
/* 12*/       cubo = calculaCubo(num);
/* 13*/       cout << "O cubo do número é: " << cubo;
/* 14*/       return 0;
/* 15*/  }
/* 16*/
/* 17*/  int calculaCubo(int n){
/* 18*/       int cb = n * n * n;
/* 19*/       return cb;
/* 20*/  }
```

Figura 7.0: Exemplo de chamada à função.

7.3.0 Protótipos de funções

No programa anterior, na linha 5, foi definido o *protótipo* da função. O protótipo da função informa ao compilador o tipo de dado retornado pela função, a quantidade de parâmetros que a função recebe, o tipo dos parâmetros e a ordem em que serão recebidos. O compilador utiliza os protótipos para validar a chamada à função (por exemplo, quando encontra a chamada da linha 12). A declaração

```
int calculaCubo(int);
```

informa ao compilador que existe uma função chamada *calculaCubo( )* que recebe um único inteiro como parâmetro e retorna um inteiro.

Todas as funções do programa devem ser prototipadas.

7.3.1 Variáveis locais e globais

Variáveis que são declaradas dentro de uma função são chamadas de variáveis *locais*. Isto significa que seu escopo (seu limite) é o corpo da função. Em outras palavras, elas não existem e não podem ser utilizadas fora da função em que foram declaradas. No exemplo da Figura 7.0, a variável *num* foi declarada dentro da função *main()*. Desta forma, ela não pode ser utilizada dentro da função *calculaCubo( )*.

Por outro lado, se uma variável for declarada fora de qualquer função, ela tem escopo *global*, o que quer dizer que pode ser utilizada em qualquer função. Na Figura 7.1 é mostrado uma exemplo de variável global (*resultado*), declarada na linha 1. Esta variável pode ser manipulada em qualquer função do programa. Variáveis globais DEVEM SER EVITADAS.

```
/*  1*/  int resultado=0;
/*  2*/
/*  3*/  int main(){
/*  4*/       ...
/*  5*/  }
/*  6*/
/*  7*/  int calculaCubo(int n){
/*  8*/       ...
/*  9*/  }
```

Figura 7.1: Exemplo de variável global.

7.4 Métodos de passagem de parâmetros

Os métodos de passagem de parâmetros são as maneiras pelas quais uma função chamadora envia dados para uma função chamada. Antes de analisar cada método, é preciso apresentar duas definições:

Parâmetros reais: são os parâmetros passados na função chamadora

Parâmetros formais: são os parâmetros declarados no cabeçalho da função; em outras palavras, são as variáveis que irão receber os valores passados pela função chamadora

A correspondência (vinculação) entre os parâmetros reais e formais é feita simplesmente pela posição. Em linguagem C, as principais formas de passagem de parâmetros são passagem por valor e passagem por referência. *Arrays* são passados por nome, que pode ser considerado um terceiro método.

7.4.0 Passagem por valor

Quando um parâmetro é passado por valor, o valor do parâmetro real é usado para inicializar o parâmetro formal correspondente. O parâmetro formal passa a se comportar como uma variável local. Na Figura 7.2 é mostrado um exemplo desse tipo de passagem. A variável *num* (parâmetro real) é copiada na variável *n* (parâmetro formal). Qualquer alteração na variável *n* na função, em nada afeta a variável *num* (lembre–se: é apenas uma cópia).

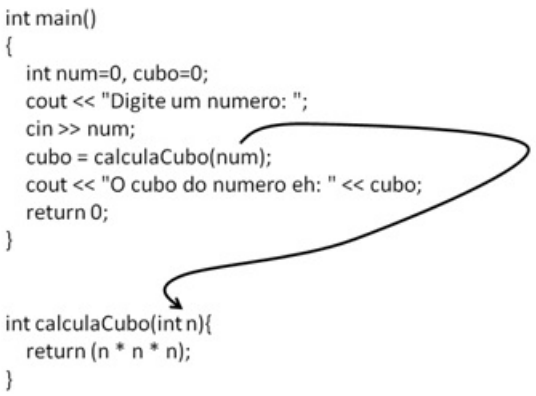


Figura 7.2: Exemplo de passagem por valor.

7.4.1 Passagem por referência

A passagem por referência, em vez de transmitir valores de dados, transmite um caminho de acesso aos dados. Na prática, é passado um endereço. Isto permite que a função chamada tenha um caminho de acesso à célula que contém o parâmetro real. De certa forma, o parâmetro real é compartilhado com a função chamada. Assim, qualquer modificação dentro da função chamada, estará modificando o parâmetro real. No exemplo da Figura 7.3 é mostrada esse tipo de passagem. Os endereços de *x* e *y* são transmitidos à função *troca( )* (observe o operador de endereço & antes do nome da variável). Uma vez que foram passados endereços, os parâmetros formais devem ser do tipo ponteiros (observe o * no cabeçalho da função na linha 14). Na linha 15, a variável *mao* recebe o conteúdo da célula de memória apontada por **a*, que é o endereço da variável *x*. Na linha 16, a célula de memória cujo endereço está guardado em **a* recebe o conteúdo da célula de memória cujo conteúdo está armazenado em **b* (variável *y*). Finalmente, a célula de memória apontada por **b* recebe o valor de *mao*. O resultado é que os valores das variáveis *x* e *y* são trocados entre si.

```
/*  1*/    void troca(int *, int *);
/*  2*/
/*  3*/    int main()
/*  4*/    {
/*  5*/        int x, y;
/*  6*/        x = 1;
/*  7*/        y = 2;
/*  8*/        troca(&x, &y);  //passa o endereco de x e y
/*  9*/        cout << "x:  " << x << endl;  // mostra 2
/* 10*/        cout << "y:  " << y << endl;  // mostra 1
/* 11*/        return 0;
/* 12*/    }
/* 13*/
/* 14*/    void troca(int *a, int *b){
/* 15*/        int mao = *a;
/* 16*/        *a = *b;
/* 17*/        *b = mao;
/* 18*/        return;
/* 19*/    }
```

Figura 7.3: Exemplo de passagem por referência.

7.4.2 Passagem *arrays* para funções

Quando uma função passa um *array* para outra função, o que está sendo passado é, na verdade, um ponteiro para o *array*. A diferença com a passagem por referência é que não se usa o operador de endereço (&) no momento da passagem. Na Figura 7.4 é mostrado um exemplo de passagem de *array* para função. Na linha 7, a função *absoluto( )* é chamada, tendo como argumento o nome do *array v*, sem dimensão ou colchetes. Implicitamente, o que está sendo passado é o endereço do *array*. No cabeçalho da função *absoluto( )* (linha 11), o *array w* é "um outro nome" para o *array v*. Tudo o que for alterado em *w* será alterado em *v*.

```
/*  1*/ void absoluto(int []);
/*  2*/
/*  3*/ int main()
/*  4*/ {
/*  5*/     int v[5];
/*  6*/     // código para ler o array omitido
/*  7*/     absoluto(v);
/*  8*/     return 0;
/*  9*/ }
/* 10*/
/* 11*/ void absoluto(int w[]){
/* 12*/     int i;
/* 13*/     for (i=0; i < 5; i++){
/* 14*/        if (w[i] < 0){
/* 15*/            w[i] = w[i] * -1;
/* 16*/        }
/* 17*/     }
/* 18*/     return;
/* 19*/ }
```

Figura 7.4: Exemplo de passagem de vetor.

O cabeçalho da função *absoluto( )* poderia ser substituido por

```
void absoluto(int w[5]);
```

ou então por

```
void absoluto(int *w);
```

que os resultados seriam os mesmos.

No caso de *arrays* de duas dimensões (matrizes), as construções anteriores não são válidas. É necessário escrever as dimensões da matriz. Uma função que recebesse uma matriz de 3 linhas e 4 colunas deveria ter o cabeçalho:

```
void absoluto(int matriz[3][4]){
    ...\\
}
```

7.5 Funções recursivas

Os programas vistos são constituídos de funções que chamam outras funções, de maneira hierárquica. Para alguns tipos de problemas, como algoritmos de busca e classificação, pode ser útil que a função chame a si mesma.

Função recursiva é um tipo de função que *chama a si mesma*. Há dois elementos essenciais em uma função recursiva:

Caso base: a parte do problema que a função sabe resolver. Se a função é chamada com o caso base, simplesmente retorna um resultado.

Caso recursivo: situação em que ainda não se chegou à solução e é necessário chamar novamente a função, agora com uma parte "menor" do problema.

As chamadas recursivas, uma vez que dividem o problema em um problema menor, em um dado momento chegam no caso base.

7.5.0 Cálculo do fatorial por recursividade

O fatorial de um número inteiro não negativo *n*, ou *n!* (n fatorial), é o produto:

```
n * (n-1) * (n-2) * ...  * 1
```

1! é igual a 1 e 0!, por definição, é 1. O fatorial pode ser calculado de maneira iterativa (estrutura de repetição) da seguinte maneira:

```
fatorial = 1;
for (cont=num; cont >= 1; cont--)
    fatorial = fatorial * cont;
```

Embora este problema seja simples de resolver de forma iterativa, ele serve de exemplo para ilustrar o conceito de recursão. A solução recursiva para o cálculo do fatorial seria:

```
n!  = n * (n-1)!
```

O programa em C correspondente:

```
int fatorial (int n){
    //caso base
    if (n <= 1){
        return 1;
    }
    //caso recursivo
    else{
        return (n * fatorial(n-1));
    }
}
```

7.6 Exercícios

0. ***(Chamada de função)*** Escreva uma função que, ao ser chamada pela *main( )*, exiba a mensagem "Olá pessoal".

1. ***(Passagem de parâmetro)*** Escreva um programa em que, na função *main( )*, seja lido um número inteiro. Passe este número (por valor) para a função *exibe( )* que deve mostrar o valor na tela.

2. ***(Passagem com retorno)*** Escreva um programa em que, na função *main( )*, sejam lidos dois números. Passe estes números para a função *somar( )* que deve somar os dois números. O resultado da soma deve ser retornado à função *main( )*, que deve exibir este valor.

3. ***(Quadrado)*** Faça um programa que, na função *main( )* sejam lidos um inteiro *n* e um caractere *c*. Chame uma função *desenhar( )* que deve mostrar um quadrado com *n* linhas e *n* colunas com o caractere dado. Exemplo:
   Entrada: 3, #
   Saída:
   ###
   ###
   ###

4. ***(Estacionamento)*** Um estacionamento cobra a taxa mínima de R$ 5,00 por até 3 horas de permanência. A partir da terceira hora, cobra R$ 3,00 por hora. O máximo que o cliente paga, no entanto, é o valor da diária, que é de R$ 20,00. (Assuma que nenhum carro fique estacionado por mais de 24 horas). Escreva um programa em que, na função *main( )*, seja informada a quantidade de horas que o carro ficou estacionado. Chame a função *calcula( )*, que deve retornar o quanto o cliente tem que pagar.

5. ***(Inversão)*** Escreva uma função que receba um inteiro e retorne o número com os dígitos invertidos. Exemplo:
   Entrada: 7631
   Saída: 1367

6. ***(Juros)*** Faça um programa em que sejam informados o valor da aplicação, a taxa de juros e a quantidade de meses que o dinheiro será investido. Chame a função *calcularJuros( )*, que deve retornar o quanto o investidor receberia se juros. (Pesquise na internet a fórmula para cálculo de juros compostos).

7. ***(Idade)*** Crie um programa em que, na função *main( )*, seja informada a sua data de nascimento. A data deve ser passada para a função *calculaIdade( )*. A função *calculaIdade( )* deve chamar a função *obterAnoCorrente( )* para pegar o ano atual com base no relógio do sistema operacional (veja na subseção 7.2.2 a função pronta). A sua idade calculada deve ser mostrada na função principal. Considere apenas o ano de nascimento, ignorando mês e dia.

8. ***(Somando intervalo)*** Faça um programa em que, na função *main( )*, sejam lidos 2 inteiros. Passe esses valores para a função *calcularIntervalo( )* que deve retornar a soma de todos os números informados, excetuando-se os números recebidos. Exiba o resultado na função *main( )*. Exemplo:
   Entrada: 2, 7
   Saída: 18
   (Saída = 3 + 4 + 5 + 6)

9. ***(Convertendo horas)*** Faça uma função que receba um valor representando horas e retorne a quantidade de minutos correspondentes. Exemplo:
   Entrada: 3.5
   Saída: 210 minutos

10. ***(Comuns)*** Escreva um programa que tenha a função *listarComuns( )*. Esta função recebe dois *arrays* de inteiros e exibe os elementos comuns a ambos. Exemplo:
    a= { 2, 5, 8, 3} e b= { 3, 9, 2, 7}
    saída: 2, 3

11. ***(Array 1)*** Faça um programa que tenha um menu na função *main( )* com as opções:
    1) Preencher *array*
    2) Exibir *array*
    3) Exibir *array* invertido
    4) Sair
    Um *array* deve ser declarado na função *main( )*. A opção 1 chama uma função que lê os elementos do *array*, a opção 2, chama uma função que exibe na tela e a opção 3 chama uma função que exibe do último elemento para o primeiro.

12. ***(Array 2)*** Faça um programa em que, na função *main( )*, seja declarado um *array* de inteiros de 10 posições. Leia, ainda na *main( )*, um número inteiro. Passe o *array* e o número lido para uma função que deve colocar o número lido na primeira posição do *array* e em seguida preencher as demais posições com o dobro do valor contido na posição anterior.

Capítulo 8

Registros (*structs*)

Registros, chamados *structs* (abreviatura de *structures*) na linguagem C, são estruturas de dados heterogêneas. Diferentemente dos *arrays*, que são coleções de dados do mesmo tipo, as *structs* podem conter dados de diferentes tipos, inclusive outras *structs*. *Structs* são muito utilizadas para definir registros que são armazenados em arquivos.

8.0 Definição de *structs*

Structs são tipos de dados derivados, pois são construídas utilizando objetos de outros tipos. Considere o seguinte exemplo:

```
struct Aluno{
    int matricula;
    char nome[50];
};
```

A palavra *struct* inicia a definição da estrutura. O identificador *Aluno* nomeia a estrutura e será usado nas futuras declarações de variáveis do tipo desta estrutura. As variáveis declaradas dentro das chaves são os membros (ou campos) da *struct*. Esquecer o ponto-e-vírgula no final da definição é um erro comum de programação.

Definir uma *struct* não faz com que o programa reserve espaço na memória para ela, apenas cria um novo tipo de dado. Variáveis do tipo registro são declaradas da mesma forma que variáveis de tipos primitivos, porém é necessário preceder o nome do tipo com a palavra *struct*. No exemplo a seguir é declarada uma variável do tipo da *struct* definida anteriormente:

```
struct Aluno aluno1, aluno2;    /*  1  */
struct Aluno lista[30];         /*  2  */
```

No caso, *Aluno* é o tipo e *aluno1* e *aluno2* são as variáveis. Na linha 2 do exemplo é declarado um *array* com capacidade para armazenar 30 elementos do tipo *Aluno*.

Em alguns compiladores pode–se omitir a palavra struct na declaração de variáveis. O código a seguir produz o mesmo efeito do que o fragmento anterior.

```
Aluno aluno1, aluno2;    /*  1  */
Aluno lista[30];         /*  2  */
```

✓Uma boa prática é iniciar o nome da *struct* em maiúscula, para diferenciar dos identificadores das variáveis.

8.1 Acessando os membros da *struct*

Dois operadores são usados para acessar membros de registros: o operador ponto (.) e o operador seta (->) usado para ponteiros de estruturas.

Um membro de uma variável *struct* é acessado por meio do operador ponto. Continuando o exemplo anterior, para entrar com a matrícula e com o nome de *aluno1*, o código é:

```
...
Aluno aluno1;
cin >> aluno1.matricula;
cin.ignore();               //necessário entre "cin" e "gets()"
gets(aluno1.nome);
...
```

✓Um erro comum de programação é tentar acessar o membro da *struct* usando apenas o nome do membro.

8.2 Inicialização

Para inicializar uma variável do tipo *struct* deve–se inicializar cada membro individualmente, com valores apropriados. Exemplo:

```
...
Aluno aluno1;
aluno1.matricula = 0;
strcpy(aluno1.nome, " ");
...
```

8.3 Operações válidas com *structs*

As operações permitidas com structs são:

0. atribuir uma variável do tipo *struct* à outra variável do mesmo tipo;
1. obter o endereço (&) de uma variável do tipo *struct*;
2. acessar os membros da variável; e
3. utilizar o operador *sizeof* para determinar o tamanho da variável.

Uma variável do tipo *struct* só pode ser atribuída (operador =) à outra do mesmo tipo. No exemplo a seguir, *aluno1* foi copiado para *aluno2* por meio do operador de atribuição (1). Isso só é possível porque as duas variáveis são do mesmo tipo.

```
...
Aluno aluno1, aluno2;
aluno1.matricula = 0;
strcpy(aluno1.nome, " ");
aluno2 = aluno1;            (1)
...
```

Os operadores relacionais de igualdade (==) e diferença (!=) não podem ser utilizados em *structs*, no entanto, os campos da variável podem ser comparados. A operação a seguir é permitida:

```
...
if (aluno1.matricula == aluno2.matricula){
    cout << "Matriculas iguais";
}
...
```

No entanto, a comparação a seguir não é válida:

```
...
if (aluno1 == aluno2){
...
```

O operador *sizeof* retorna o tamanho em *bytes* de uma variável do tipo *struct*, de acordo com a especificação do compilador utilizado. Por exemplo, a instrução:

```
...
Aluno aluno1;
cout << sizeof(aluno1);
...
```

determina o tamanho em *bytes* da variável *aluno1*. Isto é necessário quando se realiza leitura e gravação em arquivos em disco. Arquivos são tema do próximo capítulo.

8.4 Utilizando *structs* em funções

Uma *struct* pode ser passada para uma função de três maneiras: 1) passando a *struct* completa, 2) passando um campo da *struct* ou 3) passando um ponteiro para a *struct*. Quando uma estrutura ou campo dela é passado para uma função, a passagem é por valor, o que significa que as modificações dentro da função chamada não afetam a estrutura original. Passando–se a estrutura por referência, o endereço da variável é passado, então qualquer modificação dentro função chamada afeta a variável original na função chamadora. *Arrays* de estruturas, como qualquer *array*, são passados por nome (referência implícita).

8.4.0 Passagem por valor

O exemplo a seguir mostra a passagem de uma *struct* por valor. A variável *aluno* declarada na função *main()* é passada para a função *mostraDados()*, que exibe os campos da variável.

```
#include <iostream>
#include <cstdio>

using namespace std;

struct Aluno{
    int matricula;
    char nome[50];
};

void mostraDados(Aluno);  //protótipo

int main(){
    Aluno aluno1;
    mostraDados(aluno1);
    return 0;
}

void mostraDados(Aluno a){
    cout << a.matricula;
    cout << a.nome;
    return;
}
```

8.4.1 Passagem por referência

A passagem por referência ocorre quando o endereço da variável é passada para a função. Qualquer modificação na função chamada altera a variável original na função chamadora. No exemplo a seguir uma variável do tipo *Aluno* é passada para a função *lerDados()* por referência. Observe que não é necessário retornar os dados.

```
#include <iostream>
#include <cstdio>

using namespace std;

struct Aluno{
    int matricula;
    char nome[50];
};

void lerDados (Aluno *);

int main(){
    Aluno aluno1;
    lerDados(&aluno1); //passagem por referencia            (1)
    cout << "\n\nMatricula: " << aluno1.matricula ;         (5)
    cout << "\nNome: " << aluno1.nome;
    return 0;
}


void lerDados(Aluno *p){                                    (2)
    cout << "Informe a matricula: ";
    cin >> p->matricula;   //operador de seta               (3)
    cin.ignore(); //necessario antes do gets()
    cout << "Informe o nome: ";
    gets(p->nome);   //operador de seta                     (4)
    return;
}
```

No exemplo é importante notar:

0. Em (1), o endereço da variável local *aluno1* foi passado;
1. Em (2), um ponteiro é declarado para receber o endereço passado;
2. Em (3), foi utilizado o operador de seta (->), uma vez que *p* é um ponteiro;
3. Em (4), foi utilizada a função *gets()*, pois nome é um *array* de *bytes*; e
4. Em (5), será exibido a matrícula armazenada na função *lerDados()*.

8.5 Exemplo de *array* de *struct*

Arrays podem ser úteis quando é necessário manipular várias *structs* do mesmo tipo. Por exemplo, para manter um estoque de produtos, pode–se criar uma *struct Produto* e um *array* chamado *estoque* de tamanho 100, supondo que haja 100 produtos. Assim, pode–se armazenar um produto em cada posição do *array*. No fragmento de programa a seguir é mostrado como ler os 100 produtos. Como pode ser observado em (1), em se tratando de um *array* deve–se indicar a posição do elemento entre os colchetes e utilizar o operador *ponto* para acessar o campo do elemento contido na posição do *array*.

```
...
struct Produto{
```

```
    int codigo;
    float preco;
    char nome[50];
};

int main(){
    Produto estoque[100];
    //...
    for (int i=0; i < 100; i++){
        cin >> estoque[i].codigo;     (1)
        cin >> estoque[i].preco;
        cin.ignore();
        gets(estoque[i].nome);
    }
    ...
```

8.6 *Structs* como campos de *structs*

Assim como tipos primitivos (*int*, *float*, *char*) podem ser campos de *structs*, tipos compostos também podem. Desta forma, *arrays* e *structs* também podem ser campos de uma *struct*. No exemplo utilizado ao longo do capítulo, o campo *nome* da *struct Aluno* é um *array* de *char*. *Arrays* de outros tipos também são admitidos.

A seguir é mostrado um exemplo em que uma *struct* (*Pessoa*) possui um campo do tipo *struct* (*Nascimento*).

```
...
struct Nascimento{
    int dia;
    int mes;
    int ano;
};

struct Pessoa{
    int codigo;
    char nome[50];
    Nascimento nascimento;                  (1)
};

int main(){
    Pessoa pessoa;
    cout << "Dia nascimento: ";
    cin >> pessoa.nascimento.dia;           (2)
    cout << "Mes nascimento: ";
    cin >> pessoa.nascimento.mes;
    cout << "Ano nascimento: ";
    cin >> pessoa.nascimento.ano;
    cout << "\n\nData nascimento: "
        << pessoa.nascimento.dia << "/" << pessoa.nascimento.mes
        << "/" << pessoa.nascimento.ano;
    return 0;
}
...
```

A *struct Pessoa* possui um campo do tipo *Nascimento* (1), que também é uma *struct*. Logicamente a *struct Nascimento* deve ser definida antes da *struct Pessoa*. Para se "chegar" aos campos da *struct* interna, utiliza–se o operador *ponto*, como em (2), em que se está acessando o campo *dia* do campo *nascimento* da variável *pessoa*.

8.7 Funções em *structs*

Além de campos de dados, *structs* podem conter funções. Acompanhe o exemplo a seguir.

```
struct Nascimento{
    int dia;
    int mes;
    int ano;
};

struct Pessoa{
    int codigo;
    char nome[50];
    Nascimento nascimento;

    void mostraNascimento(){                    (1)
        cout << "\n\nData nascimento: "
        << nascimento.dia << "/" << nascimento.mes <<
            "/" << nascimento.ano;
    }
};

int main(){
    Pessoa pessoa;
    cout << "Dia nascimento: ";
    cin >> pessoa.nascimento.dia;
    cout << "Mes nascimento: ";
    cin >> pessoa.nascimento.mes;
    cout << "Ano nascimento: ";
    cin >> pessoa.nascimento.ano;
    pessoa.mostraNascimento();                  (2)
    return 0;
}
```

No fragmento de código anterior, uma função *mostraNascimento()* foi definida dentro da *struct Pessoa* (1). Em (2) a função é chamada, por meio do operador *ponto*, uma vez que ela é interna à *struct*. A função não pode ser usada fora de uma variável do tipo *Pessoa*.

Funções dentro de *structs* são especialmente indicadas quando estas funções manipulam dados da própria *struct*.

8.7.0 Exemplo 2

Analise o exemplo a seguir em que são criadas duas funções dentro da *struct*.

```
struct Aluno{
    int matricula;
    char nome[30];

    void inicializar(){        (1)
        matricula = 0;
        strcpy(nome, "");
    }

    void exibir(){             (2)
        cout << endl << "Matrícula: " << matricula; (3)
        cout << endl << "Nome: " << nome << endl;
    }
};

int main(){
    Aluno aluno;
    aluno.inicializar();  (4)
    aluno.exibir();       (5)
    return 0;
}
```

Em (1) é criada a função *inicializar()*, que inicializa os campos da struct com valores apropriados. Atente para o fato de não ser necessário receber os campos como parâmetros, pois uma vez que se está "dentro" da *struct*, ela tem acesso aos seus campos. Em (2), o mesmo acontece com a função *exibir()*, que mostra os campos da *struct* sem necessidade de recebê–los como parâmetros. É importante notar, também, que os campos são acessados sem o operador de ponto, como em (3).

Em (4) e (5) ocorrem as chamadas às funções. Note que a função é chamada no objeto *aluno*, por meio do operador de ponto.

✓Funções definidas dentro de uma *struct* só são podem ser chamadas por objetos desta *struct*.

8.8 Considerações finais

Registros (*structs*) são elementos que permitem ao programador criar seus próprios tipos. Além disso, são essenciais para se trabalhar com arquivos, que serão tratados no próximo capítulo.

8.9 Exercícios

0. Uma _____é um conjunto de variáveis reunidas sob um mesmo nome.

1. As variáveis declaradas na definição de um registro são chamadas _____ou _____.

2. Os membros de uma *struct* são acessados pelo operador _____ou _____.

3. Assinale V (verdadeiro) ou F (falso). Caso a afirmação seja falsa, justifique:

   (a) *Structs* podem conter variáveis de um único tipo.

   (b) Um membro de uma estrutura não pode ter o mesmo nome de um membro de outra estrutura.

(c) Estruturas são sempre passadas por referência.

(d) Estruturas não podem ser comparadas por meio dos operadores == ou !=.

(e) Membros de estruturas não podem ser comparadas por meio dos operadores == ou !=.

4. Crie uma *struct* chamada `Aluno` com os campos:
   código (int)
   nota1, nota2, nota3 e media (float)
   a *struct* deve ter uma função interna que calcula a média e armazenada no campo *media* da *struct*.

5. Crie uma *struct* chamada `Pessoa` com os campos:
   código (inteiro)
   nome (cadeia de 30 caracteres)
   O programa deve declarar uma variável do tipo da *struct* criada. A seguir, leia os campos do objeto criado. Depois disso, exiba os dados que foram lidos.

6. Reescreva o programa anterior, agora criando o objeto na função *main( )* e passando para a função *ler( )*, que faz a entrada de dados. A seguir (na *main()*), passe o objeto para função *exibir( )* que mostra os dados na tela. A passagem deve ser por valor.

7. Reescreva o programa anterior, agora fazendo passagem por referência.

8. Considere a seguinte *struct*:

```
struct Aluno{
    int matricula;
    char nome[30];
};
```

   (a) Crie uma *struct* `Turma` que tenha o código da turma e até 100 objetos do tipo `Aluno`.

   (b) Crie uma *struct* `Curso` que tenha o código do curso, o nome e até 100 objetos do tipo `Turma`.

9. Considere a seguinte *struct*:

```
struct Produto{
    int codigo;
    float preco;
    int quantidadeEstoque;
};
```

   (a) Crie uma função *baixaEstoque( )* que recebe um objeto `Produto` por referência e a quantidade vendida. Atualize o estoque, diminuindo a quantidade vendida da quantidade em estoque.

(b) Recrie a função do exercício anterior, agora recebendo um objeto `Produto` por valor e a quantidade vendida. Atualize o estoque, diminuindo a quantidade vendida da quantidade em estoque. Lembre-se de que a passagem do parâmetro por valor necessita retornar algo para atualizar o argumento de chamada.

(c) Modifique a *struct* anterior inserindo uma função *inicializa( )* dentro da *struct* . Essa função deve "zerar" todos os campos da *struct*.

(d) Crie um programa apenas com a função *main( )* que declare um objeto do tipo `Produto` e que chame a função *inicializa( )* do exercício anterior.

(e) Crie uma função que receba dois objetos do tipo `Produto` e retorne verdadeiro (*true*) e eles forem iguais ou falso (*false*) se forem diferentes. Considere que dois objetos são iguais se todos os seus campos forem iguais.

(f) Considere que um programa precise armazenar dados de até 2000 produtos. Crie uma programa apenas com a função *main( )* que: *a)*declare um *array* para armazenar os 2000 produtos e *b)*leia os dados de um produto e armazene na primeira posição do *array*.

10. Crie um programa que gerencie uma lista de clientes. Os dados de cliente são:
código (int)
nome (cadeia de caracteres)
endereço (cadeia de caracteres)
O programa deve ter um menu com as opções:

    (a) Cadastrar

    (b) Consultar por código

    (c) Consultar por nome

    (d) Exibir em ordem

    (e) Excluir cliente

    O programa deve gerenciar até 100 clientes. Não deve haver mais de um cliente com o mesmo código.

Capítulo 9

Arquivos

Os dados armazenados em variáveis e *arrays* são temporários, pois são perdidos quando o programa termina. Arquivos em memória secundária,tais como *hd's* e *pen drives*, são usados para retenção permanente de dados[1]. Arquivos são definidos pelo *iDicionário Aulete* como um conjunto de dados (textos, imagens, sons, animações, rotinas, programas etc.) gravados e armazenados como uma unidade independente e identificável.

Neste capítulo será visto o processo de criação de arquivos, bem como leitura e escrita neles.

9.0 Arquivos e *Streams*

Em C, um arquivo pode ser qualquer coisa, desde um arquivo em disco até um terminal de vídeo ou impressora. C trata cada arquivo simplesmente como um *stream* (fluxo) sequencial de *bytes*. Em outras palavras, um arquivo é um conjunto de *bytes* com uma marca de *fim de arquivo* no final. Quando um arquivo é aberto, um *stream* é associado com o arquivo. Qualquer programa automaticamente abre dois streams quando a execução começa: *standard input* e *standard output*. *Streams* fornecem um canal de comunicação entre arquivos e programas. Por exemplo, o *stream standard input* permite que o programa leia dados a partir do teclado, por meio do objeto *cin*. Já o *stream standard output* permite ao programa exibir dados na tela, por meio do objeto *cout*.

No restante deste texto, o termo arquivo será utilizado para fazer referência a arquivos em disco. As operações de ler e escrever em arquivos são semelhantes às operações de entradas e saídas padrão (*cin, cout*). A diferença é que os dados são lidos de arquivos em disco e escritos neles.

9.1 Abrindo e fechando arquivos

Arquivos são manipulados por meio de objetos *ofsteam* (arquivos de saída/escrita), *ifstream* (arquivo de entrada/leitura) ou *fstream* (que executam tanto entrada e saída). Quando o termo "saída" é usado com arquivos, estamos nos referindo à saída de *bytes* **da** memória principal **para** o arquivo. Em outras palavras, estamos escrevendo no arquivo. Da mesma forma, quando falamos em entrada (leitura), estamos falando de entrada de *bytes* **na** memória principal vindos **do** arquivo.

No restante do texto serão utilizados objetos *fstream*, que executam tanto entrada como saída. Para se declarar um objeto deste tipo o procedimento é o mesmo que se usa para declarar variáveis:

[1] Evidentemente, se a mídia se corromper, os dados poderão ser perdidos também.

```
fstream meuArquivo;
```

A primeira operação geralmente executada com um objeto *ofstream*, *ifstream* ou *fstream* é associá–lo a um arquivo físico. Este procedimento é conhecido como "abrir" o arquivo. Um arquivo aberto é representado no programa por um *stream* e qualquer operação de entrada ou saída executada neste objeto será aplicada ao arquivo físico associado a ele.

Para abrir um arquivo, a instrução é:

```
<objeto>.open(<nome do arquivo>, <modo de abertura>);
```

em que *nome do arquivo* é uma *string* representando o nome do arquivo a ser aberto e modo de abertura é um parâmetro opcional com a combinação de opções dentre as mostradas na Tabela 9.0.

Modo	**Descrição**
ios::in	Abre para operações de entrada (leitura).
ios::out	Abre para operações de saída (escrita).
ios::binary	Abre no modo binário.
ios::ate	Abre e posiciona o ponteiro de leitura/gravação no final. Se esta flag for omitida, o ponteiro se posiciona no início.
ios::app	Faz com que o que for escrito seja sempre gravado no final do arquivo. Esta opção cria o arquivo, se não existir, ou abre preservando as informações, se já existir.
ios::trunc	Apaga o arquivo anterior (se existir) e cria um novo.

Tabela 9.0: Modos de abertura de arquivo.

Exemplo:

```
#include <iostream>
#include <fstream>                                  (1)
using namespace std;

int main () {
    fstream meuArquivo;                             (2)
    meuArquivo.open ("exemplo.txt", ios::in | ios::out | ios::app); (3)
    //operações do programa
    meuArquivo.close();                             (5)
    return 0;
}
```

No exemplo, em (1) é inserida a diretiva de compilação *#include <fstream>*, necessária para se trabalhar com arquivos. Em (2) é declarado um objeto do tipo *fstream* chamado *meuArquivo*. O arquivo é aberto na linha (3), chamando–se a função *open()* do objeto. Passa–se como parâmetros para função, o nome do arquivo em disco e os modos de abertura, que no caso são *ios::in*, para que possamos ler do arquivo, *ios::out*, para que possamos escrever no arquivo e *ios::app*, para que todos os dados escritos sejam anexados ao final do arquivo. É importante fechar o arquivo quando terminarmos de trabalhar com ele. Isso é feito na linha (5), por meio da chamada à função *close()*.

9.2 Arquivos texto

O tipo mais simples de arquivo é o arquivo de texto. As informações são guardadas na forma de caracteres ASCII e podem ser lidas por qualquer editor de texto. Geralmente, são mantidos na forma de linhas de

texto, com um caractere de fim de arquivo (EOF - *end of file*) no final.

9.2.0 Escrevendo em um arquivo texto

Escrever em um arquivo texto é transferir dados da memória principal para o arquivo. A seguir é mostrado um exemplo em que se escreve uma frase em um arquivo.

```
int main(){
    fstream arq;
    arq.open("Teste.txt", ios::in | ios::out | ios::app); (1)
    arq << "Primeira linha\n";     (2)
    arq << "Segunda linha\n";      (3)
    arq.close();
    return 0;
}
```

Em (1) o objeto *fstream* é associado a um arquivo em disco chamado "exemplo.txt", que é aonde o texto será escrito. Se o arquivo não existir, ele será criado, pois foi incluída a *flag ios::app*. Arquivos, por padrão, são abertos no modo texto. Isso só não acontece se for incluída a *flag ios::binary*. Em (2), por meio do operador de deslocamento de *bits* («) um texto é enviado ao arquivo. Um segundo texto é escrito em (3). Uma vez que o arquivo foi aberto no modo *ios::app*, este segundo texto é adicionado após o que já estava escrito.

Em se tratando de arquivos, é conveniente testar se o arquivo foi aberto corretamente, como no trecho a seguir, em que se verifica se houve erro ao abrir (1). Caso *fail()* retorne verdadeiro, uma mensagem é exibida ao usuário (2) e o programa encerra retornando erro (3).

```
int main () {
    fstream arq;
    arq.open("Teste.txt", ios::in | ios::out | ios::app);
    if (arq.fail()){                    (1)
        cout << "Erro ao abrir";        (2)
        return 1;                       (3)
    }
    //demais instruções
    return 0;
}
```

9.2.1 Lendo de um arquivo texto

Ler de um arquivo texto é transferir os dados gravados para memória. No próximo exemplo, um arquivo é lido e o texto exibido na tela, linha a linha.

```
int main(){
    fstream arq;
    string texto;                       (1)
    arq.open ("Teste.txt", ios::in);    (2)
    while (arq.good()){                 (3)
        getline(arq, texto);            (4)
        cout << texto << endl;          (5)
    }
    arq.close();
    return 0;
}
```

No programa anterior foi declarada uma variável do tipo *string* (1). Esta variável armazena textos e não tem tamanho predeterminado. Em (2) o arquivo foi aberto no modo leitura (*ios::in*). Uma vez que pode haver mais de uma linha de texto, utilizou–se a estrutura de repetição *while* (3) para ler da primeira à última linha. A condição lógica utilizada foi uma chamada à função *good()* do objeto *arq*. Esta função retorna *true* caso não se tenha chegado ao fim do arquivo [2] e *false* quando o final do arquivo foi atingido. Em (4), a função *getline()* é utilizada para transferir o texto do arquivo para a variável *texto*. Esta função transfere os *bytes* até encontrar um caractere de fim de linha ('$\backslash n$'), em outras palavras, lê linha a linha. Em (5) o texto lido é exibido na tela.

9.3 Arquivos binários

Ao contrário dos arquivos de texto, os arquivos binários gravam *bytes* da mesma forma que estão representados na memória. Cabe ao programa tratar adequadamente os *bytes*. Pode–se imaginar um arquivo binário como uma sucessão de 0's e 1's com uma marca de fim de arquivo ao final. Arquivos binários são necessários para se gravar outros tipos de dados além de textos. Registros (*structs*) são gravadas na forma binária.

A primeira coisa a fazer para se trabalhar com o arquivo na forma binária, é abri–lo neste modo:

```
fstream arquivo;
arquivo.open("Teste.dat", ios::in| ios::out| ios::binary |ios::app);
```

No trecho de código acima tem–se a abertura de um arquivo na forma binária (*ios::binary*), para entrada e saída (*ios::in*| *ios::out*) sendo que novos dados serão acrescentados ao final (*ios::app*).

9.3.0 A função *sizeof()*

Quando vamos escrever em arquivo binário ou ler dados a partir dele precisamos especificar quantos *bytes* serão lidos ou escritos. Por exemplo, se vamos escrever um inteiro, devemos especificar que queremos transferir 4 *bytes* (geralmente os compiladores utilizam essa quantidade de *bytes* para armazenar inteiros). Para evitar que o programador tenha que calcular a quantidade de *bytes*, pode–se utilizar a função *sizeof()* para que o programa se encarregue desta tarefa.

A função *sizeof()* recebe como parâmetro uma *variável* ou um *tipo* e retorna a quantidade de *bytes* que ela ocupa. Exemplo:

[2] Na verdade esta função realiza também outros testes.

```
int i;
cout << sizeof(i);    (1)
cout << sizeof(int); (2)
```

As duas instruções produzem o mesmo efeito. Em (1) passou–se a variável, em (2) o tipo.

9.3.1 Escrevendo em arquivo binário

O trecho de programa a seguir escreve um inteiro no arquivo, utilizando a função *write()* do *stream*.

```
int main(){
fstream arq;
int i=0;
arq.open("MeuArquivo.dat", ios::out | ios::binary | ios::app); (1)
cin >> i;
arq.write((char *)&i, sizeof(int));  (2)
arq.close();
return 0;
}
```

No exemplo, o arquivo foi aberto como binário (1). O valor da variável *i* está sendo gravado no arquivo em (2), por meio da função *write()*. A função *write()* recebe dois parâmetros:

(char *)&<variável> : o endereço de memória onde estão os *bytes* a serem escritos. Este endereço é convertido para um ponteiro para *char*. Isto pode parecer um pouco confuso, mas basta saber que sempre será assim, o que muda é o nome da variável. Não importa que tipo de dado vai ser escrito, sempre haverá o " *(char *)&*".

sizeof(<tipo / variável>) : a quantidade de *bytes* a ser escrita. Isto poderia ser substituído por um número inteiro, mas, pelas razões já explicadas, é mais apropriado deixar que o programa calcule.

É necessário que os dados gravados em arquivos binários sejam sempre do mesmo tipo, mesmo que sejam tipos compostos. Isso garante que o programa consiga recuperar esses dados depois.

9.3.2 Lendo a partir de arquivos binários

Para se ler a partir de arquivos binários utiliza-se a função `read()`. Exemplo:

```
arquivo.read((char *)&j, sizeof(j));
```

Neste exemplo, um conteúdo é lido do arquivo e transferido para a variável *j*. A sintaxe é bem parecida com a função *write()*: indica–se para onde os *bytes* lidos vão (*&j*) e qual a quantidade de *bytes* serão transferidos (*sizeof(j)*).

No programa a seguir é realizada a leitura sequencial dos dados de uma arquivo cujo conteúdo é um conjunto de inteiros. Quando o arquivo é aberto, o "ponteiro" de leitura é posicionado no início do arquivo, o que significa que a leitura vai ser feita a partir do primeiro *byte*.

```
int main(){
    fstream arq;
    int j=0;
    while(arq.read((char *)&j, sizeof(int))){
        cout << j;
    }
    arq.close();
    return 0;
}
```

Neste exemplo, o comando de leitura foi colocado como condição do while. Isto porque, a cada operação de leitura, é retornado *true*(se houver conteúdo a ser lido) ou *false* (se a operação falhou, como no caso de se chegar ao fim do arquivo).

9.3.3 Movendo o "ponteiro" de leitura e gravação

Nem sempre queremos ler o arquivo de forma sequencial, do início ao fim ou então escrever um dado no final dele. Vamos supor que o programa precise alterar um dado no meio do arquivo. É necessário posicionar a escrita neste ponto e só então descarregar os *bytes*. O objetos *fstream* possuem duas funções para posicionar a leitura/gravação em um ponto específico do arquivo. São elas: seekp() e seekg(). A primeira se refere ao posicionamento da escrita (*put*) e a segunda, ao posicionamento da leitura (*get*). As funções recebem como parâmetros o *byte* correspondente a ser lido/escrito. Assim,

seekp(0);

posiciona a escrita no *byte* 0 (início do arquivo).

Para saber qual o posicionamento atual, existem as funções *tellp()* e *tellg()*.

9.4 Lendo e escrevendo registros

Registro é um conjunto estruturado de dados, em outras palavras, um conjunto de campos em um arquivo. Um arquivo é formado por um conjunto de registros. Para manipular os registros de um arquivo utilizam–se *structs* correspondentes.

Para ilustrar a leitura e gravação de registros analise o programa a seguir. O programa gerencia um cadastro de animais. Cada animal tem um código e um nome. É possível cadastrar um novo animal, alterar os dados de um animal cadastrado, consultar um animal por código e listar todos os animais cadastrados.

Inicialmente, define–se a *struct* que vai gerenciar os dados dos animais (1). Optou–se por uma struct de apenas dois campos para simplificação, mas as operações valem para *structs* com qualquer quantidade de campos. Na função *main()* é apresentado o menu de opções. Cada item do menu chama a função específica.

```
#include <iostream>
#include <fstream>

using namespace std;

struct Animal{      (1)
    int codigo;
    char nome[30];
};

//protótipos das funções
void cadastrar();
void consultar();
void alterar();
void listar();

int main(){
    int op=0;
    do{
        cout << "\n\n1 - Cadastrar";
        cout << "\n2 - Consultar";
        cout << "\n3 - Alterar";
        cout << "\n4 - Listar";
        cout << "\n5 - Sair";
        cout << "\nOpcao: ";
        cin >> op;
        switch(op){
            case 1 : cadastrar(); break;
            case 2 : consultar(); break;
            case 3 : alterar(); break;
            case 4 : listar(); break;
        }
    }while (op != 5);
    return 0;
}


void cadastrar(){
    Animal animal;
    fstream arq;
    arq.open("Animal.dat", ios::out | ios::binary | ios::app);  (2)
    cout << "\nCodigo: ";
    cin >> animal.codigo;
    cin.ignore();
    cout << "Nome: ";
    gets(animal.nome);
    arq.write((char *)&animal, sizeof(Animal));                 (3)
    arq.close();
}
```

```
void consultar(){
    bool achou = false;
    int codigo=0;
    Animal animal;
    fstream arq;
    arq.open("Animal.dat", ios::in  | ios::binary | ios::app);
    cout << "\nCodigo do animal a ser consultado: ";
    cin >> codigo;                                                    (4)
    while (arq.read((char *)&animal, sizeof(Animal))){                (5)
        if (codigo == animal.codigo){                                 (6)
            achou = true;
            cout << "\nCodigo: " << animal.codigo;
            cout << "\nNome..: " << animal.nome;
        }
    }
    if (! achou){                                                     (7)
        cout << "\n\nAnimal nao encontrado\n\n";
    }
    arq.close();
}


void listar(){
    int cont=0;
    Animal animal;
    fstream arq;
    arq.open("Animal.dat", ios::in | ios::binary );
    cout << "\n\n\nAnimais cadastrados:\n\n";
    while(arq.read((char *)&animal, sizeof(Animal))){             (8)
        cout << "\nCodigo: " << animal.codigo;
        cout << "\nNome..: " << animal.nome;
        cont++;
    }
    if (cont == 0){
        cout << "\n\nNenhum animal no arquivo\n\n";               (9)
    }
    arq.close();
}
```

```
void alterar(){
    fstream arq;
    Animal animal;
    int codigo=0, cont=0, pos=0;
    arq.open("Animal.dat", ios::in | ios::out | ios::binary);            (10)
    //lendo o codigo do animal a ser alterado
    cout << "\n\nCodigo: ";
    cin >> codigo;                                                       (11)
    //buscando o animal no arquivo
    while(arq.read((char *)&animal, sizeof(Animal))){                    (12)
        if (codigo == animal.codigo){    //achou
            pos = cont * sizeof(Animal); //salva a posicao em que esta   (13)
        }
        cont++;
    }
    if (cont == 0){                                                      (14)
        cout << "\n\nAnimal nao localizado\n\n";
    }
    else{                                                                (15)
        //le novos dados
        cin.ignore();
        cout << "Nome: ";
        gets(animal.nome);
        arq.clear();                                                     (16)
        arq.seekp(pos);                                                  (17)
        arq.write((char *)&animal, sizeof(Animal));                      (18)
    }
    arq.close();
}
```

Na função *cadastrar()*, os dados de um novo animal são lidos e escritos no arquivo (3). Como o arquivo foi aberto no modo *ios::app* (2), o novo registro vai ser adicionado no final.

Na função *consultar()*, o usuário informa o código do animal que quer consultar (4); o arquivo é lido sequencialmente e o código de cada registro lido é comparado com o código procurado (6). Caso o animal procurado seja encontrado, os dados são mostrados; caso ele não exista, uma mensagem é apresentada ao usuário (7).

A função *listar()* é simples: o arquivo é aberto e lido sequencialmente (8). Cada registro recuperado é exibido. Um contador foi utilizado para tratar a situação em que não haja nenhum registro (9).

A função *alterar()* é um pouco mais complexa. Note que o arquivo não foi aberto como *ios::app* (10), pois neste modo não é possível alterar registros, apenas adicionar. O código do animal que se quer alterar é lido em (11). Como não se sabe em que posição o registro está, é necessário ler sequencialmente (12) o arquivo até encontrar o registro procurado ou chegar ao fim do arquivo. Caso o registro seja encontrado, a variável *pos* salva a posição (*byte* inicial) deste registro (13). Se ao chegar no final do arquivo o registro não foi localizado, uma mensagem é exibida (14). Caso contrário (15), são lidos os novos dados e escritos "por cima" dos antigos (18). Antes disso, o arquivo foi colocado em estado consistente (16) e o "ponteiro de escrita" foi movido para posição correta(17).

Embora este seja um exemplo simples e os algoritmos possam ser melhorados, ele pode ser usado para exemplificar as operações com arquivos, que são, basicamente: incluir, alterar e consultar.

9.5 Exercícios

0. **Gravatas**. Falafel trabalha em uma empresa em que a aparência é muito importante. Por exemplo, não é elegante usar uma mesma cor de gravata duas vezes na semana. Como Falafel ocupa um cargo importante, ele quer evitar gafes, mas ele é muito esquecido. Ao escolher uma gravata, ele não lembra se já usou aquela cor na semana. Por isso, ele quer um programa de computador que armazene as cores das gravatas já usadas. Cada vez que ele vai escolher uma gravata, ele informa ao programa a cor escolhida e o programa deve dizer "Pode usar" caso a cor não tenha sido usada na semana. Neste caso, o programa salva em arquivo a cor selecionada. Caso ele tente repetir a cor, o programa deve informar "Não pode usar" e solicitar uma nova escolha.
   Faça o programa com um menu:
   1 - Escolher gravata
   2 - Mostrar as usadas na semana
   3 - Sair
   Exemplos:

Entrada	Saída
branca	"pode usar"
florida	"pode usar"
vermelha	"pode usar"
branca	"não pode usar"
amarela	"pode usar"
florida	"não pode usar"

   Utilize funções separadas para cada tarefa necessária, por exemplo, buscar se já foi usada, salvar, etc.

1. **Alunos**.Um professor de estatística costuma sortear alunos para que resolvam exercícios. Para isso, ele diz o número do aluno e este deve ir até o quadro e resolver um exercício proposto. No entanto, como são muitos exercícios, o professor nem sempre se lembra se um aluno já foi sorteado ou não. Assim, ele limitou a quantidade de vezes que um mesmo aluno pode ser premiado na mesma semana, para que os outros também possam ter oportunidade. O problema é que é difícil controlar quem já foi e quantas vezes já foi.

   Faça um programa para auxiliar os envolvidos. O programa deve:

   (a) Para cada aluno sorteado, verificar se ele já foi 3 vezes ou não

   (b) Se ele já foi 3 vezes, exibir uma mensagem dizendo "O aluno Fulano de Tal já foi 3 vezes" e solicitar novamente o número do aluno

   (c) Se ele ainda não foi 3 vezes:

      i. Caso seja a primeira vez, criar um registro no arquivo com o número, nome e a quantidade 1

      ii. Caso ele já tenha ido alguma vez, alterar o registro desse aluno incrementando a quantidade de vezes que ele foi

   (d) Sempre que solicitado, exibir uma lista completa dos alunos que já foram, com número, nome e a quantidade de vezes que foram.

Faça o programa com um menu:
1 - Informar número do aluno
2 - Mostrar lista completa
3 - Limpar arquivo
4 - Sair

Utilize *struct* para manter os dados dos alunos. Utilize funções específicas para cada tarefa do programa.

2. **Criptografia**. Huguinho e Luizinho tiveram uma ideia genial para abrir um novo negócio e ficarem milionários. Eles agora estão finalizando o projeto, mas existe um problema. Como eles residem em cidades diferentes, precisam conversar pela internet para acertar os detalhes. O medo deles é que os arquivos trocados sejam interceptados por alguém e a ideia seja roubada. Por isso, inventaram um algoritmo para codificar os arquivos de forma que ninguém possa entendê-las. A codificação do arquivo funciona da seguinte maneira:
- o texto é "quebrado" ao meio
- um novo texto é produzido combinando uma letra da primeira parte e uma letra da segunda parte, alternadamente
Exemplo:
"armazenar o time de futebol do usuário"
dividindo-se ao meio:
armazenar o time de
futebol do usuario

produziria o seguinte texto criptografado:

afrumtaezbeonla rd oo tuismuea rdieo

Faça o programa com um menu:
1 - Gravar texto
2 - Codificar texto
3 - Decodificar texto
4 - Exibir texto
5 - Sair

A opção 1 lê um texto a partir do teclado e salva em arquivo. A opção 2 lê o texto do arquivo, codifica conforme descrito e salva novamente no mesmo arquivo, codificado. A opção 3 lê o arquivo codificado, decodifica e salva novamente o texto. A opção 4 lê do arquivo e exibe na tela.

Utilize funções específicas para cada tarefa do programa.