Bacharelado em Sistemas de Informação

Algoritmos

Prof. Ms. Antonio Marcos SELMINI

Apostila de
Algoritmos / Linguagem C
2008

Prefácio

Este material é o resultado das notas de aulas das disciplinas Algoritmos e Estruturas de Dados.

O principal objetivo deste material é auxiliar os alunos na disciplina de Algoritmos ministrada na Faculdade de Informática e Administração Paulista (FIAP). Vale destacar também que este material não substitui em hipótese alguma os livros indicados como bibliografia básica, pois deverão ser consultados para uma melhor formação profissional.

Prof. Ms. Antonio Marcos Selmini

Índice

Capítulo 1 - Introdução à Microinformática 6

1.1 Introdução 6
1.2 Precursores do Computador 6
1.3 Gerações dos Computadores 7

Capítulo 2 - Programação de Computadores 9

2.1 Introdução 9
2.2 Lógica de Programação 9
2.3 Algoritmos 10
2.4 Representação de Algoritmos 11
2.5 Linguagens de Programação 14
2.6 Variáveis 15
2.7 Tipos de Dados 16
2.8 Declaração de Variáveis 17
2.9 Operadores 18
2.10 Comandos de Entrada e Saída de Dados 21
2.11 Exemplos de Algoritmos 21
2.12 Exercícios em Classe 22
2.13 Exercícios Complementares 25

Capítulo 3 - Estruturas de Seleção 27

3.1 Introdução 27
3.2 Estrutura de Seleção Simples 27
3.3 Estrutura de Seleção Composta 28
3.4 Estrutura de Seleção Agrupada 30
3.5 Exercícios em Classe 31
3.6 Exercícios Complementares 32

Capítulo 4 - Estruturas de Repetição 35

4.1 Introdução 35
4.2 Estrutura de Repetição Enquanto 35
4.3 Estrutura de Repetição Faça-Enquanto 37
4.4 Estrutura de Repetição Para 38
4.5 Exercícios em Classe 40

4.6 Exercícios Complementares 42

Capítulo 5 - Introdução a Linguagem C 44

5.1 Introdução 44
5.2 Mapa de Memória em C 45
5.3 Tipos de Dados em C 45
5.4 Inicialização de Variáveis 46
5.5 Operadores em C 46
5.6 Comandos de entrada e saída de dados 46
5.7 Estrutura de um programa em C 47
5.8 Estrutura de Seleção 47
5.8.1 Comando de Seleção if 47
5.8.2 Comando de Seleção switch 51
5.9 Estruturas de repetição 51
5.9.1 Comando while 51
5.9.2 Comando do - while 53
5.9.3 Comando for 54

Capítulo 6 - Arrays Unidimensionais (Vetores) 56

6.1 Introdução 56
6.2 Arrays unidimensionais 56
6.3 Acessando elementos de um array unidimensional 57
6.4 Inicialização de arrays 58
6.5 Exemplos de manipulação de arrays 58
6.6 Exercícios em Classe 58
6.7 Exercícios Complementares 60

Capítulo 7 - Manipulação de Strings e Caracteres 61

7.1 Introdução 61
7.2 Funções para manipulação de strings 62
7.3 Exercícios em Classe 64
7.4 Exercícios Complementares 66

Capítulo 8 - Arrays Bidimensionais (Matrizes) 67

8.1 Introdução 67
8.2 Declarando um array bidimensional 67
8.3 Exercícios em Classe 69
8.4 Exercícios Complementares 71

Capítulo 9 - Introdução aos Ponteiros 73

9.1 Definição 73
9.2 Declaração de ponteiros 73
9.3 Operadores de ponteiros (& e *) 74
9.4 Cuidados com ponteiros 75
9.5 Exercícios em Classe 76

Capítulo 10 - Funções 77

10.1 Introdução 77
10.2 Protótipos de funções 78
10.3 Passagem de Parâmetros 80
10.4 Passagem de arrays para funções 81
10.5 Escopo de variáveis 82
10.6 Exercícios em Classe 82

Capítulo 11 - Estruturas 87

11.1 Definição e declaração de estruturas em C 87
11.2 Matrizes de estruturas 89
11.3 Passando elementos de estruturas como parâmetros para funções 89
11.4 Passando estruturas inteiras para funções 90
11.5 Ponteiros para estruturas 91
11.6 Comando typedef 91
11.7 Exercícios em Classe 92
11.8 Exercícios Complementares 93

Capítulo 12 - Manipulação de arquivos em C 95

12.1 Introdução 95
12.2 Streams 95
12.3 Abrindo um arquivo 96
12.4 Fechando um arquivo 97
12.5 Escrevendo e lendo caracteres em um arquivo 97
12.6 Escrevendo e lendo strings em um arquivo 98
12.7 Funções fread() e fwrite() 99
12.8 Exercícios 100

Bibliografia 101

Capítulo 1

Introdução à Microinformática

1.1 Introdução

A cada dia presenciamos em jornais, revistas, na televisão o surgimento de novas invenções. Mas de todos esses inventos, talvez o maior e mais sensacional tenha sido o computador. O ano inicial da construção do primeiro computador é 1946, mas sabe-se que suas bases foram estabelecidas ao longo de dois séculos. O computador que conhecemos hoje, com suas funcionalidades é muito recente tendo quase 60 anos. Desses quase 60 anos sua divulgação e aquisição por parte da população tem pouco mais de 20 anos.

O ENIAC (primeiro computador) foi construído nos Estados Unidos e tinha o tamanho aproximado de um caminhão. Funcionava por poucas horas. Era construído com muitas válvulas e consumia energia suficiente para abastecer centenas de casas. Era um computador muito complicado de operá-lo e só os seus projetistas o conseguiam fazer.

Os principais fatores responsáveis pela divulgação e aquisição dos computadores são:
- Redução da dimensão física;
- Aumento da confiabilidade;
- Aumento da capacidade de processamento;
- Redução de custo para aquisição e manutenção;
- Ampliação da gama de utilidades e aplicações;

Uma pergunta que surge com freqüência em programas de televisão (programas educativos de perguntas e respostas) ou até mesmo entre os alunos é quem inventou o computador. Essa é uma pergunta difícil de responder porque não existe apenas um criador. O computador foi resultado de uma série de equipamentos e ou máquinas que foram sendo desenvolvidos por diversos personagens marcantes da história.

1.2 Precursores do Computador

Como mencionado na seção anterior dizer o inventor do computador é algo praticamente impossível. Diferente de outras invenções, o computador não "nasceu da noite para o dia", mas foi o resultado do aperfeiçoamento de várias máquinas que foram surgindo em diferentes épocas.

O surgimento dessa poderosa máquina é apenas um reflexo da evolução do homem e também uma conquista pela busca por processos de automatização e substituição do trabalho humano.

Um dos primeiros dispositivos de calcular que se tem conhecimento é o ábaco. O ábaco do grego significa "tábua de calcular". É considerado um dos primeiros precursores das máquinas de somar sendo um dispositivo auxiliar para o usuário que faz cálculos mentais.

Um personagem muito importante para o desenvolvimento dos computadores atuais é Blaise Pascal. Esse matemático francês foi responsável pela criação de uma máquina de somar, a partir da qual foram criados diversos modelos. A máquina criada por Pascal foi chamada de Pascalina e na época foi considerada a perfeição em máquinas de calcular. Vale destacar que antes da Pascalina outros cientistas criaram outras máquinas mas que não tiveram desempenho tão bom quanto a Pascalina.

Outro matemático de grande importância foi Charles Babbage. Dentre as várias invenções de Babbage destaca-se a Máquina Analítica. A Máquina Analítica tinha alguns princípios básicos muito interessantes tais como:

- Dispositivos de entrada;
- Facilidade de armazenar números para processamento;
- Processador, que na época era chamado de calculador numérico;
- Unidade organizacional que tinha como função organizar as tarefas a serem executadas;
- Dispositivo de saída com cartões perfurados;

É uma pena que a tecnologia disponível no tempo de Babbage não permitiu, pois teria sido um computador programável como conhecemos nos dias atuais.

Dentre esses dois grandes personagens citados há também uma figura feminina de destaque. Seu nome era Ada Augusta King, filha do poeta inglês Lord Byron. Ada foi amiga de Babbage. Sugeriu que a Máquina Analítica utilizasse o sistema binário e também escreveu pequenos programas para ela. Ada é considerada a primeira programadora de computadores do mundo.

Os desenvolvimentos não paravam e em toda parte do mundo cientistas desenvolviam suas engenhosas criações. Em 1930, a IBM lança uma máquina eletromecânica batizada de Harvard Mark I. O Mark I processava número com precisão de 23 dígitos e executava todas as quatro operações aritméticas, além de cálculos de logaritmos e funções trigonométricas. O Mark I também tinha programas embutidos. Era uma máquina lenta, mas era completamente automático.

Após o desenvolvimento do Mark I, o outro grande desenvolvimento americano foi o ENIAC (Electrical Numerical Integrator and Calculator). Era construído a base de válvulas e ocupava um espaço de 167 metros quadrados. Era um computador de alta velocidade e programável que usava a base de numeração decimal (base 10). Foi utilizado de 1946 a 1955.

1.3 Gerações dos Computadores

A evolução do computador segundo as tecnologias empregadas são divididas em gerações. Cada geração é caracterizada por um marco tecnológico. Basicamente os marcos tecnológicos utilizados para classificar a evolução dos computadores são três:

- Válvulas eletrônicas;
- Transistores;
- Circuitos Integrados;

1ª Geração (1946 – 1958)

- Caracterizada pelo uso de válvulas na composição dos computadores;
- Entradas e saídas baseadas em fitas perfuradas;

- Computadores aplicados a resolução de problemas técnicos, científicos e militares;
- Programação baseada em linguagem de máquina;

2ª Geração (1958 – 1964)
- Caracterizada pelo uso de transistores (talvez seja a invenção mais importante do século XX);
- Entrada de dados através de cartão perfurado;
- Surgimento dos primeiros Sistemas Operacionais;
- Utilização de discos magnéticos removíveis;
- Desenvolvimento e aplicação das primeiras linguagens de programação em substituição a linguagem de máquina;

3ª Geração (1965 –)
- Utilização dos circuitos integrados em substituição aos transistores;
- Surgimento do Sistema Operacional DOS (Disk Operating System);
- Aumento significativo das capacidades de armazenamento;

4ª Geração (...)
Para alguns historiadores a 3ª geração vai até os dias atuais. Outros afirmam que a 3ª geração termina em 1970, dando inicio a 4ª geração, caracterizada pela utilização de circuitos integrados de larga escala e utilização de técnicas de Inteligência Artificial.

Capítulo 2

Programação de Computadores

2.1 Introdução

No capítulo anterior foram apresentadas as principais características históricas e personagens que contribuíram para o surgimento dos computadores que conhecemos hoje. Como o objetivo principal da disciplina é na programação de computadores, a partir deste capítulo serão abordados os principais conceitos e técnicas envolvidas na construção de programas. O funcionamento do computador será abordado em outras disciplinas.

Uma pergunta que ainda não foi colocada em questão é: o que é um computador? Por ser um equipamento que está presente no nosso dia-a-dia e que muitas vezes nos torna totalmente dependente, muitos poderiam ficar sem apresentar uma resposta. A resposta é muito simples: Um computador é uma máquina que processa dados de entrada, gerando dados de saída. Entenda-se por processamento qualquer manipulação nos dados de entrada para gerar os dados de saída. Os dados de entrada representam as informações que são fornecidas para o computador para que ele possa manipular os dados de saída representam os resultados do processamento sobre os dados de entrada. A figura descreve a situação exposta.

Figura 2.1 – Esquema de processamento de dados.

Como o objetivo da disciplina é nas técnicas básicas de programação vamos estudar como escrever programas para a partir de uma entrada de dados gerar a saída.

2.2 Lógica de Programação

Na área de desenvolvimento de sistemas muito se fala sobre lógica de programação. Mas o que é lógica de programação e qual a sua importância para a programação de computadores?

Um programa de computador representa os passos necessários que o computador deverá seguir para resolver um problema específico. Quando escrevemos um programa de computador devemos tomar o maior cuidado possível para que os passos necessários tenham uma certa ordem de execução e também uma certa coerência. A ordem de execução e a coerência dependem do problema que o programa irá resolver. A grosso modo, escrever um programa para resolver um problema é como ensinar um leigo em um assunto como ele deverá resolver um problema.

Dessa forma, a lógica de programação é a técnica de encadear pensamentos para atingir um determinado objetivo. Esse encadeamento representa uma seqüência lógica de passos a serem "seguidos" pelo computador. É através da lógica de programação que definimos a seqüência lógica de passos.

Uma seqüência lógica são passos executados até atingir um objetivo ou solução de um problema. Em programação de computadores, o termo "instruções" normalmente é utilizado no lugar de "passos". Em ciência da computação, a instrução representa a "ordem" para que o computador execute uma determinada ação. A solução de um problema é obtida a partir de um conjunto de instruções e não apenas de uma instrução isolada.

2.3 Algoritmos

O termo algoritmos é o termo central na ciência da computação e em programação de computadores. Formalmente um algoritmo é uma seqüência lógica e finita de instruções escritas em uma linguagem de programação para se atingir a solução de um problema.

No dia-a-dia fazemos centenas de algoritmos. Por exemplo: tomar banho, tomar o café da manhã, trocar uma lâmpada, fazer uma ligação telefônica, uma receita de bolo, etc.

Um algoritmo apenas especifica os passos lógicos que devem ser seguidos pelo computador para se atingir a solução de um problema. O algoritmo não é a solução do problema porque se assim fosse, para cada problema existiria apenas um algoritmo.

Exemplo 1: Algoritmo para calcular a média de três valores inteiros:

```
Inicio
        Fornecer o primeiro valor inteiro;
        Fornecer o segundo valor inteiro;
        Fornecer o terceiro valor inteiro;
        Somar os três valores;
        Pegar o resultado da soma e dividir por 3;
        Mostrar o resultado obtido;
Fim
```

Como pode ser observado no exemplo acima um algoritmo deve ser simples e mostrar de forma coerente como deve ser obtida a solução do problema. Algumas instruções não pode ser alteradas como, por exemplo, não podemos dividir o valor antes de somar e também não podemos somar os valores se os mesmos não forem fornecidos para o programa. Caso essa inversão ocorra é o que chamamos de **erro de lógica**. É como se fossemos beber um refrigerante em lata sem abrir a lata!

Como mostrado na figura 2.1, um programa é dividido em três partes básicas: entrada, processamento e saída. Se tivermos essa idéia em mente ao construir um algoritmo, a sua construção poderá ficar mais simples.

Para o exemplo 1 temos:

- Quais são os dados de entrada?
    - Três números inteiros.
- Quais são os dados de saída?

- A média dos três valores inteiros.

- Qual é o processamento a ser realizado nos dados de entrada?
  - Somar os três valores inteiros e dividir o resultado por três.

Ao analisarmos o exemplo 1 novamente podemos identificar as três partes:

```
Inicio
        Fornecer o primeiro valor inteiro;   }
        Fornecer o segundo valor inteiro;    }  entrada
        Fornecer o terceiro valor inteiro;   }
        Somar os três valores;                       }  processamento
        Pegar o resultado da soma e dividir por 3;   }
        Mostrar o resultado obtido;  }  saída
Fim
```

2.4 Representação de Algoritmos

Como vimos na seção anterior, um algoritmo nada mais é que um conjunto de instruções escritos em uma determinada linguagem. Essa linguagem é uma forma de representação do algoritmo. A representação pode ser através de:

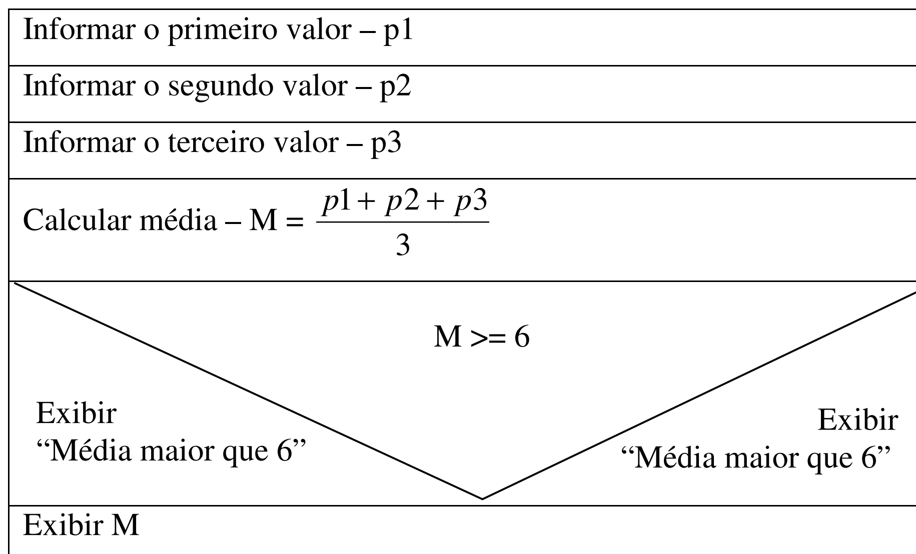
- **Linguagem de programação:** representa uma forma de escrever os algoritmos para que um computador possa entender e executar as instruções. Posteriormente estudaremos com mais detalhe os conceitos de linguagem de programação. Exemplo:

```
#include <stdio.h>
void main()
{
        int p1, p2, p3;
        float m;
        clrscr();
        printf("Informe o primeiro valor: ");
        scanf("%d", &p1);
        printf("Informe o segundo valor: ");
        scanf("%d", &p2);
        printf("Informe o terceiro valor: ");
        scanf("%d", &p3);
        m = (p1+p2+p3)/3;
        printf("O resultado é %f", m);
        getch();
}
```

O programa acima foi codificado em linguagem C e representa o código para o algoritmo do exemplo 1. Usando uma linguagem de programação, devemos estar atentos as suas regras, limitações e convenções utilizadas, o que para um iniciante pode ser algo extremamente difícil e trabalhoso.

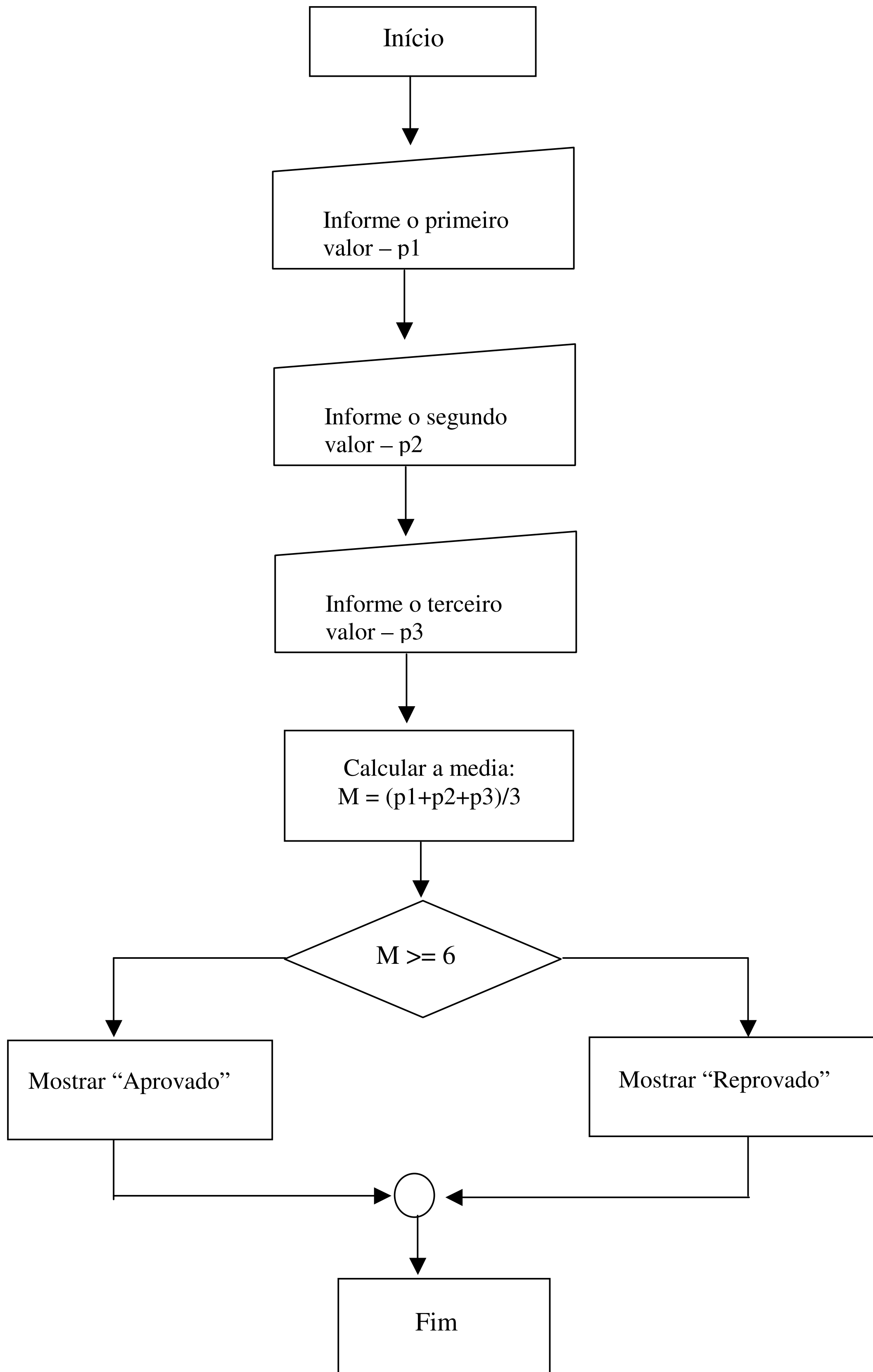
- **Graficamente:** A representação de algoritmos através de gráficos é extremamente simples e fácil de compreender, mas existe um inconveniente. A medida que os problemas a serem resolvidos tornam-se grandes, os gráficos também tornam-se grandes e trabalhosos de manipular. Outro inconveniente são as representações que devem ser memorizadas. Basicamente existem duas representações gráficas: **Diagramas de Nassin-Shneiderman-Chapin** e **Fluxogramas**.

    - **Diagramas de Nassin-Shneiderman-Chapin:**



A representação através dos Diagramas de Nassin-Shneiderman-Chapin foi muito utilizada por vários anos principalmente no inicio da programação de computadores. Como pode-se analisar na figura acima, é fácil entender a idéia do algoritmo e principalmente acompanhar as instruções de sua execução. Esse diagrama não utiliza grande variedade de símbolos, portanto a sua memorização é mais fácil.

- **Fluxograma:** O fluxograma utiliza uma variedade de símbolos diferentes. Sua compreensão também é fácil, mas como já destacado, com o aumento do problema a ser resolvido, o diagrama também fica extenso.


Início
Informe o primeiro valor – p1
Informe o segundo valor – p2
Informe o terceiro valor – p3
Calcular a media:
M = (p1+p2+p3)/3
M >= 6
Mostrar “Aprovado”
Mostrar “Reprovado”
Fim

- **Pseudo-linguagem:** é uma forma simples de representar algoritmos. A pseudo-linguagem usa a idéia de uma linguagem de programação, mas as instruções são escritas em linguagem natural (português, inglês, espanhol, etc). Outra vantagem é que na pseudo-linguagem não temos regras drásticas a serem memorizadas. Não existem convenções e a preocupação principal é apenas com lógica do programa.

```
Inicio
    inteiro p1, p2, p3;
    real m;
    imprima ("Informe o primeiro valor")
    leia(p1)
    imprima("Informe o segundo valor")
    leia(p2)
    imprima("Informe o terceiro valor")
    leia(p3)
    m = (p1+p2+p3)/3
    imprima("a média é ", m)
Fim
```

Inicialmente adotaremos a pseudo-linguagem como forma de escrever os nossos algoritmos.

2.5 Linguagens de Programação

Com o surgimento dos computadores, os primeiros programas eram escritos em linguagem de máquina. A linguagem de máquina é uma linguagem complicada de se entender e de manipular, exigindo um grande conhecimento do funcionamento da máquina.

Inicialmente poucas pessoas conseguiam escrever um programa por causa dos motivos citados acima. Os programas eram extensos, de difícil manutenção e baixa produtividade.

Para facilitar a programação de computadores, surgiram as **linguagens de programação**. Uma linguagem de programação nada mais é do que um conjunto de regras e convenções que são utilizadas para escrever instruções em uma linguagem próxima da linguagem natural (normalmente o inglês) para que o computador possa executar. Para que a máquina consiga "entender" as instruções escritas em uma linguagem de programação, as mesmas devem ser "traduzidas" para a linguagem de máquina. Dessa forma, uma linguagem de programação pode ser vista como uma interface com a linguagem de máquina.

Desde o surgimento do computador diversas linguagens de programação foram desenvolvidas com vários propósitos: aplicações comerciais, científicas, etc. Exemplos de linguagens de programação: Fortan, Cobol, Pascal, Basic, Visual Basic, Delphi, Java, entre outras.

As linguagens de programação podem ser classificadas em dois grupos principais: **linguagens de alto nível** e **linguagens de baixo nível**. As linguagens de alto nível são linguagens desenvolvidas com o propósito comercial que não manipulam ou dificilmente manipulam a linguagem de máquina. Exemplos de linguagens de alto nível: Pascal, Fortran, Delphi, Java, etc.

Por outro lado, as linguagens de baixo nível são linguagens onde as instruções são baseadas em linguagem de máquina. Um exemplo de linguagem de baixo nível e a linguagem Assembler. A linguagem de programação C é classificada por alguns pesquisadores como linguagem de nível médio por apresentar instruções em alto nível e também em baixo nível.

2.6 Variáveis

Na introdução desse capítulo foi mencionado que um algoritmo manipula dados de entrada para gerar a solução de um programa. A pergunta que será colocada agora é: como esses dados ficam armazenados para serem processados?

Existe no computador uma área de armazenamento chamada de **memória**. A memória é um dispositivo físico onde todos os dados a serem processados ou já processados são armazenados. Basicamente existem dois tipos de memória: **memória primária** e **memória secundária**. A memória primária também chama de RAM (*Random Access Memory*) é uma memória de acesso aleatório e volátil. As memórias secundárias podem ser dispositivos externos tais como: discos magnéticos, CD-ROM, dentre outros.

A grosso modo, podemos tentar visualizar a memória primária como uma caixa. Essas caixas são subdividas em caixas menores e dispostas de forma seqüencial. É nas "caixas" que as informações ficam armazenadas, ou seja, podemos ter um número, um caractere, uma seqüência de caracteres, etc. Essas caixas são chamadas de posições de memória.

O número de subdivisões é finito. É por esse motivo que algumas vezes o Sistema Operacional envia mensagens informando que há pouca memória disponível, ou seja, praticamente o número total de posições de memória estão quase todos preenchidos.

Para que o computador faça a manipulação das informações armazenadas é necessário conhecer quais são as posições de memórias que estão preenchidas. A cada posição de memória é associado um endereço físico. Esse endereço é representado por valor hexadecimal (base 16). Veja um exemplo na tabela abaixo:

Endereço Físico	Conteúdo Armazenado
1000:3AC89	"Hoje é feriado"
2000:AAFF3	25

Imagina se ao escrevermos nossos programas tivéssemos que conhecer quais as posições de memória que o nosso programa estivesse utilizando. Certamente a dificuldade seria bem maior e também o número de erros aumentaria em virtude da manipulação de erros na manipulação desses valores.

As linguagens de programação permitem que os programadores forneçam nome às posições de memória. Cada posição de memória deve ter um nome diferente de outra posição. As regras para formação dos nomes variam de uma linguagem de programação para outra. Os nomes fornecidos as posições de memória são chamados de **endereços lógicos**.

Endereço Lógico	Conteúdo Armazenado
Frase	"Hoje é feriado"
Idade	25

Quando fornecemos um nome para uma posição de memória, o computador automaticamente associa o nome a um endereço físico de memória. Dessa forma, ao invés de manipularmos os endereços físicos trabalhamos apenas com os nomes das posições. A escolha de uma posição de memória é determinada automaticamente pelo computador quando o mesmo irá executar um programa.

O conteúdo de uma posição de memória não é um valor fixo, e durante a execução de um programa esse conteúdo pode variar. Portanto, o conteúdo de uma posição de memória é variável. As posições de memória que o computador manipula durante a execução de um programa damos o nome de **variável**.

Como já discutido, toda variável em um programa deve ter um nome e as regras para a formação do nome das variáveis mudam de uma linguagem de programação para outra. Antigamente, os nomes de variáveis eram restritos a apenas oito caracteres, mas com a evolução dos sistemas operacionais essa limitação já foi superada. Como o nosso curso será baseado em linguagem C, inicialmente adotaremos algumas regras próximas dessa linguagem. Eis as regras:

- Utilize nomes sugestivos, relacionados com o conteúdo que a variável irá armazenar;
- Nomes de variáveis deverão ser iniciados com caracteres e não com números;
- Alguns símbolos especiais não são aceitos, tais como: *, !, ?, etc;
- O underline ("_") pode ser utilizado;

2.7 Tipos de Dados

Toda informação manipulada por um programa tem um tipo. O tipo é importante para que o computador possa armazenar de forma eficiente os dados a serem manipulados e também para saber quais são as operações básicas que podem ser realizadas com a variável.

A quantidade de tipos e os nomes podem variar de uma linguagem de programação para outra, mas os tipos básicos, chamados de tipos primitivos são:

- Inteiro;
- Real;
- Caractere;
- Cadeia de caracteres ou string;
- Booleano ou lógico;

Tipo Inteiro:

- Utilizado para armazenar valores numéricos sem casa decimal. Exemplo: 12, -12, 100, 0, -56, etc;
- Operações que podem ser realizadas: adição, subtração, multiplicação, divisão, potenciação e resto de divisão (esta operação só é definida para valores inteiros);

Tipo Real:
- Armazena valores numéricos com casa décima. Exemplo: 1.0, 2.456, -5.9090, 0.0, etc.
- Operações que podem ser realizadas: adição, subtração, multiplicação, divisão, potenciação;

Tipo Caractere:
- Utilizado para armazenar um único caractere (podendo ser um dígito ou não). Exemplo: ‘1’, ‘a’, ‘A’, ‘*’, ‘?’, ‘/’, ‘:’, etc.
- Os valores caracteres são representados entre apóstrofo (‘’) para diferenciar dos nomes das variáveis;

Tipo String (cadeia de caracteres):
- Uma string é formada por dois ou mais caracteres. Exemplo: “aa”, “Maria”, “Pedro”, “112”, etc;
- Toda string é representada entre aspas (“”). Isso é necessário para diferenciar do nome das variáveis;

Tipo Booleano:
- Representado por apenas dois valores excludente: verdadeiro ou true e falso ou false;

2.8 Declaração de Variáveis

Como já sabemos, uma variável é uma posição de memória que será utilizada por um programa para armazenar informações que serão manipuladas. Toda variável a ser utilizada em um programa dever ser declarada.

Declarar uma variável significa fornecer um nome para a mesma e também indicar o tipo de dado que a variável irá armazenar. Do ponto de vista da máquina, quando declaramos uma variável o computador simplesmente “reserva” um endereço físico para armazenar informações e também associa a esse endereço físico um nome (endereço lógico). Através do tipo, o computador determina a quantidade de memória para cada variável e também “sabe” quais são as operações possíveis para cada uma.

Existem algumas linguagens de programação que não exigem a declaração de variáveis. Programas escritos nessas linguagens são mais complicados de serem entendidos e também de dar manutenção. A maneira de declarar variáveis varia. A sintaxe em Porto-C para declarar uma variável é:

<tipo_de_dado> <nome_da_variável>

onde **tipo_de_dado** é qualquer tipo válido na linguagem e **nome_da_variável** será o endereço lógico a ser manipulado. Exemplo de declaração de variável:

```
inteiro a
real b
caractere c
inteiro x, y, z
```

Quando houver a necessidade de se utilizar variáveis em um programa, as mesmas deverão ser declaradas logo no início. Exemplo:

```
Início
        inteiro a
        real b
        string nome

        .
        .
        .
Fim
```

2.9 Operadores

Outro recurso disponível nas linguagens de programação são os operadores. Os operadores quando utilizados informam ao computador como manipular o conteúdo das variáveis. Basicamente os operadores são agrupados em quatro categorias:

- Operador de atribuição;
- Operadores aritméticos;
- Operadores relacionais;
- Operadores lógicos;

Operador de Atribuição

O operador de atribuição é responsável por alterar o conteúdo armazenado em uma variável. Lembre-se que alterar o conteúdo de uma variável significa alterar o conteúdo armazenado em uma posição de memória.

O operador de atribuição varia de uma linguagem de programação para outra. O operador mais utilizado é o sinal de igual (=), mas por exemplo a linguagem de programação Pascal utiliza o símbolo de dois pontos juntamente com o sinal de igual (:=). Em Portu-C iremos utilizar o símbolo = para indicar que uma variável está recebendo um valor. Exemplo:

- A = 10 (atribui / armazena o número 10 na variável A)
- B = 25 (atribui / armazena o número 25 na variável B)
- C = ‘A’ (atribui / armazena o caractere A na variável C)
- nome = “Maria” (atribui / armazena a string “Maria” na variável nome)
- F = A (atribui / armazena o conteúdo da variável A na variável F)

Exemplo 2:

```
Inicio
        inteiro a, b, c, d
        a = 2
        b = 3
        c = 10
        d = 5
Fim
```

Operadores Aritméticos
Os operadores aritméticos básicos presentes nas linguagens de programação são:

Operação	Símbolo	Exemplo
Adição	+	A + B
Subtração	-	A – B
Multiplicação	*	A * B
Divisão	/	A / B

Em Porto-C iremos adotar dois operadores adicionais:

Operação	Símbolo	Exemplo
Potenciação	^	A ^ B
Resto da divisão inteira	%	A % B
Raiz quadrada	sqrt()	sqrt(A)

Um detalhe que deve ser enfatizado é o fato de que a ordem dos operadores aritméticos é o mesmo expresso pela matemática.
1) Potenciação e radiciação
2) Multiplicação ou divisão ou resto de divisão (a operação que vier primeiro na expressão)
3) Adição ou subtração (a operação que vier primeiro)

Para dar prioridade a uma operação, utilizamos apenas parênteses. Na matemática além dos parênteses, utilizamos chaves e colchetes, mas em computação só parênteses. Exemplo:
- A + B * C
- (A + B) * C
- A – B * C + D * E
- A – B * C + D * E / F

Exemplo: Algoritmo para exemplificar o operador de atribuição e os operadores aritméticos.

```
Início
        inteiro x, y, z
        x = 10
        y = 2
        z = x / y
        x = x % y
Fim
```

Operadores Relacionais
Enquanto os operadores aritméticos são utilizados para realizar cálculos matemáticos com o conteúdo das variáveis, os operadores relacionais como o próprio nome indica são utilizados para relacionar o conteúdo das variáveis. Ao relacionar o conteúdo das variáveis conseguimos descobrir

se os conteúdos são iguais ou diferentes, se o conteúdo de uma variável é maior ou menor que o conteúdo de outra variável.

Os operadores relacionais são:

Operação	Símbolo	Exemplo
Igualdade	==	A == B
Maior	>	A > B
Maior ou igual	>=	A >= B
Menor	<	A < B
Menor ou igual	<=	A <= B
Diferente	!=	A != B

Quando utilizamos os operadores aritméticos o resultado será um valor numérico, mas quando utilizamos os operadores relacionais o resultado será um valor booleano (verdadeiro ou falso). Por exemplo: considere duas variáveis inteiras com os seguintes conteúdos A ← 3 e B ← 10.

Operação	Resultado
A == B	Falso
A != B	Verdadeiro
A >= B	Falso
A <= B	Verdadeiro

Operadores Lógicos

Os operadores lógicos são extremamente importantes no desenvolvimento de programas porque normalmente podemos ter várias expressões aritméticas e relacionais e devemos combinar os seus resultados afim de obter apenas um. Os operadores lógicos são utilizados para esse propósito.

O resultado de uma expressão com a utilização dos operadores lógicos é booleano. Em algoritmos iremos utilizar três operadores lógicos: E, OU e NÃO. A operação E resulta em um valor verdadeiro se todos os operandos tiverem valor verdadeiro, caso contrário o resultado será falso. Para a operação OU o resultado será verdadeiro se um dos operandos tiverem resultado verdadeiro e será falso somente quando todos os operandos tiverem valor falso. A operação NÃO é utilizada para inverter o resultado de uma operação. Veja a tabela abaixo:

Tabela Verdade:

p	q	p E q	NÃO(p E q)	p OU q	NÃO(p OU q)
Verdadeiro	Verdadeiro	Verdadeiro	Falso	Verdadeiro	Falso
Verdadeiro	Falso	Falso	Verdadeiro	Verdadeiro	Falso
Falso	Verdadeiro	Falso	Verdadeiro	Verdadeiro	Falso
Falso	Falso	Falso	Verdadeiro	Falso	Verdadeiro

2.10 Comandos de Entrada e Saída de Dados

Até esse ponto já sabemos como o computador armazena os dados que serão processados e também como manipular as informações. Agora nos restar saber como armazenar uma informação digitada por um operador através do teclado e também como enviar mensagens no vídeo para que o usuário possa leia e seguir as instruções do programa.

Comando para entrada de dados
A instrução para entrada de dados é muito importante para um programa, pois normalmente os dados que serão processados devem ser informados por um operador. Quando o comando para entrada de dados é executado, deve-se informar o nome da variável onde o computador irá armazenar a informação.

O comando para entrada de dados em Portu-C seguirá o mesmo padrão adotado para a linguagem C. Para este comando devemos informar dois parâmetros: o primeiro será uma string que representa o tipo de informação que será armazenado e o segundo o nome da variável que receberá o valor. O comando utilizado será **leia()**. Exemplos:

- leia("%d", &idade) → armazena um valor inteiro na variável idade
- leia("%f", &media) → armazena um valor real na variável média
- leia("%c", &letra) → armazena um caractere na variável letra.

O símbolo (&) indica para o computador o endereço físico da variável que receberá o valor. Caso o & seja omitido o valor lido pelo teclado não será armazenado na variável. As strings de controle utilizadas são:

String de controle	Descrição
%d ou %i	Valor inteiro
%f	Valor real (ponto flutuante)
%c	Valor caractere

Comando para saída de dados
Um comando de saída é uma instrução definida nas linguagens de programa que tem como objetivo principal exibir uma informação para o usuário. Essa informação é exibida no vídeo.

Em Portu-C iremos utilizar a instrução **imprima()**. Alguns exemplos com a instrução imprima():

- **imprima**("Qual sua idade?") → após a execução dessa instrução a string entre parênteses será impressa no vídeo.
- **imprima**("Informe a sua média semestral")
- **imprima**("Sua idade é %d", idade) → ao executar esse comando, a string %d será substituída pelo conteúdo armazenado na variável idade. Veja que o nome da variável não aparece entre as aspas.

2.11 Exemplos de Algoritmos

- Algoritmo para leia dois números inteiros, executar a soma e exibir o resultado:

```
Inicio
        inteiro a, b, r
        imprima("Digite o primeiro valor")
        leia("%d", &a)
        imprima("Digite o segundo valor")
        leia("%d", &b)
        r = a + b
        imprima("O resultado da soma é %d", r)
Fim
```

- Algoritmo para leia dois valores inteiros e imprima o resto da divisão do primeiro valor pelo segundo

```
Inicio
        inteiro a, b, r
        imprima("Digite o primeiro valor")
        leia("%d", &a)
        imprima("Digite o segundo valor")
        leia("%d", &b)
        r = a % b
        imprima("O resto da divisão é %d", r)
Fim
```

- Algoritmo para leia as duas notas da prova de um aluno e calcular a sua média semestral

```
Inicio
        real p1, p2, m
        imprima("Digite a primeira nota")
        leia("%f", &p1)
        imprima("Digite a segunda nota")
        leia("%f", &p2)
        m = (p1+p2)/2
        imprima("A média é %f", m)
Fim
```

2.12 Exercícios em Classe

1. Escreva um algoritmo para beber um refrigerante em lata que está na geladeira.

2. Escreva um algoritmo para trocar a lâmpada da sala de aula. Suponha que todos os itens necessários para a troca estejam disponíveis na sala.

3. Um homem precisa atravessar um rio com um barco que possui capacidade apenas para carregar ele mesmo e mais um de seus três pertences, que são: um lobo, uma cabra e um maço de alfafa. Em cada viagem só poderá ir o homem e apenas um de seus pertences. A seguinte regra deverá ser respeitada: o lobo não pode ficar sozinho com a cabra e nem a cabra sozinha com o maço de alfafa. Escreva um algoritmo para fazer a travessia dos pertences que estão em uma margem do rio para a outra (uma travessia segura).

4. Supondo que as variáveis NB, NA, NMat, SX sejam utilizadas para armazenar a nota do aluno, o nome do aluno, o número da matrícula e o sexo. Declare-as corretamente, associando o tipo primitivo adequado ao dado que será armazenado.

5. Resolva as seguintes expressões:

a) 5+9+7+8/4
b) 1-4*3/6-3^2
c) 5^2-4/2+5%2

6. Supondo que A, B e C são variáveis do tipo inteiro, com valores iguais a 5, 10 e -8, respectivamente, e uma variável real D, com valor 5. Quais os resultados das expressões aritméticas a seguir?
   a) 2*A%3-C
   b) -2*C/4
   c) 20/4/4+8^2/2
   d) 30%4*33*-1
   e) –C^2+(D*10)/A
   f) $\sqrt{A^{B/A}}$ +C*D

7. Escreva cada uma das expressões abaixo de forma que possam ser processadas por um computador:

a) $y = \dfrac{2+a}{b-3} - 2x + x^3$

b) $x = \dfrac{\dfrac{b}{a+c} + 4a}{\dfrac{d-2a}{3+c}}$

c) $x = \dfrac{-b + \sqrt{b^2 - 4ac}}{2a}$

d) $x = \dfrac{-b - b^3 - 4ac + 2a^2}{\dfrac{2a}{(b+1)^2}}$

8. Determine o valor de cada uma das expressões abaixo:

a) 2 < 5
b) 15/3 == 5
c) 21/3%2 <= 3

9. Determine o valor de cada uma das expressões abaixo:

a) 2 < 5 E 15/3 == 5
b) 2 < 5 OU 15/3 == 5
c) 20/(18/3) != (21/3)/2

d) Não(3^2/3<15-35%7)

Considerando o seguinte trecho de um algoritmo em PortuC:

```
Inicio
        inteiro pig, vari, total, a, i
        real valor_a, x

        vari = 2
        total = 10
        valor_a = 7.0
        a = 4
        i = 80
        x = 4.0
```

Informe os valores armazenados em cada uma das variáveis após a execução de cada um dos comandos abaixo:

```
        x = total / vari
        x = x + 1
        a = a + i
        pig = 10
        a = i / pig
        a = a + i % 6
        valor_a = pig * valor_a + x
```

10. Escreva um algoritmo que forneça o valor em graus Fahrenheit de uma temperatura expressa em graus Celsius dada pela expressão abaixo:

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

11. Escreva um algoritmo que leia um valor inteiro de três dígitos e mostre o valor do meio. Se o valor de entrada for 123 deverá ser exibido 2.

12. A revendedora de carros Pica Pau Ltda, paga a seus funcionários vendedores, dois salários mínimos fixo, mais uma comissão fixa de R$ 500,00 por carro vendido e mais 5% do valor das vendas. Escreva um algoritmo que calcule o valor do salário de um vendedor.

13. Em um consórcio tem-se o número total de prestações, a quantidade de prestações pagas e o valor atual da prestação. Escreva um algoritmo que determine a porcentagem de prestações pagas e também o saldo devedor do cliente.

14. O sistema de avaliação de uma disciplina é composto por três provas. A primeira prova tem peso 2, a segunda tem peso 5 e a terceira peso 3. Escreva um algoritmo que calcule a média de um aluno na disciplina.

15. Escreva um algoritmo que leia um número entre 0 e 60 e imprima o seu sucessor, sabendo que o sucessor de 60 é 0. Não pode ser utilizado nenhum comando de seleção ou repetição.

2.13 Exercícios Complementares

1. Defina Algoritmos.

2. O que são variáveis? Por que são utilizadas em um programa?

3. O que são tipos de dados? Como são aplicados em um programa?

4. Quais os valores das expressões a seguir:

a) 2+3*6
b) 12/3*2-5
c) 31/4
d) 31%4
e) 9^2

5. Indique o valor armazenado em cada variável após a execução das instruções abaixo:

```
Início
        inteiro q, w, r
        real e

        q = 10
        q = 10 + 30
        w = -1
        w = w + q
        q = q % w
        q = w / (q+40)
        e = 2*q/w
        r = 0
        r = r + 1
        r = r + 1
Fim
```

6. Determine o valor de cada uma das expressões abaixo sabendo que A, B, C contêm respectivamente 2, 7 e 3. Considere também a existência de uma variável lógica L contendo o valor falso (F).

a) B == A * C E (L OU V)
b) B > A OU B == AA
c) L E B / A >= C OU NÃO(A<=C)
d) NÃO(L) OU V E $\sqrt{A+B}$ >= C

7. Escreva um algoritmo que leia o valor do raio de uma esfera e calcule o seu volume dado pela expressão abaixo:

$V = \frac{4\pi r^3}{3}$. Considere que π vale 3.1416.

8. Escreva um algoritmo que leia quatro valores inteiros. Calcule e mostre o valor da sua média harmônica.

$$mh = \frac{4}{\frac{1}{a} + \frac{1}{b} + \frac{1}{c} + \frac{1}{d}}$$

9. Faça um algoritmo que leia um valor inteiro positivo e menor que 1000. Armazene-o numa variável inteira e determine a soma dos dígitos que formam o valor. Por exemplo: o valor 397 tem soma 19.

10. Escreva um algoritmo que leia um valor inteiro composto por três dígitos. Inverta o valor lido e o armazene em outra variável. Por exemplo: valor lido = 235 deverá ser armazenado em outra variável o valor 532.

11. Faça um algoritmo que lei um valor inteiro composto por três dígitos. Inverta o valor lido e faça a soma com o valor original. Por exemplo: valor lido foi 432. Deverá ser exibido o resultado da soma: 432 + 234

12. Faça um algoritmo que leia o valor de um depósito e o valor da taxa de juros. Calcule e exiba o valor do rendimento.

13. Faça um algoritmo que leia dois valores inteiros e positivos e armazene nas variáveis A e B respectivamente. Em seguida troque o conteúdo das variáveis, ou seja, A deverá receber o valor de B e B o valor de A.

Capítulo 3

Estruturas de Seleção

3.1 Introdução

Nos capítulos anteriores os nossos algoritmos eram caracterizados apenas por uma estrutura seqüencial, ou seja, não havia nenhum desvio condicional. O algoritmo começava com a declaração de variáveis, executava um conjunto de instruções e era finalizado.

Na prática nem sempre se tem um algoritmo seqüencial. A maioria dos problemas do dia-a-dia e muitas das aplicações comerciais necessitam de testes para se chegar a solução do problema.

Um exemplo muito simples ocorre no fechamento das notas aqui na faculdade. Todo final de ano, no boletim escolar na frente do nome de cada disciplina aparece escrito se o aluno foi aprovado, ou se está de exame ou retido. Neste caso, para cada aluno da faculdade, a partir de suas notas semestrais o sistema pode realizar uma entre três opções. A escolha de uma das opções depende da média do aluno. Vale destacar que o sistema escolherá apenas uma opção e automaticamente eliminará as outras.

Outro exemplo muito comum ocorre no pagamento de prestações, mensalidades dentre outras. Nesses sistemas, o operador deverá entrar com a data do vencimento da parcela e o sistema automaticamente detectará se o cliente em atraso e conseqüentemente calculará o valor dos juros.

Como pode ser percebido, a utilização de desvios condicionais é de extrema importância na construção de sistemas computacionais. Neste capítulo serão abordadas duas estruturas de seleção ou desvio condicional.

3.2 Estrutura de Seleção Simples

Uma estrutura de seleção é utilizada quando um algoritmo necessita escolher uma instrução ou um conjunto de instruções quando uma condição (representada por uma expressão relacional ou lógica) é satisfeita ou não.

Em Portu-C a instrução de seleção será representada pela instrução se. Sua sintaxe é:

```
se (condição)
Inicio
        <instrução_1>
        <instrução_2>
        .
        .
        <instrução_n>
Fim
```

Quando o computador encontra o bloco de instruções acima, inicialmente a condição é testada. Caso o resultado do teste seja um valor verdadeiro, o conjunto de instruções dentro do bloco *se* será executado. Caso a condição resulte em um valor falso todo o bloco de instruções é ignorado e o fluxo de execução do algoritmo segue logo abaixo do bloco *se*.

Chamamos o bloco de estrutura condicional ou de seleção porque o computador através do conjunto de valores das variáveis poderá executar ou não as instruções. O número de instruções dentro do bloco de seleção é variável podendo ter apenas uma ou várias dependendo do problema que está sendo resolvido.

Exemplo 1: algoritmo para verificar se um número inteiro fornecido pelo usuário é par.

```
Inicio
        inteiro n, r
        imprima(“digite um valor inteiro: “)
        leia(“%d”, &n)
        r = n % 2
        se( r == 0 )
        inicio
                imprima(“o número informado é par”)
        fim
Fim
```

Exemplo 2: algoritmo para verificar se um número fornecido pelo usuário é positivo.

```
Inicio
        inteiro n
        imprima(“digite um valor inteiro: “)
        leia(“%d”, &n)
        se( n >= 0 )
        inicio
                imprima(“o número informado é positivo”)
        fim
Fim
```

3.3 Estrutura de Seleção Composta

Na estrutura de seleção simples, o conjunto de instruções só é executado quando o teste lógico resultar em um valor verdadeiro. Quando o teste resultava em um falso nenhuma instrução era executada.

Podemos reestruturar o comando de seleção simples a trabalhar quando o teste lógico resultar em um valor falso. Neste caso acrescenta-se apenas um complemento ao comando *se*.

Sua nova estrutura passa a ser:

```
se (condição)
Inicio
        <instrução_1>
        <instrução_2>
        .
        .
        <instrução_n>
Fim
senão
Inicio
        <instrução_1>
        <instrução_2>
        .
        .
        <instrução_n>
Fim
```

Da mesma forma que a estrutura de seleção simples, a condição é testada. Caso o valor resulte em verdadeiro, o conjunto de instruções logo abaixo do comando *se* será executado e o conjunto de instruções pertencentes ao bloco *senão* serão ignorados. Por outro lado, caso o teste resulte em um valor falso o conjunto de instruções pertencentes ao bloco *senão* serão executados. Após a execução de um dos conjuntos de instruções o fluxo de execução do programa continua sua execução logo abaixo da estrutura de seleção.

Duas observações muito importantes a serem feitas:

- Pode-se ter uma instrução *se* sem o bloco *senão*, mas o contrário não é verdadeiro. Toda instrução *senão* deve estar associada a uma instrução *se*. Cada instrução *se* pode ter no máximo uma instrução *senão*.
- Ao escrever a instrução *senão* não se deve colocar nenhuma condição lógica, pois a sua execução sempre ocorrerá quando a condição da instrução *se* for falsa (somente nessa condição).

Exemplo 3: algoritmo para verificar se um número inteiro fornecido pelo usuário é par ou ímpar.

```
Inicio
        inteiro n, r
        imprima("digite um valor inteiro: ")
        leia("%d", &n)
        r = n % 2
        se( r == 0 )
                imprima("o número informado é par")
        senão
                imprima("o número é ímpar")
Fim
```

Exemplo 4: algoritmo para verificar se um número fornecido pelo usuário é positivo ou negativo.

```
Inicio
        inteiro n
        imprima("digite um valor inteiro: ")
        leia("%d", &n);
        se( n >= 0 )
                imprima("o número informado é positivo")
        senão
                imprima("o número informado é negativo")
Fim
```

Exemplo 5: Um time de basquete está fazendo uma peneira para selecionar alguns jogadores. Para ser aceito no time o jogador deverá ter mais que 15 anos e ter altura acima de 180cm.

```
Inicio
        inteiro idade
        real altura
        imprima("digite a sua idade: ")
        leia("%d", &idade)
        imprima("informe a sua altura: ")
        leia("%f", &altura)
        se( idade > 15 E altura > 180 )
                imprima("você foi aceito no time")
        senão
                imprima("você não foi aceito no time")
Fim
```

3.4 Estrutura de Seleção Agrupada

Dentro de uma instrução *se* ou *senão* pode-se ter qualquer outra instrução, até mesmo outra estrutura de seleção. Considere agora a seguinte situação: calcular a média de um aluno que fez duas provas. Caso a média seja maior ou igual a 6 o aluno foi aprovado, caso contrário pode-se ter duas situações possíveis. Se a média for menor que 4 o aluno foi reprovado, por outro lado, se a média for maior que 4 e menor que 6 o aluno está de exame.

A sintaxe para o comando de seleção alinhado é:

```
se(condição1)
        Instrução_1
        Instrução_2
        . . .
senão
        se(condição2)
                instrução_1
                instrução_2
                . . .
        senão
```

```
        se(condição3)
                instrução_1
                instrução_2
                . . .
```

Não se esqueça que cada instrução senão sempre está associada a uma instrução se. Através de comandos de seleção agrupados pode-se testar várias condições. A única desvantagem desse recurso é que a medida que o número de condições aumenta significativamente o número de instruções agrupadas aumentará proporcionalmente e dessa forma, a legibilidade do programa será prejudicada.

Exemplo 6: algoritmo para calcular a média do aluno conforme situação descrita.

```
Inicio
        real p1, p2, m
        imprima("Informe o valor da primeira nota: ")
        leia("%f", &p1)
        imprima("Informe o valor da segunda nota: ")
        leia("%f", &p2)
        m = (p1+p2)/2
        se( m >= 6 )
                imprima("o aluno foi aprovado")
        senão
                se( m < 4 )
                        imprima("o aluno foi reprovado")
                senão
                        imprima("o aluno ainda fará o exame")
Fim
```

3.5 Exercícios em Classe

1. Escreva o comando de seleção para cada uma das situações a seguir:
   - Se x for maior que y, multiplique o valor de x por y e armazene em y.
   - Se x for menor ou igual a y, some o valor de x com y e armazene em x. Caso contrário, subtraia o valor de x por y e armazene em y.

2. Escreva um algoritmo que leia dois valores inteiros, some os valores e imprima o resultado somente se os valores forem diferentes.

3. Escreva um algoritmo que leia um valor inteiro e imprima uma mensagem no vídeo informando se o valor é ou não múltiplo de 4.

4. Uma empresa de vendas oferece para seus clientes um desconto em função do valor da compra do cliente. Este desconto é de 20% se o valor da compra for maior ou igual a R$ 200,00 e 15% se for menor. Escreva um algoritmo que imprima o valor da compra do cliente e o valor do desconto obtido.

5. Escreva um algoritmo que leia a altura e o sexo de uma pessoa. Calcule e imprima o seu peso ideal utilizando as seguintes fórmulas:
   - Para homens: (72.7*h)-58;
   - Para mulheres: (62.1*h)-44.7;

6. Um vendedor tem seu salário calculado em função do valor total de suas vendas. Este cálculo é feito de acordo com o seguinte critério: se o valor total de suas vendas for maior que R$ 20000,00 o vendedor receberá como salário 10% do valor das vendas. Caso contrário receberá apenas 7,5% do valor das vendas. Escreva um algoritmo que calcule o valor ganho pelo vendedor.

7. Escreva um algoritmo que leia três valores e verifique se os mesmos podem formar os lados de um triângulo. Para que os valores formem os lados de um triângulo é necessário que a soma de cada lado seja menor que a soma dos outros dois.

8. Desenvolva um algoritmo que calcule o valor da expressão abaixo:

$$y = \frac{8}{2 - x}$$

9. Um hotel cobra R$ 60,00 a diária e mais uma taxa de serviços. A taxa de serviços é de:

- R$ 5,50 por diária, se o número de diárias for maior que 15;
- R$ 6,00 por diária, se o número de diárias for igual a 15;
- R$ 8,00 por diária, se o número de diárias for menor que 15.

Construa um algoritmo que mostre o valor da conta de um cliente.

10. Faça um algoritmo que calcule o valor de y:

$$y = \begin{cases} 1, & \text{Se x <= 1} \\ 2, & \text{Se 1 < x <= 2} \\ x^2, & \text{Se 2 < x <= 3} \\ x^3, & \text{Se x > 3} \end{cases}$$

11. Escreva um algoritmo que resolva as raízes de uma equação do segundo grau na forma: $ax^2 + bx + c = 0$.

12. Construa um algoritmo que leia três valores inteiros (suponha que o usuário tenha digitado três valores diferentes). Imprima o menor valor informado.

13. Escreva um algoritmo que leia três valores inteiros e diferentes. Mostre-os em ordem crescente.

3.6 Exercícios Complementares

1. Escreva o comando de seleção para cada uma das situações abaixo:

- Se x for maior que y ou, se z for menor ou igual a 30, multiplique x por 2. Caso contrário, divida x por 2 e divida z por 5;
- Se o desconto for menor que 25% e o preço do produto for maior que R$ 50,00 então, de um desconto de 25% no preço do produto.

2. Faça um algoritmo que leia dois números e mostre o resultado da divisão do primeiro valor pelo segundo. Lembre-se: não existe divisão por zero.

3. Escreva um algoritmo que leia dois valores inteiros e informe se os valores são múltiplos um do outro.

4. Escreva um algoritmo que calcule o valor do imposto de renda de um contribuinte. Considere que o valor do imposto é calculado de acordo com a tabela abaixo:

Renda Anual	Alíquota
Até R$ 10000,00	Isento
> R$ 10000,00 e <= R$ 25000,00	10%
Acima de R$ 25000,00	25%

5. Escreva um algoritmo que leia três valores inteiros. Mostre o maior e o menor valor.

6. Escreva um algoritmo que leia o ano de nascimento de uma pessoa e o ano atual. Imprima a idade da pessoa. Não esqueça de verificar se o ano de nascimento é um ano válido.

7. Construa um algoritmo que leia o salário de uma pessoa e imprima o desconto do INSS segundo a tabela abaixo:

Renda Mensal	Alíquota
<= R$ 600,00	Isento
> R$ 600,00 e <= R$ 1200,00	20%
> R$ 1200,00 e <= R$ 2000,00	25%
> R$ 200,00	30%

8. Um comerciante comprou um produto e quer vendê-lo com um lucro de 45% se o valor da compra for menor que R$ 20,00; caso contrário, o lucro será de 30%. Construa um algoritmo que leia o valor do produto e imprima o valor da venda.

9. A confederação brasileira de natação irá promover eliminatórias para o próximo mundial. Faça um algoritmo que leia a idade de um nadador e mostre sua categoria segundo a tabela abaixo:

Categoria	Idade (anos)
Infantil A	5 - 7
Infantil B	8 – 10
Juvenil A	11 – 13
Juvenil B	14 – 17
Sênior	Maiores de 18

10. Escreva um algoritmo que leia o valor de dois números inteiros e a operação aritmética desejada (adição, subtração, multiplicação e divisão). Calcule a resposta adequada.

11. Escreva um algoritmo que leia os três números (possíveis lados de um triângulo) e imprima sua classificação quanto aos lados:
    - Três lados iguais eqüilátero;
    - Dois lados iguais isóceles;
    - Três lados diferentes escaleno;

Capítulo 4

Estruturas de Repetição

4.1 Introdução

No capítulo anterior foi mostrado que no desenvolvimento de programas de computadores normalmente há a necessidade de se fazer testes lógicos para que o programa possa executar determinadas instruções, ou seja, através do teste lógico existe a seleção de instruções a serem executadas.

Até este ponto do curso, os algoritmos desenvolvidos realizavam apenas um conjunto de instruções uma única vez e o algoritmo era finalizado. Suponha a seguinte situação: em uma sala com 50 alunos, deseja-se calcular a média semestral de cada aluno considerando que todos realizaram duas provas. Para resolver esse problema com os conceitos de programação apresentados até agora seria necessário executar o mesmo programa 50 vezes. Isso não parece algo eficientes, mas resolveria o problema.

Uma outra solução e a mais recomendada é a utilização de estruturas de repetição. Uma estrutura de repetição nada mais é do que uma instrução que indica ao computador para repetir um conjunto de instruções várias vezes. O número de repetições pode ser controlado pelo programador ou até mesmo pelo operador do sistema. A quantidade de vezes que um conjunto de instruções será executado depende de um teste lógico. As estruturas de repetição em programação também são conhecidas como **laços de repetição**.

Basicamente serão estudadas três estruturas básicas de repetição. O número de estruturas de repetição e a sua sintaxe são específicos de cada linguagem de programação, mas o conceito central é o mesmo. O que difere uma estrutura de repetição da outra é a ordem de execução do teste lógico. Em algumas estruturas o teste é realizado no inicio e em outras no final.

4.2 Estrutura de Repetição Enquanto ...

A estrutura de repetição **enquanto** é utilizada para repetir um conjunto de instruções várias vezes de acordo com um teste lógico que é realizado no inicio da instrução. A sintaxe para a estrutura enquanto é:

enquanto (*condição_lógica*)
início
    Instrução_1
    Instrução_2
    . . .
    instrução_n
fim

sendo *condição_lógica* é uma expressão composta por operadores relacionais e lógicos. O conjunto de instruções dentro do bloco de repetição só será executado caso a *condição_lógica* resulte em um valor verdadeiro, caso contrário, não será executado. Ao atingir o final da instrução enquanto, o computador automaticamente retorna para o inicio da instrução e testará novamente a condição e todo o processo se repetirá até que a condição resulte em um valor falso.

Exemplo 1: algoritmo para imprima os valores de 1 até 10 no vídeo.

```
Inicio
        inteiro cont
        cont = 1
        enquanto( cont <= 10 )
        início
                imprima(“%d”, cont)
                cont = cont + 1
        fim
Fim
```

Exemplo 2: Algoritmo para calcular a média dos números pares entre 0 e 100.

```
Inicio
        inteiro cont, soma
        cont = 1
        soma = 0
        enquanto(cont <= 100)
        início
        se(cont % 2 == 0 )
                        soma = soma + cont
                cont = cont + 1
        fim
        imprima(“A soma dos números pares é %d”, soma)
Fim
```

Exemplo 3: Calcular e imprima a média dos 50 alunos de uma sala de aluno. Suponha que cada aluno fez duas provas durante o semestre.

```
Inicio
        inteiro cont
        real p1, p2, media
        cont = 1
        enquanto(cont <= 50)
        início
        imprima(“Digite a primeira nota: “)
        leia(“%f”, &p1)
        imprima(“Digite a segunda nota: “)
        leia(“%f”, &p2)
        media = (p1+p2)/2
        imprima(“A media do aluno %d é %f”, cont, media)
```

```
        cont = cont + 1
    fim
Fim
```

4.3 Estrutura de Repetição Faça-Enquanto

Como já destacado na introdução deste capítulo, o que difere uma estrutura de repetição de outra é a ordem de realização do teste lógico. Na estrutura **enquanto**, o teste lógico era realizado logo no início da estrutura e dependendo do valor do teste, o conjunto de instruções dentro da estrutura podia ser executado ou não.

Outra estrutura de repetição é **faça-enquanto**. Essa estrutura é semelhante a estrutura **enquanto**. A diferença é que na estrutura **faça-enquanto** o teste lógico é realizado no final. Dessa forma, o conjunto de instruções pertencentes a estrutura é realizado pelo menos uma vez. A sintaxe para a estrutura faça-enquanto é:

```
faça
início
    Instrução_1
    Instrução_2
    . . .
    instrução_n
fim
enquanto (condição_lógica)
```

Exemplo 4: Refazendo o exemplo 1 usando a estrutura faça-enquanto.

```
Inicio
    inteiro cont
    cont = 1
    faça
        imprima(“%d”, cont)
        cont = cont + 1
    enquanto( cont <= 10 )
Fim
```

Exemplo 5: Refazendo o exemplo 2 usando a estrutura faça-enquanto

```
Inicio
    inteiro cont, soma
    cont = 1
    soma = 0
    faça
    início
    se(cont % 2 == 0 )
        soma = soma + cont
        cont = cont + 1
```

```
        fim
        enquanto(cont <= 100)
        imprima("A soma dos números pares é %d", soma)
Fim
```

Exemplo 6: Refazendo o exemplo 3 usando a estrutura faça-enquanto

```
Inicio
        inteiro cont
        real p1, p2, media
        cont = 1
        faça
        início
        imprima("Digite a primeira nota: ")
        leia("%f", &p1)
        imprima("Digite a segunda nota: ")
        leia("%f", &p2)
        media = (p1+p2)/2
        imprima("A media do aluno %d é %f", cont, media)
                cont = cont + 1
        fim
        enquanto(cont <= 50)
Fim
```

4.4 Estrutura de Repetição Para

A estrutura de repetição para é uma das mais usadas e também a preferidas da maioria dos programadores. Como já enfatizado neste capítulo, as três estruturas têm desempenho muito semelhante mudando apenas a sintaxe de cada uma.

Tanto na estrutura **enquanto** quanto **faça-enquanto** havia um teste lógico para informar ao computador se o conjunto de instruções seria executado ou não. Além do mais, para controlar o número de execuções normalmente se utiliza uma variável que a cada ciclo de execução, a mesma é incrementada.

Na estrutura de repetição **para**, tanto a inicialização da variável de controle quanto o teste lógico e o incremento ou decremento são escritos em uma única linha. A sintaxe para a estrutura de repetição para é:

```
para(inicialização; teste_lógico; incremento/decremento)
início
        Instrução_1
        Instrução_2
        . . .
        instrução_n
fim
```

sendo: *inicialização* se refere a forma como a variável de controle do laço será inicializada; *teste_lógico* é o teste que será realizado a cada ciclo de execução do laço. Para um valor verdadeiro o conjunto de instruções dentro do laço será executado; *incremento/decremento* se refere a forma como a variável de controle do laço será alterada.

A inicialização da variável de controle ocorre apenas uma vez, na primeira vez que o laço for executado, mas tanto o teste lógico quanto o incremente ou o decremento serão executados todas as vezes.

Exemplo 7: Refazendo o exemplo 1 usando a estrutura de repetição para.

```
Inicio
        inteiro cont
        para(cont = 1; cont <= 10; cont = cont + 1)
                Imprimir(“%d”, cont)
Fim
```

Como pode ser percebido, a estrutura de repetição para simplificar um algoritmo. Esse é o motivo pela qual é a estrutura preferida dos programadores.

Dos três parâmetros pode-se omitir alguns. Os exemplos 8, 9 e 10 têm o mesmo efeito do exemplo 7.

Exemplo 8

```
Inicio
        inteiro cont
        cont = 1
        para(; cont <= 10; cont = cont + 1)
                imprima(“%d”, cont)
Fim
```

Exemplo 9

```
Inicio
        inteiro cont
        para(cont = 1; cont <= 10;)
        início
                imprima(“%d”, cont)
                cont = cont + 1
        fim
Fim
```

Exemplo 10

```
Inicio
        inteiro cont
        cont = 1
```

```
        para(; cont <= 10;)
        início
                imprima(“%d”, cont)
                cont = cont + 1
        fim
Fim
```

4.5 Exercícios em Classe

Estrutura equanto

1. Dado o conjunto de instruções a seguir, coloque o comando de repetição adequadamente:
a) executar um conjunto de instruções 10 vezes;
b) executar um conjunto de instruções N vezes, onde N é uma variável informada pelo usuário.

2. Execute o teste de mesa para os trechos abaixo:
a)

```
A = 1
S = 0
enquanto (A<5)
início
        S = S + A
        A = A + 2
fim
```

b)

```
A = 1
N = 0
S = 0
enquanto (S < 12 )
início
        S = S + A
        A = A + 1
        N = N + 1
fim
```

3. Represente cada uma das seqüências de números abaixo usando expressões matemáticas (lei formação – somatório ou produtório)
   a) 1 + 2 + 3 + 4 + ... + 20
   b) 1 * 2 * 3 * 4 * ... * 20
   c) $\frac{1}{1}+\frac{1}{2}+\frac{1}{3}+...+\frac{1}{50}$
   d) $\left(\frac{1}{1}\right)^1+\left(\frac{1}{2}\right)^2+\left(\frac{1}{3}\right)^3+...+\left(\frac{1}{n}\right)^n$
   e) $\frac{1}{1}-\frac{1}{2}+\frac{1}{3}-\frac{1}{4}+...-\frac{1}{10}$

4. Faça um algoritmo que imprima todos os números pares entre 85 e 907.

5. Escreva um algoritmo que calcule a soma dos primeiros 10 valores inteiros, ou seja, 1+2+3...+10.

6. Construa um algoritmo que, para um grupo de 50 valores inteiros fornecidos pelo usuário determine:
   a) a soma dos números positivos;
   b) a quantidade de valores negativos;

7. Criar um algoritmo que leia um número que será o limite superior de um intervalo e o incremento. Imprimir todos os números no intervalo de 0 até o limite lido. Por exemplos:

Limite superior: 20
Incremento: 5
Saída no vídeo: 0 5 10 15 20

8. Escreva um algoritmo que leia 15 números fornecidos pelo usuário. Imprima o maior valor informado.

9. Escreva um algoritmo que leia vários números fornecidos pelo usuário. Imprima o menor valor informado.

10. Escreva um algoritmo que calcule o fatorial de um número.

11. Escreva um algoritmo que calcule o valor da expressão abaixo:

$$h = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{20}$$

Estrutura faça...equanto

12. Reescrever os algoritmos do exercício 2 usando a estrutura de repetição faça...enquanto.

13. Faça um algoritmo que imprima no vídeo todos os números entre 1 e 20.

14. Escreva um algoritmo que multiplique dois valores inteiros e positivos sem utilizar o operador de multiplicação (*).

15. Criar um algoritmo que imprima os 10 primeiros termos da série de Fibonacci. A série é dada por: 1 1 2 3 ...

16. Escreva um algoritmo que leia 20 valores inteiros e imprima a soma dos números cujos quadrados são menores que do que 225.

17. Calcular e imprima a média de cada um dos 50 alunos de uma turma. Suponha que foram aplicadas 2 provas.

18. Faça um algoritmo que leia um número e imprima todos os seus divisores.

19. Escreva um algoritmo que leia 15 números fornecidos pelo usuário. Imprima quantos números maiores que 30 foram digitados.

20. Desenvolva um algoritmo que calcule o valor da expressão abaixo:

$$h = \frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$$

Estrutura para

21. Reescreva os algoritmos do exercício 2 usando a estrutura de repetição para.

22. Desenvolva um algoritmo para calcular o valor da expressão abaixo:

$$y = \frac{1}{n} + \frac{2}{n-1} + \frac{3}{n-3} + \frac{4}{n-4} + \cdots n$$

23. Faça um algoritmo para calcular o valor da expressão:

$$h = \frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \cdots \pm \frac{1}{n}$$

24. Desenvolva um algoritmo para calcular o valor da expressão:

$$x = \frac{1}{1!} + \frac{2}{2!} + \frac{3}{3!} + \cdots + \frac{n}{n!}$$

25. Chico tem 1,50m e cresce 2 centímetros por ano, enquanto que, Juca tem 1,10m e cresce 3 centímetros por ano. Escreva um algoritmo que calcule e imprima quantos anos são necessários para que Juca seja maior que Chico.

26. Criar um algoritmo que calcule e imprima o valor de b^n. O valor de n deverá ser maior ou igual a zero e o valor de b deverá ser inteiro.

4.6 Exercícios Complementares

1. Faça o teste de mesa nos algoritmos abaixo:

a)

```
A = 1
B = 4
S = 0
enquanto (A <= 6 )
início
        se (A <= B)
                S = S + A*B
        senão
                S = S - A*B
        A = A + 1
        B ← B - 1
fim
```

b)

```
G = 1
enquanto (G <= 5)
início
        C = 0
        enquanto (C < G)
                C = C + 1
        G = G + 1
fim
```

2. Reescreva os algoritmos do exercício 1 usando a estrutura de repetição faça...enquanto.

3. Reescreva os algoritmos do exercício 1 usando a estrutura de repetição para.

4. A série de Ricci difere da série de Fibonacci porque os dois primeiros valores são fornecidos pelo usuário. Os demais termos são gerados da mesma forma que a série de Fibonacci. Criar um algoritmo que imprima os 15 primeiros valores da série.

5. Criar um algoritmo que leia um número que servirá para controlar os números pares que serão impressos a partir de 2. Exemplo:

Quantidade: 4
Saída no vídeo: 2 4 6 8

6. Faça um algoritmo que leia um número e imprima a soma dos números múltiplos de 5 no intervalo aberto entre 1 e o número lido. Suponha que o número lido será maior que zero.

7. Faça um algoritmo que entre com números e imprima o triplo de cada número. O algoritmo termina quando entrar o número -1.

8. Desenvolva um algoritmo números até entrar o número -999. Para cada número imprima seus divisores.

9. Criar um algoritmo que receba a idade, a altura e o peso de 30 pessoas. Calcule e imprima:
   a) quantidade de pessoas com idade superior a 50 anos;
   b) média das alturas das pessoas com idade entre 10 e 20 anos;

10. Calcule o valor de e^x. O valor de x deverá ser informado via teclado. O valor da expressão deverá ser calculado pela soma dos 10 primeiros termos da série abaixo.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \cdots$$

Capítulo 5

Introdução a Linguagem C

5.1 Introdução

A linguagem de programação C surgiu em 1972. É uma linguagem de programação de uso geral desenvolvida por Dennis Ritche no Bell Laboratory. Os princípios básicos de C são originários de outras linguagens tais como: B, BCPL e CPL. Estas linguagens são classificadas como linguagens de alto nível, mas que permitem a manipulação das informações através dos bits.

Inicialmente a linguagem C ficou restrita as Universidades e aos centros de pesquisas. A maioria dos programadores achava que era uma linguagem de difícil manipulação. O surgimento da linguagem C está intimamente relacionada com o sistema operacional UNIX. Ken Thompson utilizou a linguagem Assembler e a linguagem B para produzir versões do sistema UNIX e a linguagem C surge devido as limitações da linguagem B.

Da mesma forma, a linguagem C deu origem a linguagem C++, desenvolvida na década de 80. A diferença é que a linguagem C++ é uma linguagem orientada a objetos que utiliza um paradigma de programa diferente do que será abordado aqui.

As várias implementações da linguagem C estão baseadas no padrão ANSI C e acrescentam diversos recursos específicos.

A linguagem C é classificada como uma linguagem de nível médio por sua capacidade de manipulação em baixo nível, ou seja, permite a manipulação de endereços de memória, bits e bytes (elementos básicos através dos quais o computador funciona). Essa classificação não faz com que C seja mais fácil ou mais difícil que as outras linguagens.

Por ser também uma linguagem de alto nível, o programa escrito (código fonte) em C deverá ser convertido em linguagem de máquina para que possa ser executado. O processo de conversão pode ser realizado por dois programas diferentes: compilador ou interpretador.

No processo de compilação, o código fonte é todo convertido em uma linguagem intermediária chamada de código objeto. Na execução do programa, o computador lê código objeto.

Na interpretação, o programa interpretador lê o código fonte linha por linha, executando cada instrução. Este processo é diferente da compilação onde todo o código fonte é convertido em código objeto antes da execução.

As linguagens interpretadas são mais lentas que as linguagens compiladas.

5.2 Mapa de Memória em C

Todo programa em C ao ser compilado, cria e usa quatro regiões na memória. Cada uma dessas regiões é distinta e possuem funções específicas. A figura abaixo ilustra as quatro regiões.

A pilha tem diversas aplicações durante a execução de um programa. É na pilha que ficam armazenados os endereços de retorno das funções. O heap é uma área livre que é utilizado através das funções de alocação dinâmica de memória.

5.3. Tipos de Dados em C

Em tem-se basicamente cinco tipos de dados:

- Caractere – **char**
- Inteiro – **int**
- Ponto flutuante (real) – **float**
- Ponto flutuante de precisão dupla (real) – **double**
- Sem valor – **void**

A declaração de uma variável em C é feita da mesma forma que foi utilizada em Porto-C: primeiro o tipo da variável e logo depois o nome da variável. Toda linha de comando em C é terminada por ponto e vírgula (;). Exemplos:

```
int x;
int a, b;
float g;
double v, s;
char nome;
```

Repare que em C não existe um tipo específico para trabalhar com cadeias de caracteres (strings). Sua formação será estudada no capítulo 7.

5.4 Inicialização de Variáveis

No momento da declaração de uma variável pode-se atribuir um valor inicial a mesma. O operador responsável pela atribuição de valores a uma variável é representado pelo símbolo de igualdade (=). Exemplos:

```
int x = 5;
int a = 2, b = 3;
float y = 12.3;
char letra = ‘a’;
```

5.5 Operadores em C

- **Operador de atribuição:**

Como já mostrado, o operador de atribuição é responsável por atribuir um valor a uma variável. Esse operador é representado pelo símbolo de igualdade. Exemplos:

```
int x = 2, y = 5;
x = 10;
y = x;
x = y = 20;
```

- **Operadores aritméticos:**

Os operadores aritméticos em C são: adição (+), subtração (-), multiplicação (*), divisão (/) e módulo da divisão ou resto (%).

- **Operadores relacionais:**

Os operadores relacionais em C são: >, >=, <, <=, == e !=.

- **Operadores lógicos:**

Os operadores lógicos são: e (&&), ou (||) e não (!).

5.6 Comandos de entrada e saída de dados

- **Comando de saída:**

Um comando de saída de dados é responsável por enviar uma mensagem, ou seja, uma string podendo ser composta por letras ou número para o vídeo. Em C o comando para saída de dado será representado pela instrução **printf()**. A sintaxe para o comando printf() é semelhante a do comando imprima() usado em Porto-C. Exemplo:

```
printf(“Esta é uma mensagem de exemplo”);
```

```
printf("Tenham todos uma\n boa noite");
printf("Sua nota é %f", y);
printf("Os valores são %d e %d", x, y );
```

- **Comando de entrada:**

Um comando de entrada é responsável por leia a informação informada via teclado por um operador e armazenar em uma posição de memória, ou seja, em uma variável já declarada. O comando utilizado para entrada de dados é **scanf()**. A sintaxe é semelhante ao do comando leia() previamente utilizado. Exemplo:

```
scanf("%d", &x );
scanf(%f", &media);
scanf("%c", &letra);
scanf("%d %d %d", x, y, z);
```

5.7 Estrutura de um programa em C

Um programa em linguagem C é composto por vários blocos de códigos e pela declaração de bibliotecas. Dentro do programa deve haver um bloco principal de código chamado **main()** que representa o início do programa. É a partir do main() que todo as outras instruções do programa serão executadas. Exemplo:

```
#include <stdio.h>
void main()
{
        int y;
        printf("digite um valor inteiro: ");
        scanf("%d", &y);
        printf("o valor informado foi %d", y );
}
```

5.8 Estrutura de Seleção

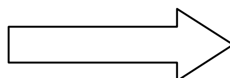
5.8.1 Comando de Seleção if

Em C tem-se dois comandos utilizados para trabalhar com seleção: if e switch. O funcionamento destas estruturas é semelhando ao apresentado em Porto-C:

- **Seleção Simples:**

Em Porto-C

```
se (condição_lógica)
Início
        Instrução_1;
        Instrução_2;
        . . .
        Instrução_n;
Fim
```



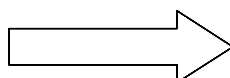
Em C

```
if (condição_lógica)
{
        Instrução_1;
        Instrução_2;
        . . .
        Instrução_n;
}
```

- **Seleção Composta:**

Em Porto-C

```
se (condição_lógica)
Início
        Instrução_1;
        Instrução_2;
        . . .
        Instrução_n;
Fim
Senão
Início
        Instrução_1;
        Instrução_2;
        . . .
        Instrução_n;
Fim
```

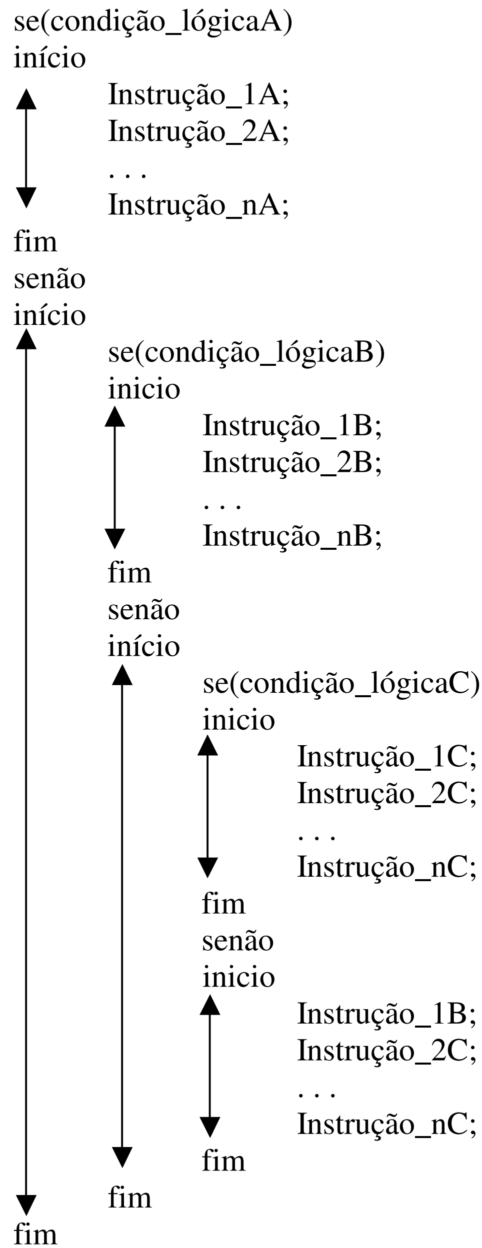


Em C

```
if (condição_lógica)
{
        Instrução_1;
        Instrução_2;
        . . .
        Instrução_n;
}
else
{
        Instrução_1;
        Instrução_2;
        . . .
        Instrução_n;
}
```

- **Seleção Encadeada ou Aninhada**

Uma seleção encadeada ou aninhada é um agrupamento de instruções de seleção, ou seja, pode-se uma instrução de seleção interna a outra instrução e assim por diante. A sintaxe em Porto-C é:



```
se(condição_lógicaA)
início
        Instrução_1A;
        Instrução_2A;
        . . .
        Instrução_nA;
fim
senão
início
        se(condição_lógicaB)
        inicio
                Instrução_1B;
                Instrução_2B;
                . . .
                Instrução_nB;
        fim
        senão
        início
                se(condição_lógicaC)
                inicio
                        Instrução_1C;
                        Instrução_2C;
                        . . .
                        Instrução_nC;
                fim
                senão
                inicio
                        Instrução_1B;
                        Instrução_2C;
                        . . .
                        Instrução_nC;
                fim
        fim
fim
```

A estrutura de seleção encadeada tem a desvantagem de ser complicada de entender a medida que o número de instruções de seleção aumenta. A sintaxe do comando de seleção encadeado em C é:

```
if(condição_lógicaA)
{
        Instrução_1A;
        Instrução_2A;
        . . .
        Instrução_nA;
}
else
{
        if(condição_lógicaB)
        {
                Instrução_1B;
                Instrução_2B;
                . . .
                Instrução_nB;
        }
        else
        {
                if(condição_lógicaC)
                {
                        Instrução_1C;
                        Instrução_2C;
                        . . .
                        Instrução_nC;
                }
                else
                {
                        if(condição_lógicaD)
                        {
                                Instrução_1D;
                                Instrução_2D;
                                . . .
                                Instrução_nD;
                        }
                        else
                        {
                                Instrução_1;
                                Instrução_2;
                                . . .
                                Instrução_n;
                        }
                }
        }
}
```

5.8.2 Comando de Seleção switch

Uma alternativa para o comando de seleção if é o comando switch. O comando switch também é utilizado para realizar seleções. A sua sintaxe é:

```
switch ( op )
{
        case valorA:
                instrução 1A;
                instrução 2A;
                ...
                instrução nA;
        break;
        case valorB:
                instrução 1B;
                instrução 2B;
                ...
                instrução nB;
        break;
        . . .
        case valorn:
                instrução 1n;
                instrução 2n;
                ...
                instrução nn;
        break;
        default:
                instrução 1;
                instrução 2;
                ...
        instrução n;
}
```

A variável *op* passada para o comando switch só pode ser de dois tipos: ou inteira ou caractere. O comando switch testa o valor de op contra cada um dos valores presentes na instrução case. Caso algum valor seja igual, todas as instruções internas ao case são executadas e o comando break é utilizado para interromper a execução do switch. Se o comando switch não encontrar nenhum case corresponde ao valor da variável que está sendo testada as instruções internas ao comando default são executadas.

5.9 Estruturas de repetição

5.9.1 Comando while

O comando while em C é a tradução para o comando "enquanto" utilizado em Porto-C. Sua sintaxe e utilização são a mesma.

Exemplo 1: algoritmo para imprima os valores de 1 até 10 no vídeo.

```
#include <stdio.h>
void main()
{
        int cont;
        cont = 1;
        while( cont <= 10 )
        {
                printf("%d", cont);
                cont  = cont + 1;
        }
}
```

Exemplo 2: Algoritmo para calcular a média dos números pares entre 0 e 100.

```
#include <stdio.h>
void main()
{
        int cont, soma;
        cont = 1;
        soma = 0;
        while(cont <= 100)
        {
        if(cont % 2 == 0 )
                        soma = soma + cont;
                cont = cont + 1;
        }
        printf("A soma dos números pares é %d", soma);
}
```

Exemplo 3: Calcular e imprima a média dos 50 alunos de uma sala de aluno. Suponha que cada aluno fez duas provas durante o semestre.

```
#include <stdio.h>
void main()
{
        int cont = 1;
        float p1, p2, media;
        while(cont <= 50)
        {
        printf("Digite a primeira nota: ");
        scanf("%f", &p1);
        printf("Digite a segunda nota: ");
        scanf("%f", &p2);
        media = (p1+p2)/2;
        printf("A media do aluno %d é %f", cont, media);
                cont = cont + 1;
```

```
	}
}
```

5.9.2 Comando do - while

O comando do - while em C é a tradução para o comando “faça - enquanto” utilizado em Porto-C. Sua sintaxe e utilização são a mesma.

Exemplo 4: Refazendo o exemplo 1 usando a estrutura do - while.

```
#include <stdio.h>
void main()
{
	int cont;
	cont = 1;
	do
	{
		printf(“%d”, cont);
		cont  = cont + 1;
	} while( cont <= 10 );
}
```

Exemplo 5: Refazendo o exemplo 2 usando a estrutura do - while

```
#include <stdio.h>
void main()
{
	int cont, soma;
	cont = 1;
	soma = 0;
	do
	{
	if(cont % 2 == 0 )
			soma = soma + cont;
		cont = cont + 1;
	} while(cont <= 100);
	printf(“A soma dos números pares é %d”, soma);
}
```

Exemplo 6: Refazendo o exemplo 3 usando a estrutura do - while

```
#include <stdio.h>
void main()
{
	int cont = 1;
	float p1, p2, media;
	do
	{
```

```
        printf("Digite a primeira nota: ");
        scanf("%f", &p1);
        printf("Digite a segunda nota: ");
        scanf("%f", &p2);
        media = (p1+p2)/2;
        printf("A media do aluno %d é %f", cont, media);
                cont = cont + 1;
        } while(cont <= 50);
}
```

5.9.3 Comando for

O comando for em C é a tradução para o comando "para" utilizado em Porto-C. Sua sintaxe e utilização são a mesma. Exemplos:

Exemplo 7: Refazendo o exemplo 1 usando a estrutura de repetição for.

```
#include <stdio.h>
void main()
{
        int cont;
        for( cont = 1; cont <= 10; cont = cont + 1)
                printf("%d", cont);
}
```

Exemplo 8

```
#include <stdio.h>
void main()
{
        int cont;
        cont = 1;
        for(; cont <= 10; cont = cont + 1)
                printf("%d", cont);
}
```

Exemplo 9

```
#include <stdio.h>
void main()
{
        int cont;
        for( cont = 1; cont <= 10;)
        {
                printf("%d", cont);
                cont = cont + 1;
        }
}
```

Exemplo 10

```
#include <stdio.h>
void main()
{
	int cont;
	cont = 1;
	for( ; cont <= 10;)
	{
		printf("%d", cont);
		cont = cont + 1;
	}
}
```

Capítulo 6

Arrays Unidimensionais (Vetores)

6.1 Introdução

Nos capítulos 3 e 4 foram apresentadas as estruturas de seleção e de repetição que são de extrema importância no desenvolvimento de algoritmos. Nas técnicas de construção de algoritmos já comentadas em nenhum momento houve a necessidade de armazenar grandes quantidades de informação.

Considere a seguinte situação prática. Um professor tem um total de 100 alunos. O professor gostaria de armazenar para processamento as duas notas referentes as provas aplicadas e também calcular e armazenar a média de cada aluno.

Do ponto de vista prático sabe-se que a área de armazenamento utilizada por um programa é a memória RAM e que um programa armazena dados na memória através da declaração de variáveis que é realizada logo no inicio do programa. No problema descrito, qual seria inicialmente o número de variáveis necessárias para armazenar todas as informações? Para quem pensou em 150 variáveis acerto! Declarar 150 variáveis parece algo muito complicado e também nada prático.

As linguagens de programação oferecem um recurso muito interessante e importante chamado de arrays. Os arrays são utilizados quando há a necessidade de manipular e conseqüentemente armazenar grandes quantidades de informação.

6.2 Arrays unidimensionais

Um array é definido formalmente como um conjunto de variáveis, ou seja, um conjunto de posições de memória contíguas que são referenciadas pelo mesmo nome. A sintaxe para a declaração de um array unidimensional é:

tipo nome_variável[*tamanho*]

onde: *tipo* representa qualquer tipo válido da linguagem C, *nome_variável* é o nome que será utilizado para referenciar os conteúdos das posições de memória e *tamanho* representa o número máximo de elementos que podem ser armazenados na variável. Exemplo de declarações:

- int y[50];
- float media[1000];

Na primeira declaração tem-se uma variável *x* do tipo inteiro armazenando no máximo 50 valores inteiros e na segunda declaração tem-se uma variável chamada *media* com capacidade para 1000 valores do tipo float.
Graficamente pode-se visualizar um array da seguinte forma:

y =	10	2	4	87	23	. . .		50	90	28

Já vimos como declarar um array e também como um array pode ser visualizado graficamente. Mas como podemos acessar os elementos individuais de um array se todos são representados pela mesma variável?

6.3 Acessando elementos de um array unidimensional

Como todos os elementos são referenciados pelo mesmo nome deve haver uma maneira de diferenciar o primeiro valor do segundo, do terceiro e assim por diante. O acesso a cada elemento do array é feito através de sua posição dentro da variável. Essa posição, chamada de índice do array, deve ser especificada através do uso de colchetes. Por exemplo:

- y[0] → 10
- y[1] →2
- y[3] → 4

Como já foi mencionado, as posições de memória ocupadas por um array são contíguas, ou seja, uma ao lado da outra. O índice do primeiro elemento do array sempre começará em zero e o último índice dependerá do tamanho total do array, especificado no momento da declaração da variável. Por exemplo, ao escrever:

int z[10]

está sendo declarado um array com 10 elementos onde o primeiro elemento está na posição z[0] e o último está na posição z[9]. Para atribuir um elemento a um array deve-se especificar o nome do array juntamente com a posição que irá armazenar o valor. Por exemplo:

z[0] = 2;
z[1] = 45;
...

Na prática a utilização de arrays sempre está associada com o uso de estruturas de repetição. Suponha um array com 1000 posições que deverá armazenar as médias de um aluno. Para armazenar os valores, deve-se executar 1000 atribuições. Com o uso de estruturas de repetição fica mais fácil realizar as operações.

O espaço total em bytes ocupado por um array dependerá do tipo do array e também do número total de posições. Com essas informações consegue-se determinar o espaço total através da expressão:

espaço = sizeof(tipo) * tamanho

onde, *sizeof*() é uma função da linguagem C utilizada para determinar o número de bytes ocupados por um *tipo* de dado da linguagem e *tamanho* representa o número de posições definidas para o array.

6.4 Inicialização de arrays

Um array como qualquer outra variável pode ser declarado e também inicializado. Como já visto, na inicialização das variáveis devemos especificar quais valores a variável irá receber. Exemplo:

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

Os dois exemplos acima produzem o mesmo efeito. Ao declarar a variável y, o compilador já atribuirá cada valor especificado entre chaves ({}) para cada uma das posições do array, ou seja, y[0] armazenará o valor 1 e y[4] o valor 5.

6.5 Exemplos de manipulação de arrays

Exemplo 1: programa em C para preencher um array unidimensional com 100 posições com valores informados por um usuário.

```
#include <stdio.h
void main()
{
        int x[100], cont;
        for( cont = 0; cont < 100; cont++ )
        {
                printf("Digite um valor: ");
                scanf("%d", &x[cont] );
        }
}
```

Exemplo 2: programa em C para imprima os valores do vetor do exemplo 1.

```
#include <stdio.h>
void main()
{
        int x[100], cont;
        clrscr();
        for( cont = 0; cont < 100; cont++ )
                printf("Posição %d = %d", cont, x[cont]);
}
```

6.6 Exercícios em Classe

1. Considere o vetor V igual a:

V =

2	6	8	3	10	9	1	21	33	14

e as variáveis x = 2 e y = 4, escreva o valor de cada uma das expressões.
a) V[x+1]

b) V[x+2]
c) V[x+3]
d) V[x*4]
e) V[V[x+y]]
f) V[8-V[2]]

2. Escreva um programa em C que preencha um vetor de 50 posições com números inteiros pares sucessivos começando em 10.

3. Escreva um programa em C que preencha um vetor de 1000 posições com valores fornecidos pelo usuário. Imprima no vídeo o maior valor armazenado.

4. Escreva um programa em C que preencha um vetor de 5000 posições com valores aleatórios. Imprima no vídeo a quantidade de números pares gerados.

5. Escreva um programa em C que preencha um vetor de 500 posições com valores aleatórios entre 1000 e 4000. Imprima o menor valor armazenado.

6. Escreva um programa em C para preencher um vetor de 100 posições com valores aleatórios pares.

7. Em uma sala de aula tem-se um total de 75 alunos. Cada aluno fez duas provas durante o semestre. Escreva um programa em C que armazene o valor das duas notas e também calcule e armazene o valor da média aritmética de cada aluno. Imprima no vídeo em formato tabular as duas notas de cada aluno, sua média e a situação ("aprovado" ou "reprovado"). Para ser aprovado a média do aluno deverá ser maior ou igual a 6.

8. Considere um vetor A de 10 posições. Preencha o vetor com valores aleatórios e em seguida faça a inversão dos valores, ou seja: A[0] = A[9], A[1] = A[8] e assim por diante. Imprima o vetor antes e após a inversão.

9. Escreva um programa em C para armazenar em um vetor os 15 primeiros termos da série numérica de Fibonacci. Os dois primeiros valores da série são 0 e 1. Imprima os valores no vídeo.

10. Considere a seguinte série numérica:

$$k = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \cdots$$

Escreva um programa em C para armazenar os 20 primeiros termos da série em um vetor. Cada termo da série deverá ser armazenado em uma posição do vetor.

11. Dado um vetor H de 10 posições. Preencha o vetor com valores aleatórios entre 0 e 50 e em seguida ordene os valores do vetor em ordem crescente. Imprima os elementos do vetor antes e após a ordenação.

6.7 Exercícios Complementares

1. Escreva um programa em C que preencha um vetor de 3000 posições com valores aleatórios. Imprima no vídeo o menor valor armazenado e também a média aritmética dos valores. A média deverá ser impressa com 4 casas decimais.

2. Escreva um programa em C que preencha um vetor de 500 posições com valores aleatórios. Imprima a quantidade de números ímpares.

3. Escreva um programa em C que preencha um vetor de 20 posições com valores aleatórios entre 100 e 200. Imprima o maior valor armazenado e também todos os seus divisores.

4. Altere o programa do exercício anterior para imprima também a diferença entre o maior e o menor elemento do vetor.

5. Escreva um programa em C que crie um algoritmo que leia um vetor de 30 números inteiros e gere um segundo vetor cujas posições pares são o dobro do vetor original e as ímpares o triplo.

6. Escreva um programa em C que armazene 50 notas e calcule quantas são 10% acima da média e quantas são 10% abaixo da média. Considere que a média seja 6.

7. Escreva um programa em C que leia a nota da primeira e da segunda prova de um aluno. Calcule sua média levando em consideração que a prova 1 tem peso 4 e a segunda tem peso 6. Imprima as notas e sua respectiva média.

8. Escreva um programa em C que leia dois vetores A e B, contendo cada um 25 elementos inteiros. Intercale esses dois conjuntos (A[0] | B[0] | A[1] | B[1] | ...) formando um vetor V de 50 elementos.

9. Fazer um algoritmo que leia a matrícula e a média de 10 alunos. Ordene da maior nota para a menor e imprima uma relação contendo todas as matrículas e médias.

10. Escreva um programa em C que preencha um vetor de 20 posições com valores aleatórios. Em seguida, imprima as seguintes informações: i) quantos números pares foram armazenados? ii) quantos números ímpares? iii) qual a média aritmética do vetor? iv) qual a porcentagem de números acima da média?

Capítulo 7

Manipulação de Strings e Caracteres

7.1 Introdução

Como já comentado anteriormente em nosso curso de Algoritmos, a linguagem C não apresenta um tipo "string". Sabemos também desde o início do nosso curso que uma string nada mais é do que uma cadeia (conjunto) de caracteres. Dessa forma temos:

"Maria"
"São Paulo"
"Faculdade de Informática"

Para manipularmos uma cadeia de caracteres em C devemos trabalhar com vetores unidimensionais. A utilização mais comum de um array unidimensional em C é na definição de uma string. Portanto, strings são vetores de do tipo *char*. Devemos estar atentos ao fato de que as strings têm como último elemento um '\0', que é adicionado automaticamente ao final da string.

A forma geral para declararmos uma string segue o mesmo padrão para a declaração de um array unidimensional:

char nome_da_string[*tamanho*];

sendo *tamanho* representa o número máximo de caracteres que a string irá armazenar. Devemos incluir neste valor o finalizador de strings.

Exemplo de declaração:

char nome[20];

No exemplo acima, temos uma variável vetor que armazena no máximo 20 caracteres. Lembre-se que o tamanho da string deve incluir o finalizador de strings ('\0').

As variáveis strings podem ser inicializadas no momento da declaração da mesma forma que as variáveis de outros tipos. Não se esqueça que uma string sempre vem entre aspas (" "). Exemplo:

char x[] = "Testando o programa";
char y[] = {'T','e','s','t','e',' ','d','e',' ','c','o','n','h','e','c','i','m','e','n','t','o','\0'};
char t[] = {84, 101, 115, 116, 101, '\0'};

Nos exemplos acima, percebemos três maneiras diferentes de inicializar uma variável string. Quando fornecemos valores decimais, estes são convertidos em caracteres conforme tabela ASCII (veja no final do capítulo).

7.2 Funções para manipulação de strings

A linguagem C apesar de não apresentar um tipo específico "string", apresenta uma série de funções que podem ser aplicadas a uma variável do tipo vetor de caracteres. Essas funções apresentam seus cabeçalhos em **string.h**. Dentre as principais funções destacam-se:

Função gets()

A função gets() é utilizada para leia uma string do teclado. A função scanf() também pode ser utilizada, mas não apresenta um bom desempenho quando a string é formada por duas ou mais palavras separadas por caracteres em branco. Dessa forma, a função gets() apresenta um melhor resultado. O cabeçalho da função gets() está em **stdio.h**. Sua forma geral é:

gets(*nome_da_string*);

Exemplo de utilização da função gets():

```
#include <stdio.h>
int main()
{
       char string[100];
       printf (“Digite o seu nome: “);
       gets (string);
       printf (“\n\n Ola %s”,string);
       return(0);
}
```

A impressão de uma string no vídeo é realizada com a função printf(). Repare que o caractere de impressão utilizado para string é %s.

Função strcpy()

Esta função é utilizada para copiar uma string para outra. A sua forma geral é:

strcpy(*string_destino*, *string_origem*);

A *string_origem* é copiada na *string_destino*. Exemplo:

```
#include <stdio.h>
#include <string.h>
int main()
{
       char str1[100], str2[100], str3[100];
       printf (“Entre com uma string: “);
       gets (str1);
       strcpy (str2, str1);      /* Copia str1 em str2 */
       strcpy (str3, “Voce digitou a string “);
```

```
        printf ("\n\n%s%s",str3,str2);
        return(0);
}
```

No programa exemplo acima, a string informada via teclado é armazenada na variável *str1*. Em seguida, o conteúdo de *str1* é copiado para a variável *str2* usando a função **strcpy()**. Perceba que neste caso teremos duas variáveis com o mesmo conteúdo. Caso a variável *str2* tenha um conteúdo inicial, este será substituído.

Função strcat()

Esta função é utilizada para concatenar duas strings, ou seja, o conteúdo de uma string é adicionado ao final de outra string. A forma geral da função é:

strcat(*string_destino*, *string_origem*);

A string de origem permanecerá inalterada e será anexada ao fim da string de destino. Exemplo:

```
#include <stdio.h>
#include <string.h>
int main()
{
        char str1[100], str2[100];
        printf ("Entre com uma string: ");
        gets (str1);
        strcpy (str2,"Voce digitou a string ");
        strcat (str2, str1);
        printf ("\n\n%s",str2);
        return(0);
}
```

Função strlen()

Uma função bastante útil na manipulação de strings é a função **strlen()** utilizada para retornar o comprimento da string informada. No comprimento retornado não é incluído o finalizador de strings. A forma geral para a função é:

strlen(*nome_da_string*);

A função **strlen()** retorna um valor do tipo inteiro. Exemplo:

```
#include <stdio.h>
#include <string.h>
int main()
{
        int tamanho;
        char str[100];
        printf( "Entre com uma string: " );
```

```
        gets( str );
        tamanho = strlen( str );
        printf( "\n\nTamanho da string digitada %d", tamanho );
        return( 0 );
}
```

Função strcmp()

Esta função é utilizada para comparar duas strings. O retorno desta função é um valor inteiro. A forma geral da função **strcmp()** é:

strcmp(*string1*, *string2*);

Caso as duas strings sejam idênticas a função retorna o valor zero. Se o valor de retorno for negativo significa que *string1* é menor que *string2*, e se o valor for positivo, *string2* é menor que *string1*. Exemplo:

```
#include <stdio.h>
#include <string.h>
int main()
{
        char str1[100], str2[100];
        printf( "Entre com uma string: " );
        gets( str1 );
        printf( "\n\nEntre com outra string: " );
        gets( str2 );
        if( strcmp(str1,str2) )
                printf( "\n\nAs duas strings são diferentes." );
        else
                printf( "\n\nAs duas strings são iguais." );

        return( 0 );
}
```

7.3 Exercícios em Classe

1. Escreva um programa em C que leia uma string (no máximo 50 caracteres) via teclado e informe a quantidade de letras "a" presentes na string (podendo ser maiúscula ou minúscula).

2. Escreva um programa em C que leia uma string (no máximo 30 caracteres) via teclado e a imprima invertido.

3. Escreva um programa em C que conte o número de vogais presentes em uma string informada via teclado por um usuário.

4. Usando a tabela de códigos ASCII, responda o que está escrito na propaganda abaixo:

    char msg[] = {69,83,84,65,77,79,83,32,67,79,78,84,82,65,84,65,78,68,79,'\0'};

Tabela ASCII – American Standard Code for Interchange Information

Char	*Dec*	*Char*	*Dec*	*Char*	*Dec*	*Char*	*Dec*
NUL ^@	0	**SPACE**	32	**@**	64	**‘**	96
^A	1	**!**	33	**A**	65	**a**	97
^B	2	**“**	34	**B**	66	**b**	98
^C	3	**#**	35	**C**	67	**c**	99
^D	4	**$**	36	**D**	68	**d**	100
^F	5	**%**	37	**E**	69	**e**	101
^F	6	**&**	38	**F**	70	**f**	102
^G	7	**‘**	39	**G**	71	**g**	103
^H	8	**(**	40	**H**	72	**h**	104
TAB ^I	9	**)**	41	**I**	73	**i**	105
^J	10	*****	42	**J**	74	**j**	106
^K	11	**+**	43	**K**	75	**k**	107
^L	12	**,**	44	**L**	76	**l**	108
^M	13	**-**	45	**M**	77	**m**	109
^N	14	**.**	46	**N**	78	**n**	110
^O	15	**/**	47	**O**	79	**o**	111
^P	16	**0**	48	**P**	80	**p**	112
^Q	17	**1**	49	**Q**	81	**q**	113
^R	18	**2**	50	**R**	82	**r**	114
^S	19	**3**	51	**S**	83	**s**	115
^T	20	**4**	52	**T**	84	**t**	116
^U	21	**5**	53	**U**	85	**u**	117
^V	22	**6**	54	**V**	86	**v**	118
^W	23	**7**	55	**W**	87	**w**	119
^X	24	**8**	56	**X**	88	**x**	120
^Y	25	**9**	57	**Y**	89	**y**	121
^Z	26	**:**	58	**Z**	90	**z**	122
^[	27	**;**	59	**[**	91	**{**	123
^|	28	**<**	60	****	92	**|**	124
^]	29	**=**	61	**]**	93	**}**	125
^^	30	**>**	62	**^**	94	**~**	126
^_	31	**?**	63	**_**	95	**del**	127

5. Faça um programa em C que leia uma palavra pelo teclado e faça a impressão conforme o exemplo a seguir para a palavra AMOR

AMOR
AMO
AM
A

6. Faça um programa em C que leia uma palavra pelo teclado e faça a impressão conforme o exemplo a seguir para a palavra AMOR

A
AM
AMO
AMOR

7. Escreva um programa em C que leia uma string via teclado. Converta os caracteres para minúsculo e informe a quantidade de caracteres que tem código ASCII par.

8. Escreva um programa em C que solicite o nome do usuário do sistema. O seu programa irá gerar uma senha que é formada apenas pelos caracteres das posições ímpares mais a soma do código ASCII dos caracteres nas posições pares.

7.4 Exercícios Complementares

1. Faça um programa em C que leia quatro strings pelo teclado. Depois, concatene todas as strings lidas em única e imprima-a no vídeo.

2. Faça um programa em C que leia uma palavra pelo teclado e informe a quantidade de caracteres que não são vogais.

3. Escreva um programa em C que leia uma palavra fornecida pelo teclado e em seguida imprima o caractere presente no meio da palavra, caso esta tenha um número ímpar de caracteres. Como exemplo, considere a palavra SONHO. O caractere a ser impresso será o N.

4. Uma string é considerada um palíndromo se quando lida da esquerda para a direita e da direita para a esquerda apresentam o mesmo valor. Por exemplo: ANA. Escreva um programa em C em que o usuário possa digitar várias palavras e informe se são palíndromos ou não. A cada palavra informada pergunte ao usuário se ele deseja continuar a execução do programa.

5. Escreva um programa em C que leia uma string (no máximo 100 caracteres) via teclado e informe a quantidade de palavras presentes na string.

Capítulo 8

Arrays Bidimensionais (Matrizes)

8.1 Introdução

Até agora, trabalhos com arrays unidimensionais (vetores). Um array bidimensional é formado por linhas e colunas, ou seja, temos uma tabela de elementos dispostos em linhas e colunas. Podemos também afirmar que um array bidimensional é basicamente um array de arrays unidimensionais.

Um array bidimensional pode ser representado pela figura abaixo:

colunas

linhas

2	10	5	0
1	256	33	85
9	66	65	63
69	63	58	70

8.2 Declarando um array bidimensional

Para declararmos um array bidimensional devemos especificar o número de linhas e o número de colunas presentes. A forma geral para declaração é:

tipo nome_do_array[*número_de_linhas*][*número_de_colunas*];

tipo pode ser qualquer tipo válido em C (int, float, double, char). O primeiro índice presente representa o número total de linhas presentes no array e o segundo índice o número total de colunas. Exemplos de declaração:

```
int x[5][5];
float y[2][3];
```

O primeiro exemplo declara um array *x* com 5 linhas e 5 colunas e, no segundo exemplo temos um array *y* com 2 linhas e 3 colunas. Tanto o índice da linha quanto da coluna começam em 0.

Para acessarmos um elemento específico em um array bidimensional devemos informar o índice da linha e o da coluna. Por exemplo: x[2][1], retorna o elemento armazenado no array *x*, na linha 2 e coluna 1. Por exemplo, considere o array *x* abaixo:

2	10	5	0
1	256	33	85
9	66	65	63
69	63	58	70

x[0][0] = 2	x[1][0] = 1	x[2][0] = 9	x[3][0] = 69
x[0][1] = 10	x[1][1] = 256	x[2][1] = 66	x[3][1] = 63
x[0][2] = 5	x[1][2] = 33	x[2][2] = 65	x[3][2] = 58
x[0][3] = 0	x[1][3] = 85	x[2][3] = 63	x[3][3] = 70

Como podemos perceber no exemplo acima, para acessar os elementos de uma determinada linha, devemos fixar o índice que representa a linha e variar o índice que representa a coluna. Da mesma forma, se quisermos acessar os elementos de uma coluna, fixamos o índice da coluna e variamos o índice da linha. Em um programa que trabalha com um array bidimensional que processa todos os elementos, usamos duas estruturas de repetição: uma para controlar o índice que representa a linha e a outra para controlar o índice que representa a coluna.

O número de bytes de memória requerido por um array bidimensional é computado utilizando a seguinte fórmula:

*bytes = número_de_linhas * número_de_colunas * tamanho_do_tipo*

Por exemplo, um array com 2 linhas e 2 colunas do tipo inteiro ocuparia 8 bytes na memória, supondo que os números inteiros ocupam apenas dois bytes (2 * 2 * 2).

Exemplo 1: o programa abaixo declara e preenche um array bidimensional com valores aleatórios entre 0 e 20.

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
	int m[2][2], l, c;
	for( l = 0; l < 2; l++ )
		for( c = 0; c < 2; c++ )
			m[l][c] = rand()%20;

	return( 0 );
}
```

Exemplo 2: Neste exemplo, o array que foi preenchido no exemplo 1 é impresso no vídeo.

```
#include <stdio.h>
int main()
{
	int m[2][2], l, c;
	for( l = 0; l < 2; l++ )
```

```
        {
                for( c = 0; c < 2; c++ )
                        printf( "%d\t", m[l][c] );

                printf("\n");
        }

}
```

No primeiro exemplo, o array *m* é preenchido com valores aleatórios gerados pela função **rand**(). Podemos também preencher o array com valores informados via teclado. Veja o exemplo abaixo:

```
#include <stdio.h>
int main()
{
        int m[2][2], l, c;
        for( l = 0; l < 2; l++ )
                for( c = 0; c < 2; c++ )
                {
                        printf("Digite um valor: ");
                        scanf("%d", &m[l][c]);
                }

        return( 0 );
}
```

Outro ponto já comentado nas aulas de vetores é que a linguagem de programação C não faz a verificação dos limites do array. Isso é tarefa do programador!

8.3 Exercícios em Classe

1. Considere uma matriz B, quadrada de ordem 3 inicialmente vazia. O programa abaixo irá preenchê-la. Após a execução do programa, desenhe a matriz com os elementos.

```
#include <stdio.h>
int main()
{
        int i, j, B[3][3];
        for( j = 0; j < 3; j++ )
                for( i = 0; i < 3; i++ )
                        B[i][j] = 2*j+i;

        return( 0 );
}
```

2. Dada a matriz: $A = \begin{bmatrix} 3 & 1 & 5 \\ 2 & 4 & 6 \\ -2 & 0 & 7 \end{bmatrix}$. Considere o programa abaixo que manipula a matriz A.

```
#include <stdio.h>
int main()
{
       int i, j, A[3][3];
       for( i = 0; i < 3; i++ )
              for( j = 0; j < 3; j++ )
                     if( A[i][j] % 2 == 0 )
                            A[i][j] = A[i][j] * 3;

       return( 0 );
}
```

Desenhe a nova matriz resultante após a execução do programa acima.

3. Escreva um programa em C que preencha uma matriz quadrada de ordem 4 com valores inteiros fornecidos pelo usuário. Mostre a soma de todos os elementos da matriz.

4. Para a matriz do exercício anterior, calcule e imprima a soma dos elementos da diagonal principal.

5. Para a matriz do exercício 3, calcule e imprima a soma dos elementos da diagonal secundária.

6. Dadas as matrizes A_{3x3} e B_{3x3} preenchidas com valores aleatórios entre 0 e 20, gere a matriz $C = A + B$.

7. Escreva um programa em C que preencha uma matriz D_{10x10} com valores aleatórios entre 0 e 1000. Imprima o maior valor armazenado bem como a sua localização.

8. Faça um programa em C que entre com valores reais para uma matriz C_{2x3}. Gere e imprima a C^t. A matriz transposta é gerada trocando linha por coluna. Veja o exemplo a seguir:

$$C = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \quad \Longrightarrow \quad C^t = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

9. Dadas as matrizes A_{3x2} e B_{2x3}, gere $C = A * B$.

10. Escreva um programa em C que leia e armazene 50 nomes. Mostre a quantidade de nomes que começam com a letra “a”.

11. Para o exercício 10, imprima apenas os nomes que começam com a letra “c”.

12. Para a matriz do exercício 10, imprima o maior nome armazenado (aquele que apresenta o maior número de caracteres).

13. Escreva um programa em C que leia o nome de 10 alunos de uma turma. Leia também a nota da primeira e da segunda prova. Calcule a média aritmética e imprima todas as informações no vídeo.

8.4 Exercícios Complementares

10. O que o programa abaixo imprime?

```
#include <stdio.h>
int main()
{
    int t, i, M[3][4];

    for (t=0; t<3; ++t)
        for (i=0; i<4; ++i)
            M[t][i] = (t*4)+i+1;

    for (t=0; t<3; ++t)
    {
        for (i=0; i<4; ++i)
            printf ("%d ", M[t][i]);
        printf ("\n");
    }
    return(0);
}
```

11. Escreva um programa em C para gerar a seguinte matriz:

$$t = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 6 & 8 \\ 3 & 6 & 9 & 12 \\ 4 & 8 & 12 & 16 \end{bmatrix}$$

12. Faça um programa em C que preencha uma matriz quadrada de ordem 5 com valores aleatórios entre 0 e 50. Em seguida, imprima todos os elementos que estão acima da diagonal secundária bem como a sua soma.

13. Escreva um programa em C que preencha uma matriz quadrada de ordem 10 com valores aleatórios entre 0 e 1000. Imprima todos os elementos da diagonal principal e da secundária.

14. Dadas as matrizes X_{2x2} e Y_{2x2} preenchidas com valores inteiros fornecidos pelo usuário, gere a matriz $Z = X - Y$.

15. Escreva um programa em C que entre com valores para uma matriz A_{3x4}. Gere e imprima a matriz B que é o triplo da matriz A.

16. Escreva um programa em C que preencha uma matriz C_{4x5} com valores reais informados pelo usuário. Calcular e imprima a soma de todos os seus elementos.

17. Dada uma matriz B (quadrada de ordem 10) de números inteiros, determine a quantidade de números ímpares e pares armazenados.

18. Faça um programa em C que leia uma matriz 12 x 4 com valores das vendas de uma loja, em que cada linha representa um mês do ano, e cada coluna, uma semana do mês. Calcule e imprima:
a) total vendido em cada mês;
b) total vendido no ano;

19. Escreva um programa em C que preencha uma matriz quadrada de ordem 5 com valores inteiros fornecidos pelo usuário. Verifique se a matriz é ou não uma matriz triangular superior. Matriz triangular superior é uma matriz onde todos os elementos de posição L < C são diferentes de 0 e todos os elementos de L > C são iguais a 0. L e C representam respectivamente linha e coluna.

20. Considere a seguinte matriz $B[3][2][3]$. Preencha a matriz com valores aleatórios entre 0 e 50. Em seguida calcule e imprima a soma dos elementos.

21. Criar um programa em C que possa armazenar as alturas de dez atletas de cinco delegações que participarão dos jogos de verão. Imprima a maior altura de cada delegação.

22. Escreva um programa em C que leia 10 nomes via teclado. Os nomes deverão ser armazenados em ordem crescente de tamanho (em relação ao número de caracteres presentes na string).

Capítulo 9

Introdução aos Ponteiros

9.1 Definição

Os **ints** guardam inteiros. Os **floats** guardam números de ponto flutuante. Os **chars** guardam caracteres. Ponteiros guardam endereços de memória. Quando você anota o endereço de um colega você está criando um ponteiro. O ponteiro é o seu pedaço de papel. Ele tem anotado um endereço. Qual é o sentido disto? Simples. Quando você anota o endereço de um colega, depois você vai usar este endereço para achá-lo. A linguagem C funciona assim. Você anota o endereço de algo numa variável ponteiro para depois usar.

Da mesma maneira, uma agenda, onde são guardados endereços de vários amigos, poderia ser vista como sendo uma matriz de ponteiros na linguagem C.

Um ponteiro também tem tipo. Veja: quando você anota um endereço de um amigo você o trata diferente de quando você anota o endereço de uma firma. Apesar de o endereço dos dois locais ter o mesmo formato (rua, número, bairro, cidade, etc.) eles indicam locais cujos conteúdos são diferentes. Então os dois endereços são ponteiros de *tipos* diferentes.

Na linguagem C quando declaramos ponteiros nós informamos ao compilador para que tipo de variável vamos apontá-lo. Um ponteiro **int** aponta para um inteiro, isto é, guarda o endereço de um inteiro.

9.2 Declaração de ponteiros

Para declarar um ponteiro temos a seguinte forma geral:

*tipo_do_ponteiro *nome_da_variável;*

É o asterisco (*) que faz o compilador saber que uma variável não vai guardar um valor e sim um endereço para aquele tipo especificado. Exemplos de declarações:

```
int *pt;
char *temp,*pt2;
```

O primeiro exemplo declara um ponteiro para um inteiro. O segundo declara dois ponteiros para caracteres. Eles ainda não foram inicializados (como toda variável do C que é apenas declarada). Isto significa que eles apontam para um lugar indefinido. Este lugar pode estar, por exemplo, na porção da memória reservada ao sistema operacional do computador. Usar o ponteiro nestas

circunstâncias pode levar a um travamento do micro, ou a algo pior. *O ponteiro deve ser inicializado (apontado para algum lugar conhecido) antes de ser usado!* Isto é de suma importância!

9.3 Operadores de ponteiros (& e *)

Para atribuir um valor a um ponteiro recém-criado poderíamos igualá-lo a um valor de memória. Mas, como saber a posição na memória de uma variável do nosso programa? Seria muito difícil saber o endereço de cada variável que usamos, mesmo porque estes endereços são determinados pelo compilador na hora da compilação e realocados na execução. Podemos então deixar que o compilador faça este trabalho por nós. Para saber o endereço de uma variável basta usar o operador **&**. Veja o exemplo:

```
int count = 10;
int *pt;
pt = &count;
```

Criamos um inteiro **count** com o valor 10 e um ponteiro para um inteiro **pt**. A expressão **&count** nos dá o endereço de **count**, o qual armazenamos em **pt**. Simples, não é? Repare que *não* alteramos o valor de **count**, que continua valendo 10.

Como nós colocamos um endereço em **pt**, ele está agora "liberado" para ser usado. Podemos, por exemplo, alterar o valor de **count** usando **pt**. Para tanto vamos usar o operador "inverso" do operador **&**. É o operador *. No exemplo acima, uma vez que fizemos **pt = &count** a expressão ***pt** é equivalente ao próprio **count**. Isto significa que, se quisermos mudar o valor de **count** para 12, basta fazer ***pt=12**.

Vamos fazer uma pausa e voltar à nossa analogia para ver o que está acontecendo.

Digamos que exista uma firma. Ela é como uma variável que já foi declarada. Você tem um papel em branco onde vai anotar o endereço da firma. O papel é um ponteiro do tipo firma. Você então liga para a firma e pede o seu endereço, o qual você vai anotar no papel. Isto é equivalente, no C, a associar o papel à firma com o operador **&**. Ou seja, o operador **&** aplicado à firma é equivalente a você ligar para a mesma e pedir o endereço. Uma vez de posse do endereço no papel você poderia, por exemplo, fazer uma visita à firma. No C você faz uma visita à firma aplicando o operador * ao papel. Uma vez dentro da firma você pode copiar seu conteúdo ou modificá-lo.

Uma observação importante: apesar do símbolo ser o mesmo, o operador * (multiplicação) não é o mesmo operador que o * (referência de ponteiros). Para começar o primeiro é binário, e o segundo é unário.

Aqui vão dois exemplos de usos simples de ponteiros:

```
#include <stdio.h>
int main()
{
       int num, valor;
```

```
       int *p;
       num = 55;
       p = &num;
       valor = *p;  //atribuição de forma indireta
       printf("\n\n%d\n",valor);
       printf("Endereço para onde o ponteiro aponta: %p\n",p);
       printf("Valor da variável apontada: %d\n",*p);
       return(0);
}
```

```
#include <stdio.h>
int main()
{
       int num,*p;
       num = 55;
       p = &num;    /* Pega o endereço de num */
       printf("\nValor inicial: %d\n",num);
       *p = 100;   /*Muda o valor de num de uma maneira indireta */
       printf("\nValor final: %d\n",num);
       return(0);
}
```

Nos exemplos acima vemos um primeiro exemplo do funcionamento dos ponteiros. No primeiro exemplo, o código **%p** usado na função **printf()** indica à função que ela deve imprima um endereço.

Podemos fazer algumas operações aritméticas com ponteiros. A primeira, e mais simples, é igualar dois ponteiros. Se temos dois ponteiros p1 e p2 podemos igualá-los fazendo p1=p2. Repare que estamos fazendo com que p1 aponte para o mesmo lugar que p2. Se quisermos que a variável apontada por p1 tenha o mesmo conteúdo da variável apontada por p2 devemos fazer *p1=*p2. Basicamente, depois que se aprende a usar os dois operadores (& e *) fica fácil entender operações com ponteiros.

9.4 Cuidados com ponteiros

O principal cuidado ao se usar um ponteiro deve ser: saiba sempre *para onde* o ponteiro está apontando. Isto inclui: nunca use um ponteiro que não foi inicializado. Um pequeno programa que demonstra como ***não*** usar um ponteiro:

```
#include <stdio.h>
int main() /* Errado - Não Execute */
{
       int x,*p;
       x = 13;
       *p = x;
       return(0);
}
```

Este programa compilará e rodará. O que acontecerá? Ninguém sabe. O ponteiro **p** pode estar apontando para qualquer lugar. Você estará gravando o número 13 em um lugar desconhecido. Com um número apenas, você provavelmente não vai ver nenhum defeito. Agora, se você começar a gravar números em posições aleatórias no seu computador, não vai demorar muito para travar o micro (se não acontecer coisa pior).

9.5 Exercícios em Classe

1. Após a execução do trecho de programa abaixo, o que será impresso?

```
#include <stdio.h>
void main( void )
{
      int x, *y;
      x = 25;
      y = &x;
      *y += 2;
      printf("%d", x );
}
```

2. Considere o trecho de programa abaixo. Quais serão os valores armazenados nas variáveis após sua execução?

```
#include <stdio.h>
void main( void )
{
      int a = 2, b = 3, temp;
      int *pa, *pb;

      pb = &b;
      pa = &a;

      temp = *pb;
      *pb = *pa;
      *pa = temp;
}
```

3. Analise o trecho de programa abaixo e informe o que será impresso após a execução.

```
#include <stdio.h>
void main( void )
{
      float x, y, *z;
      x = 10.0;
      z = y;
      *z = x + 2.5;
      printf("x = %f, y = %f", x, y);
}
```

Capítulo 10

Funções

10.1 Introdução

Até agora os nossos programas foram codificados com uma função principal (main()) onde toda a atividade do programa ocorre. Em algumas situações, utilizamos algumas funções definidas da linguagem C tais como: **rand()**, **pow()**, **sqrt()**. Lembre-se que essas funções foram utilizadas dentro da função principal.

A medida que os nossos programas vão ficando maiores, sua complexidade aumenta e portanto fica também mais difícil fazer manutenções e até mesmo entender a lógica do programa. Dessa forma, as linguagens de programação permitem que os usuários separem seus programas em blocos. A idéia é dividir programas grandes e complexos em blocos para facilitar a sua construção. Esses blocos de construção são chamados em C de **funções**. Portanto, uma função nada mais é do que um bloco de construção que realiza uma atividade específica.

A declaração de uma função em C segue o mesmo padrão adotado para a função principal do programa. A sua forma geral é:

tipo_de_retorno nome_da_função (*declaração_de_parâmetros*)
{
    corpo_da_função;
}

O *tipo_de_retorno* é o tipo de variável que a função vai retornar. O default é o tipo **int**, ou seja, uma função para qual não declaramos o tipo de retorno é considerada como retornando um inteiro. A declaração de parâmetros é feita de forma semelhante a declaração de variáveis que estamos acostumados a fazer em nossos programas. A grande diferença na declaração dos parâmetros é que devemos especificar o tipo para cada variável. Exemplo:

tipo variavel1, tipo variavel2, tipo variavel3, ..., tipo variavelN

Estas variáveis que compõem os parâmetros da função representam as entradas para a função. A saída da função está especifica no tipo de retorno. O *copo_da_função* contém as instruções que irão processar os dados de entrada (parâmetros), gerando as saídas da função.

Para que um valor seja retornado pela função, deve-se utilizar o comando **return**. Quando se chega a uma declaração **return** a função é encerrada imediatamente e, se o valor de retorno é informado, a

função retorna este valor. É importante lembrar que o valor de retorno fornecido tem que ser compatível com o tipo de retorno declarado para a função. A forma geral do comando return é:

```
return valor_de_retorno;
```

Ou

```
return;
```

Abaixo é apresentado um exemplo simples de um programa que calcula a raiz quadrada de um número inteiro, usando uma função definida.

```
#include <stdio.h>

int quadrado( int x )
{
     return x*x;
}

int main()
{
     int y, total;
     printf("Digite um valor: ");
     scanf("%d", &y );
     total = quadrado(y);
     printf("O quadrado de %d é %d", y, total);
     return( 0 );
}
```

No exemplo acima, note que o retorno da função **quadrado()** é atribuído a uma variável do tipo inteiro. A situação abaixo não pode ser realizada, pois causará erros:

```
quadrado(y) = total;
```

10.2 Protótipos de funções

No exemplo anterior onde codificamos uma função para calcular o quadrado de um número, o código da função foi escrito antes da função principal main(). Existe uma explicação para isso. No momento da compilação da função principal, nos locais onde as funções são chamadas, temos que saber o tipo de retorno das funções e os parâmetros das funções. Quando as funções definidas pelo usuário estão antes da função principal, ao compilar a função main() o compilador já compilou as funções e sabe os tipos de parâmetros e retornos das funções.

Caso não seja possível codificar as funções antes da função main(), a linguagem C apresenta o conceito de **protótipos de funções**. Os protótipos nada mais são do que declarações de funções, ou seja, você declara a função que irá usar. Dessa forma, o compilador será informado sobre a função e os tipos de valores que a função recebe e retorna. O formato para um protótipo de função é:

tipo_de_retorno nome_da_função(*lista_de_parâmetros*);

sendo que o *tipo_de_retorno*, o *nome_da_função* e a *lista_de_parâmetros* são os mesmo que você utilizará na codificação da função. O protótipo representa o cabeçalho da função (sem o código). Repare também que o protótipo de uma função termina com ponto e vírgula. Na codificação da função não podemos terminar o cabeçalho com ponto e vírgula.

Como exemplo vamos reescrever o programa que utiliza uma função definida pelo usuário para calcular o quadrado de um número, usando protótipos.

```
#include <stdio.h>

int quadrado( int x );

int main()
{
      int y, total;
      printf("Digite um valor: ");
      scanf("%d", &y );
      total = quadrado(y);
      printf("O quadrado de %d é %d", y, total);
      return( 0 );
}

int quadrado( int x )
{
      return x*x;
}
```

No exemplo acima, a função **quadrado()** foi codificada após a função main(), mas seu protótipo foi declarado antes.

Quando temos uma função que não retorna nenhum tipo de valor ou que não recebe nenhum parâmetro, utilizamos o tipo **void** na especificação. A palavra **void** do inglês significa vazio. Como exemplo, considere a função abaixo que não retorna nenhum valor e também não recebe nenhum parâmetro.

```
#include <stdio.h>

void imprima( void );

int main( main )
{
      imprima();

      return( 0 );
}
```

```
void imprima( void )
{
     printf("Testando!");
}
```

Repare no exemplo acima que só utilizamos o comando **return** quando uma função deve retornar um valor ou ser interrompida (como já foi visto).

10.3 Passagem de Parâmetros

Nos exemplos anteriores, quando chamamos as funções passamos valores paras as mesmas. Isso acontece porque valores declarados em uma função não são acessíveis dentro de outra. Em C temos dois tipos de passagem de parâmetros: **passagem por valor** e **passagem por referência**.

Na passagem por valor como o próprio nome indica, apenas uma "cópia" dos valores são passados para a função. As alterações realizadas pela função não são feitas nos valores originais do parâmetro. Como exemplo podemos citar o programa para o cálculo do quadrado de um número codificado anteriormente. Vamos considerar outro exemplo abaixo:

```
#include <stdio.h>

float somar( float x, float y );

int main( void )
{
     float a, b, total;
     printf("Digite um valor: ");
     scanf("%f", &a );
     printf("Digite um valor: ");
     scanf("%f", &b );
     total = somar( a, b );
     printf("a soma dos números é: %f", total );
     return( 0 );
}

float somar( float x, float y )
{
     float t;
     t = x + y;
     return t;
}
```

No exemplo acima, os valores declarados na função **main()** são passados para a função **somar()**, que realiza a soma e retorna o resultado. Veja que os valores originais na função **main()** não são alterados. Esse tipo de passagem de parâmetro é por valor, e é o padrão da linguagem.

Uma outra maneira de passagem de parâmetros é quando a função que recebe os parâmetros consegue alterar os valores originais na função onde os mesmos foram declarados. Esse tipo de

passagem de parâmetro é conhecido como passagem por referência. Neste caso a referência (endereço) das variáveis são passadas para as funções. Considere o exemplo abaixo:

```
#include <stdio.h>

void troca( float *x, float *y );

int main( void )
{
     float a, b;
     printf("Digite um valor: ");
     scanf("%f", &a );
     printf("Digite um valor: ");
     scanf("%f", &b );
     troca( &a, &b );
     printf("a = %f e b = %f", a, b );
     return( 0 );
}

void troca( float *x, float *y )
{
     float t;
     t = *x;
     *x = *y;
     *y = t;
}
```

O que acontece no exemplo acima é que estamos passando para a função **troca()** os endereços das variáveis *a* e *b*. Estes endereços são armazenados nos ponteiros *x* e *y* da função. Como vimos no capítulo anterior, o operador (*) é utilizado para acessar o conteúdo apontado pelos ponteiros.

Uma utilização muito comum de passagem de parâmetros por referência é quando uma função deve retornar dois ou mais valores. Lembre-se que uma função retorna apenas um valor. Dessa forma, a solução é passar os parâmetros por referência.

10.4 Passagem de arrays para funções

Da mesma forma que passamos variáveis comuns para as funções, também podemos passar arrays. Vamos considerar como exemplo a seguinte declaração para um array unidimensional:

```
int y[200];
```

Considere também que queiramos chamar a função **alterar()** e passar o array *y* como parâmetro. Para passarmos o array para a função, apenas referenciamos o nome da variável. Exemplo:

```
alterar( y );
```

O diferencial está na declaração da variável que irá receber o parâmetro. Essa declaração pode ser feita de duas maneiras diferentes:

```
1. void alterar( int t[200] );
2. void alterar( int *t );
```

Quando passamos um array para uma função em C, a passagem de parâmetros é feita por referência. Nas três situações acima, a função **alterar()** recebe um ponteiro (endereço) do primeiro elemento armazenado no array. Esse tipo de passagem de parâmetro é bem prático para a linguagem. Imagine a situação onde temos um array com uma quantidade grande de elementos, se a passagem dos parâmetros fosse por valor, o compilador teria que fazer uma cópia de todos os valores para a função, o que poderia haver uma perda de desempenho do programa.

A manipulação do array dentro da função é realizada da mesma forma que estávamos manipulando, independentes da forma com que o parâmetro é declarado.

10.5 Escopo de variáveis

As variáveis que utilizamos em nossos programas podem ser declaradas em três lugares diferentes. Dependendo da sua declaração, a abrangência das variáveis muda. Os três tipos são:
⇒ Variáveis locais;
⇒ Variáveis globais;
⇒ Parâmetros formais.

Os nossos primeiros programas em C usavam apenas a função **main()**. Dessa forma, todas as variáveis eram declaradas dentro dessa função. Essas variáveis são denominadas **variáveis locais**, e só tem validade dentro do bloco onde foram declaradas.

As **variáveis globais** são declaradas fora das funções de um programa, podendo ser acessadas e alteradas por todas as funções.

Os **parâmetros formais** se referem as variáveis que são declaradas para receber os parâmetros das funções. Essas variáveis só apresentam abrangência enquanto a função está sendo executada, ou seja, essas variáveis são locais a função.

10.6 Exercícios em Classe

1. Considere o programa em C abaixo

```
#include <stdio.h>

int teste( int a, int b );

int main( void )
{
     int x = 9, y = 5, total;
     total = teste( x, y );
```

```
     return( 0 );
}

int teste( int a, int b )
{
     int y;
     y = a % b;
     return y;
}
```

Qual será o valor armazenado na variável **total** após a execução do programa acima?

2. Analise o programa abaixo e responda:

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

int teste( int a );

int main( void )
{
     int x = 144, total;
     total = teste( x );
     return( 0 );
}

int teste( int a )
{
     int y;
     y = sqrt( a );
     return y;
}
```

a) Qual o valor armazenado na variável **total** após a execução do programa?
b) Quais são as variáveis globais do programa acima?
c) Indique todas as variáveis locais e a quais funções elas pertencem.

3. Escreva um programa em C que contenha uma função que receba como parâmetro um array unidimensional contendo 1000 números inteiros. A função deverá encontrar e retornar o maior elemento armazenado no array.

4. Reescreva apenas a função do exercício anterior, usando passagem de parâmetros por referência.

5. Escreva apenas uma função em C que receba como parâmetro uma string e retorne a quantidade de vogais presentes na string.

6. Analise o programa abaixo:

```
#include <stdio.h>

void teste( int *x, int *y );

int main( void )
{
     int a = 10, b = 7;
     teste( &a, &b );
     return( 0 );
}

void teste( int *x, int *y )
{
     *x += *y;
     *y -= 4;
}
```

Após a execução do programa acima, quais serão os valores armazenados respectivamente nas variáveis **a** e **b**?

7. Escreva um programa em C contendo uma função que receba como parâmetro um valor inteiro com três dígitos. A função deverá determinar o valor armazenado na casa das dezenas. Por exemplo: 123 = 2. Utilize passagem de parâmetros por referência.

8. Escreva um programa em C contendo uma função que receba como parâmetro um valor inteiro positivo, representando um termo na seqüência de Fibonacci. A função deverá retornar o valor correspondente ao termo. Utilize passagem de parâmetros por referência.

9. Um número é chamado triangular quando é o resultado do produto de três números consecutivos. Por exemplo: 24 = 2 x 3 x 4. Escreva apenas uma função em C que receba como parâmetro um número inteiro e positivo e verifique se o mesmo é triangular. Utilize passagem de parâmetro por valor.

10. Escreva apenas uma função em C que receba como parâmetro um array bidimensional (quadrado de ordem 10 contendo valores inteiros). Calcule e retorne a soma dos elementos da diagonal principal.

10.7 Exercícios Complementares

11. Escreva um programa em C contendo uma função que receba um número inteiro. A função deverá retornar o sucessor e o antecessor do número. Utilize passagem de parâmetro por referência.

12. Escreva um programa em C que contenha uma função que receba como parâmetro três valores inteiros. A função deverá encontrar o maior dos três valores e retornar para a função principal que deverá imprima o valor no vídeo. Utilize passagem de parâmetros por valor.

13. Reescreva o programa do exercício anterior utilizando passagem de parâmetros por referência.

14. Reescreva a função do exercício 9, utilizando passagem de parâmetro por referência.

15. Escreva apenas uma função em C que receba como parâmetro uma string e retorne a última letra.

16. Escreva apenas uma função em C que receba um valor representando uma temperatura na escala Fahrenheit e converta para a escala Centígrados. A fórmula de conversão é: $F = \frac{9C + 160}{5}$, onde **F** representa a temperatura em Fahrenheit e **C** em Centígrados.

17. Escreva um programa em C, contendo uma função que receba três valores inteiros. Seu programa deverá imprima os valores em ordem crescente. Utilize passagem de parâmetros por referência.

18. Escreva um programa em C para calcular e imprima as raízes de uma equação do 2º grau. Seu programa deverá ter duas funções: uma para o cálculo do delta e a outra para calcular as raízes da equação.

19. Escreva apenas uma função em C que receba como parâmetro três valores inteiros e positivos e verifique se os mesmo podem formar um triângulo. Lembre-se que: três números podem representar os lados de um triângulo se cada lado é menor que a soma dos outros dois. Utilize passagem de parâmetro por referência.

20. Escreva um programa em C, contendo uma função que calcule e retorne o valor da expressão: $y = \frac{5x + 3}{\sqrt{x^2 - 16}}$.

21. Escreva apenas uma função em C que receba como parâmetro uma frase e retorne o valor do caractere localizado no meio da frase, caso a frase tenha um número ímpar de caracteres.

22. Escreva um programa em C, contendo uma função que calcule e retorne o valor da expressão: $h = 1 - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + ... + \frac{1}{N}$.

23. Escreva apenas uma função em C que calcule e retorne o valor da expressão: $y = \frac{1}{n} + \frac{2}{n-1} + \frac{3}{n-2} + ... + \frac{n-1}{2} + n$.

24. Repare a seguinte característica do número 3025: 30 + 25 = 55 e 55^2 = 3025. Escreva um programa em C contendo uma função que verifique um número com quatro dígitos apresenta a característica acima. Seu programa deverá leia vários números(um de cada vez). O programa termina quando for lido um valor menor do que 1000 ou maior que 9999.

25. Uma empresa classifica seus funcionários em três níveis de acordo com um índice de produtividade. São eles: (1) Excelente, (2) Bom e (3) Regular. Cada nível acrescenta um abono ao salário base do funcionário, de acordo com a seguinte tabela:

Nível	Aumento
Excelente	80% do salário base
Bom	50% do salário base
Regular	30% do salário base

Escreva um programa em C que leia o código do funcionário (numérico), seu salário base e seu nível. O programa deverá imprima o novo salário do funcionário. O cálculo do salário deverá ser feito por uma função. O programa deverá ser executado para vários funcionários até que o código 0 seja informado.

26. Escreva um programa em C que apresente o seguinte menu de opções para os usuários:

```
Escolha uma opção:
1. somar
2. subtrair
3. Multiplicar
4. Dividir

Resultado:
```

O usuário deverá escolher uma opção válida e em seguida fornecer dois valores. Dependendo da opção escolhida, seu programa deverá executar os cálculos e exibir o resultado. Utilize uma função para cada opção.

27. A concessionária local de telefonia de uma cidade do interior gostaria de fazer um programa que pudesse controlar a central de 1000 assinantes que foi disponibilizada para a cidade. Faça um programa que funcione de acordo com o seguinte menu:

```
Menu de Opções:

   1. Inclusão de novo telefone
   2. Alteração de telefone
   3. Exclusão de telefone
   4. Impressão dos telefones cadastrados.
   5. Consulta por nome
   6. Sair
```

Para cada uma das opções do menu deverá ter uma função no seu programa. Sugestão: trabalhe com dois arrays, um para o número dos telefones e o outro para os nomes. Declare os arrays como variáveis globais.

Capítulo 11

Estruturas

11.1 Definição e declaração de estruturas em C

Até o momento vimos como agrupar várias informações (utilizando arrays) do mesmo tipo. Em C podemos agrupar também informações de tipos diferentes formando um novo tipo de dados. Esse novo tipo é chamado de **tipo personalizado** ou **tipos definidos pelo usuário**. Este tipo é conhecido como **estrutura** em C ou como **registro** em algumas linguagens de programação como o pascal.

Define-se **estrutura** como um conjunto de variáveis que são referenciadas sob o mesmo nome, oferecendo um meio de manter as informações relacionadas juntas. Como exemplo de uma estrutura podemos citar uma ficha pessoal contendo nome, telefone e endereço. Essas três informações estão relacionadas entre si para compor a ficha. A ficha em si é a estrutura.

Para se criar uma estrutura em C usa-se o comando **struct**. Sua forma geral é:

```
struct nome_da_estrutura
{
        tipo_1 nome_1;
        tipo_2 nome_2;
        ...
        tipo_n nome_n;
} variáveis_estrutura;
```

O *nome_da_estrutura* é o nome para a estrutura. As *variáveis_estrutura* são opcionais e seriam nomes de variáveis que o usuário já estaria declarando e que seriam do tipo *nome_da_estrutura*. Note que a declaração é terminada com ponto e vírgula (;). Um primeiro exemplo:

```
struct teste{
    float i;
    int f;
} a, b;
```

Neste caso, *teste* é uma estrutura com dois campos, *i* e *f*. Foram também declaradas duas variáveis, *a* e *b* que são do tipo da estrutura, isto é, *a* possui os campos *i* e *f*, o mesmo acontecendo com *b*. O exemplo abaixo cria uma estrutura de endereços:

```
struct tipo_endereco
{
```

```
        char rua [50];
        int numero;
        char bairro [20];
        char cidade [30];
        char sigla_estado [3];
        long int CEP;
};
```

Um outro exemplo é a criação de uma estrutura chamada *pessoa,* que contém os dados pessoais de um individuo:

```
struct pessoa
{
        char nome [50];
        long int telefone;
        struct tipo_endereco endereco;
};
```

Como mostrado no exemplo acima, uma estrutura pode fazer parte de outra (a *struct tipo_endereco* é usada pela *struct ficha_pessoal*).

O programa abaixo mostra como as duas estruturas declaradas acima podem ser utilizadas dentro de um programa. O programa preenche uma ficha pessoal de um cliente.

```
#include <stdio.h>
#include <string.h>
struct tipo_endereco
{
        char rua[50];
        int numero;
        char bairro[20];
        char cidade[30];
        char sigla_estado[3];
        long int CEP;
};

struct pessoa
{
        char nome[50];
        long int telefone;
        struct tipo_endereco endereco;
};

int main( void )
{
      struct pessoa ficha;
      strcpy (ficha.nome,"Luiz Osvaldo Silva");
      ficha.telefone=4921234;
```

```
        strcpy (ficha.endereco.rua,"Rua das Flores");
        ficha.endereco.numero=10;
        strcpy (ficha.endereco.bairro,"Cidade Velha");
        strcpy (ficha.endereco.cidade,"Belo Horizonte");
        strcpy (ficha.endereco.sigla_estado,"MG");
        ficha.endereco.CEP=31340230;
        return 0;
}
```

O programa acima declara uma variável ficha do tipo *pessoa* e preenche os seus dados. O exemplo mostra como podemos acessar um elemento de uma estrutura: basta usar o ponto (.). Assim, para acessar o campo telefone de ficha, escrevemos:

ficha.telefone = 4921234;

Como a *struct pessoa* possui um campo, *endereco*, que também é uma *struct*, podemos fazer acesso aos campos desta *struct* interna da seguinte maneira:

ficha.endereco.numero = 10;
ficha.endereco.CEP=31340230;

Desta forma, estamos acessando, primeiramente, o campo *endereco* da *struct ficha* e, dentro deste campo, estamos acessando o campo *numero* e o campo *CEP*.

11.2 Matrizes de estruturas

Uma estrutura é como qualquer outro tipo de dado no C. Podemos, portanto, criar matrizes de estruturas. Vamos ver como ficaria a declaração de um array de 100 do tipo *struct pessoa*:

```
struct pessoa fichas [100];
```

Poderíamos então acessar a segunda letra da sigla de estado da décima terceira ficha fazendo:

```
fichas[12].endereco.sigla_estado[1];
```

Observação: Como todas as matrizes, matrizes de estruturas começam a indexação em 0.

11.3 Passando elementos de estruturas como parâmetros para funções

Nos exemplos apresentados acima para a utilização de estruturas, as mesmas sempre foram declaradas como variáveis globais. Para passar um elemento de uma variável estrutura para uma função é simples, pois você passa apenas o valor desse elemento para a função. Exemplo:

```
struct misterio{
```

```
        char x;
        int y;
        float z;
        char s[10];
} teste;
```

`func( teste.x );` //passa apenas o valor do caractere *x* para a função
`func( teste.y );` //passa apenas o valor da variável *y* para função
`func( teste.s[2] );` //passa o valor do caractere de *s*[2]

Nos exemplos mostrados acima, temos a passagem de parâmetros por valor. Caso seja necessário passar o endereço de um elemento individual da estrutura, basta colocar o operado **&** antes da variável do tipo da estrutura. Exemplo:

`func( &teste.x );` //passa o endereço do caractere x para a função
`func( &teste.y );` //passa o endereço da variável y para a função
`func( teste.s );` //passa o endereço da string s para a função.

Não se esqueça que o operador **&** precede o nome da variável do tipo estrutura e não o nome do elemento individual da estrutura. Outro ponto a destacar é com relação a variáveis do tipo vetores de caracteres (*strings*) que já significam endereços de memória, portanto a utilização do operador **&** é dispensável.

11.4 Passando estruturas inteiras para funções

No tópico anterior passamos elementos individuais como parâmetros para funções. Da mesma forma, podemos passar para uma função a estrutura inteira. Quando uma estrutura for passada como parâmetro para uma função, não se esqueça que o tipo de argumento deve coincidir com o tipo de parâmetro. Veja o exemplo abaixo:

```
void main( void )
{
     struct exemplo{
          int a, b;
          char c;
     } d;

     d.a = 1000;
     f( d );
}

f( struct {
     int x, y;
     char h; } p )
{
   printf(“%d”, p.x);
}
```

11.5 Ponteiros para estruturas

A linguagem de programação C permite a utilização de ponteiros para estruturas da mesma forma que permite a utilização de ponteiros para os outros tipos de variáveis. Como para as outras variáveis, a declaração do ponteiro é feita utilizando o operador * na frente do nome da estrutura. Exemplo:

```
struct teste *p;
```

No exemplo acima, a variável *p* armazena o endereço de memória da estrutura *teste*. Ponteiros para estruturas apresentam duas aplicações básicas:

⇒ Gerar passagem por referência
⇒ Criação de estruturas de dados encadeadas (listas). Este será o enfoque do curso de Estruturas de Dados 1 no próximo ano.

A principal desvantagem da passagem de estruturas por valor está no tempo necessário para armazenar e retirar todos os elementos da estrutura da pilha, caso a estrutura apresente muitos membros. Estruturas grandes quando passadas por valor podem prejudicar o desempenho do programa. O mesmo não acontece para estruturas pequenas.

Uma solução para não comprometer o desempenho do programa quando uma estrutura grande precisa ser passada para uma função, é a utilização de ponteiros para estruturas. Neste caso, apenas um endereço de memória é armazenado na pilha. Exemplo da atribuição do endereço de uma estrutura para uma variável ponteiro.

```
struct teste {
     int a, b;
     float x;
} y;

struct teste *p;

p = &y;
```

Quando utilizamos ponteiros para estruturas, a forma de acessar os dados sofre alteração. Nesta situação utiliza-se o operador (→) ao operador (.). Por exemplo:

```
p→a = 10;
```

O operador seta somente é utilizado quando se está acessando um elemento de estrutura através de um ponteiro para a estrutura.

11.6 Comando typedef

A linguagem C permite que o programador defina novos nomes aos tipos de dados que estão sendo criados. Para definir esses nomes utiliza-se o comando ***typedef.*** Vale salientar que o comando

typedef é utilizado apenas para definir um novo nome para um tipo existente. A forma geral do comando ***typedef*** é:

typedef tipo nome;

sendo **tipo** qualquer tipo de dados permitido pela linguagem e **nome** é o novo nome para esse tipo. O novo nome a ser definido nada mais é do que uma opção ao nome antigo e não uma substituição. Como exemplo, podemos criar um novo nome para o tipo de dado **int** usando:

```
typedef int carro;
```

A instrução acima informa ao compilador que carro é um outro nome para o tipo int. Dessa forma, para declarar uma variável do tipo inteiro podemos ter duas opções:

```
int y;
carro y;
```

11.7 Exercícios em Classe

1. Analise o programa em C abaixo:

```
#include <stdio.h>
struct teste {
     int x, y;
};

void func ( struct teste *t );

int main( void )
{
     struct teste p;

     p.x = 10;
     p.x = 20;

     func( &p );
     return( 0 );
}

void func( struct teste *t )
{
     t → x = t → x * 2;
     t → y = t → x - t → y;
}
```

Após a execução do programa acima, quais são os valores armazenados respectivamente nas variáveis *x* e *y*?

2. Escreva um programa em C que declare uma estrutura chamada **ponto**. Esta estrutura será composta por duas variáveis *x* e *y*, que representam as coordenadas do ponto. Em seguida declare 2 pontos, leia as respectivas posições de cada um e calcule a distância entre eles. A distância entre dois pontos é dada pela seguinte expressão: $d = \sqrt{(x2 - x1)^2 + (y2 - y1)^2}$. Exiba a distância entre os pontos.

3. Escreva um programa em C para armazenar o nome e o salário de 100 funcionários de uma empresa. O seu programa deverá ter uma função que retorne o valor do maior salário pago pela empresa.

4. Um número real é um número que apresenta uma parte inteira e uma parte decimal. Por exemplo: 1.567, 20.878 etc. Deseja-se criar uma representação computacional que simule um número real.
a) Defina em C uma estrutura para representar um número real.
b) Escreva uma função em C que receba como parâmetro (passagem por referência) um número real conforme definido no item "a" e retorne apenas a parte inteira.

5. Para uma turma de 40 alunos, escreva um programa em C que permita armazenar o nome a respectiva nota de cada um dos alunos na disciplina Algoritmos. Exiba a média geral da turma e quantidade de alunos que ficaram com nota acima da média.

6. Em uma garagem existe um registro de todos os carros que estão à venda. Sabe-se que a garagem comporta no máximo 20 carros. Para cada carro tem-se registrado o modelo, a cor e o ano de fabricação. Escreva apenas uma função em C que retorne a quantidade de carros que foram fabricados antes de 2000.

11.8 Exercícios Complementares

7. Com base no exercício 2, escreva um programa em C que declare 3 pontos. Mostre qual a maior distância entre os pontos.

8. Em uma pizzaria, quando um cliente liga para fazer um pedido são anotados o nome do cliente, e o sabor da pizza (mussarela, calabresa e marguerita). Suponha que em uma hora foram feitos 6 pedidos. Escreva um programa em C, que contenha uma função que imprima a porcentagem de cada sabor de pizza pedido.

9. Os números complexos apareceram no século XVI ao longo das descobertas de procedimentos gerais para resolução de equações algébricas de terceiro e quarto grau. No século XVII os complexos são usados de maneira tímida para facilitar os cálculos. No século XVIII são mais usados na medida que se descobre que os complexos permitem a conexão de vários resultados dispersos da Matemática no conjunto dos números reais. No entanto, nada é feito para esclarecer o significado desses novos números. No século XIX, aparece a representação geométrica dos números complexos, motivada pela necessidade em Geometria, Topografia e Física, de se trabalhar com o conceito de vetor no plano. Os números complexos passam a ser aplicados em várias áreas do conhecimento humano, dentro e fora da Matemática.

Um número complexo é todo número que pode ser formado por duas partes: a parte real e a parte imaginária. A representação geral de um número complexo é:

$$z = a + bi$$

sendo **a** e **b** número reais. O número **a** é a parte real e o número **b** é a parte imaginária. Exemplos de número complexos: 2+3i, 5+i. O símbolo *i* que indica que um número é complexo tem valor $\sqrt{-1}$.

Com base nas informações do texto, responda:

a) Defina em C uma estrutura que possa representar um número complexo.
b) Escreva uma função em C que receba como parâmetro (passagem por referência) dois números complexos conforme definido no item "a" e verifique se são iguais. Por exemplo: $y = a+bi$ e $x = c+di$. Estes números são iguais se a = c e b = d.

10. Escreva um programa em C que faça a manutenção da conta bancária de 10 possíveis clientes do banco Z. Deverá ser apresentado o usuário o seguinte menu:

```
Escolha uma opção:
1. Cadastro de clientes
2. Efetuar saque
3. Efetuar depósito
4. Imprimir clientes

Digite uma opção [   ]
```

Cada uma das opções do menu deverá ser codificada dentro de uma função especifica.

Capítulo 12

Manipulação de arquivos em C

12.1 Introdução

O armazenamento de dados em variáveis e arrays é temporário, ou seja, os dados são perdidos quando uma variável local "sai do escopo" ou quando o programa termina. Portanto, os arquivos são utilizados para retenção em longo prazo de grandes quantidades de dados, mesmo depois que o programa termina. Estes dados são armazenados em dispositivos de armazenamento secundário como discos magnéticos, discos ópticos, etc.

12.2 Streams

Em C, o sistema de arquivos trabalha com vários dispositivos diferentes. Dessa forma, o sistema de arquivos deve conseguir trabalhar com esses dispositivos da mesma forma, embora esses dispositivos sejam muito diferentes. Isso acontece porque o sistema de arquivo transforma-os em um dispositivo lógico chamado *streams*. Estas streams são independentes do dispositivo.

Existem dois tipos de streams: texto e binária. Uma stream de texto nada mais é do que uma seqüência de caracteres. Por outro lado, uma stream binária é uma seqüência de bytes. Em C devemos associar uma stream com um arquivo específico através da operação de abertura.

Cada stream associada a um arquivo tem uma estrutura de controle de arquivo do tipo FILE. Essa estrutura é definida no cabeçalho STDIO.H

As principais funções para manipular um arquivo estão relacionadas na tabela abaixo:

Nome	***Função***
fopen()	*Abre um arquivo*
fclose()	*Fecha um arquivo*
putc()	*Escreve um caractere em um arquivo*
getc()	*Lê um caractere de um arquivo*
fseek()	*Posiciona o ponteiro de arquivo em um byte específico*
feof()	*Devolve verdadeiro se o fim de arquivo for atingido*
ferror()	*Devolve verdadeiro se ocorreu um erro*
rewind()	*Repõe o ponteiro de posição de arquivo no início do arquivo*
remove()	*Apaga um arquivo*
fflush()	*Descarrega um arquivo*

Para realizar as operações básicas em um arquivo (escrever e leia), seu programa precisa usar ponteiros para arquivos.

Um ponteiro de arquivo é um ponteiro para as informações sobre o arquivo tais como seu nome, status e posição atual do arquivo. Um ponteiro de arquivo é uma variável do tipo FILE que é definido em STDIO.H. A declaração dessa variável é:

```
FILE *arq;
```

12.3 Abrindo um arquivo

A abertura de um arquivo é realizada com a função **fopen()**. Esta função associa um arquivo a uma stream. O retorno da função **fopen()** é um ponteiro do tipo FILE para o arquivo que está sendo aberto. A função **fopen()** necessita de dois parâmetros: o primeiro é um ponteiro para uma cadeia de caracteres que representa o nome do arquivo e o segundo é o modo como o arquivo será aberto, ou seja, apenas para leitura, apenas para escrita ou leitura e escrita ao mesmo tempo.

Os valores para o modo de leitura estão relacionados na tabela abaixo.

Modo	***Função***
r	*Abre um arquivo texto para leitura*
w	*Cria um arquivo texto para escrita*
a	*Anexa a um arquivo texto*
rb	*Abre um arquivo binário para leitura*
wb	*Cria um arquivo binário para escrita*
ab	*Anexa a um arquivo binário*
r+	*Abre um arquivo texto para leitura/escrita*
w+	*Cria um arquivo texto para leitura/escrita*
a+	*Anexa ou cria um arquivo texto para leitura/escrita*
r+b	*Abre um arquivo binário para leitura/escrita*
w+b	*Cria um arquivo binário para leitura/escrita*
a+b	*Anexa um arquivo binário para leitura/escrita*

No momento da abertura de um arquivo, caso ocorra algum problema, a função **fopen()** devolverá um ponteiro nulo. Portanto, devemos sempre testar o valor de retorno antes de utilizarmos o ponteiro.

O trecho de código abaixo mostra a abertura de um arquivo.

```
FILE *arq;
arq = fopen( “teste”, “w” );
if( arq == NULL )
{
      printf(“O arquivo não pôde ser aberto”);
      exit(1);
}
```

Caso você utilize a função **fopen()** para abrir um arquivo no modo de escrita, e exista algum arquivo com o mesmo nome este será apagado e um novo arquivo será criado, caso não exista, o arquivo será criado. Arquivos existentes só podem ser abertos no modo de leitura. Se porventura

você quiser adicionar informações ao final do arquivo, o mesmo deve ser aberto no modo anexar. O modo de leitura/escrita não apagará um arquivo se ele já existir.

12.4 Fechando um arquivo

Para fechar um arquivo associado a uma stream usamos o comando **fclose()**. Se a operação for bem-sucedida, a função retornará o valor zero. Qualquer outro valor indica erro. A utilização do comando **fclose()** é exemplificada abaixo.

```
fclose( arq );
```

12.5 Escrevendo e lendo caracteres em um arquivo

A operação para escrever caracteres em um arquivo pode ser realizada através de duas funções equivalentes: **putc()** e **fputc()**. Ambas as funções são utilizadas para escrever caracteres em um arquivo que foi previamente aberto no modo escrita. A sintaxe para a função **putc()** é:

```
int putc( int ch, FILE *arq );
```

sendo que *ch* é o caractere que será escrito no arquivo e *arq* o ponteiro de arquivo devolvido pela função **fopen()**. Por razões mais antigas, o caractere *ch* é do tipo inteiro. Caso a operação for bem-sucedida, a função retornará o caractere escrito, caso contrário retornará **EOF** (end of file).

O processo de leitura de caracteres de um arquivo também envolve duas funções: **getc()** e **fgetc()**. Ambas as funções lêem caracteres de um arquivo previamente aberto no modo leitura. A sintaxe para a função **getc()** é:

```
int getc( FILE *arq );
```

Se a operação for bem sucedida, a função retornará um inteiro. Quando o final do arquivo for atingido um EOF será retornado.

O programa abaixo utiliza as funções apresentadas acima para abrir e escrever caracteres em um arquivo.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
      FILE *arq;
      char ch;

      arq = fopen( “teste.txt”, “w” );
      if( arq == NULL )
      {
            printf(“arquivo não pode ser aberto”);
            exit(1);
      }
```

```
	do {
		ch = getchar();
		putc( ch, arq );
	} while( ch != '$' );
	fclose( arq );

	return( 0 );
}
```

Abaixo vamos abrir o arquivo criado no programa acima e leia os caracteres gravados.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
	FILE *arq;
	char ch;

	arq = fopen( "teste.txt", "r" );
	if( arq == NULL )
	{
		printf("arquivo não pode ser aberto");
		exit(1);
	}

	//lê um caractere
	ch = getc( arq );
	while( ch != EOF )
	{
		putchar( ch );
		ch = getchar( fp );
	}

	fclose( arq );

	return( 0 );
}
```

12.6 Escrevendo e lendo strings em um arquivo

Da mesma forma que podemos escrever e leia apenas caracteres de um arquivo, C apresenta funções para leitura e escrita de strings. As funções são: **fputs()** e **fgets()**. Ambas as funções operam de forma semelhante às funções **putc()** e **getc()**.

A função **fgets()** lê uma string até que um caractere de nova linha seja lido ou que o comprimento de caracteres especificado para leitura tenham sido lidos. A string resultante será terminada por um nulo.

O programa abaixo mostra a utilização das funções **fputs()** e **fgets()**.

```
#include <stdio.h>
```

```
#include <stdlib.h>

int main()
{
      char s[80];
      FILE *arq;

      arq = fopen( "teste.txt", "w" );
      if( arq == NULL )
      {
            printf("o arquivo não pode ser aberto" );
            exit(1);
      }

      do {
            printf("Entre com uma string (enter para sair): \n");
            gets( s );
            strcat( s, "\n" );
            fputs( s, arq );
      } while( *s != '\n');

      return( 0 );
}
```

12.7 Funções fread() e fwrite()

Quando desejamos escrever ou leia tipos de dados que apresentam mais que um byte, as funções **fread()** e **fwrite()** são utilizadas. Estas funções apresentam sintaxes diferentes das funções apresentadas até agora.

```
size_t fread( void *buf, size_t num_bytes, size_t count, FILE *arq );

size_t fwrite( void *buf, size_t num_bytes, size_t count, FILE *arq );
```

Para **fread()**, *buf* é um ponteiro para a região de memória que contém os dados que serão lidos. Para **fwrite()** é um ponteiro para as informações que serão escritas no arquivo; *num_bytes* é o tamanho da informação que será lida ou escrita. O argumento *count* determina quantos itens serão copiados. O tipo de dado *size_t* é definido na biblioteca STDIO.H e é aproximadamente o mesmo que *unsigned*.

A função **fread()** devolve o número de itens lidos e função **fwrite()** o número de itens escritos. Estas funções podem ser utilizadas tanto para arquivos do tipo texto quanto para arquivos do tipo binário.

O programa abaixo escreve um valor do tipo double e um do tipo inteiro em um arquivo e depois faz a leitura dos dados.

```
#include <stdio.h>
#include <stdlib.h>

int main()
```

```
{
      double d = 12.33;
      long l = 1230456;
      FILE *arq;

      arq = fopen( "num.txt", "wb+" );
      if( arq == NULL )
      {
            printf("o arquivo não pode ser aberto" );
            exit(1);
      }

      fwrite( &d, sizeof( double ), 1, arq );
      fwrite( &l, sizeof( long ), 1, arq );
      rewind( arq );
      fread( &d, sizeof( double ), 1, arq );
      fread( &l, sizeof( long ), 1, arq );
      printf("Os valores são %f %ld", d, l );
      fclose( arq );
      return( 0 );
}
```

Uma das aplicações que aparecem com freqüência na manipulação de arquivos envolve a leitura e a escrita de valores definidos pelo usuário. Considere a seguinte declaração:

```
struct lista {
      float salário;
      char nome[40];
} custo;
```

A instrução abaixo escreve o conteúdo de *custo* no arquivo apontado por *arq*.

```
fwrite( &custo, sizeof( lista ), 1, arq );
```

12.8 Exercícios

1. Escreva um programa em C que preencha uma matriz quadrada de ordem 4 com valores aleatórios entre 0 e 100. Em seguida grave os dados em um arquivo do tipo texto.

2. Considere um registro contendo o nome o curso de 10 alunos. Escreva um programa em C que permita entrar com os dados e armazená-los em um arquivo.

Bibliografia

ARAKAKI R. e outros. Fundamentos de Programação C: Técnicas e Aplicações. Rio de Janeiro, Livros Técnicos e Científicos Editora Ltda, 1989. Campinas, São Paulo, 2000.

Curso de C - http://ead1.eee.ufmg.br/cursos/C/c.html

DEITEL, H. M., DEITEL, P.J. C++ - Como Programar. 3ª Edição, Editora Bookman, Porto Alegre, RS.

FORBELLONE, A. L. V. & EBERSPÄCHER, H. F. Lógica de programação - a construção de algoritmos e estruturas de dados. São Paulo, Makron Books, 2a Ed., 2000.

http://www.icmc.sc.usp.br/ensino/material/ Apostila de Tópicos de Linguagens de MANZANO, J. A. N. G. & OLIVEIRA, J. F. Algoritmos: lógica para desenvolvimento de programação. São Paulo: Érica, 8a Ed., 2000.

MANZANO, J. A. N. G. Estudo dirigido: Linguagem C. São Paulo: Érica, 1997.

MIZRAHI, V. V. Treinamento em Linguagem C – Curso Completo – Módulo 1. 1ª Edição, Editora Makron Books, São Paulo, SP.

MIZRAHI, V. V. Treinamento em Linguagem C – Curso Completo – Módulo 2. 1ª Edição, Editora Makron Books, São Paulo, SP.

MORAES, P. S. Lógica de programação, Centro de Computação - DSC, Unicamp, Programação. Instituto de Ciências Matemáticas e de Computação – USP, São Carlos.

SALVETTI D.D. & BARBOSA L.M. Algoritmos. Makron Books, São Paulo, 1998.

SCHILDT, H. C Completo e Total. 3ª Edição, Editora Makron Books, São Paulo, SP, 1997.

SCHILDT, H. Turbo C++. São Paulo, Makron Books, 1992.

SEBESTA R. W. Conceitos de linguagem de programação. Bookman Editora, 4ª edição, Porto Alegre, 2000.

SMULLYAN, R. Alice no País dos Enigmas, Jorge Zahar Editor, Rio de Janeiro, 2000.

ZIVIANI, N. Projeto de Algoritmos com implementações em Pascal e C. 2ª Edição, Editora Pioneira Informática, São Paulo, SP, 1993.