

CAPÍTULO I INTRODUÇÃO

Objetivos

- Familiarizar o aluno com as noções de algoritmo e de refinamentos sucessivos através de exemplos não-computacionais;
- Introduzir a noção de programa e linguagem de programação;
- Ressaltar as vantagens de representação em C;
- Identificar as componentes principais de um programa em C, baseando-se em um exemplo não computacional.

I.1 – O que é um Algoritmo

Os computadores vêm sendo cada vez mais utilizados nas mais diversas áreas, desde a medicina até a engenharia, desde a administração de empresas até o controle de máquinas industriais, desde a edição de textos pessoais até a produção de efeitos especiais em filmes.

Em todas as situações, o computador age como uma máquina capaz de coletar, manipular e devolver dados em grande quantidade e alta e alta velocidade. Estas atividades são executadas por uma gama de circuitos e componentes eletrônicos (Hardware).

Ao longo das últimas décadas, estes componentes têm se tornado cada vez mais eficientes e baratos, devido aos avanços tecnológicos.

“Embora os computadores modernos sejam capazes de executar operações difíceis e complexas, eles são máquinas dóceis e de uma inteligência restrita. Eles devem ser ensinados exatamente o que fazer, e as instruções devem ser feitas em uma linguagem simples, precisa e limitada que eles possam compreender. Estas instruções são mais conhecidas como software” (Darmell,1988).

Uma das etapas fundamentais no desenvolvimento do software é a Construção de Algoritmo.

Algoritmo é uma seqüência de operações que devem ser executadas em uma ordem definida e não-ambígua, com o propósito de solucionar um determinado problema.

Esta definição pode caracterizar tanto um receita de bolo com o conjunto de instruções para montagem de um automóvel. No nosso caso, serviria para definir o conjunto de atividades que o computador deve desempenhar para solucionar um problema.

O algoritmo é geralmente representado através de modelos de fácil compreensão pelos humanos, mas incompreensíveis para o computador. Para que o computador possa executar um algoritmo, é necessário que este algoritmo seja traduzido para uma linguagem que seja entendida pelo computador - **a linguagem de programação**. Ao traduzir-se um algoritmo para uma linguagem de programação obtém-se um **Programa**.

Este curso tem por objetivo desenvolver a capacidade de construir algoritmos de uma forma organizada (estruturada) na linguagem de programação C.

I.2 – Exemplo de Algoritmo

Boa parte das atividades do dia-a-dia pode ser descrita através de algoritmos. Por exemplo, o ato de trocar um pneu furado em um carro consiste em:

- Pegar as ferramentas no porta-malas;
- Retirar o estepe;
- Instalar o macaco;
- Levantar o carro parcialmente;
- Afrouxar os parafusos do pneu furado;

- Retirar o pneu furado;
- Instalar o novo pneu;
- Abaixar o carro
- Apertar bem as porcas;
- Guardar o pneu furado e as ferramentas;

A seqüência descrita acima pode ser completada com várias observações. Por exemplo:

- O que fazer se o macaco não estiver no porta-malas?
- O que fazer se o estepe também estiver vazio?
- Deve-se sempre puxar o freio de mão antes de executar estas operações.

Por outro lado, algumas ações devem ser levadas a cabo após executar a troca do pneu. Por exemplo:

- Limpar as mãos;
- Consertar o pneu furado;

Se o responsável pela construção do algoritmo tiver que se preocupar com todas as possibilidades do ambiente de execução do mesmo, e além disso, com todas as necessidades de tratamento pós-execução, nenhum algoritmo será completo. Sempre alguma condição poderá ser esquecida. Assim, antes de se construir um algoritmo, deve-se especificar as condições de início (dados de entrada) e as de término (dados de saída). No caso acima, teremos:

- **Condição de entrada:** um carro com todos os equipamentos de segurança em condições, parado e freado, com um pneu furado.
- **Condição de saída:** um carro parado, freado, com todos os equipamentos de segurança em condições, exceto pelo pneu estepe vazio, com todos os pneus OK.

Além disso, alguém pode argumentar que as ações não estão suficientemente detalhadas para compreensão. Para uma pessoa acostumada a trocar pneus furados as explicações são suficientes, mas será que um selvagem do deserto Kalahari, que nunca viu um carro antes, saberá utilizar corretamente um macaco para elevá-lo?

Assim, a função “levantar o carro” seria melhor descrita como:

- *Empunhe o cabo da manivela do macaco; comece a girar a manivela no sentido horário, mantendo este movimento até que o pneu do carro esteja suspenso a pelo menos 10 cm do chão; solte a manivela.*

Boa parte dos problemas tratados no computador é resolvida por seqüência de instruções (algoritmos) compostas por um agrupamento de atividades por si só complexas. Da mesma forma que foi necessário detalhar o uso do macaco para o selvagem, também estas atividades complexas devem ser detalhadas até a um nível em que o computador seja capaz de reconhecê-las e executá-las.

Devido à velocidade com que o computador processa cada uma destas instruções simplificadas, não há problema em descrever e tratar uma atividade complexa através de milhares de outras atividades básicas.

Para os seres humanos, contudo, é difícil memorizar e detalhar milhares de instruções, associando-as à atividade original. Para facilitar esta tarefa, utilizam-se duas ferramentas:

- Técnica de Refinamentos Sucessivos;
- Representação de Algoritmos.

I.3 – A Técnica de Refinamentos Sucessivos

O computador só é capaz de entender instruções muito simplificadas, escolhidas dentro de um conjunto bastante restrito. A transformação de atividades complexas, bem mais que a troca de um pneu furado, seqüência de instruções computacionais pode ser uma atividade tediosa e frustrante, na medida em que quase sempre se esquece algum detalhe.

Os seres humanos têm capacidade limitada de armazenar informações em memória. Estas informações tanto podem ser os processos necessários ao cumprimento de alguma atividade como as ferramentas necessárias para a execução dos processos individualmente. Uma boa política é então, dividir atividades complexas em processamentos-componentes menores, especificando apenas a entrada e a saída de cada um dos processamentos-componentes. Após individualizados os processamentos-componentes e suas entradas/saídas, pode-se preocupar com as ferramentas necessárias para sua execução.

Entrando, mesmo estes processos menores podem ser ainda tão complexos que exijam novas divisões. Repete-se, então, a divisão, desta vez no interior dos processamentos-componentes, gerando novos processamentos mais simplificados.

Esta atividade é feita pela construtor do algoritmo (VOCÊ!) até que os processamentos-componentes se tornem seqüências de instruções aceitáveis pelo computador.

Está técnica é conhecida como dividir-para-conquistar, ou também **Técnica de Refinamentos Sucessivos**.

Vejam graficamente o que ocorre no algoritmo que descreve atividades matinais de uma pessoa:

=>Algoritmos “Manhã de Segunda-Feira”

- Levantar da cama;
- Cuidar da higiene pessoal;
- Tomar café;
- Recolher material necessário;
- Sair de casa.

Processamento-Componente “Levantar de Cama”

- Abrir os olhos;
- Se espreguiçar;
- Ficar de pé.

Processamento-componente “Tomar Café”

- Sentar à mesa;
- Pôr café na xícara;
- Pôr margarina no pão;
- Comer o pão e tomar o café;
- Comer uma fruta.

Processamento-componente “Comer o pão e tomar o café”

- Apanhe o pão com margarina e a xícara com café;
- Enquanto o pão com margarina e o café não acabarem, coma mais um pedaço de pão e beba mais um pouco de café da xícara.

Processamento-componente “Coma uma fruta”

- Se houver algum tipo de fruta disponível na mesa, escolha um tipo;
- Pegue uma fruta do tipo escolhido;
- Coma a fruta.

Exercícios

1) Termine o detalhamento de todos os procedimentos necessários à sua “Manhã de Segunda-Feira”. Depois compare com pelo menos dois de seus colegas.

2) Use a técnica de refinamentos sucessivos para detalhar as seguintes atividades:

- Limpeza de um automóvel;
- Montagem de uma barraca de camping;
- Preparação de um bolo de aniversário.

I.4 – Representação de Algoritmos

Devido à pequena capacidade de memorização humana, os procedimentos-componentes devem ser armazenados em algum lugar, para permitir o acesso rápido e o fácil entendimento. Para isto, podem ser utilizadas desde folhas de papel até o auxílio de computadores.

Qualquer que seja o tipo de armazenamento, estes procedimentos/algoritmos devem ser descritos por algum modelo. Nos exemplos anteriores utilizamo-nos da própria língua portuguesa, que é um modelo de representação com que estamos acostumados a trabalhar.

A língua portuguesa traz, contudo, uma série de inconvenientes:

- Certas palavras têm dupla interpretação;
- Exigem-se regras de concordância bastante definidas;
- A pontuação pode alterar completamente o significado de uma frase.

Além disso, a língua falada costuma apresentar construções complexas, bastantes distintas da simplicidade das linguagens entendidas pelos computadores.

Os modelos de representação de algoritmos devem ser mais restritos, diminuindo a liberdade de expressão do “construtor de algoritmos”, mas com isso eliminando os problemas de interpretação, concordância e pontuação. Estes modelos devem ainda utilizar-se da apresentação gráfica para tornar os algoritmos mais fáceis de serem compreendidos.

A linguagem de programação C proporciona estas características para a representação de algoritmos.

I.5 – Partes de um Programa em C

Um exemplo bastante conhecido de representação de programas é o de receitas culinárias. Estas têm a seguinte estrutura:

- *Nome* da receita;
- *Ingredientes*: descreve todo o material necessário durante o preparo da receita;
- *Modo de preparar*: descreve como agrupar e tratar os ingredientes.

Além disso, toda receita vem com muitos *comentários* relativos a detalhes que poderiam passar despercebidos e estragar o resultado final. Os comentários servem para chamar a atenção do cozinheiro e explicar melhor o porque de cada passo.

Da mesma forma um bom programa terá 4 partes:

1. Cabeçalho: com o nome do programa;
2. Dicionário de Dados: definindo os dados de entrada, de saída e outros que sejam necessários ao longo do processamento do programa;
3. Corpo: definindo o tratamento dado aos dados no programa;
4. Comentários: esclarecendo aspectos pouco claros do programa em suas diversas partes (Cabeçalho, Dicionário de Dados e Corpo).

Além disso, a forma como as linhas do programa estão dispostas fisicamente, indentação, deve ser feita de maneira a tornar a listagem do programa o mais clara quanto possível, permitindo assim uma boa legibilidade do programa.

Para que um programa apresente uma boa estética deve-se seguir regras básicas, como:

- Coloque cada comando em uma linha separada;
- Insira uma linha em branco antes de cada seção do programa e sempre que apropriado em uma seqüência de comandos para destacar blocos de comandos;
- Mantenha-se fiel a indentação dos comandos de modo a enfatizar a relação entre palavras reservadas e comandos que compreendem estruturas de controle, seções de programas, etc.

CAPÍTULO II

A SOLUÇÃO DE UM PROBLEMA ATRAVÉS DO COMPUTADOR

I.1 – O que é um Algoritmo

Quando se deseja resolver um problema com auxílio do computador, é necessário a execução de uma série de etapas, elaboradas e executadas passo a passo na forma de algoritmos.

O algoritmo é descrito em português, o que permite ao programador pensar no problema a ser resolvido, e não na máquina ou linguagem de programação que serão usadas. Esta preocupação fica para uma etapa posterior.

I.2 – A Programação Estruturada

A programação estruturada consiste em uma metodologia de projeto de programa com objetivo de reduzir a complexidade dos grandes sistemas de software. Assim essa metodologia se propõe a:

- Facilitar a escrita dos programas;
- Facilitar a leitura e compreensão de programas;
- Permitir a verificação a priori dos programas;
- Facilitar a manutenção e verificação dos programas.

A idéia básica da programação estruturada é reduzir a complexidade do sistema, em três níveis:

PRIMEIRO NÍVEL

Desenvolvimento do programa em diferentes fases por refinamentos sucessivos (top-down). Isto é, o problema inicial é subdividido em subproblemas menores, etc. A cada divisão são levados em conta mais detalhes sobre a especificação do problema, sendo que os meios disponíveis para a solução (máquina e linguagem de programação) são deixados para as últimas fases.

Em cada etapa da decomposição, é preciso garantir que:

- As soluções dos subprogramas permitam obter a solução total;
- A seqüência de ações componentes faça sentido;
- A decomposição selecionada se aproxime cada vez mais daquilo que é representável pelos comandos da linguagem na qual o programa será escrito.

O processo inverso, quando em primeiro lugar é levado em conta a linguagem de programação (ou mesmo o computador que será adotado) é denominado de desenvolvimento bottom-up.

Na prática, os dois tipos de desenvolvimento (top-down e bottom-up) não são independentes e nem existe um compromisso formal do programador em usar um ou outro. Tudo depende dos recursos disponíveis, do ambiente em questão e até da formação do programador.

SEGUNDO NÍVEL

Modularização de grandes programas, isto é, divisão do programa em pequenas etapas, que são módulos ou subprogramas, que nada mais são que um conjunto de comandos agrupados com a finalidade de executar uma determinada função. A primeira etapa, por onde começa a execução do trabalho, recebe o nome “Programa Principal”, e as outras são os sub-programas propriamente ditos, executados sempre que ocorre uma chamada dos mesmos. Para que os sub-programas possam ser chamados, eles devem receber um nome, e a unidade que deseja chamá-lo o faz através de seu nome.

Um sub-programa por sua vez, pode chamar outro sub-programa e assim por diante.

A modularização possui várias vantagens, como por exemplo:

- Permite a divisão de tarefas em equipes, onde cada módulo pode ser elaborado por um programa diferente;

- Facilita o teste, pois cada módulo pode ser testado individualmente e depois incorporado ao conjunto;
- Permite o uso do mesmo módulo várias vezes.

TERCEIRO NÍVEL

Elaboração do programa usando dentro de cada módulo um número limitado de “estruturas básicas de fluxo de controle”, isto é, instruções.

Os módulos são construídos por programas. No programa, cada um de seus passos pertence a uma das três estruturas básicas:

- Operação elementar;
- Operação de controle (Estrutura) especificando uma seleção entre seqüências de passos;
- Operação de controle (Estrutura) especificando a repetição de uma seqüência de passos.

DOCUMENTAÇÃO:

Os programas devem ser lidos e entendidos por quem os confeccionou e por outras em toda a sua vida útil, uma vez que podem necessitar correção, manutenção e modificação. Para que isto ocorra, eles precisam ser muito bem documentados.

Um programa bem documentado possui as seguintes características:

- É convenientemente indentado de modo a mostrar sua estrutura lógica. Os comandos devem ser alinhados de acordo com o nível a que pertencem, ou seja, a estrutura na qual eles estão contidos deve estar destacada;
- Possui comentários que auxiliam a sua compreensão por outras pessoas não responsáveis por sua elaboração;
- Contém uma declaração de variáveis em um dicionário de dados. O dicionário de dados especifica formalmente o conteúdo de cada dado utilizado no corpo do programa e indica os valores que cada um deles pode assumir.

CAPÍTULO III CONSTRUÇÃO DE PROGRAMAS EM C

Objetivos

- Apresentar os tipos básicos de dados INT, FLOAT, CHAR;
- Apresentar a forma de definição das variáveis e constantes de um programa em C;
- Apresentar os comandos elementares de construção de programas em C, exemplificando seu uso;
- Apresentar uma forma organizada de construção de programas;

III.1 – Introdução

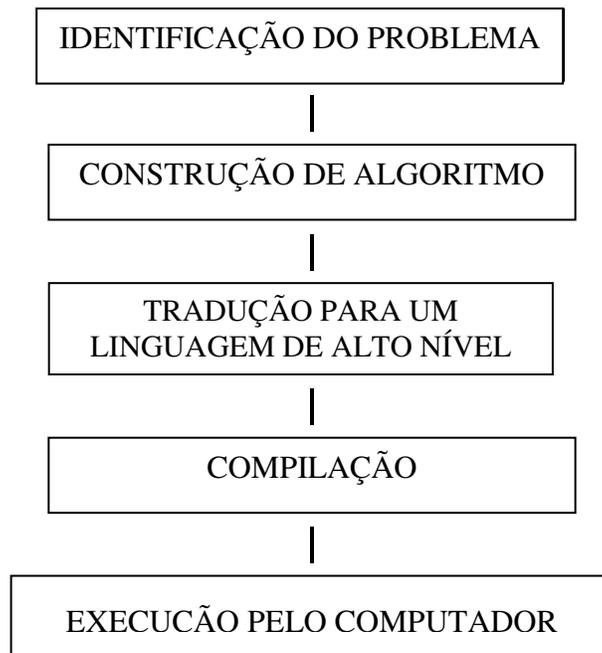
Nos capítulos anteriores mostraram-se diversos algoritmos, com suas operações mais simples sendo executadas por seres humanos (algoritmo “Troca de Pneu”). Também se discutiram as vantagens e objetivos das Técnicas de Programação Estruturada (Refinamentos Sucessivos, Modularização, Documentação, etc).

Nesse capítulo serão descritas as operações básicas que o computador pode executar. Estas operações serão utilizadas para construir algoritmos progressivamente mais complexos.

Um fator a observar é que os computadores reais, na verdade, executam um conjunto ainda mais reduzido de operações, chamadas *operações de máquina*, e que definem a *Linguagem de Máquina*.

As operações descritas aqui referem-se ao que se convencionou chamar de *Linguagens de Alto Nível*. Estas são mais próximas de linguagem humana. Para traduzir um programa escrito em linguagem de alto nível para linguagem de máquina utilizam-se outros programas denominados de *compiladores*.

Uma vez construído um algoritmo, o mesmo pode ser traduzido para uma linguagem de programação de alto nível (PACAL, COBOL, FORTRAN, C, MODULA, etc). Este último programa é então traduzido pelo compilador para linguagem de máquina, que pode ser executada pelo computador.



As operações básicas executadas em Linguagem de Alto Nível são feitas sobre tipos básicos de dados. Antes de se aprender as operações é necessário, portanto, conhecer esses tipos básicos de dados.

III.2 – Identificadores

Em geral, uma linguagem de Alto Nível possui dois tipos de elementos: os elementos definidos pela linguagem (palavras reservadas, etc) e os elementos definidos pelo programador (identificadores, comentários, etc).

Um identificador é um símbolo que pode representar uma variável, uma constante, um tipo, um nome de programa ou subprograma ou rótulo.

É formado por uma seqüência de caracteres alfanuméricos (letras e dígitos), sendo que o primeiro deve ser obrigatoriamente alfabético.

A escolha de nomes de identificadores é importante para a qualidade do programa. Um nome de identificador deve ser significativo, isto é, deve esclarecer a função que o elemento identificado exerce dentro do programa. Deste modo, o programa se torna fácil de ler, compreender e alterar.

Neste texto será adotado como regra na escrita: a) nome de identificados começados com letra maiúscula e demais caracteres minúsculos; b) nomes compostos de identificadores com a primeira letra de cada parte maiúscula, e c) palavra com letras maiúsculas. Apesar do compilador na fazer distinção entre letras maiúsculas e minúsculas.

III.3 – Tipo de Dados

As linguagens de alto nível permitem ao programador definir tipos de dados bastantes complexos, como vetores, matrizes, filas e pilhas. Todavia, todos esses tipos são compostos por outros tipos menores, com características definidas pela própria linguagem. São os tipos básicos de dados.

Os dados em um algoritmo/programa podem ser de dois tipos: **constantes** ou **variáveis**.

Constantes são valores que parecem em meio ao programa e que não têm seu valor alterado durante a execução do mesmo. Um exemplo típico é o do número π (PI).

Variáveis são elementos (posição de memória da máquina) que recebem o valor de uma variável ou de uma expressão, podendo este valor ser alterado durante a execução do programa. Uma variável pode ser imaginada como uma posição da máquina, identificada pelo “nome da variável”.

Exemplo:

Soma = 0; significa que uma posição de memória da máquina, identificada pelo nome **Soma**, recebe o valor 0.

III.3.1 – Tipo Básicos de Dados

No C, são adotados quatro tipos básicos de dados, isto é, são quatro os tipos primitivos de dados que podem ser usados. Esses tipos podem criar outros tipos chamados de “tipos construídos”, que serão vistos posteriormente.

1. TIPO INTEIRO (int)

Designa um intervalo (dependente do compilador e da máquina) dos números inteiros (negativo, positivo ou nulo). Possui como operações associadas:

- Soma: +
- Multiplicação: *
- Subtração: -
- Divisão inteira: /

2. TIPO PONTO FLUTUANTE (float)

Designa um intervalo dos números racionais (a precisão e o intervalo são dependentes da máquina e do compilador). Possui como operações associadas:

- Soma: +
- Multiplicação: *
- Subtração: -
- Divisão: /

3. TIPO CHARACTER (char)

Designa qualquer caracter (letras, números e caracteres especiais como *, &, \$, etc). Existem operações especiais associadas à manipulação de caracteres, mas não serão abordadas neste momento.

4. TIPO BOOLEANO (int)

Na linguagem C, o tipo INT é usado também como expressão booleana. Para isso, o zero é interpretado como falso e os outros números como verdadeiro.

III.4 – Definição das Variáveis de um Programa

Quando se escreve um programa, é necessário definir o tipo de todas as variáveis que serão usadas. Essa definição de tipo é feita através de uma “Declaração de Variáveis”, onde são especificados os valores que podem ser atribuídos a cada uma das variáveis (se valores inteiros, reais, etc). Em C, quando um computador está executando um programa e encontra uma variável qualquer, se esta não tiver tido o seu nome e tipo declarado anteriormente, será emitida uma mensagem de erro, abortando a execução. Assim, toda variável deve ser declarada antes de ser utilizada.

A declaração de variáveis e constantes segue a seguinte estrutura:

```
#define <identificador> <valor>  
<tipo> <identificador>;
```

Exemplo:

```
#define pi 3, 1415926
#define NotaMaxima 10
int soma, cont1, cont2, matriculaAluno;
float mediaAluno, notaProva1, notaProva2;
```

EXERCÍCIO

Assinale os identificadores válidos:

- Valor
- Salário-Liquido
- SalarioLiquido
- X2
- NotaAluno
- 3X3
- AH!!
- xy
- X234

III.5 – Operações Elementares

As operações elementares são os comandos de atribuições, cálculos de expressões, e operações de entrada e saída. Essas operações são chamadas de elementares por na alterarem o fluxo normal da execução do programa, ou seja, após a execução de uma operação elementar, a próxima instrução a ser executada é a seguinte do programa.

Toda instrução possui uma parte sintática (diz respeito à forma como a instrução é escrita) e uma parte semântica (diz respeito ao que a instrução faz).

III.5.1 – Comando de Atribuição

O comando de atribuição é a maneira de se especificar que a uma determinada variável será atribuído (fornecido) um certo valor. Este comando. Em C, é indicado pelo símbolo de atribuição “=”.

Forma geral do comando de atribuição:

$$\langle \text{variável} \rangle = \langle \text{valor} \rangle;$$

O tipo dos dados à esquerda e à direita do comando de atribuição deve ser compatível. A única exceção ocorre para casos em que um valor inteiro é atribuído a uma variável do tipo real.

III.5.2 – Cálculo de Expressão

As variáveis e constantes podem ser combinadas com os operadores associados a cada tipo de dados, gerando expressões. É o valor calculado das expressões que é atribuído à variável ao lado esquerdo do comando de atribuição.

III.5.3 – Expressão Aritmética

A expressão aritmética é aquela cujos operadores e/ou variáveis de tipo numérico.

A notação é a mesma das operações matemáticas, com as seguintes exceções:

- Não é permitido omitir o operador de multiplicação:

$A * C$ é diferente de AC

- Nas operações aritméticas, as operações guardam sempre uma relação de prioridade, que é:

Prioridade	Operação
1	Multiplicação, divisão
2	Adição, subtração

Para se obter uma seqüência de cálculos diferente, vários níveis de parênteses podem ser usados para quebrar as prioridades definidas. Não é permitido o uso de colchetes e chaves, uma vez que estes símbolos serão utilizados com outras finalidades.

- **Funções Pré-definidas**

Além das operações básicas citadas podem-se usar nas expressões aritméticas algumas funções bastante comuns na matemática, definidos dentro da linguagem.

Nome da função	Tipo do Argumento	Tipo de Retorno
abs	int	int
abs	float	float
sin	int/float	float
cos	int/float	float
atan	int/float	float
log	int/float	float
exp	int/float	float
sqrt	int/float	float
floor	float	int
ceil	float	int

Exemplos:

- ABS é o valor absoluto (módulo) de um número. Exemplos:

$$\text{abs}(-4) = 4$$

- SQRT é raiz quadrada. Exemplos:

$$\text{sqrt}(4) = 2$$

- FLOOR é o maior inteiro não maior que o número de entrada. Exemplo:

$$\text{floor}(10.9) = 10$$

- CEIL é o menor número não menor que o número de entrada. Exemplo:

$$\text{ceil}(9.9) = 10$$

- SIN, COS e ATAN são as funções trigonométricas seno, coseno e arco-tangente respectivamente.

Os ângulos são sempre dados em radianos.

- LOG e EXP são o logaritmo neperiano e exponencial respectivamente.

Exercício

Calcule o valor das seguintes expressões aritméticas, considerando os seguintes valores:

A = 10 B = 3 X = 2.5 Y = 1.2

a) X/2

b) ceil(X-Y)

c) EXP(Y*(B+2))-6

d) floor(2.9+ceil(0.3+Y)*2+A)

III.5.2.2 – Expressão Relacional

Trabalha com operadores relacionais e é formada por três partes: uma expressão (da esquerda), um operador relacional e outra expressão (da direita). Note-se que uma variável ou constante isolada pode ser entendida como uma expressão.

São exemplos de operadores relacionais:

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

As expressões relacionais têm como resultados finais valores do tipo inteiro (0 para falso e 1 para verdadeiro).

As variáveis do tipo CHAR também podem ser comparadas entre si, obedecendo à ordenação do padrão ASCII.

III.5.2.3 – Expressão Lógica

É comum nos programas surgirem situações em que a execução de uma ação, ou seqüência de subações, está sujeita a uma certa condição. Esta é representada no texto do programa por meio de uma expressão lógica.

Denomina-se de Expressão Lógica à expressão cujos operadores são lógicos e cujos operandos são expressões relacionais e/ou variáveis do tipo inteiro.

Os operadores lógicos são:

&& (AND) || (OR) ! (NOT)

Os resultados obtidos das expressões lógicas também são valores do tipo inteiro (0 para falso e 1 para verdadeiro).

Exemplos:

Se P e Q são operandos inteiras então:

P	Q	P && Q	P Q	!P	!(P && Q)
1	1	1	1	0	0
1	0	0	1	0	1
0	1	0	1	1	1
0	0	0	0	1	1

Exercícios

Dadas as variáveis e seus valores:

A = 10 B = 3 X = 3.5 Y = 0.3

L1 = 1 L2 = 0

Calcule o valor das seguintes expressões:

a) $(A < B)$

b) $(X < A)$

c) $((X + A) < (B * Y))$

d) $((X < A) \parallel (B > Y))$

e) $L1 \ \&\& \ L2$

f) $L1 \parallel L2$

g) $(L1 \ \&\& \ (X < Y))$

h) $((! L2) \parallel (! (X < (Y + 2))))$

- **Prioridade de Cálculo das Expressões**

Caso se elimine os parâmetros das expressões dos exercícios anteriores, podem-se criar dúvidas sobre que operações serão executadas primeiro. A tabela a seguir apresenta a ordem de prioridade de cálculo de expressões.

Prioridade	Operador
1	!
2	*, /, %, &&
3	+, -,
4	==, !=, <, <=, >=, >

III.5.3 – Operações de Entrada de Dados

O computador armazena os valores das variáveis (e o próprio programa!) em sua memória. O computador alcança estes valores do ambiente externo através de operações de entrada de dados. A operação de entrada de dados transfere um ou mais dados do ambiente externo (teclado, disquete) para uma ou mais variáveis na memória, funcionando como um comando de atribuições. Os dados lidos devem ser do mesmo tipo das variáveis especificadas no comando de leitura. A forma geral de descrição desta operação é:

```
scanf("<formato1> <formato2> ... <formatoN>", &var1, &var2, ..., &varN);
```

A tabela a seguir nos mostra alguns dos possíveis formatos a serem usados:

Código	Significado
%c	Lê um único caracter
%d	Lê um inteiro decimal
%i	Lê um inteiro decimal
%e	Lê um número em ponto flutuante
%f	Lê um número em ponto flutuante
%g	Lê um número em ponto flutuante
%o	Lê um número octal
%s	Lê uma string
%x	Lê um número hexadecimal
%p	Lê um ponteiro
%n	Recebe um valor inteiro igual ao número de caracteres lidos até então
%u	Lê um número sem sinal
%[]	Busca por um conjunto de caracteres

Exemplo:

Tome o seguinte trecho de um programa em escrito em C.

```
scanf(“%d %d %d”, &X, &Y, &Z);
```

```
scanf(“%d”, &W);
```

Simulação da execução:

TELA	“MEMÓRIA”
	X = ?, Y = ?, Z = ?, W = ?
10 20 30	X = 10, Y = 20, Z = 30, W = ?
40	X = 10, Y = 20, Z = 30, W = 40

III.5.4 – Operações de Saída de Dados

Quando o computador termina de processar as informações e obtém os valores finais (por exemplo, o salário líquido), os valores são automaticamente armazenados em posições de memória (valores de variáveis). Entretanto, o usuário não tem acesso a estes valores.

Para permitir o acesso do usuário aos resultados, o programador deve ordenar sua apresentação. Esta pode ser em vídeo ou em impressoras.

A operação de saída de dados transfere para o mundo exterior um ou mais valores ou variáveis desejadas. Apresenta a seguinte forma geral:

```
printf(“<formato1> <formato2> ... <formatoN>”, var1, var2, ..., varN);
```

A tabela a seguir nos mostra alguns dos possíveis formatos a serem usados:

Código	Formato
%c	caracter
%d	Inteiros decimais com sinal
%i	Inteiros decimais com sinal
%e	Notação científica (e minúsculo)
%E	Notação científica (E maiúscula)
%f	Ponto flutuante decimal
%g	Usa %e ou %f, o que for mais curto
%G	Usa %G ou %F, o que for mais curto
%o	Octal sem sinal
%s	String de caracteres
%u	Inteiros decimais sem sinal
%x	Hexadecimal sem sinal (letras minúsculas)
%X	Hexadecimal se sinal (letras maiúsculas)
%p	Apresenta um ponteiro
%n	O argumento associado é um ponteiro para inteiro no qual o número de caracteres escritos até esse ponto é colocado
%%	Escreve o símbolo %

Exemplo:

Tome o seguinte trecho do programa:

```
printf(“Entre com dois numeros inteiros: ”);
```

```
scanf("%d %d", &num1, &num2);
printf("%d ", num1);
printf("%d\n", num2); //o '\n' faz com que ele pule para a próxima linha
printf("Soma: %d", num1+num2);
printf("\n");
printf("Fim Programa");
```

Simulação da execução:

TELA

```
Entre com dois números inteiros:1 2
1 2
Soma: 3

Fim Programa
```

III.6 – Um Primeiro Programa

Um aluno do curso de Construção de Programas deseja calcular a sua média final, conhecidas as suas três notas parciais. Escreva um programa computacional que executa esta operação.

OBS.: As notas parciais podem ser fracionárias; a média final é a média aritmética das três notas parciais.

```
main(){

    float nota1, nota2, nota3, media_final;

    scanf("%f %f %f", &nota1, &nota2, &nota3);

    media_final = (nota1 + nota2 + nota3) / 3;    //aqui é feito o cálculo da média

    printf("A Media Final eh: %d", media_final);

}
```

Esse exemplo já permite identificar as partes de um programa. São elas: cabeçalho, dicionário de dados, corpo e comentários.

III.6.1 – Dicionário de Dados

No dicionário de dados se definem todas as variáveis e constantes utilizadas no programa. Também os nomes das variáveis devem ser auto-explicativos. No caso de não ficar explícito a função da variável com o nome, deve-se utilizar comentários explicativos.

A forma de declaração de dados já foi definida anteriormente.

III.6.2 – Corpo

È onde se descrevem as operações que se devem efetuar sobre os dados (entradas, atribuições, cálculo de expressões e saídas) de forma a obter informações desejadas. No caso de algum trecho do programa não ter uma leitura imediata e fácil, mesmo por quem não o construiu, também dev-se utilizar comentários explicativos.

III.6.3 – Corpo

Os comentários são a última parte do programa, mas não a menos importante. Como se explicou, os comentários se localizam em todas as partes do programa (cabeçalho, dicionário de dados e corpo). Sua função é explicar a função de cada operação cuja utilidade não esteja clara a uma leitura imediata. Também o programa como um todo merece um conjunto de comentários que descrevem sua utilidade, o que se faz com um grupo de comentários associados ao cabeçalho do programa.

Em qualquer parte do programa, pode se identificar os comentários por estarem escritos entre /* e */ ou à frente de //. Aquelas funcionam como delimitadores dos comentários.

Em exemplos pequenos como o acima, a utilização de nomes mnemônicos nas variáveis é suficiente para garantir a legibilidade do programa. Em programas maiores, contudo, isto pode não ocorrer: neste caso a utilização de comentários é essencial.

O critério para o uso ou não de comentários é subjetivo, mas pode ser bem traduzido pelo seguinte questionamento: *‘Será que um programador médio, não tão capaz e inteligente quanto eu mesmo, será capaz de ler e entender cada linha de código escrito com a mesma facilidade? Será que eu serei capaz de entender a função de cada variável e cada linha de código daqui a um ano, às 21:00 horas de uma sexta-feira de carnaval, quando este programa der um defeito e eu for o responsável pela sua correção?’*

Muitos programadores acostumados a restrições de linguagens antigas têm o mau costume de definir nomes de variáveis muito pequenos e pouco explicativos (uma ou duas letras), também só incluem comentários quando são diretamente obrigados pela sua chefia, e isto após a escrita do seu programa, quando já esqueceram boa parte das funções do programa e suas variáveis.

A experiência demonstra que o tempo gasto em escrever nomes maiores e mais explicativos é bem menor que aquele gasto tentando lembrar a função e o tipo de uma variável de nome XY2 na milésima primeira linha de código, ou procurando a definição desta variável na folha de número 257 em um bloco de listagem com 500 páginas.

Na mesma linha de raciocínio, os comentários podem e devem ser definidos durante a construção do programa. Ao utilizar as técnicas de programação estruturada, o programador terá todos os comentários apropriados a cada parte do programa antes de iniciar seu detalhamento final.

- **Comentários no cabeçalho**

Um grupo de comentários são aqueles relativos ao conjunto do programa. Antes de algumas informações sobre o autor e a data de construção do programa, existem 5 comentários fundamentais em qualquer programa:

- Função: deve-se descrever o objetivo do programa com um texto conciso;
- Entradas: descrevem-se todos os dados cuja leitura é necessária durante a execução do programa;
- Saídas: descrevem-se todas as saídas de resultados obtidos ao longo do programa;
- Condição de parada: muitas vezes os programas executam tarefas repetitivas. A não ser que haja uma condição de parada estipulada, o programa continuaria indefinidamente. Esta condição deve ser explicitada nos comentários do cabeçalho;

- Restrições: alguns programas exigem que os dados estejam dispostos de uma forma particular ou com alguma característica em particular. Este é o caso de dados cuja entrada deve ser ordenada por alguma chave ou dados que não podem ter valor nulo. Caso haja uma ocorrência deste tipo em programa, esta restrição deve ser incluída nos comentários do cabeçalho.

Exemplos:

3)Escreva um programa que calcule a renda líquida de uma pessoa, conhecidas a renda bruta, o percentual de impostos e a parcela a deduzir de uma determinada pessoa física. Imprima o resultado junto com o número de CPF que também são dados de entrada.

```
main(){
    /*Autor: Pedro II
    Data: 13 de maio de 1888
    Função: calcular a renda liquida de um contribuinte.
    Entrada: numero do CPF, renda bruta, percentual de impostos e parcela a deduzir.
    Saída: numero do CPF e receita liquida.*/

    float rendaLiquida, rendaBruta, parcelaDeducao, impostoFinal, percentualImposto;
    int numCPF;

    scanf("%d %f %f %f", &numCPF, &percentualImposto, &parcelaDeducao);

    impostoFianl = rendaBruta * (percentualImposto / 100) – parcelaDeducao;

    receitaLiquida = receitaBruta – impostoFinal;

    printf("%d %f", numCPF, receitaLiquida);

}
```

III.7 – Comandos Estruturados

III.7.1 – Seqüência de Comandos

Uma seqüência finita de comandos onde cada instrução é executada uma vez, um após o outro sem desvio. A seqüência é delimitada por '{' e '}', com cada comando separado por ';'.

```
{
    <comando1>;
    <comando2>;
    .
    .
    .
    <comandoN>;
}
```

Um programa é um exemplo de uma seqüência de comandos.

III.7.2 – Estrutura de Seleção

Quando uma ação ou conjunto de ações dentro do programa depender de uma teste, que decide pela sua execução ou não, deve-se utilizar o comando (estrutura) de seleção.

Comando if / else

Sua forma geral é:

```
if (expressão)
    Comando1
else
    Comando2
```

O *comando1* só será executado se a *expressão* for verdadeira. Caso contrário será executado o *comando2*.

Jamais ambas expressões de comandos serão executadas, e também nunca teremos nenhuma expressão sem ser executada.

Exemplo:

Calcular a média final de um aluno do curso de programação a partir de suas três notas parciais. Caso sua nota final seja inferior (superior) a 6.0, imprima uma aviso de que o aluno foi reprovado (aprovado).

Algoritmo em português:

- Leitura das notas
- Cálculo das médias
- Se média inferior a 6.0: aluno reprovado
- Senão: aluno aprovado

Algoritmo traduzido para C.

```
main(){

    int matricula;
    float n1, n2, n3, mediaFinal;

    printf("Entre com as notas parciais: ");

    scanf("%d %f %f %f", &matricula, &n1, &n2, &n3);

    mediaFinal = (n1 + n2 + n3) / 3.0;

    if(mediaFinal < 6.0){
        printf("O aluno %d foi reprovado.");
    }
    else{
        printf("O aluno %d foi aprovado.");
    }
}
```

```
}

```

Observe que os delimitadores '{' e '}' foram omitidos nas cláusulas if / else, isto porque sempre que uma seqüência for composta por apenas um comando os delimitadores são opcionais, porém é sempre bom colocá-los. O ';' nunca é colocado antes da cláusula else, visto que o ';' representa o fim do comando if e o else nada mais é que a continuação do comando if / else.

A cláusula else é opcional, podendo a estrutura reduzir-se a:

```
if (expressão)
    Comando
```

Neste caso, se a condição for falsa o comando seguinte ao comando if será executado.

Exemplo:

Ler dois números e ordená-los.

```
main(){

    int num1, num2, aux;

    printf("Entre com os números: ");

    scanf("%d %d", &num1, &num2);

    printf("Valores desordenados: %d %d.\n", num1, num2);

    if(num1 > num2){
        aux = num1;
        num1 = num2;
        num2 = aux;
    }

    printf("Valores ordenados: %d %d.\n", num1, num2);

}
```

O programa lê dois números e faz a troca, ordenando-os, caso num1 seja maior que num2. Caso contrário a troca não é executada e o comando seguinte ao if é executado.

OBS.: Para que uma variável participe de uma operação de comparação é necessário que inicialmente à ela seja atribuído algum valor, através de leitura ou atribuição.

Programa MaiorDeTres:

```
main(){
    /* Funcao: le tres números e identificar o maior.
       Entrada: tres numeros inteiros.
       Saida: o maior dos tres numeros inteiros. */

    int n1, n2, n3;
```

```
printf("Entre com os tres numeros: ");
scanf("%d %d %d", &n1, &n2, &n3);

if((n1 > n2) && (n1 > n3)){
    printf("O maior eh: %d.", &n1);
}

if((n2 > n1) && (n2 > n3)){
    printf("O maior eh: %d.", &n2);
}

if((n3 > n1) && (n3 > n2)){
    printf("O maior eh: %d.", &n3);
}

}
```

Note que este programa embora funcione não é eficiente, pois caso o número n1 seja maior mais dois teste são feitos desnecessariamente.

Um a segunda versão mais eficiente utilizando comandos de seleção aninhados é dada no seguinte exemplo:

```
main(){
    /* Funcao: le tres números e identificar o maior.
    Entrada: tres numeros inteiros.
    Saida: o maior dos tres numeros inteiros. */

    int n1, n2, n3;

    printf("Entre com os tres numeros: ");

    scanf("%d %d %d", &n1, &n2, &n3);

    if (n1 > n2){
        if (n1 > n3){
            printf("O maior eh: %d.", n1);
        }
        else {
            printf("O maior eh: %d.", n3);
        }
    }
    else{
        if (n2 > n3) {
            printf("O maior eh: %d.", n2);
        }
    }
}
```

```

        else {
            printf("O maior eh: %d.", n3);
        }
    }
}

```

Comando Switch

Sua forma geral é:

```

switch (expressão) {
    case valor1: comando1
    case valor2: comando2
        .
        .
        .
    case valorN: comandoN
    default : comando
}

```

O comando switch permite que seja realizada uma entre as n seqüências de operações, dependendo do valor do seletor.

O valor do seletor é o comparado com cada valor de seqüência de valores. Se algum dos valores comparados for igual ao valor do seletor a seqüência de operações a ele relacionada será executada.

O caso intitulado default é executado se nenhum outro for satisfeito. O default é opcional.

Exemplo:

Ler uma vogal e exibir a vogal subsequente. Considere letras maiúsculas.

```

main(){
    /* Funcao: ler uma vogal e exibir a vogal subsequente
    Entrada: uma vogal
    Saida: uma vogal subsequente */

    char cor;

    printf("Entre com uma vogal maiuscula: ");

    scanf("%c", &cor);

    switch (cor) {
        case 'A': cor = 'E';
        case 'E': cor = 'I';
        case 'I': cor = 'O';
        case 'O': cor = 'U';
        case 'U': cor = 'A';
    }

    printf("Vogal subsequente: %c", cor);
}

```

```
}
```

Exemplo:

Ler mês e ano e exibir o número de dias do mês / ano digitado.

```
main(){  
  
    int dia, mes, ano;  
  
    printf("Entre com o mes: ");  
  
    scanf("%d", &mes);  
    scanf("%d", &ano);  
  
    switch (mes) {  
        case 1: case 3: case 5: case 7: case 8: case 10: case 12: dia = 31;  
        case 4: case 6: case 9: case 11: dia = 30;  
        case 2: {  
            if (ano % 4 == 0){  
                dia = 29;  
            }  
            else {  
                dia = 28;  
            }  
        }  
    }  
  
    printf("O mes %d do ano %d possui %d dias.", mês, ano, dia);  
  
}
```

III.7.3 – Estrutura de Repetição

Quando um conjunto de ações é executado repetidas vezes, tem-se uma “repetição”.

A estrutura de repetição, assim como a estrutura de seleção, envolve sempre a avaliação de uma condição. Existem três tipos básicos de repetição em C: “while”, “do-while” e “for”.

Comando while

Sua forma geral é:

```
while(expressão)  
    comando
```

A estrutura while é usada para repetir “n” vezes uma ou mais instruções, tendo como vantagem o fato de não ser necessário conhecer o valor de “n” (número de repetições).

Antes da seqüência ser executada, a condição é avaliada. Se na primeira avaliação a condição for falsa a seqüência de operações não é executada uma única vez. Caso a condição seja verdadeira a seqüência de operações é executada repetidamente até que a condição se torne falsa.

Logo, em algum momento esta condição deve ser modificada dentro do laço while, caso contrário a seqüência será executada indefinidamente provocando um loop no programa.

Exemplo:

Dado um número externo, imprimir a tabuada desse número, de zero a nove:

```
main(){  
  
    int numero, multiplicador, produto;  
  
    scanf("%d", &numero);  
  
    multiplicador = 0;  
  
    while (multiplicador <= 9){  
        produto = numero * multiplicador;  
        printf("numero * multiplicador = %d", produto);  
        multiplicador = multiplicador + 1;  
    }  
  
}
```

Neste exemplo o valor da variável multiplicador é modificado a cada iteração do laço while. A seqüência de operações do comando while é executada até que o valor do multiplicador seja maior que 9, isto é, a condição se torne falsa.

Comando do-while

Assim como o comando while, este comando é usado quando não se conhece o número de vezes que uma seqüência de comandos será executada.

Sal forma geral é:

```
do  
    comando  
while (expressão);
```

O do-while testa avalia a expressão no final, depois de fazer cada passagem pelo corpo do laço, por tanto o corpo é executado ao menos suma vez.

Exemplo:

1) Dada a série 2, 3, 4, 9, 16, 29, 54, 99... onde cada elemento a partir do quarto é a soma dos três termos que lhe antecedem, faça um programa que leia três termos iniciais quaisquer, e a partir daí obtenha e imprima mais 20 termos da referida série.

```
main(){  
  
    int num1, num2, num3, num, n;  
  
    printf("Entre com os tres primeiros termos: ");
```

```

scanf("%d %d %d", &num1, &num2, &num3);

printf("%d %d %d ", num1, num2, num3);

n = 0;

do {
    num = num1 + num2 + num3;
    printf("%d ", num);
    num1 = num2;
    num2 = num3;
    num3 = num;
    n = n + 1;
} while (n < 20);

}

```

2)

```

main(){

    char resp;

    do {
        printf("deseja continuar? – S/N \n");
        resp = getchar();
    } while ((resp != 'S') || (resp != 'N'));

}

```

Neste exemplo novos caracteres são lidos até que seja teclado 'S' ou 'N'.

Comando for

O comando for é útil quando se deseja realizar uma seqüência de operações um número fixo e conhecido de vezes.

Apresenta-se da seguinte forma:

```

for (<inicialização>; <condição>; <incremento>)
    comando

```

A inicialização é são os comandos de atribuições feitas às variáveis desejadas. A condição é uma expressão relacional que determina quando o laço acaba. O incremento define como as variáveis de controle variam com cada passo do laço.

No exemplos

```

for (i = 1; i <= 10; i++)
    printf("%d \n", &i);

```

e

```

for (i = 10; i >= 1; i--)
    printf("%d \n", &i);

```

Ocorrem 10 repetições. No primeiro for a variável de controle recebe inicialmente o valor 1 e é incrementada automaticamente de 1 a cada repetição até que o primeiro valor acima de 10 seja obtido. Isto ocorre porque antes da seqüência ser executada, o valor da <inicialização> é atribuído à variável de controle e a cada iteração seu valor é comparado na <condição>, a seqüência de comando será executada se a variável de controle satisfizer a <condição>. No segundo for a variável é decrementada de 1.

A variável de controle deve ser do tipo enumerável e deve ser declarada no dicionário de dados.

Exemplos:

1) Ler 3 notas e imprimir a média.

```
#define numNotas 3

main(){

    int i;
    float nota, soma, media;

    soma = 0;

    for (i = 0; i < numNotas; i++){
        printf("Entre com a nota: ");
        scanf("%f", &nota);
        soma = soma + nota;
    }

    media = soma / numNotas;

    printf("A media eh: %f", media);

}
```

2) Ler n notas e imprimir a média

```
main(){

    int n, i;
    float nota, soma;

    soma = 0;

    printf("Entre com o numero de notas: ");

    scanf("%d", &n);

    for (i = 0; i < n; i++){
```

```
        printf("Entre com a nota: ");
        scanf("%d", &nota);
        soma = soma + nota;
    }

    if (n > 0){
        printf("A media eh : %f", soma / n);
    }

}
```

III.8 – Um Algoritmo para a construção de Programas

Os estudos da psicologia sobre a memória humana apontam que uma pessoa com capacidade de memorização mediana consegue definir, de forma imediata, cerca de 8 “fatos” ao mesmo tempo, isto é, apenas 8 “fatos” ou “valores” podem ser recordados imediatamente se a pessoa for questionada sobre eles.

Para poder recordar mais fatos simultaneamente, existem dois caminhos: 1) a pessoa deve “decorar” os valores, de forma a “acessá-los” de forma automática; 2) deve ser dado um tempo para que a pessoa consiga “lembra” o valor solicitado.

Na construção de programas, mesmo para a resolução de programas simples como os estudantes aqui, a quantidade de “fatos” a se recordar ultrapassa facilmente estes limites. Além disso, geralmente não se dispõe de tempo para se decorar cada uma das entradas e / ou saídas do problema, suas variáveis, a estrutura do algoritmo (uso de repetições e seleções), etc.

Um dos objetivos das técnicas de programação estruturada é permitir a pessoas normais o desenvolvimento de programas (e, por conseguinte, de algoritmos) com boa produtividade. Como já foi apontado anteriormente, duas ferramentas são fundamentais para este processo: as técnicas de refinamentos sucessivos e de representação de algoritmos.

Ao se desenvolver pequenos algoritmos, estas ferramentas podem ser dispensadas, porque a memória é capaz de tratar todos os elementos (variáveis e estruturas de seleção e repetição, por exemplo) programador sente dificuldades em trabalhar nesta maneira pouco metódica. A seguir, apresenta-se uma estratégia para auxiliar o desenvolvimento de programas;

Metodologia para o Desenvolvimento de Programas

- 1) Ler especificação do problema até o final, de forma a compreender de que se trata o assunto;
- 2) Ler novamente, tomando o cuidado de fazer e ressaltar os pontos fundamentais (entrada, saída, restrições e condições de parada);
- 3) Montar o cabeçalho do programa, inclusive (e principalmente!) os comentários pertinentes: função, entrada, saída, restrições e condições de parada.
- 4) Escrever em português, da forma que desejar, a seqüência de ações necessárias a obtenção dos resultados desejados. Importante: evite, neste momento, a utilização de nomes de variáveis. Utilize a técnica de refinamentos sucessivos para transformar procedimentos complexos em outros mais simples, até obter um texto baseado apenas em ações simples, estruturas de seleção e repetição. Ao fim desse passo, deverá ser possível visualizar as estruturas de repetição e seleção necessárias ao algoritmo final.
- 5) Monte o dicionário de dados do programa, baseando-se no seguinte roteiro:
 - 5.1) defina as constantes existentes no problema (ex: número de elementos de um conjunto, valores máximos ou mínimos de algum dado, etc.);

5.2) defina as variáveis necessárias para receber os valores de entrada (uma variável para cada dado de entrada). Estas entradas já estão disponíveis nos comentários do cabeçalho;

5.3) defina as variáveis de saída necessárias para escrever os resultados desejados. Novamente, os comentários do cabeçalho servirão de base para este passo;

5.4) defina as variáveis necessárias ao cálculo (o algoritmo em português pode ser útil para esta tarefa). Por exemplo: para calcular a média de um conjunto de valores, necessita-se do valor acumulado destes valores e do número de elementos do conjunto;

5.5) defina as variáveis necessárias ao controle de fluxo de processamento do programa (o algoritmo em português vai ajudar). Por exemplo: se o programa deve parar ao processar o centésimo elemento de entrada, então será necessária uma variável para controle do número de elementos processados (um contador). As variáveis de controle estão sempre associadas às condições testadas nas estruturas de seleção e repetição do problema, portanto, utilize o algoritmo em português para auxiliar também este passo.

6) Comece a traduzir o algoritmo em português escrito em 4 partes para o C, utilizando as constantes e variáveis descritas no dicionário de dados montado no passo 5.

7) Verifique se todas as variáveis foram devidamente inicializadas (veja observações abaixo). Caso não tenham sido, identifique o momento ideal de sua inicialização e inclua-a no algoritmo escrito no passo 6.

8) Verifique a correção do programa através de um ou mais testes de mesa (“teste chinês”).

OBSERVAÇÕES:

- RESISTA a tentação de saltar as fases desta metodologia. Criando o hábito de utilizá-la, você poderá identificar adaptações que se ajustem melhor a sua forma de trabalho, mas, ACREDITE, todos os passos descritos acima são necessários para a construção de um programa.
- O cabeçalho do programa, incluindo seus comentários (função, entrada, saída, parada e restrição) deve ser escrito antes do dicionário de dados e corpo do programa. NÃO OS IMAGINE COMO ACESSÓRIOS OU PERDA DE TEMPO. ELES TE AJUDARÃO NO RESTANTE DE SEU TRABALHO!!!
- Os programas básicos descritos neste capítulo e nos capítulos subseqüentes vão auxiliá-lo a definir a estrutura do problema. Estude-os com atenção!
- Sempre que identificar uma estrutura de repetição, haverá uma ou mais variáveis associadas à sua condição de parada. Verifique, então, que: a) estas variáveis tenham recebido um valor inicial antes do início da repetição; b) em algum ponto da repetição o valor de pelo menos uma destas variáveis é atualizado de maneira a tornar falsa a condição de continuidade da repetição.
- Utilize nomes de variáveis que descrevam a sua função; nunca use a mesma variável para executar duas funções diferentes.

III.9 – Programas Exemplos

A) Calcular as raízes de uma equação do segundo grau, lidos os valores dos coeficientes A, B e C:

Algoritmo:

- Leitura de valores
- Cálculo de delta
- Se delta é maior ou igual a zero: Calcular as raízes reais
- Senão: Não há raízes reais

- Calcular as raízes reais:
 - Se delta é igual a zero: calcular raiz única.
 - Se não: calcular raízes distintas

```
main(){
  /*Funcao: Calcular as raízes reais de uma equação de segundo grau.
  Entrada: os coeficientes A, B e C
  Saída: as raizes reais da equação*/

  float A, B, C, delta, raizDelta, raiz1, raiz2;

  Scanf("%f %f %f", &A, &B, &C);

  //calcular delta
  delta = B * B - 4 * A * C;

  if (delta >= 0){
    if (delta == 0){
      raiz1 = (-1) * B / (2 * A);
      printf("As duas raizes sao iguais a: %f.", raiz1);
    }
    else{
      raizDelta = sqrt(delta);
      raiz1 = ((-1) * B + raizDelta) / (2 * A);
      raiz2 = ((-1) * B - raizDelta) / (2 * A);
      printf("As duas raizes são: %f e %f", raiz1, raiz2);
    }
  }
  else {
    printf("Não existem raizes reais desta esquacao!");
  }
}
```

B) Imprimir a primeira, segunda e terceira potências dos números pares entre 2 e 50, inclusive:

```
main(){

  int numero, potencia1, potencia2, potencia3;

  numero = 2;

  while (numero <= 50){
    potencia1 = numero;
    potencia2 = potencia1 * numero;
    potencia3 = potencia2 * numero;
  }
}
```

```

        printf("%d %d %d", potencia1, potencia2, pontecia3);
        numero = numero + 2;
    }
}

```

C) Imprimir da primeira a vigésima potência dos números inteiros pares entre 2 e 50, inclusive:

```

main(){
    int numero, potencia, i;

    numero = 2;

    while (numero <= 50){
        potencia = 1;
        for(i=1; i<=20; i++){
            potencia = potencia * numero;
            printf("A %d .a potencia de %d eh %d", i, numero, potencia);
        }
        numero = numero + 2;
    }
}

```

D) Calcular a soma de uma progressão aritmética de 50 termos, sendo lidos o primeiro termo e a razão:

```

#define numTermos 50

main(){
    /*Condição de parada: ter somado os 50 termos da PA*/

    float primeiroTermo, razao, soma, termo;
    int contaTermos;

    scanf("%f %f", &primeiroTermo, razao);

    contaTermos = 1;

    termo = primeiroTermo;

    soma = primeiroTermo;

    do {
        termo = termo + razao;
        soma = soma + temo;
    }
}

```

```

        contaTermos = contaTermos + 1;
    } while(contaTermos != 50);
}

```

E) Calcular o Máximo divisor comum entre dois números:

O MDC de dois números pode ser obtido escolhendo o maior deles e subtraindo-lhe o valor do menos.

Esta operação é repetida até que os dois sejam iguais, cujo valor será o MDC dos números iniciais:

```

33 15      45 18
18 15      27 18
03 15      09 18
03 12      09 09
03 09      MDC = 09
03 06
03 03
MDC = 03

```

```

main(){

    int numero1, numero2, auxiliar1, auxiliar2;

    scanf("%d %d", &numero1, &numero2);

    auxiliar1 = numero1;
    auxiliar2 = numero2;

    while (auxiliar1 != auxiliar2){
        if (auxiliar1 > auxiliar2){
            auxiliar1 = auxiliar1 - auxiliar2;
        }
        else {
            auxiliar2 = auxiliar2 - auxiliar1;
        }
    }

}

```

F) Cálculo do maior número dentre N lidos.

Dado um número N do meio externo, representando a quantidade de número a serem lidos, o maior desses números é encontrado através do seguinte programa:

```

main(){
    /*Restricao: o numero N não pode ser negativo ou zero
    Parada: apos a leitura do N-esimo numero*/

    int n, i;

```

```

float numero, maior;

printf("Entre com o numero de termos a serem lidos: ");
scanf("%d", &n);

printf("Entre com o primeiro numero: ");
scanf("%f", &numero);

maior = numero;

for(i=2; i<=n; i++){
    printf("Entre com proximo numero: ");
    scanf("%f", &numero);
    if(numero > maior){
        maior = numero;
    }
}

printf("Maior = %f", maior);

}

```

G) Cálculo do maior número positivo sem saber quantos são previamente.

Dada uma quantidade indefinida de valores positivos a serem lidos, o programa a seguir determina qual é o maior deles.

```
#define delimitador -1
```

```

main(){
    /*Restricao: os números devem ser inteiros positivos
    Parada: ao ser lido o valor -1*/

    int numero, maior;

    printf("Entre com o primeiro numero: \n");
    printf("Digite -1 para encerrar a entrada de dados\n");

    scanf("%d", &numero);

    maior = numero;

    while (numero != delimitador){
        if (numero > maior){
            maior = numero;
        }
        printf("Entre com o próximo numero: \n");
        printf("Digite -1 para encerrar a entrada de dados.\n");
    }
}

```

```
        scanf("%d", &numero);
    }

    if (maior !=delimitador){
        printf("O maior numero eh: %d", maior);
    }
    else {
        printf("Não foi lido nenhum numero positivo");
    }
}
}
```

Lista de exercícios

- 1)Construa um programa que apresente os 100 primeiros números ímpares.
- 2)Construa um programa que calcule e apresente todos os divisores de um número lido.
- 3)Construa um programa que apresente os 200 primeiros números primos.
- 4)Construa um programa que apresente os 100 primeiros números de uma Progressão Geométrica, sendo lidos o primeiro termo e a razão.
- 5)Fazer um programa que calcule e escreva uma tabela de graus centígrados em função de graus farenheit que variam de 50 a 150 de 1 em 1.
- 6)Um comerciante deseja fazer o levantamento do lucro das mercadorias que ele comercializa. Para isto, ele reuniu as seguintes informações: número de mercadorias, preço de compra e preço de venda. Fazer um programa que:
 - Determine e escreva quantas mercadorias proporcionam:
lucro < 10%
10% < lucro < 20%
lucro > 20%
 - Determine e escreva o valor total de compra e de venda de todas as mercadorias, assim como o lucro total.
- 7)Suponha que a população de um país A seja de 9.000 habitantes com uma taxa anual de crescimento de 3% e que a população de um país B seja, aproximadamente, de 20.000 de habitantes com uma taxa anual de crescimento de 1,5%, fazer um programa que calcule e escreva número de anos necessários para que a população de país A ultrapasse ou se iguale à população do país B, mantidas estas taxas de crescimento.
- 8)Uma certa firma fez uma pesquisa de mercado para saber se as pessoas gostaram ou não de um novo produto lançado no mercado. Para isso, obteve, para cada pessoa entrevistada, informações a respeito do sexo do entrevistado e sua resposta (S = Sim ou N = Não). Sabe-se que foram entrevistados 2000 pessoa, fazer um programa que calcule e escreva:

- O número de pessoas que responderam sim
- O número de pessoas que responderam não
- A porcentagem de pessoas do sexo feminino que responderam sim
- A porcentagem de pessoas do sexo masculino que responderam não

9)O sistema de avaliação de uma determinada disciplina obedece aos seguintes critérios:

- Durante o semestre são dadas três notas.
- A nota final é obtida pela média aritmética das notas dadas.
- É considerado aprovado o aluno que obtiver nota final superior ou igual a 60 e que tiver comparecido a um mínimo de 40 aulas.

Fazer um programa que:

- Leia um conjunto de dados contendo o número de matrícula, as três notas e frequência (número de aulas frequentadas) de 100 alunos
- Calcule:
 - a) a nota final de cada aluno
 - b) a maior e a menor nota da turma
 - c) a nota média da turma
 - d) o total de alunos reprovados
 - e) a porcentagem de alunos reprovados por frequência insuficiente
- Escreva:
 - a) para cada aluno, o número de matrícula, a frequência, a nota final e o código (A = aprovado e R = reprovado).
 - b) o que foi calculado de (a) a (e).

10)Escreva um programa que encontre o maior e o segundo maior número de uma série de números lidos do teclado.

11)Faça um programa para ler uma seqüência de números inteiros positivos (um por vez), e verificar se eles são pares ou ímpares.

12)Dada a série de Fibonacci

$$S = 2 \ 3 \ 5 \ 8 \ 13 \ 21 \ 34 \ 55 \ \dots$$

Escreva um programa que gere essa série até o n-ésimo termo (n será lido).

Dica:

- leia os dois primeiros termos,
- esta série é tal que cada número a partir do terceiro é a soma dos dois anteriores.

13)Escreva um programa para ler 50 dados do tipo base e altura, que contém as medidas de diferentes retângulos. Imprimir o perímetro e a área de cada retângulo.

14)Escreva um programa para calcular o valor de π usando a fórmula:

$$\pi = \sqrt{\sum}$$

refazer essa parada!!!!!!

Escreva um programa para ler uma seqüência de 30 códigos de vendedores de uma firma comercial, juntamente com os respectivos totais de vendas diárias e verificar quantos deles estão na faixa compreendida entre R\$ 500,00 e R\$ 1500,00. Imprimir uma lista listagem contendo todo os códigos dos vendedores situados no intervalo especificado, e a quantidade de vendedores neste intervalo.

16) Leia uma lista de dados contendo o número da conta e o saldo médio dos correntistas de uma caderneta de poupança. Imprimir uma listagem contendo o total de:

- a) clientes bons: $500,00 \leq \text{SALDO MÈDIO} \Rightarrow 1000,00$
- b) clientes razoáveis: $\text{SALDO MÈDIO} < 500,00$
- c) clientes prioritários: $\text{SALDO MÈDIO} > 1000,00$

Imprima também a lista dos clientes prioritários. O flag de saída para o comando de repetição será referente ao SALDO MÈDIO, sairá quando este for igual a zero.

17) Um serviço de correspondência sentimental deseja obter informações sobre seus clientes. Cada cliente forneceu as seguintes informações:

MATRICULA, SEXO ('M' ou 'F'), IDADE, ALTURA (em m), PESO (em Kg), COR DOS OLHOS (1-azul, 2-castanhos, 3-outros), COR DOS CABELOS (1-castanhos, 2-loiros, 3-outros).

Escrever um programa que leia estes dados e imprima as matrículas de:

- a) todos os homens loiros de olhos azuis e altura maior que 1.75 m, pesando entre 75 Kg e 95 Kg.
- b) Todas as mulheres de olhos castanhos, altura entre 1.65 m e 1.75 m, pesando menos de 60 Kg.

Utilizar um flag para indicar o final da entrada de dados.

18) Escreva um programa para ler uma seqüência de dados contendo a matrícula e a altura dos funcionários de uma empresa. Obter uma listagem para o Departamento de Educação Física e Desportos contendo a matrícula e a medida de todos funcionários altos (acima de 1.80 m). Imprimir a altura média dos funcionários e o total de:

- Funcionário baixos (altura inferior a 1.65 m);
- Funcionários de altura mediana ($1.65 \text{ m} \leq \text{ALTURA} \leq 1.80 \text{ m}$).

19) Escreva um programa para ser fornecido ao DETRAN, para provimento das informações abaixo descritas, referentes a acidentes em estradas:

- a) Percentagem de motoristas não-habilitados;
- b) Percentagem de homens no volante;
- c) Percentagem de motoristas inexperientes (menos de 20 anos).

Os dados de entrada contêm o ano de nascimento, o sexo e o código de habilitação dos motoristas, sendo:

- CÓDIGO 1 – motorista habilitado;
- CÓDIGO 2 – motorista não-habilitado.

20) Faça um programa que calcule e escreva o valor de S:

$$S = 1/1 + 3/2 + 5/3 + 7/4 + \dots + 99/50$$

Acertar essa parada!!!!!!

21) Faça um programa que calcule o valor da soma:

$$S = (2)^1/50 + (2)^2/49 + (2)^3/48 + \dots + (2)^50/1$$

Acertar essa parada!!!!!!

22) Faça um programa que calcule a seguinte soma:
Escrever a soma!!!!

23) Faça um programa que calcule a seguinte soma:
Escrever a soma!!!!

24) Construa um programa que:

- Leia diversos pares de números naturais positivos;
- Para cada par de valores, calcule o valor do quociente e do resto da divisão inteira de num_1 por num_2; (num_1 > num_2)
- Apresentar o resultado para cada par de valores.

25) Escreva um programa que leia o número de inscrições e a altura de moças para um concurso de beleza. Calcule e escreva as duas maiores alturas e quantas moças as possuem.

26) Um banco deseja saber as seguintes informações:

- a) o cliente com maior volume de depósitos em cada agência;
- b) a agência com maior saldo (depósito-saldo) em cada região;
- c) a agência que possui o cliente com maior saldo no banco.

Fazer um programa que determine essas informações para o banco, tendo como dado de entrada a região, agência, conta, movimento (depósito) e valor.

CAPÍTULO IV MODULARIZAÇÃO

Objetivos

- Definir o que é modularização e quais as suas vantagens;
- Apresentar a estrutura de um subprograma;
- Apresentar o método de modularização através de função e procedimentos, enfatizando o conceito de passagem de parâmetros;
- Apresentar exemplos de modularização;
- Definir estratégias que contribuam para assegurar a qualidade da modularização.

IV.1 – A Necessidade de Modularização na Programação

Estudos da psicologia mostram que o ser humano normal é capaz de resolver problemas que envolvam 7 mais ou menos 2 variáveis simultaneamente. Um problema é complexo se envolve mais do que 9 ou 10 variáveis. Como, então, podemos resolver problemas deste tipo?

A solução que encontramos para solucionar estes problemas é dividir em problema complexo em vários subproblemas elementares, onde não estejamos sujeitos à essa limitação humana. Esta estratégia de solução de problemas é comumente conhecida com o a técnica de Dividir para Conquistar, a qual é o fundamento básico para a Modularização.

Neste sentido, a modularização de programas consiste da divisão do programa em módulos ou subprogramas, que nada mais são que um conjunto de comandos agrupados com a finalidade de executar ma determinada função.

Além disso, a modularização é uma técnica de programação que permite a aplicação direta dos refinamentos sucessivos, isto é, possibilita que haja um mapeamento entre os diversos níveis de algoritmos criados quando utilizamos refinamentos e o próprio ato de programar.

Um outro problema que nos defrontamos quando estamos programando é a repetição de código. Você já deve ter notado que certos trechos dos seus programas se repetem em vários momentos, tal qual foram escritos num primeiro instante ou de uma maneira muito aproximada. Esta repetição de código, além de ser algo desagradável para o programador, torna o programa redundante e dificulta, em última instância, a sua legibilidade.

A modularização é uma solução para este problema pois um módulo pode ser ativado em pontos diferentes de um mesmo programa.

Para falarmos um pouco mais sobre as vantagens da modularização necessitamos definir alguns conceitos sobre qualidade de programas. Um programa é de boa qualidade quando apresenta as seguintes características:

- **Confiabilidade:** o programa está correto e é confiável.
- **Legibilidade:** o programa é fácil de ser lido e entendido por um programador de nível médio.
- **Manutenibilidade:** o programa é fácil de ser modificado. Quando o usuário requiere modificações no programa, fruto de novas necessidades, a atualização do programa de ser natural.
- **Flexibilidade:** o programa é fácil de ser reutilizado; não possui muitas restrições.

Neste contexto, podemos criar agora uma série de vantagens da modularização:

- Facilita o projeto de sistemas – incentiva a técnica de refinamentos sucessivos.
- Minimiza a manutenção de programas – modificações no programa precisam ser realizadas apenas no módulo correspondente, sem ter de alterar o resto do programa.

- Melhora a legibilidade dos programas – não há necessidade de se entrar em detalhes de programação para se entender o que o programa faz.
- Viabiliza a validação – testes e correções de cada segmento podem ser realizados em separado, só reunindo os módulos quando todos estiverem devidamente validados.
- Permitir a divisão de tarefas em uma equipe – cada módulo pode ser realizado por um programador diferente.
- Facilita a reutilização de código:
 - Em trechos onde ocorre repetição de código no programa.
 - Em trechos onde ocorre repetição parcial de código, onde só são modificados os argumentos utilizados.
- Estimula o encapsulamento de informações: o programador que usará o módulo só precisa saber o que o módulo faz e não como faz.

IV.2 – Modularização em C

Agora que já estamos familiarizados com o conceito de modularização e conhecemos as suas vantagens, podemos nos ater a técnica de modularização que é implementada em C.

A ferramenta disponível em C para modularização são as funções.

Funções são segmentos de programa que executam uma determinada tarefa. Alguns exemplos de funções já usadas aqui são: printf, scanf e sqrt.

As funções escritas pelo programados, e que são usadas para fazer a modularização, são denominadas funções de usuário ou rotina de usuário.

IV.2.1 – A Estrutura das Funções de Usuário

A estrutura de uma função de usuário é muito parecida com a estrutura dos programas que apresentamos até agora. Uma função de usuário constitui-se de um nome pelo qual faremos referência a ela, de uma lista de argumentos passados a ela, e de um bloco de instruções que definem o que será feito por essa função.

Nas funções podemos declarar variáveis, denominadas locais, que têm vida e escopo somente durante a execução da função. Estas variáveis só podem ser usadas na função onde foram declaradas.

Um exemplo de função é dado a seguir:

```
float media2 (float a, float b){
    float media;

    media = (a + b) / 2.0;

    return(media);
}
```

Nesse exemplo o nome da função é media2 e ela recebe dois argumentos do tipo float. A média desses valores recebidos é calculada e retornada para o programa que fez a chamada a função. Esse valor de retorno é do tipo float, conforme é indicado antes do nome da função.

Depois de definida a função, podemos chamá-la dentro de um programa. Um exemplo de chamada da função media2 é dado abaixo:

```
main() {
```

```

float num1, num2, media;

printf("Digite dois numeros:\n");

scanf("%f %f", &num1, &num2);

media = media2(num1, num2); //aqui é feita a chamada da função

printf("A media dos números fornecidos eh: %f", media);

}

```

IV.2.2 – Definição de Função

A sintaxe de definição de uma função é a seguinte:

```

tipo_de_retorno nome(tipo1 arg1, tipo2 arg2, ....., tipoN argN)  {

    <comandos>

}

```

A primeira linha é dita declaração da função, nela encontramos o tipo que a função retorna, o nome dela e a lista de argumentos. Após a declaração, encontramos a lista de comandos entre as chaves.

O tipo de retorno serve para especificar qual o tipo de dados retornado pela função, podendo ser do tipo int, float, etc. Nos casos em que a função não retorna nenhum tipo, devemos defini-la como void, ou seja, sem retorno.

A lista de argumentos serve para dizer quais são os valores recebidos pela função e seus respectivos tipos.

IV.2.3 – Onde colocar as Funções

As funções podem ser declaradas no mesmo arquivo da main ou num outro arquivo separado.

IV.2.3.1 – Funções no mesmo arquivo

Quando uma função é declarada no mesmo arquivo existe a possibilidade dela ser declarada antes ou depois da main.

IV.2.3.1.1 – Funções no mesmo arquivo e antes da main

Quando as funções são declaradas antes da main não é necessário adicionar nenhuma outra instrução. A sintaxe fica assim:

```

tipo nome_funcao(lista_de_argumentos){
    comandos          //definição da função
}

void main(){
    .
    .
    .
}

```

```
    variavel = nome_funcao(lista_de_argumentos); //chamada da função
    .
    .
    .
}
```

IV.2.3.1.2 – Funções no mesmo arquivo e depois da main

Quando a função é definida depois da main é necessário incluir um protótipo da função chamada. Um protótipo constitui-se do tipo de retorno da função, do nome da função e quantidade de argumentos junto com seus tipo. A sintaxe dessa declaração é a seguinte:

```
tipo nome_funcao(lista_de_argumentos);

void main(){
    .
    .
    .
    variavel = nome_funcao(lista_de_argumentos); //chamada da função
    .
    .
    .
}

tipo nome_funcao(lista_de_argumentos){
    comandos //definição da função
}
}
```

IV.2.3.2 – Funções em arquivo separado

Nesse caso é necessário incluir o arquivo onde se encontra a função ao arquivo onde ela está sendo usada. Essa inclusão é feita através do comando #include que diz para o compilador na hora da compilação que ele precisa fazer alguma(s) inclusão(s).

Supondo que a função esteja no arquivo funcoes.h, a sintaxe ficaria assim:

```
#include "funcoes.h"

main(){
    .
    .
    .
    variavel = nome_funcao(lista_de_argumentos);
    .
    .
    .
}
```

IV.2.4 – Hierarquia das funções

Quando uma função chama a outra, dizemos que a função que chama tem uma hierarquia superior a da função chamada.

As funções com hierarquia inferior devem ser declaradas antes das funções com hierarquia superior. Isso deve ser feito para que ao se fazer a chamada de uma função, ela já seja conhecida.

IV.2.5 – Recursão

Função recursiva é uma função que faz chamada a si mesma.

É importante ressaltar que a cada nova chamada da função recursiva um novo conjunto de variáveis é criado.

Exemplo: um bom exemplo do uso de função recursiva é o cálculo do fatorial de um número.

```
int fatorial (int numero){  
  
    if (numero < 0){  
        return 0;  
    }  
  
    if (numero == 0){  
        return 1;  
    }  
  
    return (numero * (fatorial (numero-1)));  
}
```

O primeiro “if” avalia se o número de entrada é menor que zero, caso seja a função retorna zero pois não existe fatorial de número negativo. Já quando o número é igual a zero, o segundo “if” faz com que a função retorne um pois o fatorial de zero é um. Para os números maiores que zero, a função retorna a multiplicação do número pelo fatorial do seu antecessor. Repare que para esse último caso, a recursão irá terminar quando o antecessor do número for zero pois a chamada de “fatorial(0)” não chamará outra vez a função fatorial conforme visto anteriormente.

CAPÍTULO V ESTRUTURAS

Objetivos

- Apresentar o que é uma estrutura e para que ela serve
- Mostrar a declaração de uma estrutura e como acessar seus elementos
- Mostrar a relação entre estruturas e funções

V.1 – O que é uma estrutura

Uma estrutura é uma coleção de variáveis, que podem ou não ser de tipos diferentes, colocadas sob um único nome para manipulá-las. As estruturas ajudam na organização do código e facilitam a vida do programador pois juntam variáveis relacionadas e permitem que elas sejam tratadas como uma unidade maior.

Um bom exemplo do uso de estruturas é o armazenamento de datas. Uma data é caracterizada por um dia, um mês e um ano. Caso o programador não pudesse utilizar uma estrutura, ele teria que declarar três variáveis separadamente e trabalhar com elas em paralelo. Isso poderia gerar alguma confusão e, em alguns casos, perda de algumas informações. Mas isso pode ser evitado quando o programador cria uma estrutura de nome data, por exemplo, e dentro dela declara três variáveis: dia, mês e ano. Com isso, ele passaria a trabalhar com uma variável do tipo data que por sua vez possuiria as três variáveis já citadas anteriormente.

V.2 – Declaração de uma estrutura

A sintaxe de declaração de uma estrutura é a seguinte:

```
struct nome_da_estrutura {  
  
    declaração_das_variáveis  
  
};
```

O nome da estrutura deve ser algo que lembre o que ela representa para que outros programadores possam entender o seu significado. No exemplo anterior, um bom nome para a estrutura seria “data” e ela ficaria assim:

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

Para usarmos essa estrutura durante o programa precisamos declarar uma variável de seu tipo. Feito isso, poderemos manipulá-la conforme veremos na próxima sessão. A declaração de uma variável do tipo de uma estrutura é feita assim:

```
struct nome_da_estrutura nome_da_variavel;
```

No exemplo a seguir veremos a declaração de uma variável do tipo estrutura data:

```
main(){
```

```

struct data aniversario;
    .
    .
    .
}

```

V.3 – Manipulação dos elementos da estrutura

A manipulação dos elementos de uma estrutura é muito simples. Os elementos da estrutura podem ser modificados durante a declaração da variável. Como ilustração disso, tomemos o exemplo o seguinte exemplo:

```

struct ponto {
    int x;
    int y;
};

main() {

    struct ponto maxPt = { 500, 1000 };    //aqui a variável maxPt é declarada e as
                                           //suas variáveis internas da estrutura passam a
                                           //valer: x = 500 e y = 1000

}

```

Outra forma de modificar os elementos da estrutura é mostrada no próximo exemplo.

Exemplo: tomemos a estrutura data declarada anteriormente.

```

main(){

    struct data nascimento;

    printf("\nDigite o dia do seu nascimento: ");
    scanf ("%d", &nascimento.dia); //observe que "nascimento.dia" indica que você está
                                   //armazenando o dado de entrada na variável "dia" que está
                                   //dentro da variável aniversario, que por sua vez é do tipo
                                   // struct data.

    printf("\nDigite o mes do seu nascimento: ");
    scanf ("%d", &nascimento.mes);

    printf("\nDigite o ano do seu nascimento: ");
    scanf ("%d", &nascimento.ano);

    printf("\nVoce nasceu no dia %d do mes %d do ano %d\n", nascimento.dia, nascimento.mes,
nascimento.ano);

}

```

V.4 – O comando typedef

Esse comando é bastante útil para atribuirmos novos nomes a tipos de dados. Ele permite, por exemplo, que você atribua ao tipo “int” o nome “tipoDeEntrada” para designar que o tipo de entrada de uma determinada função será o “int”.

A sintaxe desse comando é muito simples e pode ser vista a seguir:

```
typedef tipo novo_nome;
```

No exemplo citado acima, a renomeação do tipo “int” ficaria assim:

```
typedef int tipoDeEntrada;
```

Assim, quando fizermos as declarações

```
int entrada;
```

e

```
tipoDeEntrada entrada;
```

estamos declarando a mesma coisa.

Esse comando nos permite também renomear estruturas criadas por nós, para que a declaração de uma variável do tipo dessa estrutura seja menos trabalhosa.

Tomando como exemplo a estrutura declarada anteriormente para representar um ponto, poderíamos usar o typedef do seguinte modo:

```
typedef struct ponto {  
    int x;  
    int y;  
} ponto;
```

Dessa forma uma variável que antes era declarada fazendo

```
struct ponto centro;
```

pode ser declarada usando

```
ponto centro;
```

Outra forma de se usar o typedef com estruturas é fazer:

```
typedef struct ponto ponto;  
struct ponto {  
    int x;  
    int y;  
};
```

A declaração de uma variável ficaria do mesmo jeito do caso anterior.

V.5 – Estruturas e Funções

As estruturas são muito úteis quando estamos trabalhando com funções pois elas podem agrupar dados que serão passados como parâmetro para as funções e permitem que uma função retorne mais de um valor.

V.5.1 – Estruturas como parâmetros

A passagem de estrutura como parâmetro para função é muito utilizada pois permite que os dados com características comuns cheguem agrupados e também por diminuir o número de parâmetros da função facilitando o seu entendimento.

A declaração de uma função que recebe uma estrutura como parâmetro é feita da mesma forma que a de uma função que recebe apenas tipos básicos de dados como parâmetro.

A seguir mostraremos um exemplo de uma função que recebe uma variável do tipo ponto declarado anteriormente e retorna o quadrante ao qual ele pertence .

```
int qualQuadrante (ponto p){
    /*A função retorna 0 quando p for parte do limite de quadrantes, 1 quando p pertencer ao
    1º quadrante, 2 quando pertencer ao 2º quadrante, 3 se pertencer ao 3º quadrante e 4
    caso seja do 4º quadrante.*/

    if (p.x == 0 || p.y == 0){
        return 0;
    }

    if (p.x > 0){
        if(p.y > 0){
            return 1;
        }
        else {
            return 4;
        }
    }
    else {
        if(p.y > 0){
            return 2;
        }
        else {
            return 3;
        }
    }
}
}
```

V.5.2 – Estruturas como tipo de retorno

Estruturas são usadas com frequência quando queremos que uma função retorne mais do que valor. Nesse caso precisamos declarar que o tipo de retorno da função é uma determinada estrutura.

Tome como exemplo uma função que recebe três números inteiros distintos e retorna o maior e o menor deles.

Para esse exemplo usaremos uma nova estrutura:

```
typedef struct maiorEmenor {
    int maior;
    int menor;
} mEm;
```

Agora vejamos como fica o protótipo dessa função:

```
mEm maior_menor (int , int , int );
```

ou

```
struct maiorEmenor maior_menor(int , int , int );
```

Agora podemos definir a função, ela fica assim:

```
mEm maior_menor (int a, int b, int c){
    mEm s;

    if (a>b){
        if (a>c){
            if (c>b){
                s.maior = a;
                s.menor = b;
            }
            else {
                s.maior = a;
                s.menor = c;
            }
        }
        else {
            s.maior = c;
            s.menor = b;
        }
    }
    else {
        if (a<c){
            if (c>b){
                s.maior = c;
                s.menor = a;
            }
            else {
                s.maior = b;
                s.menor = a;
            }
        }
        else {
            s.maior = b;
            s.menor = c;
        }
    }

    return s;
}
```

V.6 – Tipos Abstratos de Dados

Tipos abstratos de dados são tipos de dados criados pelo programador de acordo com as suas necessidades para resolver um determinado problema. A definição de um tipo abstrato de dados é constituída de:

- Definir os elementos básicos
- Implementar construtores para gerar novos elementos a partir dos elementos básicos
- Implementar operações sobre esses elementos

Podemos definir um tipo abstrato de dados para representar os pontos do plano XY. Para isso considere a estrutura denominada ponto.

```
typedef ponto {
    int x;
    int y;
} ponto;
```

Agora já sabemos como são os elementos básicos desse tipo. O próximo passo é implementar uma função que opere elementos desse tipo. Um exemplo de uma operação entre elementos desse tipo é o cálculo da distância entre dois pontos.

```
float distanciaPontos (ponto a, ponto b){

    float distancia, deltaX, deltaY;

    deltaX = a.x - b.x;
    deltaY = a.y - b.y;

    distancia = sqrt(deltaX*deltaX+deltaY*deltaY);

    return distancia;
}
```

Podemos ainda verificar se um dado ponto representa a origem ou não e também calcular seu simétrico em relação à origem.

```
int origem(ponto p){

    if (p.x && p.y){
        return 0;
    }
    else {
        return 1;
    }
}

ponto simetrico (ponto p){

    p.x = -p.x;
    p.y = -p.y;

    return p;
}
```

Por último vamos fazer o cálculo do deslocamento de um ponto.

```
ponto desloca (ponto p, int deslocamento){  
    p.x = p.x + deslocamento;  
    p.y = p.y + deslocamento;  
    return p;  
}
```

CAPÍTULO VI VETORES

Objetivos

- Introduzir a estrutura de dados vetor, explicar seu conceito e mostrar sua utilidade;
- Apresentar as operações básicas sobre vetores.

VI.1 – Conceito de vetores e sua utilidade

Um vetor é uma estrutura de dados composta homogênea, isto é, ela possui, em geral, mais de um elemento (por isso, composta) sendo que estes elementos são sempre de um mesmo tipo (por isso, homogênea). Tal qual na matemática, os vetores possuem uma dimensão (que corresponde ao número de elementos do vetor) e seus elementos são acessados através de índices. A figura abaixo representa um vetor com sete elementos inteiros.

Índice: 0	1	2	3	4	5	6
9	5	15	6	3	2	12

Mas, você já deve estar questionando, para que servem os vetores?

Nem sempre os tipos básicos de dados (int, float, char) são suficientes para exprimir estruturas de dados em algoritmos. Considere o problema em que um professor de uma turma de 50 alunos deseja saber quais desses são bons alunos (tiverem nota acima da média da turma). Com os instrumentos que dispomos até agora necessitaríamos ler as notas dos 50 por duas vezes: uma para calcular a média e outra para verificar quais alunos tiveram notas superior à média. Esta dupla leitura não é muito conveniente, concorda?! A solução para este problema se dá através da utilização de vetores.

A utilização de um vetor se faz necessária sempre que for preciso armazenar uma quantidade finita de dados de um tipo em comum, para posterior processamento.

VI.2 – Declaração de Vetor

Um vetor é definido da seguinte maneira:

```
tipo_componentes nome_vetor[tamanho];
```

tipo_componentes indica de que tipo são os elementos do vetor, o *tamanho* indica quantos elementos o vetor poderá armazenar. O vetor apresentado na sessão anterior possui a seguinte definição:

```
int vetor_inteiros[7];
```

O número de componentes de um vetor é fixo e determinado na sua declaração.

O tamanho de um vetor é determinado em sua declaração, antes da execução do programa (tamanho físico), isto significa que o tamanho de um vetor não poderá ser lido ou alterado durante a execução do programa. Verifica-se então que para armazenar dados em vetores é necessário conhecer antecipadamente a quantidade máxima de elementos a serem processados. Quando este número não for conhecido, deve-se maximizar o tamanho do vetor baseado em critérios intuitivos (bom senso) relacionado ao problema em questão.

Por exemplo, para calcular quais as notas dos alunos de uma turma são maiores que a média, seria preciso ter-se uma idéia de quantos alunos no máximo uma turma poderia possuir.

VI.3 – Acesso aos elementos do vetor

Os elementos do vetor são acessados através da citação do nome da variável (que indica qual vetor estamos acessando) e de um índice (que determina qual elemento queremos acessar). Este índice não pode ser um valor maior que o tamanho do vetor menos um nem tampouco um valor menor que zero.

A sintaxe de acesso a um vetor é dada por:

```
nome_vetor[índice]
```

Exemplo:

```
vetor[3] = 987;  
vetor[índice+1] = vetor[3*2];
```

VI.4 – Operações básicas sobre vetores

Em C não existem operações pré-definidas para a manipulação de vetores como um todo. Por exemplo, não é permitido ler o conteúdo de um vetor inteiro com o comando `scanf`. As operações só podem ser feitas para cada elemento do vetor.

Portanto, as operações básicas de vetor devem ser simuladas através de subprogramas. A seguir serão apresentadas uma série de operações básicas para vetores.

Como sabemos, no parâmetro associado por valor é uma cópia do argumento e a duplicação de estruturas do tipo vetor pode ocasionar problemas de memória e ineficiência do programa. A maneira de resolver estes dois problemas é forçar a passagem do vetor por referência, porém conceitualmente o uso desse artifício é incorreto, devendo ser implementado somente após ter sido testado e nunca desacompanhado de um comentário.

Você verá nos procedimentos apresentados a seguir que um vetor apesar de não ser modificado no seu interior (é um argumento de entrada) este é passado por referência, como por exemplo o procedimento para escrita de um vetor.

Daqui em diante usaremos uma estrutura de dados que contenha o vetor e o número de elementos desse vetor. Essa estrutura é da seguinte forma:

```
typedef vetor {  
    tipo_elemento vetor[TAMANHO];  
    int numero_elementos;  
} vetor;
```

Onde TAMANHO indica o tamanho do vetor.

VI.4.1 – Leitura e Escrita

Um exemplo de uma função que faz a leitura dos elementos de um vetor de inteiros.

```
void leitura_vetor (vetor vet){  
  
    int i;  
  
    for (i = 0; i < vet.numero_elementos; i++){
```

```
        scanf("%d", &vet.vetor[i]);
    }
}
```

Agora mostraremos uma função que imprime os elementos de um vetor de inteiros.

```
void imprime_vetor (vetor vet){

    int i;
    for (i = 0; i < vet.numero_elementos; i++){
        printf("%d ", vet.vetor[i]);
    }
}
```

VI.4.2 – Adição de dois Vetores Numéricos

```
void soma_vetor (vetor vet1, vetor vet2, vetor vet_soma){

    int i;

    if(vet1.numero_elementos != vet2.numero_elementos){
        printf("\n Vetores com numero de elementos diferente.\n");
        return;
    }

    vet_soma = vet1.numero_elementos;

    for (i = 0; i < vet1.numero_elementos; i++){
        vet_soma[i] = vet1.vetor[i] + vet2.vetor[i];
    }
}
```

VI.4.3 – Produto Escalar de dois Vetores Numéricos

```
int produto_escalar (vetor vet1, vetor vet2){

    int prod_escalar, i;

    if(vet1.numero_elementos != vet2.numero_elementos){
        printf("\n Vetores com numero de elementos diferente.\n");
        return;
    }

    for (i = 0; i < vet1.numero_elementos; i++){
        prod_escalar = prod_escalar + vet1.vetor[i] * vet2.vetor[i];
    }

    return prod_escalar;
}
```

VI.4.4 – Pesquisa Seqüencial

int pesquisa_binaria (vetor vet, int elemento){
/*Funcao: percorre seqüencialmente o vetor, procurando por um elemento. Esta funcao retorna o valor -1 caso o elemento nao seja encontrado. Caso contrario, retorna o índice da posicao do vetor onde o elemento foi encontrado.*/

```
    int i, encontrou, pesq_seq;  
  
    encontrou = 0;  
    pesq_seq = -1;  
  
    for (i = 0; i < vet.numero_elementos && !encontrou; i++){  
        if (vet.vetor[i] == elemento){  
            pesq_seq = i;  
            encontrou = 1;  
        }  
    }  
  
    return pesq_seq;  
}
```

VI.4.5 – Pesquisa Binária

int pesquisa_binaria (vetor vet, int elemento){
/*Funcao: Percorrer o vetor ordenado (ordenado crescente), particionando-o ao meio e procurando determinado elemento. Esta funcao retorna o valor -1 caso o elemento nao seja encontrado. Caso contrario, retorna o índice do vetor onde o elemento foi encontrado.*/

```
    int inicio, fim, meio;  
  
    inicio = 0;  
    fim = vet.numero_elementos - 1;  
  
    while (inicio <= fim){  
        meio = (inicio + fim) / 2;  
        if (elemento < vet.vetor[meio]){  
            fim = meio - 1;  
        }  
        else {  
            if (elemento > vet.vetor[meio]){  
                inicio = meio + 1;  
            }  
            else {  
                return meio;    //encontrou!!!  
            }  
        }  
    }  
}
```

```

    return -1;
}

```

VI.4.6 – Inserção em Posição Pré-determinada do Vetor

```

void insere (vetor vet, int elemento, int posicao){
/*Funcao: inserir um novo elemento numa posicao pre-estabelecida.*/

    int i;

    if ((posicao < 0) || (posicao > vet.numero_elementos-1)){
        printf("\nEntrada de Dados Incorreta\n");
    }
    else {
        for (i = numero_elementos - 2; i > posicao - 1; i--){
            vet.vetor[i + 1] = vet.vetor[i];
        }
        vet.vetor[posicao] = elemento;
    }
}

```

VI.4.7 – Exclusão de Elementos de Vetor

```

void excluir (vetor vet, int elemento){
/*Funcao: Percorrer o vetor sequencialmente procurando o elemento. Caso encontre-o retira-lo. Caso contrario, manter o vetor inalterado.*/

```

```

    int i, posicao, encontrou;

    encontrou = 0;

    posicao = 0;

    while ((posicao < vet.numero_elementos) && (!encontrou)){
        if (vet.vetor[posicao] == elemento){
            encontrou = 1;
            for (i = posicao; i < vet.numero_elementos - 1; i++){
                vet.vetor[i] = vet.vetor[i + 1];
            }
        }
        posicao += 1;
    }
}

```

VI.4.8 – Ordenação pelo Método do Menor

```

void ordenacao_menor (vetor vet){

```

```

int posicao, i, indice_menor, menor;

for (posicao = 0; posicao < vet.numero_elementos - 1; posicao++){
    menor = vet.vetor[posicao];
    for (i = posicao + 1; i < tamanho; i++){
        if (vet.vetor[i] < menor){
            menor = vet.vetor[i];
            indice_menor = i;
        }
    }
    vet.vetor[indice_menor] = vet.vetor[posicao];
    vet.vetor[posicao] = menor;
}
}

```

VI.4.9 – Ordenação pelo Método da Bolha

```

void ordenacao_bolha (vetor vet) {

    int a, b, t;

    for (a = 1; a < vet.numero_elementos; a++){
        for (b = vet.numero_elementos - 1; b >= a; --b){
            if (vet.vetor[b - 1] > vet.vetor[b]){
                t = vet.vetor[b - 1];
                vet.vetor[b - 1] = vet.vetor[b];
                vet.vetor[b] = t;
            }
        }
    }
}

```

VI.5 – Exemplo de programa que utiliza vetores

A)Este programa lê dois vetores de mesma dimensão, calcula a sua soma e seu produto escalar, apresenta em seguida os resultados.

```
#define tamanho 100
```

```

typedef vetor {
    int vetor[tamanho];
    int numero_elementos;
} vetor;

```

```
void leitura_vetor_inteiros (vetor vet){
```

```
    int i;

    for (i = 0; i < vet.numero_elementos; i++){
        scanf ("%d", vet.vetor[i]);
    }
}
```

```
void escrita_vetor_inteiros (vetor vet){

    int i;

    printf("\n");

    for (i = 0; i < vet.numero_elementos; i++){
        printf("%d ", vet.vetor[i]);
    }

    printf("\n");

}
```

```
void soma_vetores_inteiros (vetor vet1, vetor vet2, vetor vetor_soma){

    int i;

    if(vet1.numero_elementos != vet2.numero_elementos){
        printf("\n Vetores com tamanhos diferentes \n");
        return;
    }

    for (i = 0; i < vet1.numero_elementos; i++){
        vetor_soma.vetor[i] = vet1.vetor[i] + vet2.vetor[i];
    }

}
```

```
int produto_escalar (vetor vet1, vetor vet2){

    int i, prod_esc;

    prod_esc = 0;
```

```
    if(vet1.numero_elementos != vet2.numero_elementos){
        printf("\n Vetores com tamanhos diferentes \n");
        return;
    }

    for (i = 0; i < vet1.numero_elementos; i++){
        prod_esc = prod_esc + vet1.vetor[i] * vet2.vetor[i];
    }

    return prod_esc;
}

main(){

    vetor vetor1, vetor2, vetor_soma;
    int prod_esc;

    vetor1.numero_elementos = tamanho;
    vetor2.numero_elementos = tamanho;
    vetor_soma.numero_elementos = tamanho;

    leitura_vetor_inteiros(vetor1);

    leitura_vetor_inteiros(vetor2);

    soma_vetores_inteiros (vetor1, vetor2, vetor_soma);

    escrita_vetor_inteiros (vetor_soma);

    prod_esc = produto_escalar (vetor1, vetor1);

    printf("\n%d\n", prod_esc);
}
```

Exercícios

- 1) Faça um programa em C que leia 100 números reais e os imprima na ordem inversa em que foram lidos.
- 2) Crie uma função em C que some os elementos de um vetor com 100 elementos inteiros.
- 3) Faça um programa em C que leia as notas e as matrículas dos alunos de uma turma numaprova e obtenha:

- a) a melhor nota e o aluno que a obteve;
 - b) a pior nota e o aluno que a obteve;
 - c) a média da turma;
 - d) os alunos que obtiveram nota superior à média da turma;
- OBS.: considere que uma turma tem no máximo 50 alunos.

4) Faça um programa em C que determine os 100 primeiros números primos. Considere que:

- a) um número é primo quando só é divisível por si próprio;
- b) é suficiente testar a divisibilidade de um número X por números primos que o antecedem até um limite igual a raiz de X.

5) Faça um programa em C que leia duas listas de caracteres com até 100 elementos e coloque cada um num vetor respectivo. Após ter sido realizada a leitura das duas listas, deve ser lida a ordem a partir da qual deve ser inserida a segunda lista na primeira lista de caracteres. Considere o seguinte exemplo:

```
-abcdjklm    (primeira lista)
-efghi       (segunda lista)
-5
Resultado: abcd fghijklm (lista final)
```

6) Faça um programa em C que leia dois vetores ordenados em M e N elementos respectivamente os intercale gerando u, novo vetor também ordenado. Considere o seguinte exemplo:

Dados:

```
-5    (M)
-4    (N)
-1 7 13 14 30 (primeiro vetor)
-2 3 7 16    (segundo vetor)
Resultado: 1 2 3 7 7 13 14 16 30 (vetor resultado)
```

OBS.: o terceiro vetor não pode ser gerado copiando-se primeiramente os dois vetores e fazendo a ordenação posteriormente.

7) Faça um programa em C que leia N ($N < 1000$) números de matrículas que fazem PD1 e os coloque em ordem crescente num vetor a medida que vão sendo lidas as matrículas. Posteriormente deseja-se identificar se um certo conjunto de M alunos fazem PD1 (utilize o algoritmo de pesquisa binária para fazer esta verificação).

8) Faça um programa em C que:

- a) leia N números reais colocando-os num vetor de 100 elementos (considere $N < 1000$);
 - b) ordene crescentemente os elementos de índices ímpares (considerar apenas os N elementos);
 - c) escreva os N números após o ajuste da letra b.
- OBS.: deve ser empregado o método da bolha para a ordenação.

9) Crie um procedimento em C que cumpra a seguinte especificação:

a) Dados de entrada:

- 1- um vetor de caracteres com um máximo de 100 elementos;
- 2- o tamanho corrente do vetor (suponha que o vetor inicie na primeira posição);
- 3- um caracter a ser inserido;
- 4- a posição do vetor onde o caracter deve ser inserido.

b)Dados de saída:

1-o vetor com caracter inserido na posição;

2-o vetor deve ser mantido inalterado se a posição onde deveria ser inserido o caracter for superior ao tamanho corrente do vetor ou se o vetor já estiver completo.

10)Um armazém trabalha com 100 mercadorias diferentes identificadas pelos números de i a 100. O dono anota as quantidades de cada mercadoria vendida no mês. Ele tem uma tabela que indica para cada mercadoria o preço de venda. Escreva um algoritmo que calcule o faturamento mensal do armazém , onde:

$$\text{Faturamento} = \sum(\text{quantidade}_i * \text{preço}_i), \text{ com } i = 1, \dots, 100.$$

11)Dado um vetor A de 128 elementos, verificar se existe um elemento igual a K no vetor. Se existir, imprimir a posição onde foi encontrado o elemento. Caso contrário, imprimir “Um elemento igual a K não foi encontrado”. K e o vetor são lidos.

12)Escreva um algoritmo que:

-leia um conjunto A de 100 elementos reais;

-construa e imprima um outro conjunto B formado da seguinte maneira:

*os elementos de ordem par são os correspondentes de A divididos por 2;

*os elementos de ordem ímpar são os correspondentes da A multiplicado por 3.

13)Dado um vetor contendo um frase com oitenta caracteres (incluindo brancos). Escrever um algoritmo para:

-contar quantos brancos existem na frase;

-contar quantas vezes aparece a letra ‘A’;

-contar quantas vezes aparece um mesmo par de letras na frase e quais são eles.

14)Faça um algoritmo que leia um vetor de 200 elementos numéricos e verifique se existem elementos iguais a 33. Se existirem, escrever as posições em que estão armazenados.

15)Escrever um algoritmo que faça reserva aéreas de uma companhia. Além da leitura do número dos vôos e da quantidade de lugares disponíveis, a ler vários pedidos de reserva, constituídos do número da carteira de identidade do cliente e o número do vôo desejado. Para cada cliente, verificar se a disponibilidade no vôo desejado. Em caso afirmativo, imprimir o número da carteira de identidade do cliente e o número do vôo, atualizando o número de lugares disponíveis. Caso contrário, avisar o cliente da inexistência de lugares. Indicando o fim dos pedidos de reservas, existe um passageiro cujo número de identidade é 999999. Considerar fixo i igual a 30 o número de vôos da companhia.

16)Faça um algoritmo para ler um vetor de comprimento N (N sendo par menor ou igual a 50) e imprimir o seu conteúdo depois de feita a troca dos elementos das posições pares com os elementos das posições ímpares.

17)Faça um programa que, dado um nome terminado por ponto, devolva o número de vogais nele presentes.

18)Faça um programa que, dada uma rase terminada por ponto. Retire todos os espaços em braço da mesma, retornando a frase assim modificada.

19)Faça um programa que, dada duas listas de nomes, compare-as e devolva o número de vezes que cada palavra da segunda lista aparece na primeira lista. Assuma que cada nome seja composto de no máximo 15 caracteres.

Próximos temas a serem abordados:

- Matrizes
- Ponteiros
- Recursividade
- Variáveis globais e escopo de visibilidade
- Assertivas e prova de correção