



FIAA-FACULDADES INTEGRADAS ANGLO-AMERICANO
CURSO DE SISTEMAS DE INFORMAÇÃO

Apostila de

Algoritmo Estruturado

4ª edição – Agosto de 2003
Profª Salete Buffoni

Sumário

CAPÍTULO 1 - PROGRAMAÇÃO ESTRUTURADA

1.1. Introdução.....	4
1.2. Conceitos Básicos.....	4
1.2.1. Hardware.....	6
1.2.2. Software.....	7
1.3. O Desenvolvimento de Sistemas de Informação e o Programador.....	8

CAPÍTULO 2 - SOLUÇÃO DE PROBLEMAS: ALGORITMOS

2.1. Problemas e soluções.....	10
2.2. Definição de Algoritmo.....	11
2.3. Técnicas de Representação de Algoritmos.....	12
2.4. Algoritmo x Programa.....	14
2.5. Linguagem de Programação.....	14
2.6. Processo de Criação e Execução de um Programa.....	15
2.7. Estratégias a serem Seguidas na Resolução de Problemas.....	17
2.7.1. Critérios de Qualidade de um Programa	17
2.7.2. Metodologia de Solução.....	17

CAPÍTULO 3 - ALGORITMOS: COMANDOS BÁSICOS

3.1. Identificadores, Constantes, Variáveis e Tipos Básicos.....	19
3.2. Declaração de Variáveis.....	20
3.2.1. Palavras Reservadas.....	21
3.3. Comando Simples.....	21
3.4. Comando Composto.....	21
3.5. Operadores.....	22
3.6. Comandos de Entrada e Saída.....	24
3.7. Estrutura de um Algoritmo em Pseudo-Código e DNS.....	25

CAPÍTULO 4 - ESTRUTURAS BÁSICAS DE CONTROLE

4.1. Seqüência.....	28
4.2. Seleção.....	28
4.2.1. Seleção Simples.....	28
4.2.2. Seleção Composta.....	29
4.2.3. Aninhamento de Seleções.....	30
4.3. Estruturas de Repetição- LAÇOS(LOOPS).....	30
4.3.1. Laço Enquanto.....	30
4.3.2. Controlados por Contador.....	32
4.3.3. Repetição com Teste no Final.....	32
4.4. Escape do Laço.....	33
4.5. Regras Práticas para a Construção de Algoritmos Legíveis.....	35

CAPÍTULO 5 - ESTRUTURAS DE DADOS HOMOGÊNEAS: VETORES E MATRIZES

5.1. Repetição com variável de controle (Para).....	51
5.2. Seleção dentre múltiplas alternativas.....	53
5.3. Vetores.....	55
5.3.1. Declaração de Vetores.....	55
5.3.2. Trabalhando com Vetores.....	56
5.4. Matrizes.....	63
5.4.1. Declaração de Matrizes.....	63
5.4.2. Trabalhando com Matrizes.....	63

CAPÍTULO 6 - MODULARIZAÇÃO DE ALGORITMOS

6.1. Introdução.....	69
6.2. Procedimentos.....	69
6.3. Funções.....	71
6.3.1. Protótipos de Funções.....	75
6.3.2. Escopo de Variáveis.....	76
6.3.2.1. Variáveis Locais.....	76
6.3.2.2. Parâmetros Formais.....	77
6.3.2.3. Variáveis Globais.....	77
6.3.3. Passagem de Parâmetros por Valor e Passagem por Referência.....	78
6.4. Funções Recursivas.....	80
6.5. Estruturação dos Módulos de um Algoritmo.....	80

CAPÍTULO 7 - ALGORITMOS DE PESQUISA E ORDENAÇÃO

7.1. Introdução.....	81
7.2. Algoritmos de Pesquisa.....	81
7.2.1. Pesquisa Seqüencial Simples.....	82
7.2.2. Pesquisa Seqüencial Ordenada.....	83
7.2.3. Pesquisa Binária.....	85
7.3. Algoritmos de Ordenação.....	86
7.3.1. Métodos de Seleção Direta.....	87
7.3.2. Método de Inserção Direta.....	88
7.3.3. Método de Bolha.....	88

REFERÊNCIAS.....	89
-------------------------	-----------

CAPÍTULO 1 - PROGRAMAÇÃO ESTRUTURADA

1.1. INTRODUÇÃO

Programação de computadores é uma atividade que vem se desenvolvendo há mais de 40 anos desde o surgimento das primeiras linguagens de programação. Inicialmente, aprendia-se a programar utilizando-se a própria linguagem como ferramenta. A linguagem Fortran, criada em 1954, e com seu primeiro compilador em 1957, foi durante muito tempo utilizada com este fim, inclusive no Brasil. Nas décadas de 60 e 70, com o aparecimento de algumas outras linguagens, o Fluxograma, um método gráfico que hoje está em desuso, foi grandemente utilizado em complemento ao anterior. Hoje em dia, depois de centenas de linguagens de programação criadas e de vários métodos para se representar a solução de problemas, aprende-se a programar construindo e desenvolvendo algoritmos.

Na disciplina Algoritmos, com o auxílio desta apostila, você vai aprender os passos básicos e as técnicas para construção de algoritmos e programas, utilizando dois métodos para sua representação. Espera-se que com isso ao fim de um período de quatro meses, você tenha adquirido a capacidade de transformar qualquer problema de média complexidade e extensão em um algoritmo de boa qualidade. Antes de entrarmos no assunto propriamente dito, vamos apresentar alguns conceitos básicos sobre construção de algoritmos e tecer breves considerações sobre os dois componentes do sistema de computação ou de processamento eletrônico de dados: O Hardware e o Software. Se você deseja saber algo sobre a história da computação sugerimos os capítulos iniciais das referências [1] e [3].

O objetivo desse curso é a Lógica de Programação dando uma base teórica e prática, suficientemente boa, para que, o aluno domine os algoritmos e esteja habilitado a aprender uma linguagem de programação. Será mostrado também um grupo de algoritmos clássicos para tarefas cotidianas, tais como: ordenação e pesquisa.

1.2. CONCEITOS BÁSICOS

- **Dados:** são elementos brutos que podem ser processados por um computador digital para se obter alguma conclusão ou resultado, ou seja, uma informação.

- **Computador Digital:** Máquina(hardware) muito rápida que pode processar dados, realizando cálculos e operações repetitivas, se dotadas de programação adequada (software), e que fornece resultados corretos e precisos.
- **Informação:** é o resultado do processamento dos dados pelo computador. Uma informação pode ser considerada como dado para outro processamento e, por isso, muitas vezes é referenciada como dado de saída.
- **Hardware:** é um termo de origem americana que engloba todo o equipamento principal e periférico de um computador. O termo é utilizado também para equipamentos sem software. O hardware, normalmente, é fixo e difícil de ser modificado.
- **Software:** é um outro termo de origem americana que engloba programas, documentação, procedimentos e dados, utilizados em um computador para resolução de um determinado problema. O *software*, ao contrário do *hardware*, é maleável e por isso mais facilmente modificável pelo programador, para adaptação a novos dados, novos requisitos do problema a ser resolvido, ou novo hardware onde vai funcionar (ser executado).
- **Programa de Computador:** seqüência de instruções não ambígua e finita em uma linguagem de programação específica que, quando executada, resolve um problema determinado.
- **Linguagem de máquina:** é a linguagem binária (composta de zeros e uns) utilizada pelos computadores, para representar dados, programas e informações. É tediosa, difícil de se compreender e fácil de gerar erros na programação.
- **Linguagem Assembler:** é uma linguagem particular para cada processador (computador), que codifica as instruções em linguagem de máquina e facilita um pouco o trabalho do programador. É também considerada uma linguagem de baixo nível, mas ainda hoje é utilizada.
- **Linguagem de programação de 3ª geração:** também chamada de linguagem de alto nível, engloba todas as linguagens de programação que utilizam compiladores ou interpretadores. Possuem instruções mais poderosas que a linguagem Assembler, facilitando ainda mais o trabalho do programador.
Exemplos: Fortran, Cobol, Algol, PLI, Pascal, Modula2, Lips, C, ADA, etc.
- **Linguagem de programação de 4ª geração:** são linguagens não procedurais, utilizadas para sistemas de gerenciamento de banco de dados, planilhas e outros aplicativos que utilizam comandos mais poderosos ainda que as linguagens de 3ª geração e, por isso, são ainda mais fáceis de programar. *Exemplos:* SQL, DBASE, CLIPPER, INGRES, etc.

- **Compilador:** é um programa utilizado para traduzir os programas escritos pelo programador nas linguagens de alto nível (programa fonte) para a linguagem de máquina (programa executável), a fim de poder ser executado pelo computador.
- **Interpretador:** é um programa que traduz os programas escritos pelo programador para a linguagem de máquina, no momento da execução (não existindo assim o programa executável).

1.2.1. HARDWARE

O hardware de um computador, segundo a arquitetura básica de Von Neumann, é composto de três elementos principais :

- A Unidade Central de Processamento (UCP ou CPU)
- A Memória Principal (MP)
- Os dispositivos de Entrada e Saída de Dados (E/S ou I/O)

A UCP é composta de dois elementos principais:

- A Unidade Aritmética e Lógica (ULA ou ALU) – que é responsável pela realização de todas as operações lógicas e aritméticas no computador;
- A Unidade de Controle (UC ou CU) – que é responsável pelo controle de todos os elementos do computador e dos periféricos, recebendo sinais de estado (status) destes e gerando sinais de controle para estes.

Exemplos de UCP: 80.286, 80.386, 80.486, PENTIUM, POWER PC, 68.040, etc.

A **MP** é o elemento que armazena os programas, os dados e as informações (resultados) durante a execução de um programa. Cada posição da memória possui um endereço e um conteúdo. Ao ser desligado o computador, todo o conteúdo da memória principal é perdido.

Os periféricos, ou dispositivos de **entrada e saída**, como sugere o nome, são responsáveis pela entrada de dados para UCP/MP e a saída de informações da UCP/MP. Os periféricos de E/S estão, normalmente, relacionados à memória secundária e podem ser de entrada, de saída ou de emprego duplo (entrada e saída).

Exemplos: Unidade Controladora de Disco Flexível, Unidade de Fita Magnética, Scanner, Mouse, Plotter, Impressora, Monitor, Teclado, Modem, caneta Ótica, Unidade Controladora de disco Rígido (winchester), Unidade de CD-ROM, etc.

A memória secundária (MS) é responsável pelo armazenamento, em caráter permanente, de programas e dados. É onde se localizam os arquivos de dados, de programas ou de saída (resultados).

Exemplos de MS são: discos flexíveis (disquetes), discos rígidos (winchester), cartuchos (fitas streamer), fitas magnéticas, CD-ROM, etc.

Para os que desejarem, este assunto é apresentado em maior profundidade na referência [3]. A Figura 1.1 abaixo ilustra o inter-relacionamento dos componentes acima citados:

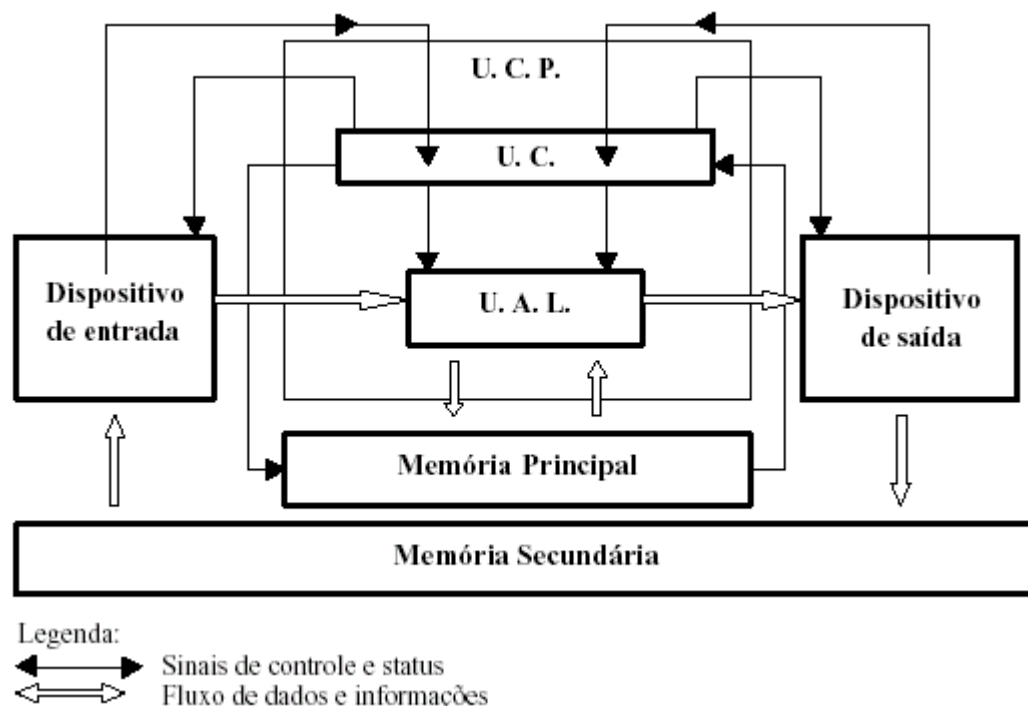


Figura 1.1- O Computador segundo Von Neumann.

1.2.2. SOFTWARE

O software de um computador é o que determina o seu uso e os resultados que serão produzidos e apresentados. Em um computador digital existem diversos tipos diferentes de software, em camadas, com finalidades e complexidades diferentes. Normalmente, quanto mais relacionado e próximo ao hardware, mais complexo e difícil de se desenvolver e manter ele é.

A Figura 1.2, que se segue, procura ilustrar o que foi dito. Partindo-se do hardware que está no centro, o Sistema Operacional com seus diversos programas é a primeira camada de software do nosso computador. É muito complexo desenvolver-se um sistema operacional como DOS, UNIX, VMS, WINDOWS 2000 e outros que você já deve ter ouvido falar.

Externamente ao sistema operacional vamos encontrar os compiladores e interpretadores das linguagens de programação que, de certa forma, traduzem os programas aplicativos para a linguagem de máquina que o computador entende. Nesta segunda camada encontramos também os processadores ou editores de texto (Wordstar, WordPerfect, Fácil, Carta Certa, Word, etc.), os gerenciadores de bancos de dados (Dbase, Dialog, Paradox, FoxBase, Access, etc.), as planilhas (Lotus123, Quattro, Excel, etc.) e muitos outros programas utilitários (PCTools, Norton, Utilities, Quik Link, etc.). Na camada mais externa encontramos os programas aplicativos que podem ser desenvolvidos utilizando-se os recursos da camada anterior e nas linguagens de programação, utilizando os interpretadores ou os compiladores para poderem ser executados.

Para o desenvolvimento destes programas é que se faz uso das técnicas de construção de algoritmos, de forma a se garantir que os programas serão gerados com um mínimo de erro e poderão ser mantidos, sem dificuldade, por um programador, que não o tenha desenvolvido.

Ao contrário do que muitos pensam, um computador não faz nada sozinho. Ele é uma máquina rápida, que resolve problemas bem definidos e repetitivos, mesmo complexos, mas somente se for bem programado. Ou seja: se temos LIXO na entrada (quer na forma de maus programas ou dados ruins), teremos LIXO na saída (nossos resultados).

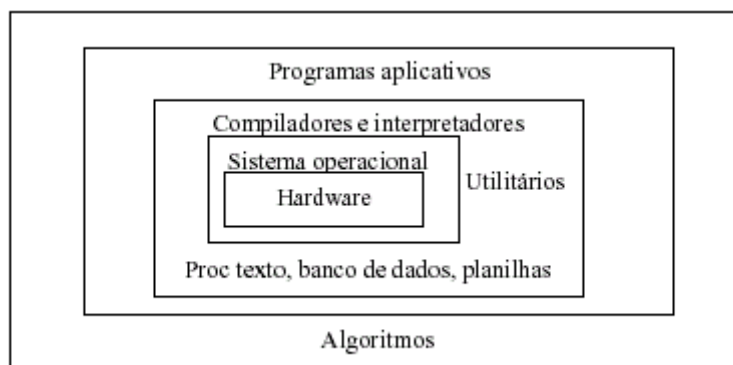


Figura 1.2- As camadas de software em um computador.

1.3. O DESENVOLVIMENTO DE SISTEMAS DE INFORMAÇÃO E O PROGRAMADOR

Um sistema de informação pode ser definido como um sistema baseado em computador que auxilia, automatiza e otimiza o funcionamento de qualquer atividade através da(o):

- Redução da participação do homem em atividades rotineiras e repetitivas; Aumento da facilidade na coleta e armazenamento de dados e na rapidez de recuperação e manuseio;
- Redução do número de erros produzidos em operações de coleta, arquivamento e recuperação de dados e de informações;
- Aumento de facilidade e flexibilidade na geração de relatórios, entre outros.

Uma pergunta que muitos se fazem é: Qual o papel do programador e do analista de sistemas no desenvolvimento de sistemas de informação? Para respondê-la, vejamos o Ciclo de Vida de um sistema. Para desenvolvimento de qualquer sistema informatizado de boa qualidade há que se passar pelas seguintes fases:

- 1ª Fase: Estudo de Viabilidade (Estudos Iniciais)
- 2ª Fase: Análise detalhada do sistema (Projeto Lógico)
- 3ª Fase: Projeto preliminar do sistema (Projeto Físico)
- 4ª Fase: Projeto detalhado do sistema (Algoritmos)
- 5ª Fase: Codificação ou implementação (na linguagem escolhida)
- 6ª Fase: Testes
- 7ª Fase: Implementação, operação e manutenção

As três primeiras fases normalmente são de responsabilidade dos Analistas de Sistemas. Nas outras podem trabalhar indistintamente, dependendo da complexidade e da situação, programadores, analistas ou programadores e analistas. A construção dos algoritmos aparece então na fase do projeto detalhado do sistema. Após definidos e criados todos os algoritmos temos de codificá-los na linguagem escolhida. Para essa tarefa o programador deverá conhecer a máquina e o compilador a serem utilizados, e esse assunto será coberto na matéria de Linguagem de Programação.

No desenvolvimento de um projeto, quanto mais tarde um erro é detectado, mais dinheiro e tempo se gasta para repará-lo. Assim, a responsabilidade do programador é maior na criação dos algoritmos do que na sua implementação pois, quando bem projetados, não se perde muito tempo em refazê-los reimplantá-los e retestá-los, assegurando assim um final feliz e no prazo previsto para o projeto. Entretanto, num projeto, o trabalho de desenvolvimento de algoritmos e programas poderá ter de ser todo refeito se houver problemas nas três primeiras fases, mostrando assim a importância do trabalho do Analista de Sistemas.

CAPÍTULO 2 - RESOLUÇÃO DE PROBLEMAS: ALGORITMOS

2.1. PROBLEMAS E SOLUÇÕES

Vamos supor que Temos que pregar um prego em um pedaço de madeira. Para realizar esta tarefa teremos que segurar o prego sobre a madeira e bater com o martelo tantas vezes quantas forem necessárias até que o prego entre por inteiro.

Uma solução para este problema seria:

1º repetir a seguinte seqüência de ações:

segurar o prego sobre a madeira com a mão esquerda;

bater com o martelo no prego, com a mão direita;

verificar se o prego já está todo dentro da madeira.

O qual pode se notar nesse exemplo é que haverá uma repetição de ações até que uma determinada condição esteja satisfeita (o prego esteja dentro da madeira).

Suponha que você precise realizar uma seleção de candidatos para um emprego e há dois requisitos a serem preenchidos. Você deverá contratar os que preencherem os dois requisitos, anotar os dados de quem preencher apenas um dos requisitos e dispensar os que não preencherem nenhum dos dois requisitos. Poderíamos escrever uma solução para este problema da seguinte forma:

1º chamar o candidato;

2º se preencher os dois requisitos então contratar;

3º caso contrário, se preenche um ou outro requisito então anotar seus dados;

4º senão dispensá-lo.

O que se pode observar nesta solução é que estamos fazendo uma seleção do candidato que satisfaz as duas condições (requisitos) para o emprego, e também uma segunda seleção (possivelmente para uso futuro) dos candidatos que satisfazem apenas uma das condições.

Estas seqüências de passos, que resolvem cada um dos exemplos propostos é o que pode ser chamado de algoritmo para solução do problema. Elas estão a exemplificar as três formas básicas de se definir uma solução para qualquer problema:

SEQUÊNCIA**REPETIÇÃO****SELEÇÃO**

Naturalmente que os algoritmos para problemas resolvidos com o auxílio de computadores não serão tão simples e nem apresentados com os exemplos mostrados. Veremos isto nos capítulos que se seguem, mas a figura abaixo dá uma idéia da utilidade dos Algoritmos.

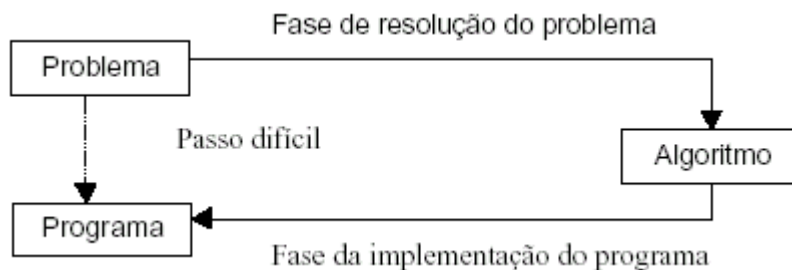


Figura 2.1- A solução de problema com o uso do computador.

2.2. DEFINIÇÃO DE ALGORITMO

- O conceito central da programação e da Ciência da Computação é o conceito de algoritmos, isto é, programar é basicamente construir algoritmos.
- É a descrição, de forma lógica, dos passos a serem executados no cumprimento de determinada tarefa.
- O algoritmo pode ser usado como uma ferramenta genérica para representar a solução de tarefas independente do desejo de automatizá-las, mas em geral está associado ao processamento eletrônico de dados, onde representa o rascunho para programas (Software).
- Serve como modelo para programas, pois sua linguagem é intermediária à linguagem humana e às linguagens de programação, sendo então, uma boa ferramenta na validação da lógica de tarefas a serem automatizadas.
- Um algoritmo é uma receita para um processo computacional e consiste de uma série de operações primitivas, interconectadas devidamente, sobre um conjunto de objetos. Os objetos manipulados por essas receitas são as variáveis.
- Algoritmo pode ter vários níveis de abstrações de acordo com a necessidade de representar ou encapsular detalhes inerentes às linguagens de programação.
- Como qualquer modelo, um algoritmo é uma abstração da realidade. A abstração é o processo de identificar as propriedades relevantes do fenômeno que está sendo modelado. Usando o modelo abstrato, podemos nos centrar unicamente nas propriedades relevantes para nós, dependendo da finalidade da abstração, e ignorar as irrelevantes.

- É a forma pela qual descrevemos soluções de problemas do **nosso mundo**, afim de, serem implementadas utilizando os recursos do **mundo computacional**. Como este possui severas limitações em relação ao nosso mundo, exige que, sejam impostas algumas regras básicas na forma de solucionar os problemas, para que, possamos utilizar os recursos de **hardware** e **software** disponíveis. Pois, os algoritmos, apesar de servirem para representar a solução de qualquer problema, no caso do Processamento de Dados, eles devem seguir as regras básicas de programação para que sejam compatíveis com as **linguagens de programação**.
- Um algoritmo é formalmente uma seqüência finita de passos que levam a execução de uma tarefa. Podemos pensar em algoritmo como uma receita, uma seqüência de instruções que dão cabo de uma meta específica. Estas tarefas não podem ser redundantes nem subjetivas na sua definição, devem ser claras e precisas. Como exemplos de algoritmos podemos citar os algoritmos das operações básicas (adição, multiplicação, divisão e subtração) de números reais decimais. Outros exemplos seriam os manuais de aparelhos eletrônicos, como um videocassete, que explicam passo a passo como, por exemplo, gravar um evento.
- De um modo informal, um algoritmo é um procedimento computacional bem definido que toma como parâmetro de entrada um valor (ou um conjunto de valores) e que produz como saída um valor (ou um conjunto de valores). Ou seja, é uma seqüência de passos computacionais que transformam um "input" num "output". Sob um outro ponto de vista um algoritmo é uma ferramenta que permite resolver um problema computacional específico.
- Um Algoritmo é a descrição de um padrão de comportamento, expresso em termos de um repertório bem definido e finito de ações "primitivas", as quais damos por certo que podem ser executadas [2].

As definições acima mostram que um algoritmo, para ser assim definido, precisa:

1. Ter início e fim
2. Ser descrito em termos de ações não ambíguas e bem definidas
3. Que as ações sigam uma seqüência ordenada

Exemplos de algoritmos (muitos podem ser retirados do nosso dia a dia):

As instruções de operação de um equipamento;

Uma receita de bolo ou de algum prato;

Instruções de montagem;

2.3.TÉCNICAS DE REPRESENTAÇÃO DE ALGORITMOS

Os algoritmos podem ser representados de diversas formas, a saber:

1. Em uma língua (português, inglês): é utilizado nas receitas, instruções , etc. Para solução de problemas em computação apresenta um inconveniente: a ambigüidade de alguns termos.
2. Uma linguagem de programação (Pascal, Cobol, C, etc.): é utilizada por alguns programadores já experientes, porém apresenta alguns inconvenientes: utiliza apenas as instruções existentes na linguagem específica e é muito detalhada nas preocupações com a sintaxe, desviando a atenção do programador da solução do problema em si.
3. Representações gráficas: são bastante recomendáveis já que um desenho muitas substitui, com vantagem, mil palavras. Alguns exemplos (ver Figura 2.2 abaixo):

- a) fluxograma (hoje em dia estão caindo em desuso)
- b) diagramas de Nassi-Schneiderman (utilizaremos no nosso curso)
- c) método de Jackson
- d) diagramas de Warnier-Or

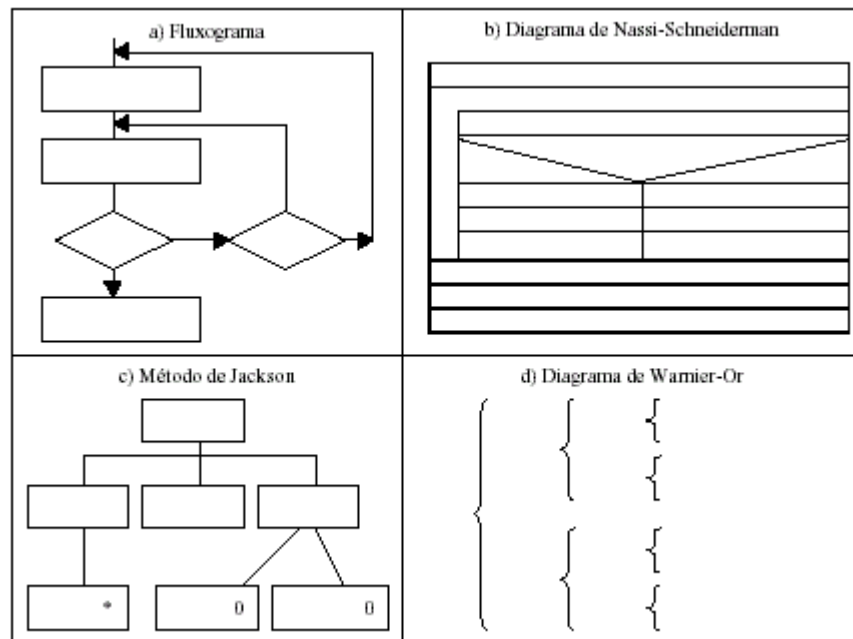


Figura 2.2 - Métodos gráficos para representação de algoritmos

2.3.1- PSEUDO-CÓDIGO

Os algoritmos são descritos em uma linguagem chamada **pseudo-código**. Este nome é uma alusão à posterior implementação em uma linguagem de programação, ou seja, quando formos programar em uma linguagem, por exemplo Visual Basic, estaremos gerando código em Visual Basic. Por isso os algoritmos são independentes das linguagens de programação. Ao contrário de uma linguagem de programação não existe um formalismo rígido de como deve ser escrito o algoritmo.

O algoritmo deve ser fácil de se interpretar e fácil de codificar. Ou seja, ele deve ser o intermediário entre a linguagem falada e a linguagem de programação. Utilizaremos em nosso curso um pseudo-código que chamaremos de nosso pseudo-código. Existem várias pseudolinguagens de programação como mostra a referência [2] que utiliza o **Portugol (Português + Algol + Pascal)**, para o estudo dos algoritmos.

2.4. ALGORITMO X PROGRAMA

Um algoritmo é uma seqüência lógica de ações a serem executadas para se executar uma determinada tarefa. Um Programa é a formalização de um algoritmo em uma determinada linguagem de programação, segundo suas regras de sintaxe e semântica, de forma a permitir que o computador possa entender a seqüência de ações.

2.5. LINGUAGEM DE PROGRAMAÇÃO

Conjunto de símbolos (comandos, identificadores, caracteres etc) e regras de sintaxe que permitem a construção de sentenças que descrevem de forma precisa ações compreensíveis e executáveis para o computador.

2.5.1. LINGUAGEM DE PROGRAMAÇÃO = SÍMBOLOS + REGRAS DE SINTAXE

Uma linguagem de programação é uma notação formal para descrição de algoritmos que serão executados por um computador. Como todas as notações formais, uma linguagem de programação tem dois componentes: Sintaxe e Semântica. A sintaxe consiste em um conjunto de regras formais, que especificam a composição de programas a partir de letras, dígitos, e outros símbolos. Por exemplo, regras de sintaxe podem especificar que cada parênteses aberto em uma expressão aritmética deve corresponder a um parênteses fechado, e que dois comandos quaisquer devem ser separados por um ponto e vírgula. As regras de semântica especificam o “significado” de qualquer programa, sintaticamente válido, escrito na linguagem.

2.5.2. TIPOS DE LINGUAGENS DE PROGRAMAÇÃO

Existem diversas linguagens de programação, cada uma com suas características específicas e com níveis de complexidade e objetivos diferentes:

- Linguagem de Máquina Única compreendida pelo computador. Especifica de cada computador.
- Linguagens de Baixo Nível Ex.: Assembly
- Linguagens de Alto Nível Utiliza instruções próximas da linguagem humana de forma a facilitar o raciocínio.

Ex.: Uso Científico : Fortran
Propósito Geral : Pascal, C, Basic
Uso Comercial : Cobol, Clipper

2.6. PROCESSO DE CRIAÇÃO E EXECUÇÃO DE UM PROGRAMA

Embora seja teoricamente possível a construção de computadores especiais, capazes de executar programas escritos em uma linguagem de programação qualquer, os computadores, existentes hoje em dia são capazes de executar somente programas em linguagem de baixo nível, a Linguagem de Máquina.

As Linguagens de Máquina são projetadas levando-se em conta os seguintes aspectos:

- Rapidez de execução de programas
- Custo de sua implementação
- Flexibilidade com que permite a construção de programas de nível mais alto.

Por outro lado, linguagens de programação de alto nível são projetadas em função de :

- Facilidade de construção de programas
- Confiabilidade dos programas

O PROBLEMA É : Como a linguagem de nível mais alto pode ser implementada em um computador, cuja linguagem é bastante diferente e de nível mais baixo ?

SOLUÇÃO : Através da tradução de programas escritos em linguagens de alto nível para a linguagem de baixo nível do computador.

Para isso existem três tipos de programas tradutores : Montadores, Interpretadores e Compiladores.

MONTADOR Efetua a tradução de linguagem de montagem (Assembly) para a linguagem de máquina.

1. Obtém próxima instrução do Assembly
2. Traduz para as instruções correspondentes em linguagem de máquina
3. Executa as instruções em linguagem de máquina
4. Repete o passo 1 até o fim do programa

INTERPRETADOR Efetua a tradução a de uma linguagem de alto nível para linguagem de máquina da seguinte forma:

1. Obtém próxima instrução do código-fonte em linguagem de alto nível

2. Traduz para as instruções correspondentes em linguagem de máquina
3. Executa as instruções em linguagem de máquina
4. Repete o passo 1 até o fim do programa

COMPILADOR Efetua a tradução de todo o código-fonte em linguagem de alto nível para as instruções correspondentes em linguagem de máquina, gerando o código-objeto do programa. Em seguida é necessário o uso de um outro programa (Link-Editor) que é responsável pela junção de diversos códigos-objeto em um único programa executável.

2.6.1. POSSIBILIDADES DE ERROS NUM PROGRAMA

Erros de Compilação : Erros de digitação e de uso da sintaxe da linguagem

Erros de Link-Edição : Erro no uso de bibliotecas de sub-programas necessárias ao programa principal.

Erros de Execução : Erro na lógica do programa (algoritmo).

2.7. ESTRATÉGIAS A SEREM SEGUIDAS NA RESOLUÇÃO DE PROBLEMAS

2.7.1. CRITÉRIOS DE QUALIDADE DE UM PROGRAMA

Refere-se à precisão das informações manipuladas pelo programa, ou seja, os resultados gerados pelo processamento do programa devem estar corretos, caso contrário o programa simplesmente não tem sentido.

Clareza : Refere-se à facilidade de leitura do programa. Se um programa for escrito com clareza, deverá ser possível a outro programador seguir a lógica do programa sem muito esforço, assim como o próprio autor do programa entendê-lo após ter estado um longo período afastado dele.

Simplicidade : A clareza e precisão de um programa são normalmente melhoradas tornando as coisas o mais simples possível, consistentes com os objetivos do programa. Muitas vezes torna-se necessário sacrificar alguma eficiência de processamento, de forma a manter a estrutura do programa mais simples.

Eficiência : Refere-se à velocidade de processamento e a correta utilização da memória. Um programa deve ter performance SUFICIENTE para atender às necessidade do problema e do usuário, bem como deve utilizar os recursos de memória de forma moderada, dentro das limitações do problema.

Modularização : Durante a fase de projeto, a solução do problema total vai sendo fatorada em soluções de subproblemas, o que permite geralmente dividir o problema em forma natural em

módulos com subfunções claramente delimitadas, que podem ser implementados separadamente por diversos programadores de uma equipe, ou seja, a **modularização** consiste no particionamento do programa em módulos menores bem identificáveis e com funções específicas, de forma que o conjunto desses módulos e a interação entre eles permite a resolução do problema de forma mais simples e clara.

Generalidade : É interessante que um programa seja tão genérico quanto possível de forma a permitir a reutilização de seus componentes em outros projetos.

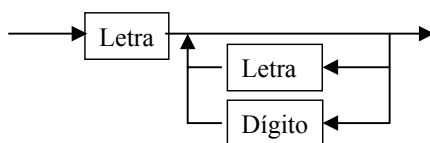
2.7.2. METODOLOGIA DE SOLUÇÃO

- 1- Entender o problema;
 - 2- Formular um esboço da solução;
 - 3- Fazer uma primeira aproximação das variáveis necessárias;
 - 4- Rever os passos originais, detalhando;
 - 5- Se o algoritmo estiver suficientemente detalhado, testar com um conjunto de dados significativos;
 - 6- Implementar numa linguagem de programação.
- Utilizaremos em nosso curso uma pseudolinguagem de programação chamada **PORTUGOL**, cujo objetivo é obter-se uma notação para algoritmos a ser utilizada na definição, na criação, no desenvolvimento e na documentação de um programa, e não criar mais uma linguagem de programação.

CAPÍTULO 3 - ALGORÍTMOS

3.1. IDENTIFICADORES, CONSTANTES, VARIÁVEIS E TIPOS BÁSICOS

- **Identificadores:** Representam os nomes escolhidos para rotular as variáveis, procedimentos e funções, normalmente, obedecem as seguintes regras :



1. O primeiro caracter deve ser uma letra
2. Os nomes devem ser formados por caracteres pertencentes ao seguinte conjunto :
{a,b,c,...z,A,B,C,...Z,0,1,2,...,9,_}
3. Os nomes escolhidos devem explicitar seu conteúdo.

EX: A, B1, BC3D,SOMA, CONTADOR

obs.: Um exemplo de identificador inválido seria 2AB ou qualquer outro iniciado por um dígito

- **CONSTANTE (“constant”)** – Uma constante (Const), como sugere o nome, é um identificador que armazena um valor fixo e imutável, durante a execução de um algoritmo ou programa. Podemos associá-lo a uma posição de memória (endereço) que tem um conteúdo fixo. Este conteúdo poderá ser um número (real ou inteiro), uma cadeia de caracteres (texto) ou um valor lógico (será definido abaixo).
- **VARIÁVEL (“variable”)** – Uma variável (Var) é um identificador que, como sugere o nome, possui o conteúdo variável durante a execução de um algoritmo ou programa. Podemos associar uma variável a uma posição da memória (endereço) e poderemos armazenar (guardar) neste endereço qualquer valor do conjunto de valores de um tipo básico associado a ela. Uma variável pode assumir vários valores diferentes ao longo da execução do programa, mas, em um determinado momento, possui apenas um valor. Unidades básicas de armazenamento das informações a nível de linguagens de programação. Os tipos de dados e variáveis utilizados dependem da finalidade dos algoritmos, mas, podemos definir alguns, pelo fato de serem largamente utilizados e implementados na maioria das linguagens:

Esse tipo básico poderá ser:

INTEIRO (“int, short int ou long int”): : qualquer número inteiro, negativo, nulo ou positivo.

Ex: -2, -1, 0...

Operações: soma(+), subtração(-), multiplicação(*), divisão inteira(/), resto(%) e comparações.

REAL (“float ou double”): qualquer número real, negativo, nulo ou positivo.

Ex: 2.5, 3.1

Operações: soma(+), subtração(-), multiplicação(*), divisão exata(/) e comparações.

CARACTER (“char”): qualquer conjunto de caracteres alfanuméricos.

Ex: A, B, "ABACATE "

Operações: comparações

TEXTO OU CADEIA DE CARACTERES (“STRING”): uma variável deste tipo poderá armazenar uma cadeia de caracteres de qualquer tamanho. Caso seja imprescindível para o entendimento pode-se acrescentar, entre parênteses, a quantidade máxima de caracteres. (Exemplo: texto (10)).

Obs.: Os textos deverão ser representados sempre entre apóstrofes para que não se confundam com os valores numéricos. Veja que o inteiro 5, é diferente do texto ‘5’.

LÓGICO (“BOOLEAN”): tipo especial de variável que armazena apenas os valores V e F, onde V representa VERDADEIRO e FALSO

Ex: e, ou, não

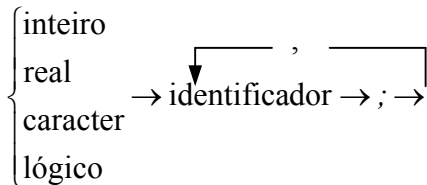
Operações: Verdadeiro ou Falso

3.2. DECLARAÇÃO DE VARIÁVEIS

Consiste na definição dos nomes e valores das constantes e dos nomes e tipos das variáveis que serão utilizadas pelos algoritmos, previamente à sua utilização, incluindo comentário, quando se fizerem necessários.

Na maioria das linguagens de programação, quando o computador está executando um programa e encontra uma referência a uma variável ou a uma constante qualquer, se esta não tiver sido previamente definida, ele não saberá o que fazer com ela. Da mesma forma, um programador que estiver implementando um algoritmo, em alguma linguagem de programação, terá o seu trabalho simplificado se todas as constantes e variáveis referenciadas no algoritmo tiverem sido previamente declaradas. As constantes são declaradas antes das variáveis. Vejamos os formatos da declaração e alguns exemplos.

O significado da declaração de variáveis corresponde à criação de locais na memória rotulada com o nome da variável (identificador) e marcada com o tipo de valores que ela pode conter. Para que os programas manipulem valores, estes devem ser armazenados em variáveis e para isso, devemos declará-las de acordo com a sintaxe:



Ex:

Inteiro X1;

obs.: X1 é o nome de um local de memória que só pode conter valores do tipo inteiro

real SOMA, MÉDIA;

caracter frase, nome;

inteiro X1;

real A,B;

lógico TEM;

3.2.1. PALAVRAS RESERVADAS

São palavras que terão uso específico no nosso pseudo-código e que não deverão ser usadas como identificadores, para não causar confusão na interpretação.

Exemplo: Algoritmo, Programa, Bloco, Procedimento, Inteiro, Real, Texto, Const, Var, Tipo, Início, Imprima, Se, Então, Senão, Enquanto, Repita, Variando, Faça, Caso, Até, Vetor, Matriz, Registro, Fim, Execute, Procedimento, Função, etc....

O significado de cada um desses termos será visto e entendido nos itens e capítulos que se seguem.

3.3. COMANDO SIMPLES

É uma instrução simples.

leia(x);

3.4. COMANDO COMPOSTO

um grupo de comandos simples que executam alguma tarefa.

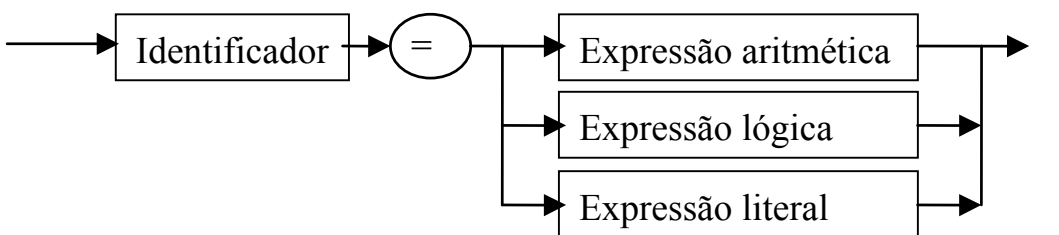
Início

```
leia(x);
y = 2*x;
fim.
```

3.5. OPERADORES

Na solução da grande maioria dos problemas é necessário que as variáveis tenham seus valores consultados ou alterados e, para isto, devemos definir um conjunto de **OPERADORES**, sendo eles:

- OPERADOR DE ATRIBUIÇÃO**



Ex: O valor da expressão é atribuído ao identificador(variável).

```
X = 2; y = 5-x;
```

Este comando permite que se forneça ou altere o valor de uma determinada variável, onde o tipo desse valor seja compatível ao tipo de variável na qual está sendo armazenado, de acordo com o especificado na declaração.

```
NUM = 8 {A variável NUM recebe o valor 8}
```

```
NOME = "Guilherme" {A variável NOME recebe o valor 'Guilherme'}
```

```
CONT = 0
```

```
AUXNOME = NOME {A variável AUXNOME recebe o conteúdo da variável NOME}
```

```
ACHOU = falso {A variável ACHOU recebe o valor falso}
```

- OPERADORES ARITMÉTICOS:**

+ = Adição	/ = Quociente da divisão de inteiros 5/2=2
* = Multiplicação	% = Resto da divisão de inteiros 5%2=1
- = Subtração ou inverso do sinal.	a**b = Exponenciação a ^b
/ = Divisão	TRUNCA(A) - A parte inteira de um número Fracionário.
SINAL(A) - Fornece o valor -1, +1 ou zero conforme o valor de A seja negativo, positivo ou igual a zero.	ARREDONDA (A) - Transforma por arredondamento, um número fracionário em inteiro.

Obs. FUNÇÕES PRIMITIVAS: As funções mais comuns de matemática são também definidas e válidas no nosso Pseudo Código.

Exemplos:

LOG (X) , {dá o logaritmo na base 10 de X}

SEN (X), {dá o seno de X}

ABS (X), {dá o valor absoluto de X}

INT (X), {dá a parte inteira de um real X}

ARREDONDA (x), {arredonda um real para inteiro}

RAIZ (X), {dá a raiz quadrada de X} etc. ...

- **OPERADORES RELACIONAIS:**

São utilizados para relacionar variáveis ou expressões, resultando num valor lógico (Verdadeiro ou Falso), sendo eles:

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

Exemplos:

NUM = 3

NOME = 'DENISE'

NUM > 5 é falso (0) {3 não é maior que 5}

NOME < 'DENIZ' é verdadeiro (1) {DENISE vem antes de DENIZ}

(5 + 3) >= 7 é verdadeiro

NUM != (4 - 1) é falso {lembre-se que NUM "recebeu" 3}

- **OPERADORES LÓGICOS:**

São utilizados para avaliar expressões lógicas, sendo eles:

e ("and") \cap : conjunção

ou ("or") \cup : disjunção

não ("not") \emptyset : negação

Exemplos:

ACHOU = falso

NUM =9

(4>5) e (5>3) =>falso e verdadeiro (1) = falso (0)

não ACHOU =>verdadeiro

- **PRIORIDADE DE OPERADORES:**

Durante a execução de uma expressão que envolve vários operadores, é necessário a existência de prioridades, caso contrário poderemos obter valores que não representam o resultado esperado.

A maioria das linguagens de programação utiliza as seguintes prioridades de operadores :

1º - Efetuar operações embutidas em parênteses "mais internos"

2º - Efetuar Funções

3º - Efetuar multiplicação e/ou divisão

4º - Efetuar adição e/ou subtração

5º - Operadores Relacionais

6º - Operadores Lógicos

Ou seja,

Primeiro: Parênteses e Funções

Segundo: Expressões Aritméticas

1) +, - (Unitários)

2) **

3) *, /

4) +, - (binário)

Terceiro: Comparações

Quarto: Não

Quinto: e

Sexto: ou

OBS: O programador tem plena liberdade para incluir novas variáveis, operadores ou funções para adaptar o algoritmo as suas necessidades, lembrando sempre, de que, estes devem ser compatíveis com a linguagem de programação a ser utilizada.

3.6. COMANDOS DE ENTRADA E SAÍDA

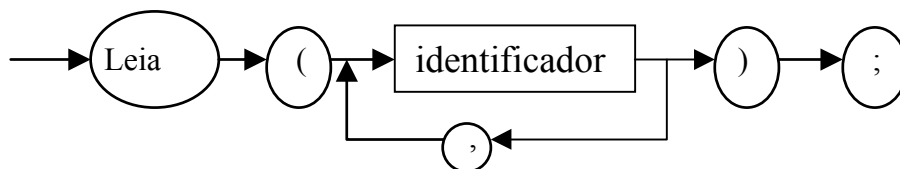
No algoritmo é preciso representar a troca de informações que ocorrerá entre o mundo da máquina e o nosso mundo, para isso, devemos utilizar comandos de entrada e saída, sendo que, a nível de algoritmo esses comandos representam apenas a entrada e a saída da informação, independente do dispositivo utilizado (teclado, discos, impressora, monitor,...), mas, sabemos que nas linguagens de programação essa independência não existe, ou seja, nas linguagens de programação temos comandos específicos para cada tipo de unidade de Entrada/Saída.

- **Comando de Entrada de Dados**

Para que possamos obter dados do meio exterior para uso do computador (memória principal), estes têm de vir através dos dispositivos de entrada. Da mesma forma, as informações que são produzidas, tem de ser levadas ao meio externo (um arquivo, um a impressora, uma tela etc.) através de um dispositivo de saída. Para isso, utilizamos dois comandos assim definidos:

Comando **Leia**(scanf no c): lê, do meio externo, a próxima informação disponível para leitura e armazena na(s) variável(eis) discriminada(s) após o comando, entre parênteses. Mais tarde aprenderemos como especificar a leitura de um dado que está armazenado em um arquivo e de que arquivo está sendo lido o dado.

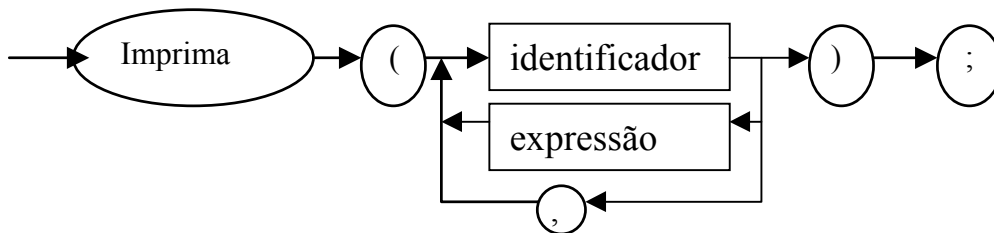
Leia(variável_1, variável_2,...);



Ex: leia (v); O valor da variável (v) é dado por um dispositivo de entrada.

- **Comando de Saída de Dados**

Comando **Imprima** (printf no c): Imprime (na tela ou na impressora) o conteúdo da(s) variável(eis) especificada(s) após o comando, entre parênteses. Não será preocupação nossa a formatação de relatórios, mas o comando permite a impressão de texto (entre “), se for necessária para clareza ou especificação do que está sendo impresso.



Ex: imprima (x); O valor atual da variável (x) é informado para um dispositivo de saída.

3.7. ESTRUTURA DE UM ALGORITMO EM PSEUDO-CÓDIGO E DNS

As estruturas de controle introduzidas nesta unidade facilitarão a expressão de algoritmos e permitirão que exemplos mais significativos sejam apresentados e propostos, visando o aperfeiçoamento do desenvolvimento do processo de aprendizado da construção de algoritmos.

Descreveremos a seguir os blocos e os comandos básicos de controle do PORTUGOL. Serão introduzidos paralelamente os fluxogramas de CHAPIN criados por Ned Chapin a partir dos

trabalhos de Nassi & Shneiderman e o fluxograma tradicional, como formas alternativas de descrição da estrutura dos comandos.

- **PORTUGOL**

Algoritmo

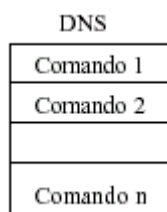
Início

<declarações de variáveis>;

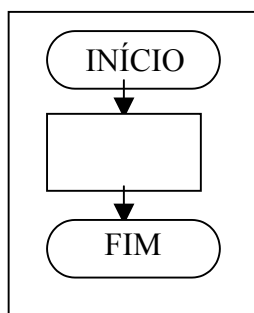
<comandos>;

Fim.

- **DNS**



- **TRADICIONAL**



O diagrama tradicional é uma forma padronizada para representar os passos lógicos de um determinado processamento.

Com o diagrama podemos definir uma seqüência de símbolos, com significado bem definido, portanto, sua principal função é a de facilitar a visualização dos passos de um processamento.

- Blocos pré-definidos (teste, desvio, impressão);
- Muita flexibilidade;
- Descrição do raciocínio por meio de símbolos gráficos.



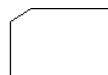
Cálculo



Decisão



Display



Entrada de Dados



Saída de Dados



Início/Término

Problemas

- Desenhos confusos e complexos;
- Difícil manutenção (perigosa);
- Não é utilizado na prática;

Observações:

1 - As declarações têm o mesmo formato tanto para o PC, DNS e fluxograma , e por isso não foram repetidas acima.

2 - Quando fazemos um programa, uma boa idéia é usar comentários que ajudem a elucidar o funcionamento do mesmo. Os comentários devem vir entre `/* */`

Ex=

`/* Um Primeiro Programa */`. O compilador C desconsidera qualquer coisa que esteja começando com `/*` e terminando `/*`.

3.8. DOCUMENTAÇÃO

Alguns princípios básicos abaixo mencionados, deverão ser usados desde a criação do primeiro algoritmo:

- Ao criar as constantes e variáveis, utilizar nomes significativos e comentar, se necessário for;
- Utilizar indentação (três espaços como sugestão), para mostrar a estrutura lógica e seqüência de comandos, quando usando o nosso pseudo-código. Com o DNS não é preciso;
- Utilizar, opcionalmente, um dicionário de dados que especifique formalmente o conteúdo de cada constante e variável de todo algoritmo ou programa;
- Utilizar parênteses para evitar ambigüidade nas expressões;

CAPÍTULO 4 - ESTRUTURAS BÁSICAS DE CONTROLE

No capítulo anterior, foram apresentados exemplos das três estruturas necessárias para representação da solução de qualquer algoritmo: a seqüência, a seleção e a repetição. Em se tratando de algoritmo para computadores, as três também formam a base das estruturas de controle de execução. Vejamos como formalizaremos a representação e o uso dessas estruturas no nosso pseudo-código e nos Diagramas de Nassi-Schneiderman.

4.1. SEQÜÊNCIA

Grupo de comandos em que são executados um após o outro.

- **Nosso Pseudo-código**

```
Início
  comando1;
  comando2;
  .
  .
  comandoN;
Fim.
```

- **DNS**

Formato (DNS):

comando1
comando2
comandoN

4.2. SELEÇÃO

Também chamada de estrutura de decisão ou de processamento condicional, a estrutura de seleção é utilizada quando a execução de um comando (ou uma seqüência de comandos) depende de um teste anterior (uma ou mais comparações). A seleção pode ser simples ou composta.

4.2.1. SELEÇÃO SIMPLES

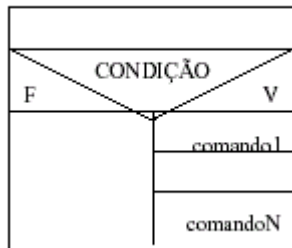
Quando a execução de um comando (ou de uma seqüência de comandos) depender de uma condição verdadeira, e não há comandos a executar se a condição for falsa.

- **Nosso Pseudo-código**

```
Se <condição> Então
    (comandos);
Fim Se;
```

- **DNS**

Formato (DNS):



onde a CONDIÇÃO poderá ser também uma expressão lógica.

Exemplo1: leia um número inteiro e o imprima se ele for diferente de 0 (ZERO)

Exemplo2:

```
Se (a<5) Então
    a=5;
Fim Se;
```

4.2.2. SELEÇÃO COMPOSTA

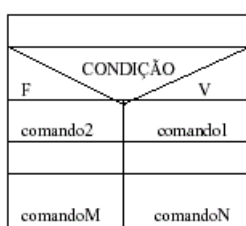
Quando se executa um comando (ou seqüência de comando) se uma condição é verdadeira, e se executa um outro comando (ou seqüência de comandos) se a condição é falsa.

- **Nosso Pseudo-Código**

```
Se <condição> Então
    (Comandos);
Senão
    (comandos);
Fim Se;
```

- **DNS**

Formato (DNS):



4.2.3. ANINHAMENTO DE SELEÇÕES

A estrutura de Seleção permite o aninhamento, ou seja, o comando a ser executado dentro de uma seleção (por exemplo, no "Senão") pode ser uma outra seleção. Outro aninhamento poderá ocorrer também com esta última seleção e assim por diante. Nos casos de vários aninhamentos subseqüentes, uma boa indentação será fundamental para o entendimento de algoritmo quando utilizando pseudo-código.

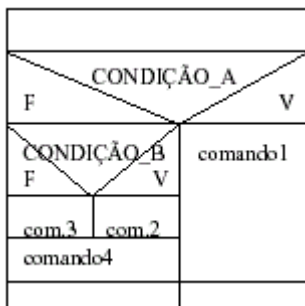
- **Nosso Pseudo-Código**

```

Início
-
Se CONDIÇÃO_A Então {V}
  comando1;
Senão {F}
  Se CONDIÇÃO_B Então
    comando2;
  Senão
    comando3;
  Fim Se;
Fim se;
-
-
Fim.

```

- **DNS**



obs: o comando se é equivalente ao if na linguagem c.

4.3. ESTRUTURAS DE REPETIÇÃO - LAÇOS(LOOPS)

4.3.1. LAÇO ENQUANTO(WHILE)

A estrutura de repetição (enquanto) é utilizada quando um conjunto de comandos deve ser executado repetidamente, enquanto uma determinada condição (expressão lógica) permanecer verdadeira. Dependendo do resultado do teste da condição, o conjunto de comandos poderá não ser executado nem uma vez (se for falsa no primeiro teste), ou será executado várias vezes (enquanto for verdadeira). Chama-se a isso um laço ("loop").

Da mesma forma que a estrutura de seleção, ela permite o aninhamento de repetições, ou seja, a existência de uma estrutura de repetição dentro de outra. Poderão haver também aninhamentos de seleções dentro de estruturas repetitivas e vice-versa. Dois cuidados ao criar estruturas de repetição (enquanto):

1. Inicializar a(s) variável(eis) que controla(m) o laço antes do início do laço;
2. Inicializar a(s) variável(eis) que controla(m) o laço dentro do laço (seja por leitura ou por atribuição), pois se isto não for feito cairemos no que chamamos um laço infinito e de lá o nosso programa não sairá.

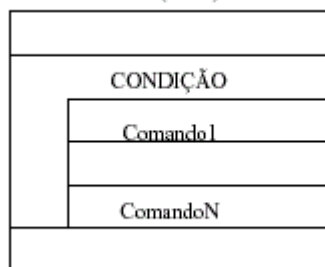
- **Nosso Pseudo-Código**

```

Enquanto <Condição> Faça
    (Comandos);
Fim Enquanto;
```

- **DNS**

Formato (DNS):



Exemplo:

Início

```

Inteiro x;
x = 2;
Enquanto (x<10) Faça
    Imprima (x);
    x=x+1;
Fim Enquanto;
Imprima(x);
```

Fim.

Exercício: faça um trecho de algoritmo para ler e imprimir o nome e a idade de várias pessoas, até encontrar alguém com 65 anos, quando deverá ser impresso, além do nome, uma mensagem informando o fato.

4.3.2. CONTROLADOS POR CONTADOR

Uma variável é fornecida com o n.º de vezes que será repetido o laço.

Algoritmo Abono_por_Contador

Início

Leia (Numero_de_Funcionários);

Contador = 0;

Enquanto Contador < Número_de_Funcionários Faça

Leia (Nome);

Contador = Contador+1;

Fim Enquanto;

Fim.

4.3.3. REPETIÇÃO COM TESTE NO FINAL

Quando se deseja executar a série de comandos uma vez pelo menos, pode se fazer o teste no final. Essa variação tem um uso bastante efetivo na validação de dados de entrada, pelo teclado, mas pode ser sempre substituída por um enquanto. Uma vantagem do repita é que não é preciso inicializar a(s) variável(eis) de controle do laço antes de entrar no mesmo. Deve-se, contudo, ter o cuidado de modificá-la(s) dentro do laço para que não caiamos em um laço infinito. Executa uma instrução e faz um teste lógico. Dependendo da resposta, fica repetindo o processo até o teste lógico dar Verdadeiro.

- **Nosso Pseudo-Código**

repita

C1;

C2;

.

.

Cn;

até <condição>;

- **DNS**

C1
C2
.
.
Cn
<condição>

- **Significado deste comando:**

Os comandos C1, C2, ..., Cn são executados pelo menos uma vez. Quando a condição é encontrada, ela é testada, se for verdadeira o comando seguinte será executado, se for falsa, os comandos C1, C2, ..., Cn são reexecutados até que a condição se torne verdadeira.

O comando repita-até é equivalente ao comando enquanto, conforme será mostrado no exemplo abaixo.

- Repita - até

Início

```

Inteiro x;
X = 2;
Repita
    Imprima (x);
    X = x+1;
Até (x>=10);
Imprima (x);

```

Fim.

- **Enquanto (while)**

Início

```

Inteiro: x;
x←2;
Enquanto x<10 faça
    Imprima (x);
    x←x+1;
Fim Enquanto;
Imprima (x);

```

Fim.

Observa-se que a utilização de um comando **repita** no lugar de um comando **enquanto** corresponde a utilizar como <condição> para o repita a negação da <condição> do enquanto. Numa estrutura **Enquanto**, os comandos são executados 0 ou mais vezes. Numa estrutura **Repita**, os comandos são executados 1 ou mais vezes.

4.4. ESCAPE DO LAÇO-Abandone(exit)

Comando Abandone: Força uma saída de um laço de repetição.

- **Nosso Pseudo-Código**

abandone

- CHAPIN

ABANDONE

Exemplo:

```

Enquanto (a>b) faça
  Leia (a,b);
  Imprima (a,b);
  Se b==0
  Então abandone;
  Senão a=a/b;
Fim Enquanto;
```

O comando abandone só tem sentido dentro de um comando de repetição (enquanto, repita, para). Além disso, estará sempre associado ao teste de uma condição com comando se.

- **Significado do comando abandone:** quando o abandone é encontrado, o próximo comando a ser executado é o primeiro comando logo após o fim do comando de repetição mais interno onde aparece o abandone.

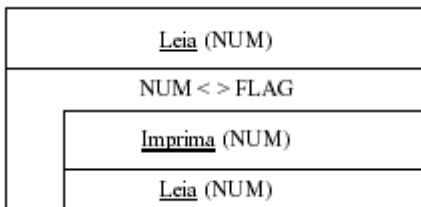
4.5. Sinalizador ('Flags')

O sinalizador ou 'flag' é um valor que, quando lido, sinaliza a saída de um laço, o fim de um programa, o fim de uma lista de dados, ou o fim de um arquivo. Para identificá-lo iremos defini-lo como uma constante nos nossos algoritmos.

Exemplo: faça um trecho de algoritmo utilizando o DNS que leia e imprima uma lista de números inteiros terminada pelo número 999.

```

{Declarações;}
Constante FLAG = 999;
Inteiro : NUM;
```



4.6. REGRAS PRÁTICAS PARA A CONSTRUÇÃO DE ALGORITMOS LEGÍVEIS

- 1- Procure incorporar comentários no algoritmo, pelo menos para descrever o significado das variáveis utilizadas.

Exemplo: /* Cálculo da média */

- 2- Escolha nomes de variáveis que sejam significativos, isto é, que traduzam o tipo de informação a ser armazenada na variável.

Exemplo: NOTA

MATRÍCULA

- 3- Procure alinhar os comandos de acordo com o nível a que pertençam, isto é, destaque a estrutura na qual estão contidos.

Exemplo:

```
B=A*C;
```

```
Enquanto P==1 Faça
```

```
    P=P+1;
```

```
    F=X+NOTA/2;
```

```
Fim Enquanto;
```

```
Imprima (F,NOTA);
```

.

- .4)- Tenha em mente que seus algoritmos deverão ser lidos e entendidos por outras pessoas (e por você mesmo) de tal forma que possam ser corrigidos, receber manutenção e serem modificados.

- 5) Escreva comentários no momento em que estiver escrevendo o algoritmo. Um algoritmo não documentado é um dos piores erros que um programador pode cometer.

- 6) Use comentários no início para explicar o que o algoritmo faz. Alguns comentários seriam;

- Uma descrição do que faz o algoritmo
- Como utilizá-lo
- Explicação do significado das variáveis mais importantes
- Estrutura de dados utilizadas
- Os nomes de quaisquer métodos especiais utilizados

- Autor
- Data da escrita

7) Utilize espaços em branco para melhorar a legibilidade. Espaços em branco, inclusive linhas em branco são valiosíssimos para melhorar a aparência de um algoritmo.

8) Escolha nomes representativos para suas variáveis.

9) Um comando por linha é suficiente

- O algoritmo fica mais legível
- O algoritmo fica mais fácil de ser depurado.

-

Exemplo:

A=14,2;I=1;enquanto I<10 faça X=X+1;K=I*K;I=I+1; fim enquanto

O mesmo exemplo com cada comando em uma linha

A=14, 2;

I=1;

Enquanto I<10 Faça

X=X+1;

K=I*K;

I=I+1;

Fim Enquanto;

10) Utilize parênteses para aumentar a legibilidade e prevenir-se contra erros.

Exemplos:

Com poucos parênteses	Com parênteses extras
A*B*C/(C*D*E)	(A*B*C)/(C*D*E)
A**B**C	(A**B)**C
A+B<C	(A+B)<C

11) Toda vez que uma modificação for feita no algoritmo, os comentários devem ser alterados

4.7. UMA REFLEXÃO ANTES DE RESOLVER DE PROBLEMAS

Passo 1 – Leia cuidadosamente a especificação do problema até o final

Passo 2- Caso não tenha entendido o problema, pergunte.

Passo 3- levantar e analisar todas as saídas exigidas na especificação do problema (impressões)

Passo 4- levantar e analisar todas as entradas citadas na especificação do problema(leituras)

Passo 5- Verificar se é necessário gerar valores internamente ao algoritmo e levantar as variáveis necessárias e os valores iniciais de cada uma;

Passo 6- Levantar e analisar todas as transformações necessárias para, dadas as entradas e valores gerados internamente, produzir as saídas especificadas;

Passo 7- Testar cada passo do algoritmo

Passo 8 – Avaliação geral, elaborando o algoritmo e revendo comentários



FACULDADES INTEGRADAS ANGLO-AMERICANO

TRABALHO DE ALGORITMOS**ENTREGA: Dia da TV1****PROFESSORA: SALETE BUFFONI**

- 1- Como ocorre o armazenamento de uma informação na memória de um computador?
- 2- Caracterize as gerações de computadores quanto a:
 - tamanho;
 - velocidade de operação;
 - custo;
 - consumo de energia;
- 3- Caracterize os sistemas de computadores quanto a sua arquitetura básica de Hardware e Software.
- 4- Cite e comente 3 dispositivos de entrada de dados
- 5- Cite e comente 3 dispositivos de saída de dados.
- 6- Comente os tipos de memórias existentes no ambiente de informática.
- 7- Através de um desenho, identifique as principais partes de um disquete e cite alguns cuidados em seu manuseio.
- 8- Através de um **DIAGRAMA DE BLOCOS**, descreva um computador em termos de partes fundamentais.
- 9- Conceitue as linguagens quanto ao seu nível.

EXERCÍCIOS INTRODUTÓRIOS - I

1) O resultado da expressão $3 + 6 * 13 = 117$ ou 81 ?

2) Sendo A,B,C e D variáveis do tipo inteiro, cujos conteúdos são: A=3, B=4, C=13 e D=4. Quais os valores fornecidos por cada uma das expressões aritméticas abaixo?

a) $100*(B/A)+C$

d) $(C+D)/C**2$

b) $A\%5 - D/2$

e) $B\%(A+1)$

c) $\text{trunca}(0,3+5)*2$

f) $\text{senal}(D-C)$

3) Qual é a primeira operação executada em cada um dos comandos abaixo?

a) $X+Y-Z;$

b) $A+B/C**2;$

c) $\text{JOÃO}+\text{JOSÉ}/\text{JOEL};$

d) $X+Y+B**2+R*3$

4) Avalie a seguinte expressão aritmética: $6+4/2*3$. O resultado desta expressão é 15; 12; 24?

5) Qual o resultado das seguintes expressões

a. $1 + 7 * 2 ** 2 - 1 = 28$

b. $3 * (1 - 2) + 4 * 2 = 5$

6) Avalie as expressões abaixo sabendo que $X = 2$ e $Y = 3$

a. $3 * y - x ** 2 * (7 - y ** 2)$

b. $2 * x * (3 - y) / (x ** 2 - y)$

7) Identifique as entradas e saídas das expressões abaixo. Escreva-as em forma computacional

a. $r = 2y^2 + 3z$

b. $x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$

8) Quais valores serão impressos após a execução do seguinte trecho de algoritmo

```
I=0;
Enquanto I<20 Faça
    Imprima (I,I**2);
    I=I+1;
Fim Enquanto;
Imprima ("I=",I);
Imprima ("UFA! CHEGUEI AQUI"); /*CUIDADO*/
```

9) Dado o algoritmo abaixo, que calcula o volume de uma esfera de raio R:

```
Início
    Real VOLUME, PI, R;
    Constante PI = 3,1416;
    R=0;
    Enquanto R<= 6 Faça
        VOLUME=4/3*PI*(R**3);
        Imprima (R,VOLUME);
        R=R+2;
    Fim Enquanto;
Fim.
```

Completar os demais valores que serão impressos

R	VOLUME DA ESFERA
0	0
2	33,51

10) Identifique o tipo de cada uma das constantes abaixo:

- a) 21
- b) "BOLA"
- c) "VERDADEIRO"
- d) $0,21 \times 10^2$
- e) falso

11) Sendo A, B, X e Y variáveis do tipo numérico, quais os resultados fornecidos por cada uma das seguintes funções, onde A= 10, B=3, X=2 e Y=1,2

a) A/B e A%B

b) X/2 e X%2

c) SINAL(X+Y-A), SINAL(A-B**2+Y) e SINAL(A-4*X)

d) ARREDONDA(A-X), ARREDONDA(B+Y) e ARREDONDA(Y-X)

e) TRUNCA(X-3,2) e TRUNCA(A/3+1)

f) ABS(A-B**3) e ABS(A-B)

d) (B+Y)/(X+1)

EXERCÍCIOS INTRODUTÓRIOS - II

1- Sendo P, Q, R e S variáveis do tipo numérico, cujos conteúdos são iguais a 2, 3, 12 e 4.5, respectivamente, quais os valores fornecidos por cada uma das expressões aritméticas abaixo?

a) $100 * Q / P + R$

b) $P * R \% 5 - Q / 2$

c) SINAL(S-R)

d) $\text{sqrt}(R + P ** 2) + \text{ARREDONDA}(S)$

e) $S \% (P + 1) - Q * R$

f) $1 + \text{sqrt}(P ** 3 + 2 * R) - \text{TRUNCA}(S - 1)$

g) $1 + (R + S) / (Q ** 2) * \text{SINAL}(2 * P * Q - R)$

2- Dadas as variáveis numéricas X, Y e Z e as variáveis literais NOME e COR, observar os resultados obtidos para as relações a partir dos valores atribuídos a estas variáveis

VARIÁVEIS					RELAÇÕES		
X	Y	Z	COR	NOME	$X ** 2 + Y > Z$	$COR == "AZUL"$	$NOME <> "JOSÉ"$
1	2	5	"AZUL"	"PAULO"			
4	3	1	"VERDE"	"JOSÉ"			
1	1	2	"BRANCO"	"PEDRO"			
1	2	1	"AZUL"	"JOSÉ"			

3- Sendo

Real SOMA, NUM, X;

Caracter NOME, COR,DIA;

Lógico TESTE,COD,TUDO;

Assinalar os comandos de atribuição considerados inválidos:

a) NOME = 5;

f) $X = X + 1$;

b) $SOMA = NUM + 2 * X$;

g) $NUM = \text{"*ABC*"}$

c) $TESTE = COD$ ou $X ** 2 \neq SOMA$

h) $SOMA + 2 = X ** 2$;

e) $TUDO = SOMA$

i) $X = NOME$

Revisão de Avaliação de Expressões e Exercícios

ATRIBUIÇÃO E EXPRESSÕES

- As variáveis podem ter seus valores alterados durante a execução do algoritmo, isto é feito através de uma operação chamada atribuição.
- Em uma atribuição temos dois elementos fundamentais: o primeiro é o elemento para o qual estaremos realizando a atribuição e o segundo elemento é o que nos fornece o valor a ser atribuído. O primeiro elemento é sempre uma variável e o segundo é uma expressão
- O tipo da variável deve ser respeitado durante uma atribuição, ou seja, a variável deve ser compatível com o valor a ser atribuído. Inicialmente o valor a ser atribuído à variável pode vir de outra variável ou de uma expressão.

Ex.:

$X = 0$;

$X = Y$;

$x = 1 + Y$;

EXPRESSÕES

- Quando juntamos vários operadores, operandos e funções temos uma expressão.
- As expressões são classificadas de acordo com o tipo de valor resultante de sua avaliação.
- Também podem ser de 3 tipos: *aritméticas*, *lógicas* e *literais*

Expressão Aritmética

- O resultado de sua avaliação é um valor numérico
- Expressão aritmética:
 - É uma constante numérica.
Ex: 4
 - Uma variável numérica.
Ex: $x*y*z$
 - Uma função numérica.
Ex: $\cos(a)$
 - Uma combinação das anteriores através dos op. Aritméticos.
Ex: $4 + x*y*z * \cos(a)$

Prioridade das operações (ou precedência de operadores)

- Avalie a seguinte expressão aritmética:
 $6+4/2*3$ Qual o resultado desta expressão? 15; 12; 24;
- Desta forma é necessário estabelecer uma prioridade para avaliar as expressões.
- Ordem de prioridade:
 - 1º. efetuar as funções
 - 2º. efetuar as potências e raízes
 - 3º. efetuar as multiplicações e divisões
 - 4º. efetuar as adições e subtrações

Operações com mesma prioridade são executadas da esquerda para a direita. Para modificar a ordem de prioridades da pré-definida, utilizam-se parênteses. As operações entre parênteses devem ser avaliadas em 1º. lugar. No exemplo anterior, para obter o resultado 15 a expressão deveria ser: $(6 + 4) / 2 * 3$ e para obter o resultado 24: $(6 + 4 / 2) * 3$

Exercícios:

1- Calcule

- a) $\text{sen}(180^\circ)$ b) $\text{abs}(-1)$ c) $\text{abs}(2)$ d) $\text{sqrt}(4)$ e) $4\%2$ f) $5\%2$
g) $3\%2$ h) $7\%7$ i) $6\%3$ j) $4 / 2 = 2$ k) $5/2$ l) $6/3$

2- Qual o resultado das seguintes expressões:

- a. $1 + 7 * 2 ** 2 - 1$
b. $3 * (1 - 2) + 4 * 2$

3- Avalie as expressões abaixo sabendo que $X = 2$ e $Y = 3$

- a. $3 * y - x ** 2 * (7 - y ** 2)$
b. $2 * x * (3 - y) / (x ** 2 - y)$

EXERCÍCIOS COMPLEMENTARES - I

1. Faça um algoritmo que leia quatro idades e calcule a média das mesmas
2. Altere o algoritmo de media de idade para ler também os nomes e ao final mostrar a mensagem com os três nomes mais a média de suas idades.
3. Construa um algoritmo que calcule a média do TV1 da turma de lógica de programação.
4. Faça um algoritmo que faça a soma dos números inteiros de 1 a 18
5. Faça um programa que leia 4 números inteiros e apresente:
 - Média dos ímpares
 - Maior número par
 - Diferença do maior menos o menor número
6. Faça um programa que leia o nome e a idade de 3 pessoas e apresente:
 - Maior idade
 - Nome da pessoa mais nova
 - Média das idades
7. Faça um programa que leia a medida do lado de um quadrado e calcule e apresente a área e o perímetro desta figura. Obs: Perímetro é a soma de todos os lados de uma figura geométrica.
8. Faça um programa que leia o raio de uma circunferência e calcule e apresente sua a área e perímetro.
9. Faça um programa que leia o valor dos lados de um triângulo retângulo e calcule e apresente a sua área.

$$\text{Área} = (\text{base} \cdot \text{altura}) / 2$$

EXERCÍCIOS COMPLEMENTARES - II

1- Identificar o tipo de cada uma das constantes abaixo:

- a) " Barbara" b) 2 c) 2.33

2) O que está errado no trecho de algoritmo abaixo

Início

```
Inteiro N, PAR, X;  
Leia (N) ;  
x←N % 2;  
Se X==0 Então  
    PAR=verdadeiro;  
Senão PAR=falso;  
Fim Se;
```

Fim.

3) Uma pesquisa sobre algumas características físicas da população de uma determinada região coletou os seguintes dados referentes a cada habitante para serem analisados

- Sexo (masculino, feminino)
- Cabelos(Louros, preto, castanho, ruivo)
- Cor dos Olhos (Azul, verde, castanho)
- Idade em anos

Para cada habitante foi digitada uma linha com esses dados e a última linha, que não corresponde a ninguém conterá o valor de idade igual a -1. Fazer um algoritmo que determine e imprima:

- a) A maior idade dos habitantes
- b) A porcentagem de indivíduos do sexo feminino cuja idade está entre 18 e 35 anos inclusive e que tenham olhos verdes e cabelos louros

4) Faça um algoritmo que calcule a soma dos números pares de 100 até 200

5) Num frigorífico existem 90 bois. Cada boi traz preso em seu pescoço um cartão contando seu número de identificação e seu peso. Fazer um algoritmo que escreva o número e peso do boi mais gordo e do boi mais magro.

6) Faça um algoritmo que calcule a soma dos números inteiros de 1 até 18

7) Fazer um algoritmo que: Leia um número indeterminado de linhas contendo cada uma a idade de um indivíduo . A última linha que não entrará nos cálculos contém o valor da idade igual a zero. Calcule e imprima a idade média deste grupo de indivíduos.

EXERCÍCIOS COMPLEMENTARES - III

- 1) Faça um algoritmo que leia um número indeterminado de números, o algoritmo encerra quando for digitado o número 99. O algoritmo deve fornecer ao final o percentual de números pares e ímpares digitados.
- 2) Faça um algoritmo que leia um número e escreva a mensagem "é primo" ou "não é primo".
Número primo: Nro divisível por 1 e por ele próprio.
- 3) Faça um algoritmo que leia 2 números inteiros de cada vez. A leitura deve ser feita enquanto não for digitado o número 0. Os números lidos deverão ser apenas números ímpares e, quando forem consecutivos deverão ser multiplicados. Ao finalizar o algoritmo deverá escrever a soma das multiplicações.
- 4) Faça um algoritmo que leia um valor e some 10 valores pares a partir do valor lido, escrevendo no final o resultado obtido. OBS: o valor lido pode ser ímpar.
- 5) Faça um algoritmo que fique lendo valores até encontrar o valor zero, com cada valor lido faça a soma de 10 valores subseqüentes e mostre a soma e a média desses valores.
- 6) Fazer um algoritmo que fique lendo valores, um de cada vez e faça o seguinte:
 - se for 0 (zero) escreva "desligue tudo" e termine o algoritmo;
 - se for 1 (um) escreva "ligar TV";
 - se for 2 (dois) escreva "ligar rádio";
 - se for 3 (três) escreva "ligar a luz".
 - Se for outro valor escreva "tente novamente"

- 7) Escrever um algoritmo que lê 10 valores, um de cada vez, e conta quantos deles estão no intervalo fechado entre 10 e 20 e quantos deles estão fora do intervalo, no final escreva estes resultados.
- 8) Escrever um algoritmo que lê o nro de matrícula dos vendedores de uma empresa, seu salário fixo e o total de vendas de cada vendedor. Cada vendedor recebe um salário fixo, mais uma comissão proporcional as vendas por ele efetuadas. A comissão é de 3 % sobre o total de vendas até 1.000 e 5 % sobre o que ultrapassa este valor. Escrever o nro do vendedor, o total de suas vendas, seu salário fixo e seu salário total. OBS: A leitura deve ser encerrada quando ler a matrícula do vendedor nro 99.
- 9) Escreva os 20 primeiros termos da série: 1,3,9,27,...
- 10) Faça um algoritmo que leia vários valores, um de cada vez e mostre quantos destes nros foram negativos, quantos positivos, bem como a soma dos nros positivos e a soma dos nros negativos. O valor lido 0 (zero) significa fim de algoritmo.
- 11) Leia o nome e altura, um de cada vez, para as moças inscritas em um concurso de beleza. Ao final do algoritmo escreva também as duas maiores alturas e nome das moças que a possuem. O algoritmo termina quando o nome da moça lido for "FIM".
- 12) Tem-se um conjunto de dados contendo altura e sexo de 50 pessoas, fazer um algoritmo que calcule e escreva:
 - a) a maior e a menor altura do grupo
 - b) média de altura das mulheres
 - c) o nro de homens
- 13) Uma pesquisa sobre algumas características físicas da população coletou os seguintes dados: sexo (masc, fem), cor dos olhos (azuis, verdes, castanhos); cor dos cabelos (louros, castanhos, pretos); e idade em anos. Para cada habitante, foi digitada uma linha com esses dados e a última linha, que não corresponde a ninguém, conterà o valor de idade igual a -1. Fazer um algoritmo que determine e escreva :
 - a) a maior idade dos habitantes
 - b) percentual de indivíduos do sexo feminino, cuja idade está entre 18 e 30 anos inclusive, e que tenham olhos verdes e cabelos louros.
 - c) percentual de indivíduos do sexo masculino
 - d) somatório das idades
 - e) média de idade da população

- 14) Fazer um algoritmo que calcula o preço à vista e o preço no cartão de crédito de uma mercadoria. Sendo que sobre o preço à vista é dado um desconto de (5%) e para o pagamento a prazo é embutido acréscimo de 30% sobre o preço da mercadoria.

EXERCÍCIOS PARA RESOLUÇÃO EM SALA DE AULA

- 1) Construir um algoritmo para calcular as raízes de uma equação do segundo grau sendo que os valores A, B e C são lidos a partir de um arquivo de entrada
- 2) Sendo $H = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{N}$, fazer um algoritmo para gerar o número H. O número N é lido em cartão.

- 3) Fazer um algoritmo que:
- leia um número real x de um cartão
 - determine e imprima o seguinte somatório

$$S = X - \frac{X}{1!} + \frac{X}{2!} - \frac{X}{3!} + \dots \text{ Usando os 20 primeiros termos da série}$$

- 4) O valor aproximado de π pode ser calculado usando-se a série

$$S = 1 - \frac{1}{3^3} + \frac{1}{5^3} - \frac{1}{7^3} + \frac{1}{9^3} \dots$$

$$\pi = \sqrt[3]{S * 32}$$

- 5) Faça um algoritmo para um programa que calcule e imprima o valor de π usando os 51 primeiros termos da série acima

- 6) Fazer um algoritmo para calcular o valor de S dado por :

$$S = \frac{1}{N} + \frac{2}{N-1} + \frac{3}{N-2} + \dots + \frac{N-1}{2} + \frac{N}{1}$$

Sendo N lido a partir de um arquivo de entrada

- 7) Implementar um algoritmo capaz de encontrar o maior dentre 3 números inteiros quaisquer. Suponha todos serem distintos.
- 8) Implementar um algoritmo que leia 3 números quaisquer e os imprima em ordem crescente.

CAPÍTULO 5- ALGORÍTMOS BASEADOS EM ESTRUTURAS DE DADOS HOMOGÊNEAS: VETORES E MATRIZES

Antes de tratarmos das estruturas de dados homogêneas, vejamos algumas novas estruturas de controle, que faremos uso quando trabalharmos com vetores e matrizes.

5.1- REPETIÇÃO COM VARIÁVEL DE CONTROLE –PARA (for)

Repete uma instrução um pré-determinado número de vezes.

para **v** de **i** até **l** passo **p** faça

onde:

v: variável de controle.

i: valor inicial de v.

l: valor final de v.

p: valor do incremento de v.

sintaxe do comando:

- **Nosso Pseudo-Código**

Para v de i até l passo p Faça

C1;

C2;

.

.

Cn;

Fim Para;

- **DNS**

v de i até l passo p
C1
C2
.
.
Cn

Significado do comando: v, i, l, p são variáveis quaisquer e que, de acordo com as regras da programação estruturada, não devem ser modificadas nos comandos C1, C2, . . ., Cn. O comando **para** é, na verdade, o comando enquanto utilizando-se uma variável de controle, escrito numa notação compactada. Neste caso existirá sempre uma inicialização da variável de controle, um teste para verificar se a variável atingiu o limite e um acréscimo na variável.

Exemplo para comparação entre **Enquanto** e **Para**

Início

```

Inteiro x;
X = 1 /*inicialização*/
Enquanto x<=10 Faça /*teste*/
    Leia(v[x]);
    x = x+1; {acrécimo}
Fim Enquanto;
```

Fim.

equivale a:

Início

```

Inteiro x;
Para x de 1 até 10 Passo 1 Faça
    Leia(v[x]);
Fim Para
```

Fim.

Nota:

- 1) Assim como no comando enquanto, se o valor inicial (i) já for superior ao limite (l), os comandos C1, C2,...,Cn não serão executados.
- 2) Toda vez que o **fim para** é encontrado, a variável de controle (i) é incrementada pelo passo p e o teste ($v \leq l$) é feito.
- 3) O valor da variável de controle torna-se indefinido assim que é executado o comando para. Neste aspecto o comando para é diferente do comando enquanto. Por exemplo, com o trecho de algoritmo seguinte

```

Para I de 1 até 10 Passo 1 Faça
    Imprima (I);
Fim Para;
Imprima (I);

```

serão impressos dentro do comando para: 1,2,3,4,5,6,7,8,9,10 e fora do comando para será impresso um valor indefinido

Quando o passo (p) for igual a 1, não será necessário escrever esta especificação no comando.

No exemplo acima teríamos:

```

Para I de 1 até 10 Faça
    Imprima (I);
Fim Para;

```

5.2. SELEÇÃO DENTRE AS MÚLTIPLAS ALTERNATIVAS-CASO (CASE)

Em nossos algoritmos, quando uma variável ou expressão aritmética puder assumir vários valores diferentes, e a execução do programa for diferente dependendo do valor obtido ou assumido por esta variável ou expressão, poderemos utilizar a estrutura que se segue, ao invés de vários "Se" aninhados.

- **Nosso Pseudo-Código**

```

Início
  Conforme
    Caso IDENTIFICADOR = Valor_1
      comando1;
    Caso IDENTIFICADOR = valor_2
      comando2;
    Caso IDENTIFICADOR = valor_3
      comando3;
    -
  Senão
    comandoN;

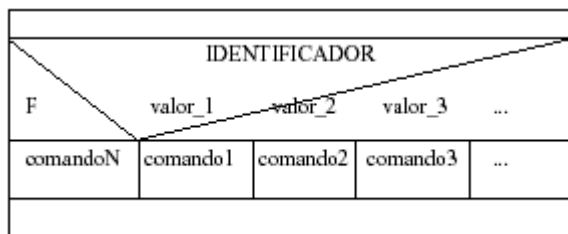
```

```

Fim Conforme;
-
-
Fim.

```

- **DNS**



Onde: - valor_1, valor_2, etc, pode ser um único valor ou uma faixa de valores discretos. Os testes de cada caso são feitos em seqüência, de cima para baixo em PC e da esquerda para a direita no DNS.

1. Após obter-se um valor verdadeiro para um dos casos, cessam os testes dos casos e o próximo comando a ser executado é o que vem a seguir do Fim Conforme.

Esta estrutura também pode ser utilizada testando-se as condições ao invés de valores, para determinar qual o comando a ser executado após o teste de cada condição (Ver exemplo b). Em programação, uma aplicação sempre interessante desta estrutura é no tratamento de Menus de Opções, quando para cada seleção feita pelo usuário do programa, haverá uma seqüência de ações diferentes a ser executada. Outra aplicação é na impressão de diferentes mensagens de erro.

Exemplos:

- a) Início {trecho de algoritmo para tratamento de três possíveis erros conhecidos na execução de um programa}

Conforme

```

Caso ERRO = 1
    Imprima (MSG1);
Caso ERRO = 2
    Imprima (MSG2);
Caso ERRO = 3
    Imprima (MSG3);
Senão
    Imprima ("ERRO NÃO CONHECIDO");

```

Fim Conforme;

- b) Início {trecho de algoritmo para impressão da situação de um aluno a partir da MÉDIA obtida ao fim do curso}

Conforme

```
Caso MÉDIA > 7.0
```

```
    Imprima ("APROVADO COM DIPLOMA");
```

```
Caso MÉDIA > = 5.0
```

```
    Imprima ("APROVADO COM CERTIFICADO");
```

```
Senão
```

```
    Imprima ("REPROVADO");
```

```
Fim Conforme;
```

5.3. VETORES

Um vetor ou agregado homogêneo, ou ainda variável composta homogênea, é uma estrutura de dados que contém elementos de mesmo tipo, que podem ser referenciados como um todo. Ao declararmos um vetor, estamos reservando na memória principal do computador uma série de células para uso da variável daquele tipo. O nome do vetor aponta para a base das células e o seu início dá a posição relativa do elemento referenciado ao primeiro (base).

Nem sempre os tipos básicos (inteiro, real, caracter e lógico) são suficientes para exprimir estruturas de dados em algoritmos. Por exemplo consideremos um problema em que um professor com 5 alunos deseja imprimir a nota e a média de seus alunos. Nesse caso seria necessário se considerar cinco variáveis reais para contar as notas dos cinco alunos. Imagine que o número de alunos da turma seja 80. Só a declaração destas variáveis tornaria impraticável a redação do algoritmo. Daí a necessidade de novos tipos serem criados. Um destes tipos, o vetor, será estudado.

Os vetores podem ser unidimensionais ou multi-dimensionais (matrizes). Um vetor unidimensional, como uma lista de notas dos 50 alunos de uma turma, tem apenas um índice. Se existirem porém várias turmas poderemos utilizar um vetor com dois índices (o número da turma e o número do aluno da turma). Abordaremos este assunto ainda neste capítulo.

5.2. DECLARAÇÃO DE VETORES

```
tipo_da_variável nome_da_variável [li:lf];
```

li - Posição Inicial do Vetor

lf - Posição Final do Vetor

Quando o C vê uma declaração como esta ele reserva um espaço na memória suficientemente grande para armazenar o número de células especificadas em tamanho. Por exemplo, se declararmos:

```
float exemplo [20];
```

o C irá reservar $4 \times 20 = 80$ bytes. Estes bytes são reservados de maneira contígua. Na linguagem C a numeração começa sempre em zero. Isto significa que, no exemplo acima, os dados serão indexados de 0 a 19. Para acessá-los vamos escrever:

```

exemplo[0]
exemplo[1]
.
.
exemplo[19]
    
```

Mas ninguém o impede de escrever:

```

exemplo[30]
exemplo[103]
    
```

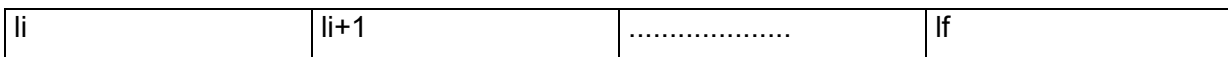
Por quê? Porque o C não verifica se o índice que você usou está dentro dos limites válidos. Este é um cuidado que você deve tomar. Se o programador não tiver atenção com os limites de validade para os índices ele corre o risco de ter variáveis sobrescritas ou de ver o computador travar. Bugs terríveis podem surgir. Vamos ver agora um exemplo de utilização de vetores:

Exemplo:

```
Real Notas[1:80];
```

O número de elementos de um vetor é dado por: **lf-li+1**

Isto significa que as posições do vetor são identificadas a partir de li, com incrementos unitários, até lf:



Exemplo: Qual o número de elementos e o que representa a especificação e declaração abaixo?

```
Character Nome[5:9]
```

Solução:

O vetor tem $9-5+1=5$ elementos e pode ser representado como na Figura 5.1.

NOME

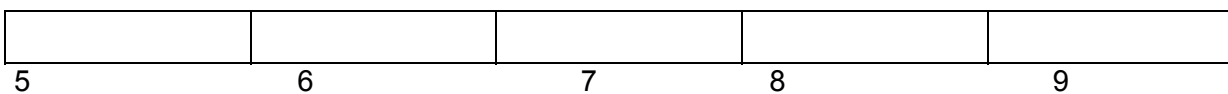


Figura 5.1 - Representação da estrutura de um vetor.

Cada elemento de um vetor é tratado como se fosse uma variável simples. Para referência a um elemento do vetor, utiliza-se o nome do vetor e a identificação do elemento (índice) entre colchetes. Por exemplo, se desejarmos atribuir o valor “ FMI” ao elemento identificado pelo índice 6 do vetor anterior, teremos:

NOME[6] ← “FMI” que produzirá

	FMI			
5	6	7	8	9

Ex.: Algoritmo para ler as notas de 3 alunos e imprimir os nomes daqueles cujas notas forem maior que a média.

- **Algoritmo Notas {Sem Vetor}**

Início

```

Character Nome1, Nome2, Nome3;
Leia (Nome1, Nota1);
Leia (Nome2, Nota2);
Leia (Nome3, Nota3);
Média = (Nota1+Nota2+Nota3)/3.0 ;
Se Nota1 > Média Então
    Imprima (Nome1);
Fim Se;
Se Nota2 > Média Então
    Imprima (Nome2);
Fim Se;
Se Nota3 > Média Então
    Imprima (Nome3);
Fim Se;

```

Fim.

- **Algoritmo Notas {Com Vetor}**

Início

```

Character Nome[0:2]
Real Nota[0:2];
Para I de 0 até 2 Faça
    Leia (Nome[I], Nota[I]);
Fim Para;
/*Cálculo da Média*/
Soma = 0.0 ;
Para I de 0 até 2 Faça

```



```

    Soma = Soma + Nota [I] ;
Fim Para;
Média = Soma/3.0;
Para I de 0 até 2 Faça
    Se Nota[I] > Média Então
        Imprima (Nome[I]);
    Fim Se;
Fim Para;
Fim.

```

1- Para inicializar um vetor com valores diferentes, utilizamos um comando “ construtor”
 NOTA = { 5.7, 9.5, 10.0, ..., 7.5}; que equivale a: NOTA [0] = 5.7, NOTA[2] =10.0;....etc.

1) Dado o vetor CRR de caracteres abaixo,

CRR

!	U	O	T	R	E	C	A
0	1	2	3	4	5	6	7

Qual será a sua configuração depois de executados os comandos:

```

Para I de 0 até 3 passo 1 Faça
    AUX =CRR[I];
    CRR[I]=CRR[6-I+1];
    CRR[6-I+1] =AUX;
Fim Para;

```

2) Dados dois vetores R[9] e S[19], escreva um algoritmo que: Leia os vetores R e S e Gere o vetor X correspondente à união dos vetores R e S.

Início

```

inteiro R[0:9],S[0:19],X[0:29];
Inteiro I;
Leia (R , S);
Para I de 0 até 9 Passo 1 Faça
    X[I] = R[I];
Fim Para
Para I de 0 até 19 Passo 1 Faça

```

```
        X[I+10] = S[I];
Fim Para;
Imprima (X);
Fim.
```

a) Gere o vetor Y com os elementos comuns de R e S

```
Início
inteiro R[0:9],Y[0:9],S[0:19],I,J;
leia (R,S);
Y = 0
para I de 0 até 9 faça
    K = 1;
    para J de 0 até 19 faça
        se R[I] == S[J]
            então enquanto K<=I faça
                se Y[K] != R[I];
                    então se Y[K]==0;
                        então Y[K] =R[I];
                    fim se;
                senão abandone;
            fim se;
            K =K+1;
        fim enquanto;
    fim se;
    se Y[K]==R[I] então abandone;
    fim se;
fim para;
fim para;
imprima (Y);
Fim.
```

- 3) Um armazém trabalha com 100 mercadorias diferentes identificadas pelos números inteiros de 1 a 100. O dono do armazém anota a quantidade de cada mercadoria vendida durante o

mês. Ele tem uma tabela que indica para cada mercadoria o preço de venda. Escreva o algoritmo para calcular o faturamento mensal de armazém, isto é:

$$FATURAMENTO = \sum_{i=1}^{100} (QUANTIDADE_i * PREÇO_i)$$

A tabela de preço e quantidade são fornecidas em dois conjuntos, sendo que um conjunto contém a quantidade vendida e o outro o preço de cada mercadoria.

Solução

Início

```
/* calcula o faturamento*/  
real QUANTIDADE[0:99], PREÇO[0:99]/*Tabela de qtdes vendidas e Preço*/  
real FATURAMENTO;  
inteiro I; /* indexador*/  
FATURAMENTO = 0;  
leia (QUANTIDADE, PREÇO);  
para I de 0 até 99 faça  
    FATURAMENTO = FATURAMENTO +QUANTIDADE[I]*PREÇO[I];  
fim para;  
imprima (FATURAMENTO);
```

Fim.

EXERCÍCIOS SOBRE VETORES

1. Faça um algoritmo que leia 18 números inteiros e calcule e imprima a soma dos mesmos.
2. Faça um algoritmo que leia 18 números inteiros e calcule e imprima a média dos mesmos
3. Faça um programa que leia 4 números inteiros e apresente:
 - Média dos ímpares
 - Maior número par
 - Diferença do maior menos o menor número

4. Faça um programa que leia um conteúdo com 100 números inteiros e apresente a média dos números.
5. Faça um programa que leia 2 conjuntos com 50 números inteiros e gere um terceiro conjunto com a média dos elementos de A e B. Apresente o C.
6. Faça um programa que leia um conjunto A com 30 números inteiros e que armazene em um conjunto B do mesmo tipo os elementos de A multiplicados por 3. apresente o conjunto B.
7. Faça um programa que leia um conjunto A com 100 números inteiros e que gere um conjunto B com os elementos de A de acordo com a seguinte regra. Se o elemento de A for par, armazene em B multiplicando por 5. Se o elemento de A for ímpar, armazene em B somando 5.
8. Faça um programa que leia dois conjuntos A e B com 20 inteiros e troque os valores de A com B.
9. Faça um programa que leia dois conjuntos A e B com 15 números inteiros e gere um conjunto de 30 elementos, onde os 15 primeiros elementos são de A e o resto de B. Apresente o conjunto C.

EXERCÍCIOS COMPLEMENTARES

- 1) Faça um programa que leia dois números e apresente o maior deles.
- 2) Fazer um algoritmo que descubra o maior valor de um vetor de 20 posições
- 3) Faça um algoritmo que calcule o fatorial de um número.
- 4) Implementar um algoritmo que, a partir da área de um quadrado lido, forneça, como saída, o comprimento do lado de um quadrado com metade de sua área.
- 5) Implementar um algoritmo que calcule o **menor caminho** a ser percorrido entre três cidades, passando somente uma vez por cada uma delas e começando em qualquer uma. Como entrada são fornecidas as distâncias entre pares de cidades. Exemplo: Entre 3 cidades A, B e C devem ser lidas as distâncias entre AB, AC e BC (é claramente dispensável ler as distâncias BA, CA e CB, já que são iguais a AB, AC e BC, respectivamente).

Dica : Para saber qual a distância total percorrida em um caminho basta somar duas distâncias lidas. Exemplo : AB e BC atende à resolução do problema, já que inicia-se em A, passando por B e chegando em C. A distância do caminho AB, BC é exatamente a mesma de BC, AB (invertendo origem e destino), logo não é necessário somar a distância do caminho nas duas direções.

6) Construa um algoritmo que receba três número e mostre o maior deles.

5.4 - MATRIZES

Definição: Conjunto de dados referenciados por um mesmo nome e que necessita de mais de um índice para ter seus elementos individualizados.

- Quando temos mais de uma dimensão então utilizamos as matrizes, que de diferente no uso e na declaração só tem estas dimensões a mais explícitas :
- A sintaxe é a seguinte:

<Tipo básico> *identificador* [*l*₁:*l*₁,*l*₂:*l*₂,.....*l*_n:*l*_n] ;

Esta especificação corresponde apenas à criação do modelo, e, para efetivar a estrutura de dados dentro do algoritmo, é necessária uma declaração dando um nome à variável que segue o modelo.

Ex:

Real MAT[1:3, 1:4] ;

ou seja,

		MAT			
		1	2	3	4
1					
2					
3					

O número de dimensões da matriz será igual ao número de vírgulas (,) da declaração mais 1. O número de elementos é igual ao produto do número de elementos de cada dimensão: $(l_1-l_{i1}+1)*(l_2-l_{i2}+1)*...*(l_n-l_{in}+1)$

Exercícios

1) Qual é o número de elementos e dimensões das matrizes especificadas abaixo:

a) real MAT1[0:2, 1:5] ;

b) Caracter MAT2[1:3, 2:4, 3:4]

Para referenciar um elemento da matriz são necessários tantos índices quantas são as dimensões da matriz. Numa matriz bidimensional (duas dimensões) o primeiro índice indica a linha e o segundo, a coluna. No exemplo anterior, MAT1[1,3] se refere ao elemento da linha número 1 (segunda linha no caso) e coluna número 3 (terceira coluna)

MAT1

	1	2	3	4	5
0					
1					
2					

Para matrizes com três dimensões, repete-se a estrutura bidimensional tantas vezes quantos são os elementos da terceira dimensão, numerando-as de acordo com os limites especificados na declaração de tipo.

Exemplo1:O que será impresso no algoritmo abaixo:

```

início
  inteiro M1[1:3,1:2];
  inteiro: I, J;
  M1[1,1] = 1;
  M1[1,2] = 2;
  M1[2,1] = 3;
  M1[2,2] = 4;
  M1[3,1] = 5;
  M1[3,2] = 6;
  Imprima(M1);
  Para I de 1 até 2 Faça
    Para J de 1 até 3 Faça
      Imprima (M1[J,I]);
    Fim Para;
  Fim Para;
Fim.

```

Exemplo 2:

Dada a matriz MAT abaixo

		MAT			
		1	2	3	4
1	O	Q	*	I	
2	E	A	E	S	
3	R	E	U	T	
4	A	*	*	S	

Qual será a configuração de MAT depois de executado o algoritmo:

Início

```

Inteiro I, J;
Caracter AUX;
Caracter M1[1:4,1:4];
Leia (MAT);
Para I de 1 até 4 Passo 1 Faça
    Para J de I+1 até 4 Passo 1 Faça
        AUX = MAT[I,J];
        MAT[I,J] = MAT[J,I];
        MAT[J,I] = AUX;
    Fim Para;
Fim Para;
AUX=MAT[1,1];
MAT[1,1]=MAT[4,4];
MAT[4,4]= AUX;
AUX = MAT[2,2];
MAT[2,2] = MAT[3,3];
MAT[3,3] = AUX

```

Fim.

Notas:

1- Para percorrer a matriz linha por linha:

Fixar a linha

Variar a coluna

2- Para percorrer a matriz coluna por coluna

Fixar a coluna

Variar a linha

```
Inteiro exemplo[1:3,0:4];
1. I =1;
   Enquanto I<=3 Faça
     J=0;
     Enquanto J<= 4 Faça
       exemplo[I,J] =I+J;
       J=J+1;
     Fim Enquanto;
     I=I+1;
   Fim Enquanto;
2. J =0;
   Enquanto J<=4 Faça
     I=1;
     Enquanto I<= 3 Faça
       exemplo[I,J] =I+J;
       I=I+1;
     Fim Enquanto;
     J=J+1;
   Fim Enquanto;
```

ou então

```
Para j de 0 até 4 Passo 1 Faça
Para I de 1 até 3 Passo 1 Faça
  exemplo[I,J]←I+J;
Fim Para;
Fim Para;
```

Como no caso de vetores, também para matrizes podemos ter comandos concisos para inicialização, leitura e impressão.

Por exemplo, seja MAT definida por:

```
Inteiro MAT[1:10,1:8];
```

O comando:


```
Leia (MAT);
```

é equivalente ao seguinte trecho:

```
Para i de 1 até 10 Passo 1 Faça
```

```
Para j de 1 até 8 Passo 1 Faça
```

```
    Leia (MAT[I,J]);
```

```
Fim Para;
```

```
Fim Para;
```

Do mesmo modo, pode-se inicializar uma matriz com todos os seus elementos iguais a um determinado valor escrevendo

```
MAT =0;
```

Exercícios Resolvidos

1) Dada uma matriz MAT de 4 x 5 elementos, faça um algoritmo para somar os elementos de cada linha gerando o vetor SOMALINHA. Em seguida, somar os elementos do vetor SOMALINHA na variável TOTAL que deve ser impressa no final:

Exemplo

$$SOMALINHA_i = \sum_{j=1}^5 MAT_{ij} \quad (i = 1,2,3,4)$$

$$TOTAL = \sum_{i=1}^4 SOMALINHA_i$$

$$\begin{bmatrix} 1 & 2 & -1 & 2 & 3 \\ 1 & 3 & 4 & 2 & 0 \\ 8 & 5 & 1 & 3 & 2 \\ 1 & -2 & 3 & 4 & 5 \end{bmatrix} \rightarrow \begin{matrix} 7 \\ 10 \\ 19 \\ 11 \end{matrix}$$

↓

TOTAL 47

Solução:

Início

```
Real MAT[1:4, 1:5];
```

```
Real SOMALINHA[1:4];
```

```
Real: TOTAL;
```

```
Inteiro: I, J;
```

```
SOMALINHA=0;
```

```
TOTAL=0;
```

```
Leia (MAT);
```

```

Para I de 1 até 4 Faça
    Para J de 1 até 5 Faça
        SOMALINHA[I]= SOMALINHA[I]+MAT[I,J];
    Fim Para;
TOTAL=TOTAL+SOMALINHA[I];
Fim Para;
Imprima ("TOTAL=", TOTAL);
Fim.

```

2) Escreva um algoritmo que leia duas matrizes reais de dimensão 3 x 5, calcule e imprima a soma das matrizes.

```

Início
    Real A[1:3,1:5], B[1:3,1:5], C[1:3,1:5];
    Inteiro: I, J;
    Leia (A,B);
    I=1;
    Enquanto I<=3 faça
        J=1
        Enquanto J<=5 faça
            C[I,J] =A[I,J]+B[I,J];
            J=J+1;
        Fim Enquanto;
        I=I+1;
    Fim Enquanto;
    Imprima ( "C=" ,C );
Fim.

```

3) Escreva um algoritmo para um programa que leia uma matriz quadrada 20 x 20 de elementos reais, divida cada elemento de uma linha da matriz pelo elemento da diagonal principal desta linha e imprima a matriz assim modificada.

Obs. $\begin{bmatrix} \# \\ \# \end{bmatrix}$ Elementos da diagonal principal ; M[1,1], M[2,2], M[3,3], M[4,4]

```

Início
    real M[1:20,1:20];
    Inteiro: I, J;
    Real: DIAGONAL;
    Leia (M);

```

```

Para I de 1 até 20 Faça
  DIAGONAL =M[I,I];
  Para J de 1 até 20 Faça
    M[I,J] =M[I,J]/DIAGONAL;
  Fim Para;
Fim Para;
Imprima ("M=",M);
Fim.

```

10. Dadas as declarações abaixo, indicar as dimensões e o número de elementos:

a) $a[0:2,1:3]$

b) $\text{beta}[1:2, 1:1, 5:6];$

c) $\text{alfa}[2:10];$

11. dada a matriz
$$\begin{bmatrix} O & Q & * & I \\ E & * & E & S \\ R & E & U & T \\ A & * & * & S \end{bmatrix} \text{ MAT}$$

Qual será a configuração MAT depois de executado o algoritmo:

Início

Inteiro: I, J;

Character: AUX;

Character MAT[1:4, 1:4];

Para I de 1 até 4 Faça

 Para j de I+1 até 4 Faça

 AUX=MAT[I,J];

 MAT[I,J] =MAT[J,I];

 MAT[J,I] =AUX

 Fim Para;

Fim Para;

AUX =MAT[1,1];

MAT[1,1] =MAT[4,4];

MAT[4,4] =AUX;

AUX=MAT[2,2];

MAT[2,2] =MAT[3,3];

MAT[3,3] =AUX;

Fim.

Exercício Proposto

1. A matriz dada a seguir contém, em cada linha, as cinco notas de provas obtidas por um aluno durante o período letivo. O índice das linhas corresponde ao número do aluno. Assim, por exemplo, o aluno número 3 obteve as notas, 8,9 – 3,8 – 8,7 – 6,0 – 6,5

$\left[\begin{array}{ccccc} 6,0 & 7,0 & 5,0 & 2,5 & 8,0 \\ 8,0 & 3,4 & 8,6 & 8,8 & 7,5 \\ 8,9 & 3,8 & 8,7 & 6,0 & 6,5 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 10,0 & 10,0 & 9,5 & 9,0 & 9,0 \end{array} \right]$	NOTAS
--	-------

A turma contém 30 alunos

As notas são lidas a partir de um arquivo de entrada, sendo que cada linha da matriz contém as cinco notas de provas obtidas por alunos.

Faça um algoritmo que calcule a média de cada aluno e a média geral da turma e que imprima o resultado da seguinte forma.

<i>Nº do Aluno</i>	<i>MÉDIA/ ALUNO</i>	<i>MÉDIA/ TURMA</i>
<i>1</i>	<i>7,63</i>	<i>XXX</i>
<i>2</i>	<i>7,60</i>	<i>XXX</i>
<i>⋮</i>	<i>⋮</i>	<i>⋮</i>

CAPÍTULO 6 - MODULARIZAÇÃO DE ALGORITMOS

6.1. INTRODUÇÃO

Vimos que os algoritmos estruturados são desenvolvidos levando-se em conta algumas premissas básicas:

- 1) Desenvolver o algoritmo em diferentes fases de detalhamento crescente, do geral ao particular, por refinamentos sucessivos (desenvolvimento "top-down" ou de cima para baixo).
- 2) Decompor o algoritmo em módulos funcionais, organizados de preferência em um sistema hierárquico. Esses módulos trazem vantagens adicionais para testes, pois testa-se um módulo (ou seja, uma parte do programa) de cada vez, independentemente; e para reutilização de um módulo em outros algoritmos e programas, no futuro.

Passaremos agora a formalizar a utilização de módulos nos nossos algoritmos e verificar as vantagens que os mesmos trazem para entendimento, construção, codificação, teste e reutilização dos mesmos. A grande maioria das linguagens de programação que são utilizadas, tem esta facilidade, seja com o nome de Sub-rotinas, Subprogramas, Procedimentos, Funções, Módulos, Blocos, etc., sempre é possível subdividir-se um programa de modo a facilitar o entendimento, permitir a reutilização, evitando-se a repetição de blocos dos programas. No nosso pseudo-código definiremos dois tipos de módulos: Os procedimentos ("procedures") e as funções ("functions").

Os procedimentos e funções são normalmente definidos antes de serem utilizados(chamados) pelo programa principal. Em se tratando de algoritmos, entretanto, poderemos definir nossos procedimentos e funções em qualquer parte do algoritmo principal ou depois dele, adotando os formatos que se seguem e as normas de documentação.

6.2. PROCEDIMENTOS

Um procedimento é uma seqüência de comandos precedida por uma seqüência de declarações que possui um identificador (nome do procedimento), uma lista de parâmetros opcional e pode realizar qualquer tipo de processamento que o programador ou analista deseje. As variáveis, os tipos e as constantes declaradas dentro de um procedimento só são acessáveis dentro dos comandos do procedimento. São chamadas variáveis locais. Em um Algoritmo, as variáveis, tipos e constantes declaradas logo após o identificador do algoritmo, são acessíveis e visíveis dentro de qualquer procedimento que esteja sendo usado pelo algoritmo. São chamadas variáveis Globais. É uma boa técnica ao se construir um procedimento, não fazer uso no mesmo, de variáveis globais e referenciar e utilizar apenas as variáveis locais e os parâmetros do procedimento. Isso fará com que cada procedimento, ao ser modificado, não afete outras partes

do Algoritmo, e da mesma forma as variáveis globais poderão ser modificadas sem que haja efeitos colaterais nos procedimentos.

Sintaxe de declaração

```
Procedimento < nome do Procedimento >
  Início
    <declarações>;
    C1;
    C2;
    .
    Cn;
  Fim { nome do procedimento}.
```

Exemplo: O que será impresso no algoritmo abaixo?

```
Início
  Inteiro: X,Y,A,B,C,D;
  Procedimento troca;
  Início
    Inteiro: AUX,X;
    AUX←X;
    X←Y;
    Y←AUX;
  Fim troca.
  A←5;
  B←3;
  Imprima (A, B);
  X←A;
  Y←B;
  Troca;
  A←X;
  B←Y;
  Imprima (A, B);
  C←4;
  D←9;
  Imprima (C, D);
  X←C;
  Y←D;
  Troca;
  C←X;
  D←Y;
  Imprima (C, D);
Fim.
```

Os módulos que estivemos utilizando na apostila, nos capítulos anteriores, são procedimentos sem parâmetros, ou seja, que fazem uso de variáveis globais e sua única importante vantagem é facilitar o entendimento e a solução do algoritmo.

Utilizando parâmetros e não variáveis globais dentro do procedimento, podemos, por assim dizer, isolá-lo do meio exterior e obter vantagens adicionais na codificação, pois, uma vez que foram bem definidos os parâmetros, um programador poderá desenvolver seu procedimento sem se preocupar com o algoritmo que o vai chamar. Poderemos também testá-lo individualmente e isoladamente para verificar sua correção. Isto será comprovado na disciplina LP. A isto se denomina Encapsulamento. Uma lista de parâmetros consiste em uma lista de variáveis e seus respectivos tipos.

```
Procedimento < nome do Procedimento> (<lista de parâmetros>)  
  < especificação de parâmetros>  
  Início  
    <declaração de variáveis locais>  
    C1;  
    C2;  
    .  
    .  
    Cn;  
  Fim. { nome do procedimento}
```

Exemplo de Procedimento:

```
Procedimento TROCAINTEIROS (Inteiro NUM1, inteiro NUM2)  
  Início  
    Inteiro AUXNUM;  
    AUXNUM = NUM1;  
    NUM1 = NUM2;  
    NUM2 = AUXNUM;  
  Fim.
```

6.3. FUNÇÕES

Funções são as estruturas que permitem ao usuário separar seus programas em blocos. Se não as tivéssemos, os programas teriam que ser curtos e de pequena complexidade. Para fazermos programas grandes e complexos temos de construí-los bloco a bloco.

Uma função é um módulo que tem por objetivo calcular e retornar ao algoritmo, procedimento ou função "chamador" um valor. A chamada de uma função para execução é feita

apenas citando-se o seu identificador no meio de uma expressão. Vide as funções previamente existentes no nosso pseudo-código. Uma função é dita recursiva quando chama a ela própria.

Uma função no C tem a seguinte forma geral:

tipo_de_retorno nome_da_função (declaração_de_parâmetros)

Início

corpo_da_função

Fim.

Aqui no nosso Pseudo-Código utilizaremos a seguinte forma

- **Pseudo - Código**

Tipo-de-retorno<nome da função>(<declaração dos parâmetros>)

Início

Corpo da Função

Fim.

Onde:

- A lista de parâmetros é semelhante à dos procedimentos
- Antes do nome da função dá o tipo do valor que a função retorna ao algoritmo ou procedimento chamador.

O tipo-de-retorno é o tipo de variável que a função vai retornar. O default é o tipo **inteiro (int)**, ou seja, uma função para qual não declaramos o tipo de retorno é considerada como retornando um inteiro. A declaração de parâmetros é uma lista com a seguinte forma geral:

tipo nome1, tipo nome2, ... , tipo nomeN

Repare que o tipo deve ser especificado para cada uma das N variáveis de entrada. É na declaração de parâmetros que informamos ao compilador quais serão as entradas da função (assim como informamos a saída no tipo-de-retorno).

O corpo da função é a sua alma. É nele que as entradas são processadas, saídas são geradas ou outras coisas são feitas.

O Comando retorna (return)

O comando **return** tem a seguinte forma geral:

retorna valor_de_retorno; ou retorna;

Digamos que uma função está sendo executada. Quando se chega a uma declaração **retorna (return)** a função é encerrada imediatamente e, se o valor de retorno é informado, a função retorna este valor. É importante lembrar que o valor de retorno fornecido tem que ser compatível com o tipo de retorno declarado para a função.

Uma função pode ter mais de uma declaração **retorna (return)**. Isto se torna claro quando pensamos que a função é terminada quando o programa chega à primeira declaração **retorna (return)**. Abaixo estão dois exemplos de uso do **retorna (return)**:

Exemplo de função:

```

Inteiro VALORABSOLUTO (inteiro X)
Início
  Se X > = 0 Então
    VALORABSOLUTO =X;
  Senão
    VALORABSOLUTO = (-X);
  Fim Se;
Fim.

```

Exercício para casa: Escrever e executar em C o programinha acima

- Algoritmo para calcular o quadrado de um número

- **Em Pseudo-Código**

```

Inteiro Square(inteiro a)
Início
  retorna (a*a);
Fim.

Algoritmo Principal
Início
  inteiro num;
  imprima ("Entre com um numero: ");
  leia (num);
  num=Square(num);
  imprima ("O seu quadrado vale:", num);
Fim

```

- **Em C**

```

#include <stdio.h>
int Square (int a)
{
  return (a*a);
}
int main ()
{
  int num;
  printf ("Entre com um numero: ");
  scanf ("%d", &num);
  num=Square(num);
  printf ("\n\nO seu quadrado vale: %d\n", num);
  return 0;
}

```

- Algoritmo para verificar de um número a é divisível por 2.

- **Em Pseudo-Código**

```

inteiro EPar (inteiro a)
início
se (a%2) então      /* Verifica se a e divisivel por dois */
    return 0;      /* Retorna 0 se nao for divisivel, a%1=1 */
senão
    return 1;      /* Retorna 1 se for divisivel , a%0=0*/
fim se;
fim.

```

Algoritmo Principal

```

Início
inteiro num;
imprima ("Entre com numero: ");
leia (num);
se (EPar(num)) então
    imprima ("O numero e par.");
senão
    imprima ("O numero e impar.");
fim se;
Fim.

```

- **Em C**

```

#include <stdio.h>
int EPar (int a)
{
    if (a%2)          /* Verifica se a e divisivel por dois */
        return 0;    /* Retorna 0 se nao for divisivel */
    else
        return 1;    /* Retorna 1 se for divisivel */
}
int main ()
{
    int num;
    printf ("Entre com numero: ");
    scanf ("%d", &num);
    if (EPar(num))
        printf ("\n\nO numero e par.\n");
    else
        printf ("\n\nO numero e impar.\n");
    return 0;
}

```

É importante notar que, como as funções retornam valores, podemos aproveitá-los para fazer atribuições, ou mesmo para que estes valores participem de expressões. Mas *não* podemos fazer:

```
func(a,b)=x; /* Errado! */
```

6.3.1. Protótipos de Funções

Até agora, nos exemplos apresentados, escrevemos as funções antes de escrevermos a função principal, **main()**. Isto é, as funções estão fisicamente antes da função principal **main()**.

Isto foi feito por uma razão. Imagine-se na pele do compilador. Se você fosse compilar a função **main()**, onde são chamadas as funções, você teria que saber com antecedência quais são os tipos de retorno e quais são os parâmetros das funções para que você pudesse gerar o código corretamente. Foi por isto que as funções foram colocadas antes da função **main()**: quando o compilador chegasse à função **main()** ele já teria compilado as funções e já saberia seus formatos. Mas, muitas vezes, não poderemos nos dar ao luxo de escrever nesta ordem. Muitas vezes teremos o nosso programa espalhado por vários arquivos. Ou seja, estaremos chamando funções em um arquivo que serão compiladas em outro arquivo. Como manter a coerência? A solução são os protótipos de funções. Protótipos são nada mais, nada menos, que declarações de funções. Isto é, você declara uma função que irá usar. O compilador toma então conhecimento do formato daquela função antes de compilá-la. O código correto será então gerado. Um protótipo tem o seguinte formato:

tipo_de_retorno nome_da_função (declaração_de_parâmetros);

onde o tipo-de-retorno, o nome-da-função e a declaração-de-parâmetros são os mesmos que você pretende usar quando realmente escrever a função. Repare que os protótipos têm uma nítida semelhança com as declarações de variáveis. Vamos implementar agora um dos exemplos da seção anterior com algumas alterações e com protótipos:

- **Em Pseudo-Código**

```
inteiro Square (inteiro a);
Algoritmo Principal
início
    inteiro num;
    imprima ("Entre com um numero: ");
    leia (num);
    num=Square(num);
    imprima ("O seu quadrado vale:", num);
    retorna (0);
fim.

inteiro Square (inteiro a)
início
    retorna (a*a);
fim.
```

- **Em C**

```
#include <stdio.h>
int Square (int a);
int main ()
{
    int num;
    printf ("Entre com um numero: ");
    scanf ("%f",&num);
    num=Square(num);
    printf ("\n\nO seu quadrado vale: %f\n",num);
    return 0;
}
int Square (int a)
{
    return (a*a);
}
```

Observe que a função **Square()** está colocada depois do algoritmo principal, mas o seu protótipo está antes. Sem isto este programa não funcionaria corretamente.

Usando protótipos você pode construir funções que retornam quaisquer tipos de variáveis. É bom ressaltar que funções podem também retornar ponteiros sem qualquer problema. Os protótipos não só ajudam o compilador. Eles ajudam a você também: usando protótipos, o compilador evita erros, não deixando que o programador use funções com os parâmetros errados e com o tipo de retorno errado, o que é uma grande ajuda!

6.3.2. Escopo de Variáveis

O escopo é o conjunto de regras que determinam o uso e a validade de variáveis nas diversas partes do programa.

6.3.2.1. Variáveis locais

O primeiro tipo de variáveis que veremos são as variáveis locais. Estas são aquelas que só têm validade dentro do bloco no qual são declaradas. Podemos declarar variáveis dentro de qualquer bloco. Só para lembrar: um bloco começa quando abrimos uma chave e termina quando fechamos a chave. Até agora só tínhamos visto variáveis locais para funções completas. Mas um comando **para(for)** pode ter variáveis locais e que não serão conhecidas fora dali. A declaração de variáveis locais é a primeira coisa que devemos colocar num bloco. A característica que torna as variáveis locais tão importantes é justamente a de serem exclusivas do bloco. Podemos ter quantos blocos quisermos com uma variável local chamada **x**, por exemplo, e elas não apresentarão conflito entre elas.

A palavra reservada do C **auto** serve para dizer que uma variável é local. Mas não precisaremos usá-la pois as variáveis declaradas dentro de um bloco já são consideradas locais.

Abaixo vemos um exemplo de variáveis locais:

- **Em Pseudo-Código**

```
funcao1 (...)  
Início  
  inteiro abc, x;  
  ...  
Fim.  
  
funcao2 (...)  
Início  
  inteiro abc;  
  ...  
Fim.  
  
inteiro principal ()  
Início  
  inteiro a, x, y;  
  para (...)  
  {  
    real a, b, c;  
    ...  
  }  
  ...  
Fim.
```

- **Em C**

```

func1 (...)
{
    int abc,x;
    ...
}
func (...)
{
    int abc;
    ...
}
int main ()
{
    int a,x,y;
    for (...)
        {
            float a,b,c;
            ...
        }
    ...
}

```

No programa acima temos três funções. As variáveis locais de cada uma delas não irão interferir com as variáveis locais de outras funções. Assim, a variável **abc** de **func1()** não tem nada a ver (e pode ser tratada independentemente) com a variável **abc** de **func2()**. A variável **x** de **func1()** é também completamente independente da variável **x** da função principal **main()**. As variáveis **a**, **b** e **c** são locais ao bloco **para(for)**. Isto quer dizer que só são conhecidas dentro deste bloco e são desconhecidas no resto da função Principal(**main()**). Quando usarmos a variável **a** dentro do bloco **para (for)** estaremos usando a variável **a** local ao **para(for)** e não a variável **a** da função **main()**.

6.3.2.2. Parâmetros formais

O segundo tipo de variável que veremos são os parâmetros formais. Estes são declarados como sendo as entradas de uma função. Não há motivo para se preocupar com o escopo deles. É fácil: o parâmetro formal é uma variável local da função. Você pode também alterar o valor de um parâmetro formal, pois esta alteração não terá efeito na variável que foi passada à função. Isto tem sentido, pois quando o C passa parâmetros para uma função, são passadas apenas cópias das variáveis. Isto é, os parâmetros formais existem independentemente das variáveis que foram passadas para a função. Eles tomam apenas uma cópia dos valores passados para a função.

6.3.2.3. Variáveis globais

Variáveis globais são declaradas, como já sabemos, fora de todas as funções do programa. Elas são conhecidas e podem ser alteradas por todas as funções do programa. Quando uma função tem uma variável local com o mesmo nome de uma variável global a função dará preferência à variável local. Vamos ver um exemplo no nosso pseudo-Código:

```

inteiro z,k;

```

```

func1 (...)
{
    inteiro x,y;
    ...
}

func2 (...)
{
    inteiro x,y,z;
    ...
    z=10;
    ...
}

principal
{
    inteiro count;
    ...
}

```

No exemplo acima as variáveis **z** e **k** são globais. Veja que **func2()** tem uma variável local chamada **z**. Quando temos então, em **func2()**, o comando **z=10** quem recebe o valor de 10 é a variável *local*, não afetando o valor da variável global **z**.

Evite *ao máximo* o uso de variáveis globais. Elas ocupam memória o tempo todo (as locais só ocupam memória enquanto estão sendo usadas) e tornam o programa mais difícil de ser entendido e menos geral.

6.3.3. Passagem de parâmetros por valor e passagem por referência

Já vimos que, na linguagem C, quando chamamos uma função os parâmetros formais da função copiam os valores dos parâmetros que são passados para a função. Isto quer dizer que não são alterados os valores que os parâmetros têm fora da função. Este tipo de chamada de função é denominado chamada por valor. Isto ocorre porque são passados para a função apenas os valores dos parâmetros e não os próprios parâmetros. Veja o exemplo abaixo:

Exemplo

- **Em Pseudo-Código**

```

real sqr (real num);
int main ()
{
    real num,sq;
    imprima ("Entre com um numero: ");
    leia (num);
    sq=sqr(num);
    imprima ("O numero original e:",num);
    imprima ("O seu quadrado vale:",sq);
}

real sqr (real num)
{
    num=num*num;
    retorna num;
}

```

- **Em C**

```

#include <stdio.h>
float sqr (float num);
void main ()
{
    float num,sq;
    printf ("Entre com um numero: ");
    scanf ("%f",&num);
    sq=sqr(num);
    printf ("\n\nO numero original e: %f\n",num);
    printf ("O seu quadrado vale: %f\n",sq);
}
float sqr (float num)
{
    num=num*num;
    return num;
}

```

No exemplo acima o parâmetro formal **num** da função **sqr()** sofre alterações dentro da função, mas a variável **num** da função principal **main()** permanece inalterada: é uma chamada por valor.

Outro tipo de passagem de parâmetros para uma função ocorre quando alterações nos parâmetros formais, dentro da função, alteram os valores dos parâmetros que foram passados para a função. Este tipo de chamada de função tem o nome de "chamada por referência". Este nome vem do fato de que, neste tipo de chamada, não se passa para a função os valores das variáveis, mas sim suas referências (a função usa as referências para alterar os valores das variáveis fora da função).

O C só faz chamadas por valor. Isto é bom quando queremos usar os parâmetros formais à vontade dentro da função, sem termos que nos preocupar em estar alterando os valores dos parâmetros que foram passados para a função. Mas isto também pode ser ruim às vezes, porque podemos querer mudar os valores dos parâmetros fora da função também. O C++ tem um recurso que permite ao programador fazer chamadas por referência. Há entretanto, no C, um recurso de programação que podemos usar para simular uma chamada por referência. Quando queremos alterar as variáveis que são passadas para uma função, nós podemos declarar seus parâmetros formais como sendo *ponteiros*. Os ponteiros são a "referência" que precisamos para poder alterar a variável fora da função. O único inconveniente é que, quando usarmos a função, teremos de lembrar de colocar um **&** na frente das variáveis que estivermos passando para a função. Veja um exemplo:

```

#include <stdio.h>
void Swap (int *a,int *b);
void main (void)
{
    int num1,num2;
    num1=100;
    num2=200;
    Swap (&num1,&num2);
    printf ("\n\nEles agora valem %d %d\n",num1,num2);
}
void Swap (int *a,int *b)
{
    int temp;
    temp=*a;
    *a=*b;
    *b=temp;
}

```

Não é muito difícil. O que está acontecendo é que passamos para a função Swap o endereço das variáveis num1 e num2. Estes endereços são copiados nos ponteiros a e b. Através do operador * estamos acessando o conteúdo apontado pelos ponteiros e modificando-o. Mas, quem é este conteúdo? Nada mais que os valores armazenados em num1 e num2, que, portanto, estão sendo modificados!

6.4. FUNÇÕES RECURSIVAS

Existem casos em que um procedimento ou função chama a si próprio. Diz-se então que o procedimento ou função é recursivo. Por exemplo, o fatorial de um número n pode ser definido recursivamente, ou seja:

$$n! = \begin{cases} n \cdot (n-1)! \\ 1 & \text{se } n = 0 \end{cases}$$

```
inteiro fat(inteiro n)
Início
  Se n==0 Então
    fat=1;
  Senão
    fat=n*fat(n-1);
  Fim Se;
Fim.
```

Por exemplo, o fatorial de 3 será calculado a partir da chamada à função pelo comando `x=fat(3)` que retornará em fat o valor 6.

6.5. ESTRUTURAÇÃO DOS MÓDULOS DE UM ALGORITMO

Um procedimento nada mais é que um algoritmo hierarquicamente subordinado a um outro algoritmo, comumente chamado de Módulo principal ou programa principal ou ainda algoritmo principal.

Da mesma forma, um procedimento poderá conter outros procedimentos (e também funções) aninhados. Assim, ao definirmos as fases do nosso algoritmo, no processo de refinamentos sucessivos, se transformarmos as mesmas em procedimentos e os refinamentos de cada procedimento em outros procedimentos, obteremos, ao fim do processo de criação, um algoritmo estruturado, com módulos hierarquicamente subordinados e muito menos suscetível a erros de lógica na fase de codificação do programa.

CAPÍTULO 7 - ALGORITMOS DE PESQUISA E ORDENAÇÃO

7.1. INTRODUÇÃO

Quando temos um Vetor (ou Matriz) com muitos elementos e precisamos descobrir se um determinado elemento que procuramos se encontra no vetor, uma solução que certamente nos vem à mente é comparar o elemento que procuramos com cada elemento do vetor, até que encontremos ou até que concluamos que o elemento procurado não está no vetor.

Esta é a base do raciocínio dos algoritmos de pesquisa ou busca ("Search"), que como sugere o nome, "Pesquisam", em um vetor, a existência ou não existência de um elemento procurado. A diferença entre um e outro algoritmo de busca, fica por conta da rapidez com que "varremos" o vetor para encontrar o elemento ou para concluirmos que ele não existe.

Um fator que influencia em muito nessa rapidez é a disposição dos elementos no vetor. Se estão desordenados, seguramente teremos que verificar do primeiro ao último elemento para concluir, com certeza, que o elemento não existe. Já se estão ordenados, ao encontrarmos um elemento maior (ou menor) que o elemento procurado, poderemos concluir pela sua não existência. Os algoritmos de ordenação ou classificação ("Sort"), por sua vez, são utilizados para ordenar os elementos de um vetor de forma a facilitar a pesquisa posterior de um elemento, no conjunto de elementos existentes.

Existem algoritmos para pesquisa e ordenação para as mais variadas estruturas de dados. Na nossa disciplina, entretanto, trataremos apenas os algoritmos de pesquisa e ordenação em vetores, que também podem ser utilizados em matrizes, desde que sofram pequenos ajustes.

7.2. ALGORITMOS DE PESQUISA

Para fazermos qualquer pesquisa em vetor (ou matriz) precisamos de quatro parâmetros:

- a) O vetor no qual realizaremos a pesquisa
- b) O número de elementos desse vetor que devem ser pesquisados. (Lembre-se que muitas vezes um vetor de 1000 elementos, só tem 700 carregados, e podemos evitar tratamento de 300 elementos com "lixo").
- c) O elemento procurado
- d) Um índice que vai ser preenchido com a posição onde o elemento foi encontrado ou retornará com 0 (zero) caso o elemento não exista.

Como os algoritmos de pesquisa poderão ser utilizados muitas vezes na solução de diferentes problemas, vamos defini-los como Procedimentos de um Algoritmo Principal hipotético, com os seguintes argumentos:

Declarações:

Const

MAXELEM = 10000 {Número máximo de elementos do vetor, apenas para limitar o tamanho do vetor}

<Tipo básico> Nome do vetor[1:MAXELEM]

inteiro TOTELEM {Corresponde ao parâmetro b) Total de elementos a ser pesquisado}

<Tipo básico> ELEMPROC {Corresponde ao Parâmetro c) Elemento procurado}

Inteiro POS {Corresponde ao Parâmetro d) Posição em VET onde ELEMPROC foi encontrado, ou 0 se não o foi}

7.2.1. PESQUISA SEQÜENCIAL SIMPLES

Na pesquisa seqüencial simples, como o vetor a ser pesquisado não está ordenado pelo elemento procurado, teremos de comparar um a um o ELEMPROC com cada elemento de VET. Portanto para um elemento inexistente teremos de fazer TOTELEM testes e para um elemento existente faremos, na média, TOTELEM/2 testes.

Procedimento PESQSEQ (<tipo básico>VET);

Início

Inteiro TOTELEM, POS, J;

<Tipo básico> ELEMPROC;

Leia(ELEMPROC);

/*Executa a Pesquisa da primeira ocorrência de ELEMPROC em VET e retorna em POS o índice onde foi encontrada ou 0 se não existe*/

Enquanto (Pos==0) e (J<=TOTELEM) Faça

Se VET[J]==ELEMPROC Então

POS = J;

Senão

J = J+1;

Fim Se;

Fim Enquanto;

Fim.

Se precisamos determinar todas as ocorrências de um elemento em um vetor, o problema se simplifica, pois teremos que, obrigatoriamente, varrer o vetor até o fim (Elimina-se do laço o teste $POS = =0$), mas teremos de guardar em um vetor todas as posições onde o elemento foi encontrado ou, o que é mais usual, processaremos a ocorrência dentro do procedimento (após $POS = J$). O trecho de procedimento abaixo exemplifica:

- **Pesquisa Seqüencial com Repetição**

Trata todas as ocorrências de um elemento procurado no vetor

Início

```
POS = 0; /*Indicará ao algoritmo chamador a não ocorrência*/
```

```
J = 1;
```

```
Enquanto J < = TOTELEM Faça
```

```
  Se VET[J] == ELEMPROC Então
```

```
    POS = J;
```

```
    /*Imprime, Acumula, etc.*/
```

```
  Fim Se;
```

```
    J = J + 1;
```

```
Fim Enquanto;
```

Fim.

7.2.2. PESQUISA SEQÜENCIAL ORDENADA

Para se utilizar a Pesquisa Seqüencial Ordenada, o vetor de busca tem de estar ordenado pelo campo chave da pesquisa. Com isso, ao se encontrar no vetor, um elemento maior do que o elemento procurado, poderemos abandonar a busca pois, com certeza, não mais o encontraremos. Devido a isso o número de testes para elementos existentes ou inexistentes será, na média, de $TOTELEM/2$, e, por isso, melhor que o anterior para o caso de elemento inexistente.

```
Procedimento PESQORD (<tipo básico> VET, Inteiro TOTELEM, inteiro POS, <Tipo básico>
ELEMPROC);
```

```
Início
```

```
  Inteiro J;
```

```
  /*Executa a Pesquisa da primeira ocorrência de ELEMPROC em VET e
  retorna em POS o índice onde foi encontrada ou 0 se não existe. VET
  tem de estar ordenado pelo campo chave*/
```

```
  POS = 0;
```

```
  J = 1;
```

```
  Enquanto J<=TOTELEM e POS==0 Faça
```

```
    Se VET[J]>=ELEMPROC Então
```

```
      Se VET[J] == ELEMPROC Então
```

```
        POS=J;
```

```
      Senão
```

```
        Imprima(" O elemento não está na lista");
```

```
        Abandone;
```

```
      Fim Se;
```

```
    Senão
```

```
      J=J+1;
```

```
    Fim Se;
```

```
  Fim Enquanto;
```

```
Fim.
```

Exercício resolvido - Dado um vetor A de 128 elementos, verificar se existe um elemento igual a K (chave) no vetor. Se existir, imprimir a posição onde foi encontrada a chave; se não, imprimir: “chave K não encontrada”. O vetor A e chave K são lidos de cartões.

```
início
```

```
  inteiro A[1:128] inteiro;
```

```
  inteiro I,K;
```

```
  lógico ACHOU;
```

```
  leia (K);
```

```
  leia (A);
```

```
  ACHOU = falso;
```

```
  para I de 1 até 128 passo 1 faça
```

```
    se A[I]==K então
```

```
      imprima (K, " Está na posição" , I);
```

```
      ACHOU = verdadeiro;
```

```
      abandone;
```

```
    fim se;
```

```
  fim para ;
```

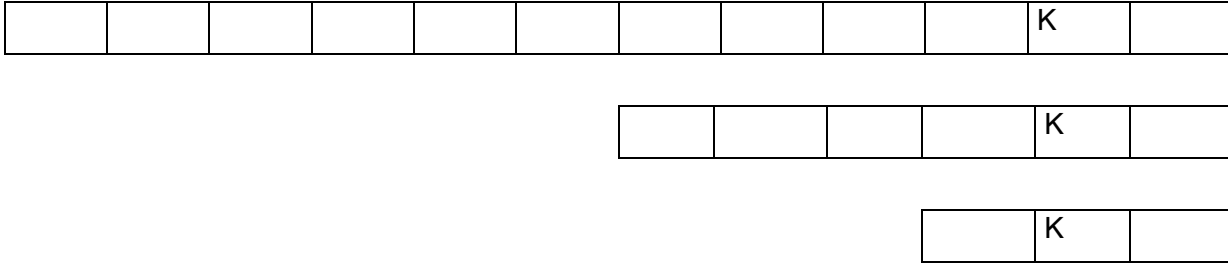
```
  se não ACHOU então
```

```
    imprima ("A CHAVE", K,"NÃO ESTÁ NO VETOR");
```

```
  fim se;
```

```
fim.
```

Neste exemplo, a pesquisa a um elemento ficará mais eficiente se utilizarmos a pesquisa binária, desde que o vetor já esteja ordenado. Nesta pesquisa procuramos o elemento K dividindo o vetor em duas partes e testando em qual das duas ele deveria estar. Procedendo da mesma forma para a parte provável, e assim sucessivamente:



7.2.3. PESQUISA BINÁRIA

A pesquisa binária é um método que também só se aplica a vetores previamente ordenados. A sua grande vantagem é a rapidez e, por isso, ela é muito recomendada para vetores grandes. Neste método, a primeira comparação é feita com o elemento do meio do vetor, eliminando-se assim metade do mesmo para a busca. Seguem-se comparações sucessivas ao elemento do meio do segmento onde pode estar o elemento procurado. Assim, a cada comparação feita, metade do vetor é eliminada.

Procedimento PESQBIN (V: VET, Inteiro: TOTELEM, POS;<Tipo básico> : ELEMPROC)

Início

```
/*Executa a Pesquisa binária da primeira ocorrência da ELEMPROC em VET
e retorna em POS se o índice foi encontrado ou 0 se não existe. VET
tem de estar ordenado pelo campo chave*/
```

```
Inteiro: PRI, ULT, MED {primeiro, último e elemento do meio}
```

```
POS = 0;
```

```
PRI = 1;
```

```
ULT = TOTELEM;
```

```
Enquanto (PRI <= ULT) e (POS == 0) Faça
```

```
    MED = (PRI + ULT)/2 /*quociente da divisão inteira*/
```

```
    Se VET[MED] == ELEMPROC Então
```

```
        POS = MED;
```

```
    Senão Se VET[MED] > ELEMPROC Então
```

```
        /*O ELEMPROC está na primeira metade*/
```

```
        ULT = MED - 1;
```

```
    Senão
```

```
        /*O ELEMPROC está na Segunda metade*/
```

```
        PRI = MED + 1;
```

```
    Fim Se;
```

```
Fim Se;
```

```
Fim Enquanto;
```

```
Fim.
```

- Exercício 1 do item 7.2.2 utilizando-se pesquisa binária.

```

início /*pesquisa binária*/
  inteiro COMEÇO, /*indicador do primeiro elemento da parte do vetor a
                  considerar*/
      FIM, /*indicador do último elemento da parte do vetor a
            considerar*/
      MEIO, /*indicador do elemento do meio da parte do vetor
            Considerada*/
      K; /*elemento procurado*/
Inteiro A[1:128];
leia (A,K);
COMEÇO =1;
FIM=128;
repita
  MEIO= (COMEÇO+FIM)/2;
  se K<A[MEIO] então
    FIM=MEIO-1
  senão
    COMEÇO=MEIO+1;
  fim se
até A[MEIO]==K ou COMEÇO >FIM;
se A[MEIO]!=K então
  imprima (" Não existe o elemento");
senão
  imprima (" Está na posição:" , MEIO);
fim se
fim.

```

7.3. ALGORITMOS DE ORDENAÇÃO

Como já foi dito, o propósito dos algoritmos de ordenação é o de facilitar e acelerar a busca posterior de um elemento no vetor. Os algoritmos de ordenação são utilizados normalmente uma vez em cada execução do programa, ou poucas vezes, se comparados com os de Pesquisa, por isso o uso de métodos elementares e demorados não é tão problemático como nas pesquisas. Na nossa disciplina, veremos três algoritmos para classificação interna de vetores, que têm tempo de execução proporcional ao quadrado do número de elementos a serem ordenados:

- Ordenação por seleção (método seleção direta)
- Ordenação por inserção (método inserção direta)

- Ordenação por troca (método da bolha)

Para fazermos qualquer ordenação em vetor (ou matriz) precisaremos de dois parâmetros:

a) O vetor que será ordenado;

b) O número de elementos desse vetor que devem ser ordenados. (Novamente para evitar tratamento de "lixo")

Da mesma forma que os algoritmos de pesquisa, os algoritmos de ordenação poderão ser utilizados muitas vezes na solução de diferentes problemas e também vamos defini-los como um Procedimento do mesmo Algoritmo Principal, hipotético, com os seguintes argumentos:

Declarações

```
Constante MAXELEM = 1000; /*Número máximo de elementos do vetor, apenas limitar
o tamanho do vetor*/
<tipo básico> VET[1: MAXELEM];/*Corresponde ao Parâmetro a) o vetor a ser
ordenado*/
Inteiro TOTLEM /*Corresponde ao Parâmetro b)*/
```

7.3.1. MÉTODO DE SELEÇÃO DIRETA

Este é um dos mais simples métodos existentes. Primeiro descobre-se o menor elemento do vetor e troca-se com o primeiro elemento. A partir daí, com os elementos remanescentes, repete-se o processo até que todo o vetor esteja ordenado. Um exemplo bastante elucidativo é o de algumas cartas de baralho na mesa viradas para cima (à vista). Seleciona-se a menor (ou maior) para colocá-la na primeira posição. Depois, seleciona-se, sucessivamente, a menor carta dentre as remanescentes e se a coloca sobre a primeira.

Procedimento SELEÇÃO_DIRETA (V: VET, Inteiro: TOTELEM)

Início

```
/*Faz a ordenação crescente de TOTELEM elementos de um vetor VET*/
```

```
Inteiro MIN, I, J; /*índices*/
```

```
<Tipo Básico> AUX;
```

```
Para K de 1 até (TOTELEM - 1) Faça
```

```
    MIN ←K;
```

```
    Para J de (K + 1) até TOTELEM Faça
```

```
        Se VET[J] < VET [MIN] Então
```

```
            MIN = J;
```

```
        Fim Se;
```

```
    Fim Para;
```

```
    AUX = VET[MIN];
```

```
    VET[MIN] = VET[K];
```

```
    VET[K] = AUX;
```

```
Fim Para;
```

Fim.

7.3.2. MÉTODO DE INSERÇÃO DIRETA

Nesse método, considera-se que um elemento do vetor (o primeiro) está ordenado e "insere-se" os demais considerando os anteriores já ordenados. Voltando-se ao exemplo do baralho é como se agora as cartas sobre a mesa estão com a face para baixo e cada carta é arrumada ordenadamente na mão em relação as que já estão na mão.

Este método pode ser utilizado, com vantagem, quando se está lendo elementos de um arquivo para serem posteriormente ordenados. Bastam leves modificações no algoritmo, que leremos o arquivo e já o armazenaremos na memória ordenadamente.

Procedimento INSERÇÃO_DIRETA (V: VET, Inteiro : TOTELEM)

Início

/*Faz a ordenação de TOTELEM elementos de um vetor VET*/

<Tipo básico> AUX;

Inteiro K, J; /*Índice*/

Para K de 2 até TOTELEM Faça

AUX = VET[K];

J = K - 1;

Enquanto (J > 0) Faça

Se AUX < VET[J] Então

VET [J + 1] = VET [J];

J = J - 1;

Senão

Abandone;

Fim Se;

Fim Enquanto;

VET [J + 1] = AUX;

Fim Para;

Fim.

7.3.3. MÉTODO DA BOLHA

Este método caracteriza-se por efetuar a comparação sucessiva de pares subseqüentes de elementos, trocando-os de posição, se estiverem fora de ordem. Dos três este é o menos eficiente, já que cada troca de elementos exige três comandos e o número de trocas é (TOTELEM-1) vezes maior que no método de seleção direta, por exemplo. Além disso o método é mais complicado de entender que o de seleção direta. Segue-se exemplo de uma implementação de algoritmo de pesquisa utilizando-se esse método.

Procedimento BOLHACRESCENTE (V: VET, Inteiro: TOTELEM)

Início

/*Faz o ordenamento de TOTELEM elementos de um valor VET*/

Inteiro P,J; /*contador de passadas e índice*/

<Tipo básico> AUX;

Para P de 1 até (TOTELEM - 1) Faça

Para J de 1 até (TOTELEM - P) Faça

Se VET[J + 1]<VET[J] Então

AUX = VET[J]; /*Troca dos elementos*/

VET[J] =VET[J + 1];

VET[J + 1] =AUX;

Fim Se;

Fim Para;

Fim Para;

Fim.

- Exercício Resolvido

1) Classificar um vetor numérico VET de 6 elementos em ordem crescente:

início /*método da bolha*/

inteiro VET[1:6]

inteiro: AUX, /* auxiliar para troca de elementos*/

BOLHA, /* indicador de mais alto elemento fora de ordem*/

LSUP, /*indicador do tamanho do vetor a ser pesquisado, sendo o
valor inicial igual a 6*/

J; /*indicador do elemento do vetor*/

leia (VET);

LSUP = 6;

enquanto LSUP>1 faça

BOLHA = 0;

para J de 1 até LSUP-1 faça

se VET[J]>VET[J+1] então /* troca elemento j com j+1*/

AUX = VET[J];

VET[J] = VET[J+1];

VET[J+1] = AUX;

BOLHA = J;

fim se;

fim para

LSUP = BOLHA; /*aponta para última posição trocada*/

fim enquanto;

imprima (VET);

fim.

REFERÊNCIAS

- [1] Farrer H. et ali. **Algoritmos Estruturados.**, Editora Guanabara S. A., Rio de Janeiro, 1985.
- [2] Guimarães, Angelo M & Lages, Newton A. de C. **Algoritmos e Estrutura de dados.** LTC - Livros Técnicos e Científicos Editora S.A.; 216 – 1985
- [3] Tremblay, J. P; Bunt R. **Ciência dos Computadores - Uma Abordagem Algorítmica.** Editora McGraw Hill do Brasil, são Paulo, 1983.
- [4] Villas, Marcos V, & Villasboas, Luiz F.P. **Programação Conceitos, Técnicas e Linguagens.** Ed. Campus; 195 p. 1988.
- [5] Filho, José Vanni, **Construção de Algoritmos.** PUC-Rio . 4ª edição – Setembro de 95
- [6] Manzano, J. ; Oliveira, J. **Algoritmos** . Editora Érica. Décima Quarta Edição. 2002.
- [7] Santos, H. **Curso de Linguagem C – UFMG.** 2003