UNIVERSIDADE DO ESTADO DE SANTA CATARINA

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

CURSO DE TECNOLOGIA EM SISTEMAS DE INFORMAÇÃO

INTRODUÇÃO A

CIÊNCIA DA COMPUTAÇÃO

PREFÁCIO

Bem-vindo à disciplina de Introdução à Ciência da Computação do curso de Tecnologia em Sistemas de Informação da UDESC. Neste “novo mundo”, para muitos que aqui ingressaram, eis que aqui está uma das disciplinas mais complexas da primeira fase, principalmente por se tratar de algo fora do convencional, lecionado em salas de aula normais.

Esta diferença dá-se justamente por se tratar da disciplina-base do curso, pois é aqui que vocês aprenderão a programar, ou seja, todo o conteúdo adquirido nesta disciplina os acompanhará para o resto de suas vidas, caso vocês permaneçam na área de programação.

Um auxílio, tanto para nós professores, que podemos dinamizar as aulas e cumprir o conteúdo dentro do prazo estipulado com uma maior gama de exemplos e aulas práticas, como para vocês alunos, que tem em mãos um resumo do conteúdo, com exemplos e vários exercícios para serem feitos.

Exercícios estes que são a chave para o sucesso do acadêmico no curso, pois são mais de 130 exercícios de lógica, além dos exercícios avaliativos e exercícios extras que não se encontram neste material que são aplicados durante o semestre.

Mas mesmo com a apostila e fazendo todos os exercícios, o acadêmico somente terá o seu diferencial caso ele pesquise na rede, em livros, em apostilas e faça algo mais, não somente o que lhe foi pedido, o que lhe foi mandado, pois assim ele será apenas mais um dentre vários acadêmicos que se formam a cada semestre. Enfim, o sucesso depende de cada um, de seu esforço e sua vontade, não somente aqui na universidade, mas em todas as áreas da vida.

São Bento do Sul, SC.

SUMÁRIO

UNIDADE 1 - INICIAÇÃO AOS COMPUTADORES

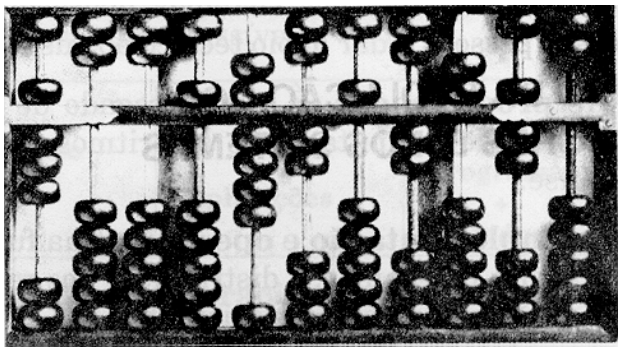
1.1 - HISTÓRICO DA COMPUTAÇÃO

A Informática é uma ciência que passou a ser tratada como tal há poucos anos, sendo que a ela está associada uma série de fatos e descobertas anteriores, que serviram para que atualmente seja uma das ciências à qual o homem está dedicando maior atenção e atribuindo cada vez mais importância.

1.1.1 - Fatos e Personagens Históricos

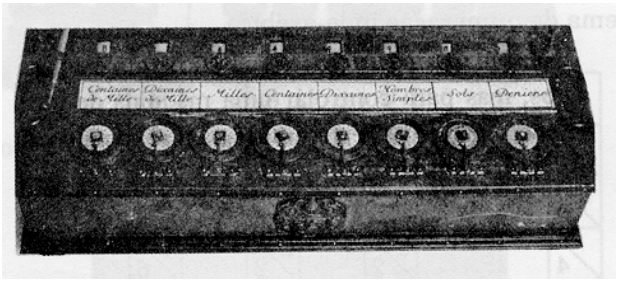
Já há muito tempo o homem vem tentando livrar-se dos trabalhos manuais e repetitivos, entre os quais estão as operações de cálculo e a redação de relatórios.

- O **ábaco** mais antigo data aproximadamente do ano 3500 a.C., no vale entre Tigre e o Eufrates. Por volta do ano 2600 a.C. apareceu o ábaco chinês, que evoluiu rapidamente. O ábaco constituiu o primeiro dispositivo manual de cálculo, sendo que sua principal função era representar números no sistema decimal e realizar operações com eles.



- No final do século XVI, o matemático escocês John Napier (l550-1617), inventor dos logaritmos naturais, idealizou um dispositivo baseado em bastões que continham números capaz de multiplicar e dividir de forma automática. Também idealizou um calculador com cartões que permitia efetuar multiplicações e que recebeu o nome de **Estruturas de Napier**.

- Poucos anos depois, em 1642, o matemático e filósofo francês Blaise Pascal (l623-1662) inventou a primeira máquina automática de calcular, constituída de rodas dentadas, baseada no funcionamento do ábaco.
  Essa máquina realizava operações de soma e subtração mostrando o resultado numa série de janelinhas. Num primeiro momento foi chamada Pascalina, recebendo mais tarde o nome de **Máquina Aritmética de Pascal**.



- Em 1650, Patridge, com base nas experiências de Napier, inventou a **régua de cálculo,** pequena régua deslizante sobre uma base fixa em que havia diversas escalas para a realização de determinadas operações. Esse dispositivo de cálculo foi muito utilizado até os anos setenta, quando foi substituído pelas calculadoras eletrônicas.

- Poucos anos depois, 1672, o matemático alemão Gottfried Wilhelm von Leibnitz (l646-1716) aprimorou a máquina de Pascal, obtendo a **calculadora universal,** que somava, subtraía, multiplicava, dividia e extraía a raiz quadrada.

- Já no século XIX, em, 1801, Joseph Marie Jackuard (l752 - 1834) construiu um tear automático com entrada de dados através de cartões perfurados para controlar a confecção dos tecidos e seus desenhos. Podemos considerá-lo a primeira máquina mecânica programada.

- No ano de 1822, Charles Babbage (l792-1871), matemático inglês e professor da Universidade de Cambridge, projetou a sua **Máquina de Diferenças,** um dispositivo mecânico baseado em rodas dentadas, para a avaliação de funções e obtenção de tabelas. Devido às deficiências tecnológicas da época, essa máquina não chegou a ser fabricada.

- Em 1833, Babbage projetou a sua **Máquina Analítica ou Diferencial,** semelhante ao computador atual, pois dispunha de programa, memória, unidade de controle e periféricos de entrada e saída. A idéia da sua construção surgiu da necessidade de se realizar automaticamente tabelas de logaritmos e funções trigonométricas. Essa máquina - pelos mesmos motivos da sua antecessora, a Máquina de Diferenças - não chegou a ser construída. Devido a esse projeto, Babbage é considerado o **Pai da informática**.

- Em 1854, George Boole, matemático inglês, desenvolveu a teoria da Álgebra de Boole, que permitiu a seus sucessores a representação de circuitos de comutação e o desenvolvimento da chamada **Teoria dos Circuitos Lógicos**.

- Em 1937, Howard H. Aiken, da Universidade de Harvard, desenvolve a idéia de Babbage junto com cientistas do seu departamento e engenheiros da IBM. Como resultado desse desenvolvimento, construíram o primeiro computador eletromecânico baseado em relês e engrenagens, denominado Calculadora Automática de Seqüência Controlada *(Automatic Sequence Controlled Calculator* - ASCC), que recebeu o nome de MARK-I. O MARK-1 acabou de ser construído em 1944 e possuía unidades de entrada, memória principal e unidade aritmética de controle e de saída. Utilizava como entrada cartões e fitas perfuradas, tinha 17 metros de comprimento por 2 metros de altura, pesava cerca de 70 toneladas, era constituído de 700.000 peças móveis e sua fiação alcançava os 800.000 metros. Somava dois números em menos de um segundo e os multiplicava em seis segundos.

- Em 1940, John W. Mauchly e J. Presper Eckert Jr., junto com cientistas da Universidade da Pensylvania, construíram na Escola Moore de Engenharia Elétrica, o primeiro computador eletrônico, denominado ENIAC *(Electronic Numerical Integrator and Calculator),* que entrou em funcionamento em 1945. Foi um projeto do Exército dos Estados Unidos para o cálculo da trajetória de projéteis através de tabelas.

- Em 1951, Mauchly constrói o primeiro computador da série a ser posto à venda, o UNIVAC-I (Computador Automático Universal), que utilizava fitas magnéticas.

- Em 1952, são construídos os computares MANIAC-I, MANIAC-II e o UNIVAC-H (este último com memória de núcleos de ferrite) e com eles acaba a pré-história da Informática.

1.1.2 - Evolução da Eletrônica

Desde o surgimento do UNIVAC-I, em 1951, como primeiro computador comercial, até hoje, quando existe uma infinidade de modelos cada vez mais potentes, baratos e pequenos, quase todas as transformações foram impulsionadas por descobertas ou avanços no campo da eletrônica. Esses avanços podem ser resumidos em:

Em 1904, o inglês Fleming inventou a **válvula a vácuo**, que foi utilizada como elemento de controle e para integrar dispositivos biestáveis.

Nos anos cinqüenta, com a descoberta dos semicondutores, surgiu o **diodo** e o **transistor**, permitindo a redução do tamanho dos circuitos e aumentando a confiabilidade dos equipamentos.

Anos depois, começou a miniaturização com a construção dos **circuitos integrados** que consistiam em tratamentos físico-químicos sobre uma película de silício, permitindo configurar diferentes circuitos de portas lógicas. Com esse elemento teve início a ciência do projeto lógico de circuitos com baixa escala de integração ou SSI *(Short Scale Integration)* que permitia introduzir em cada circuito em média de 10 portas lógicas.

Surgiu a integração em escala média ou MSI *(Medium Scale* Integration) na qual passaram a integrar-se numa única pastilha de circuito integrado entre 100 e 1.000 portas lógicas.

Anos mais tarde, conseguiu-se introduzir num mesmo circuito entre 1.000 e 10.000 portas, com o que se passou à integração em grande escala ou LSI *(Long Scale Integration).*

Quando foram ultrapassadas as 10.000 portas por circuito, passou-se à altíssima escala de integração ou VLSI (*Very Long Scale Integration).*

Em 1971 apareceu o **microprocessador,** com o que se conseguiu implementar toda a CPU de um computador num único elemento integrado.

1.1.3 - Gerações de Computadores

Os saltos tecnológicos ocorridos implicaram numa classificação dos computadores em gerações:

- **1ª. Geração** (l940-1952). É constituída por todos os computadores construídos à base de válvulas a vácuo e cuja aplicação fundamental se deu nos campos científico e militar. Utilizavam como linguagem de programação a linguagem de máquina e como única memória para armazenar informação os cartões perfurados e as linhas de retardo de mercúrio.

- **2ª. Geração** (l952-1964). A substituição da válvula pelo transistor deu início à chamada segunda geração de computadores. As máquinas ganharam mais potência e contabilidade, com redução de tamanho e consumo. Começaram a ser utilizadas linguagens de programação mais evoluídas, como as linguagens de montagem (assembly) e algumas das chamadas de alto nível (COBOL, ALGOL e FORTRAN). Além disso, a fita magnética e os tambores magnéticos começaram a ser utilizados como memória.

- **3ª. Geração** (l964-1971). Nesta geração o elemento mais significativo é o circuito integrado surgido em 1964 e que consiste no encapsulamento de uma grande quantidade de componentes discretos (resistências, condensadores, diodos e transistores), instalando-se um ou vários circuitos numa pastilha de silicone ou plástico. A miniaturização se estende a todos os circuitos do computador, aparecendo os minicomputadores. Foram utilizadas temologias SSI e MSI. A par disso o software evoluiu consideravelmente, com um grande desenvolvimento dos sistemas operacionais, nos quais se incluiu a multiprogramação, o tempo real e o modo interativo. Passaram a se utilizar as memórias de semicondutores e os discos magnéticos.

- **4ª. Geração** (l971-1981). Em 1971 aparece o microprocessador, que consiste na inclusão de toda a CPU de um computador num único circuito integrado. A tecnologia utilizada é a LSI que permitiu a fabricação de microcomputadores e computadores pessoais, bem como computadores compactos. É o início da utilização do disquete *(floppy disk)* como unidade de armazenamento. Surgiu uma grande quantidade de linguagens de programação de todos os tipos e as redes de transmissão de dados para a interligação de computadores (Telemática/Informática).

- **5ª. Geração** (l98l-?). Em 1981 os principais países produtores de novas tecnologias anunciaram uma nova geração cujas principais características residem na utilização de:
  - Componentes com altíssima escala de integração (tecnologia VLSI).
  - Inteligência artificial.

- Linguagem natural.
- Altíssima velocidade de processamento.
- Etc.

1.2 - APRESENTAÇÃO DE UM MODELO DE COMPUTADOR

1.2.1 - Introdução

A informática é a técnica propulsora do processo de transformação tecnológica e cultural que hoje muda, e continuará mudando a realidade subjetiva e social, mas ela é apenas a parte imersa do *íceberg.* Isso se deve ao fato de que, enquanto o uso de palavras como *software, hardware, CPU, microprocessador, CD-ROM* é apresentado como meta a ser alcançada na corrida à "alfabetização", as características e os efeitos da introdução das novas tecnologias parecem estar cada vez mais reservados ao conhecimento de poucos. O processo em questão requer uma visão global, unitária, uma observação que, ao contrário do que vem ocorrendo, seja fruto de diversas abordagens disciplinares e científicas.

1.2.2 - Definição e Origem do termo Informática

Ao longo da história o homem tem precisado constantemente tratar e transmitir informação, por isso nunca parou de criar máquinas e métodos para processá-la. Com esta finalidade, surge a informática como uma ciência encarregada do estudo e desenvolvimento dessas máquinas e métodos.

A informática nasceu da idéia de auxiliar o homem nos trabalhos rotineiros e repetitivos, em geral de cálculo e gerenciamento.

O termo **Informática** foi criado na França em 1962, e provém da contração das palavras:

Information auto**matique** (Informação automática).

Uma das definições mais comumente aceitas:

Informática é a ciência que estuda o tratamento automático e racional da informação.

Entre as principais funções da informática destacam-se:

- desenvolvimento de novas máquinas;
- desenvolvimento de novos métodos de trabalho;
- construção de aplicações automáticas;
- melhoria de métodos e aplicações existentes.

1.2.3 - Elementos e Conceitos Fundamentais

Do ponto de vista informático, o elemento físico utilizado para o tratamento de dados e a obtenção de informação é o computador.

O **computador** é uma máquina composta de elementos físicos do tipo eletrônico, capaz de realizar uma grande variedade de trabalhos com alta velocidade e precisão, desde que receba as instruções adequadas.

Ao conjunto de ordens dadas a um computador para a realização de um determinado processo dá-se o nome de **programa.** Ao conjunto de um ou vários programas que realizam determinado trabalho completo dá-se o nome de **aplicação informática.**

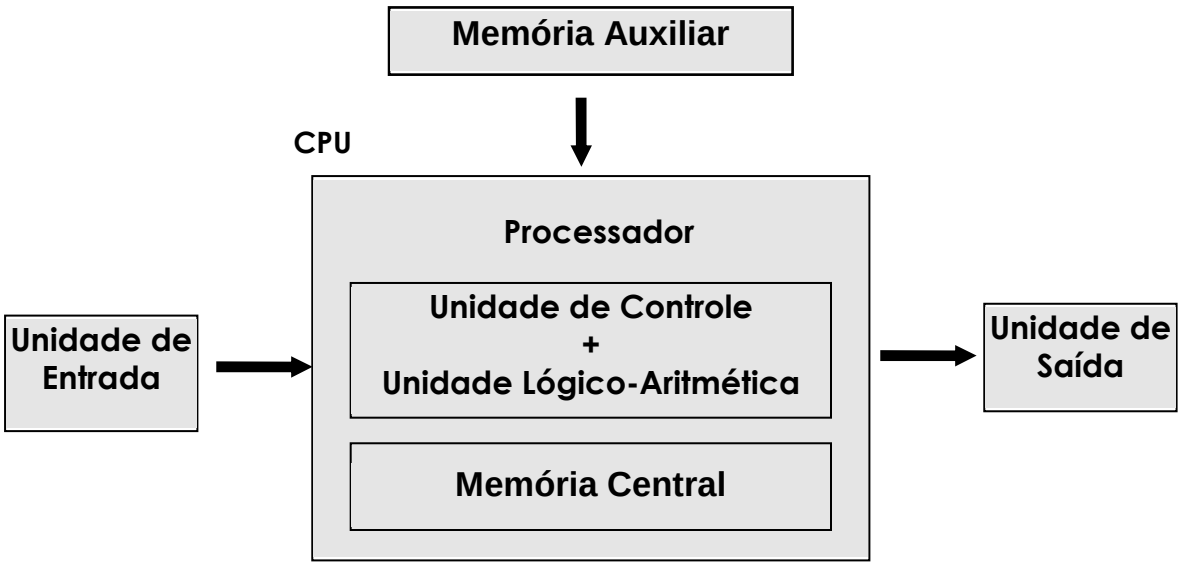
A **informação** é o elemento a ser tratado e é definida como tudo aquilo que permite adquirir qualquer tipo de conhecimento e portanto, existirá informação sempre que se der a conhecer algo que até então se desconhecia.

1.2.4 - Hardware

O hardware representa a parte física de um sistema informático, ou seja, todos os elementos que o compõem.

Tudo que você "ver e tocar" no computador é hardware, ou seja, teclado, monitor, impressora, etc.

O esquema básico do hardware é o seguinte:



1.2.5 - Software

Para que um computador funcione é necessário de programas. Estes programas são chamados de softwares. Software é o conjunto de instruções, organizadas em um ou mais arquivos, que manda o computador executar tarefas que solucionam determinados problemas. Os softwares são compostos de elementos lógicos que direcionam a ação do hardware. Os softwares são classificados em: básico, aplicativos, linguagens e utilitários.

Software básico (sistema): são programas que administram, operam e mantém o funcionamento do computador, ao mesmo tempo em que nos auxiliam a usá-lo. Dentre os softwares básicos destaca-se o sistema operacional, cujo conjunto de programas tem a finalidade de tornar o computador operacional, isto é, são os programas que gerenciam os recursos da máquina e executam tarefas definidas, tais como: gravar dados, apagar, etc..., servindo de ligação entre o computador e o usuário, criando ambiente necessário dentro do qual os softwares e aplicativos são executados. Exemplo: DOS, Windows 95, OS/2, Unix, etc.

Software de aplicação: qualquer programa que nos possibilite tirar proveito do computador, como o *Word* um processador de textos, planilhas eletrônicas, editores gráficos, gerenciadores de banco de dados, jogos, etc.

Utilitários: são programas que tem a finalidade de dar apoio a operação do computador. Executa rotinas para tarefas realizadas freqüentemente como: compactação/descompactação de arquivos, detecção e eliminação de vírus, entre outros.

Linguagens de programação: as linguagens surgiram da necessidade de comunicação entre o homem e o computador. É a forma que possibilita o homem a desenvolver aplicações, por exemplo: folha de pagamento, sistema contábil, sistemas acadêmicos, etc. As linguagens são os meios de implementação desses programas. Exemplo: Pascal, C++, Cobol, Visual Basic, etc.

1.3 - UNIDADES BÁSICAS DO COMPUTADOR DIGITAL

Conforme visto na unidade 1.2.4, os componentes básicos de um computador são:

- **Unidade Central de Processamento (CPU)**: é o elemento principal do computador e sua função consiste em coordenar, controlar ou realizar todas as operações do sistema.

- **Unidades de Entrada e Saída**: também chamadas de periféricos, são responsáveis pela entrada ou saída de informações no computador.

- **Memória**: são os dispositivos capazes de armazenar informações.

Veremos detalhadamente cada componente na seqüência.

1.4 - UNIDADE CENTRAL DE PROCESSAMENTO

A unidade central de processamento (CPU) é o verdadeiro cérebro do computador sendo que sua tarefa consiste em coordenar e controlar ou realizar das as operações do sistema. A CPU compõe-se de elementos cuja natureza é exclusivamente eletrônica (circuitos).

A CPU é o cérebro do computador e é nesta unidade que ocorrem as decisões, cálculos, etc.

Suas partes principais são as seguintes:

O **Processador**: Que por sua vez é composto de:
- Unidade de controle (UC).
- Unidade Lógica e Aritmética (ULA).

- A **Memória Principal**: A unidade central de processamento também incorpora um certo número de registros rápidos (pequenas unidades de memória) de finalidade especial, que são utilizados internamente.

1.4.1 - Unidade de Controle (UC)

A unidade de controle (UC) é o centro nervoso de computador, controla o fluxo de dados entre as unidades da CPU, buscando uma a uma as instruções e os dados armazenados na memória principal e distribuindo para os módulos responsáveis pela sua execução, verificando, por exemplo, se serão enviados para a unidade de entrada/ saída ou para unidade lógico-aritmética.

1.4.2 - Unidade Lógico-Aritmética (ULA)

Esta unidade encarrega-se da realização das operações elementares de tipo aritmético (geralmente somas ou subtrações) e de tipo lógico (geralmente comparações).

1.5 - MEMÓRIA PRIMÁRIA E SECUNDÁRIA

O termo memória se aplica a todo dispositivo capaz de armazenar informações.

1.5.1 - Memória Primária (ou Principal)

É a parte da unidade central de processamento de um computador onde estão armazenadas as instruções e os dados necessários para que um determinado processo possa ser realizado.

A memória principal é constituída por um grande número de células ou posições de memória numeradas de forma consecutiva, capazes de reter a informação necessária enquanto o computador estiver ligado.

Classificam-se em dois tipos:
- Memória RAM
- Memória ROM (Read Only Memory)

1.5.1.1 - Memória RAM (Random Access Memory)

Esta memória retém os dados e programas que estão sendo executados, tornando o tempo de leitura e gravação extremamente rápidos. Este tipo de memória é denominado volátil, pois seu conteúdo é perdido ao se desligar o micro ou até mesmo devido a uma queda súbita de energia.

1.5.1.2 - Memória ROM (Read Only Memory)

É a memória de leitura, pois as informações são gravadas no momento da fabricação e não mais serão alteradas. Contém basicamente informações necessárias para o funcionamento do computador, como rotinas que verificam se os meios físicos estão aptos para o funcionamento.

1.5.2 - Memória Secundária (ou Auxiliar)

As memórias auxiliares são usadas em computação para definir qualquer dispositivo utilizado como meio para guardar informações que não podem ser perdidas, mesmo quando o computador é desligado por qualquer motivo. Este tipo de memória é conhecido ainda como memória de massa ou externa.

O tempo de leitura e gravação são mais lentos que o da memória principal, tendo em vista que a principal está relacionada diretamente com a CPU, enquanto que a auxiliar encontra-se fora da CPU, e necessita de um meio externo, acionador de leitura e gravação, para o armazenamento das informações desejadas.

Exemplo de memórias auxiliares:

- Fita magnética
- Disco magnético - Rígidos (winchester) ou Flexível (disquete)
- Cartão Perfurado
- CD-ROM

Discos Magnéticos

São elementos físicos compostos por uma base de plástico ou metal recoberta por uma fina camada de material magnético (normalmente óxido de ferro) onde se registra a informação em pontos magnetizáveis.

Os discos magnéticos são reutilizáveis, uma vez que a informação pode ser apagada ou gravada quantas vezes for necessário.

Os discos magnéticos são circulares, divididos em trilhas e setores. Divide-se em:

- **Discos Flexíveis** (ou Disquetes): Possuem um orifício central que serve para encaixá-lo em um mecanismo de rotação e um pequeno orifício de controle que serve de índice para referenciar o começo da trilha. Possuem três tamanhos e são medidos em polegadas, de acordo com seu tamanho: 8", 5 ¼" e 3 ½". As operações de leitura e gravação é feito por meio de um acionador de discos flexíveis chamado "driver".Dentre as vantagens podemos citar:
  - Portabilidade que nos permite utilizar a mesma massa de dados em vários computadores;
  - O custo dos discos flexíveis é relativamente baixo.

- **Discos Rígidos** (ou Winchester): Estes discos são acoplados no interior do computador. Tem como principais vantagens:
  - a grande capacidade de armazenamento;
  - maior velocidade de acesso se comparados com os discos flexíveis.

1.6 - PERIFÉRICOS OU MEIOS DE ENTRADA E SAÍDA

Em informática, dispositivos de entrada e saída é todo dispositivo que permite a comunicação do computador com o mundo exterior. Existem vários tipos de entrada e saída como: mouse, impressoras, teclados, scanner, etc.
A transferência de dados, realizada nessa comunicação, pode ser efetuada através de blocos de informação ou palavra a palavra. A transferência é realizada através dos controladores dos dispositivos sob supervisão da CPU.

Os periféricos são utilizados para introduzir ou extrair informações no computador.

Podemos distinguir três categorias de periféricos:

- Dispositivos de Entrada
- Dispositivos de Saída
- Dispositivos de Entrada/Saída

1.6.1 - Dispositivos de Entrada

São os utilizados para introduzir no computador a informação que vai ser objeto de tratamento.
Exemplos: teclado, mouse, scanner, mesa digitalizadora, canetas óticas, leitora de código de barras, telas de superfície sensíveis ao toque, etc.

1.6.2 - Dispositivos de Saída

Os dispositivos de saída convertem as informações internamente armazenados no computador e os transforma em informações úteis ao mundo exterior.
Exemplos: Impressora, vídeo, plotter, etc.

Vídeo

Este periférico é tipicamente de saída de informação. Embora a primeira vista nos confunda o fato de ao digitarmos termos a impressão de que estamos introduzindo os dados a partir da tela.

O menor ponto que se pode obter em um dado vídeo é chamado de pixel (picture element) e depende da qualidade do monitor e da placa de vídeo que o controla. Com isso podemos ter um monitor de baixa, média ou alta resolução.

Observação: A linguagem C trabalha com o vídeo através de 80 colunas por 25 linhas.

Impressora

Obviamente, a utilidade deste periférico está na produção de informação escrita; portanto, podemos afirmar que ela é uma unidade unicamente de saída.
Podem ser divididas em impressoras de impacto (matricial) e de não impacto(jato de tinta, lazer, etc.).

1.7 - NOÇÕES DE SISTEMA OPERACIONAL

Um computador recém saído da linha de montagem, sem software residente, não pode fazer absolutamente nada. Não aceita caracteres digitados via teclado e nem os exibe na tela. Não pode sequer carregar ou muito menos executar um programa. Diante da máquina "crua", mesmo programadores experientes encontrarão dificuldades para fazerem alguma coisa, enquanto os usuários sem formação técnica perdem-se completamente. Uma vez que o hardware “puro” apresenta-se como a menos cooperativa das interfaces, as pessoas raramente comunicam-se com ele. Usuários e programadores lidam com o hardware por intermédio de um programa do sistema chamado **Sistema Operacional**.

O sistema operacional serve como interface entre o software aplicativo e o hardware.

A mais evidente função do sistema operacional é servir como interface com o hardware. Isso, porém, não é tudo que ele faz. Os recursos básicos do computador consistem no hardware, software e os dados. Todos esses recursos são gerenciados pelos sistemas operacionais modernos, especialmente na máquina de grande porte.

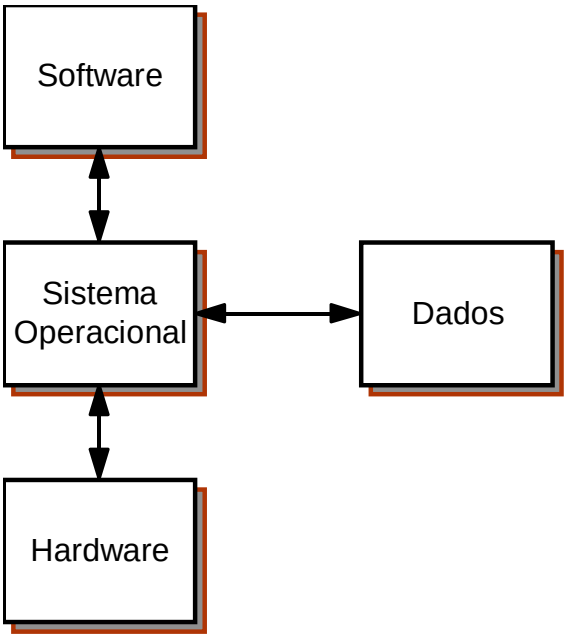


Figura: Em muitos computadores, o sistema operacional é um gerenciador de recursos, alocando o hardware, o software e os dados.

O sistema operacional é a interface básica entre o usuário e o computador. Por isto, a medida que a utilização de um computador exige uma maior complexidade, torna-se mais importante um certo conhecimento mínimo do sistema operacional.

1.8 - NOÇÕES BÁSICAS DE REDE LOCAL

Uma rede local é distinguida de outras redes pela área que ela serve, a velocidade com que as informações são transmitidas, a facilidade com que novos equipamentos são adicionados e a simplicidade do meio de transmissão básico.

Rede Local é uma interligação de vários equipamentos que compartilham recursos entre sí.

Geralmente uma rede local serve à uma área geograficamente limitada, isto é, um ambiente de trabalho, em edifício, um campus universitário, uma fábrica, etc. As distâncias podem variar de metros até alguns poucos quilômetros. A velocidade de transmissão é da ordem de milhões de *bits* (binary digit) por segundo, em contrastes com a rede de longa distância que transportam dados a uma velocidade que varia de centenas de bits por segundo a milhares de bits por segundo.

A maioria dos produtos existentes na área utiliza uma forma simples de interligação física entre os equipamentos e talvez esta seja uma das características mais atrativas das redes locais, uma vez que o sonho

de todo gerente de rede é ter a facilidade de conectar novos equipamentos sem ter que ligar novos cabos e sem ter que implementar novos protocolos e procedimentos.

Estas características, no entanto, não são suficientes para garantir o sucesso de uma rede local. Para o usuário final, é muito importante ter um mecanismo de transmissão de informação eficiente sem que haja a necessidade de conhecer os detalhes técnicos para efetuar a ligação com a rede.

Uma rede local pode ser descrita através das seguintes características:

- estar completamente contida dentro de uma área geográfica limitada;
- equipamentos são interconectados de forma independentes;
- proporciona alto grau de interconexão entre os equipamentos da rede;
- é usada para transmissão de informação geralmente na forma digital;
- a interface com a rede é feita através de equipamentos e meios de transmissão relativamente baratos;
- é possível a comunicação entre dois equipamentos quaisquer da rede.

Uma das aplicações básicas de rede local é o compartilhamento de recursos, isto é, o compartilhamento de um periférico mais caro, tal como disco ou impressora entre diversos micros.

EXERCÍCIOS

1. Faça uma síntese através de uma tabela da evolução do computador.

2. Repita e exercício anterior para representar as gerações dos computadores.

3. Conceitue informática e relacione algumas atividades de sua vida em que ela faz parte.

4. Conceitue computador, programa, aplicação informática e informação.

5. Conceitue *hardware* e elabore seu esquema básico.

6. Conceitue *software* e relacione suas aplicações.

7. Relacione as unidades básicas de um computador.

8. Conceitue CPU e suas subdivisões (UC e ULA).

9. Conceitue memória, memória principal e memória secundária.

10. Sobre memória secundária os discos magnéticos são, hoje em dia, os mais utilizados. Defina discos magnéticos e as vantagens de cada um.

11. O que são periféricos? Exemplifique.

12. Conceitue sistema operacional e faça um esquema sobre o mesmo.

13. Conceitue rede local e a vantagem de utilização da mesma.

UNIDADE 2 - INTRODUÇÃO À PROGRAMAÇÃO

2.1 - CONCEITO DE ALGORITMO

A automação é o processo em que uma tarefa deixa de ser desempenhada pelo homem e passa a ser realizada por máquinas, sejam estas dispositivos mecânicos, eletrônicos (como os computadores) ou de natureza mista.

Para que a automação de uma tarefa seja bem sucedida é necessário que a máquina que passará a realizá-la seja capaz de desempenhar cada uma das etapas constituintes do processo a ser automatizado com eficiência, de modo a garantir a **repetibilidade** do mesmo. Assim, é necessário que seja especificado com clareza e exatidão o que deve ser realizado em cada uma das fases do processo a ser automatizado, bem como a seqüência em que estas fases devem ser realizadas.

À especificação da seqüência ordenada de passos que deve ser seguida para a realização de uma tarefa, garantindo a sua repetibilidade, dá-se o nome de algoritmo.

Ao contrário do que se pode pensar, o conceito de algoritmo não foi criado para satisfazer às necessidades da computação. Pelo contrário, a programação de computadores é apenas um dos campos de aplicação dos algoritmos. Na verdade, há inúmeros casos que podem exemplificar o uso (involuntário ou não) de algoritmos para a padronização do exercício de tarefas rotineiras.

Para que um computador possa desempenhar uma tarefa é necessário que esta seja detalhada passo a passo, numa forma compreensível pela máquina, utilizando aquilo que se chama de **programa.** Neste sentido, um programa de computador nada mais é que um algoritmo escrito numa forma compreensível pelo computador.

2.2 - APRESENTAÇÃO DE UMA LINGUAGEM HIPOTÉTICA

Existem diversas formas de representação de algoritmos, mas não há um consenso com relação à melhor delas.

Algumas formas de representação de algoritmos tratam os problemas apenas em nível lógico, abstraindo-se de detalhes de implementação muitas vezes relacionados com alguma linguagem de programação específica. Por outro lado, existem formas de representação de algoritmos que possuem uma maior riqueza de detalhes e muitas vezes acabam por obscurecer a idéia principal, o algoritmo, dificultando seu entendimento.

Dentre as formas de representação de algoritmos mais conhecidas sobressaltam:

- **Descrição Narrativa;**
- **Fluxograma Convencional;**
- **Pseudocódigo,** também conhecido como **Linguagem Estruturada** ou **Portugol.**

2.2.1 - Descrição Narrativa

Nesta forma de representação os algoritmos são expressos diretamente em **linguagem natural.** Como exemplo, têm-se os algoritmos seguintes:

- *Receita de bolo:*

  Misture os ingredientes
  Unte a forma com manteiga
  Despeje a mistura na forma
  Se houver coco ralado então despeje sobre a mistura
  Leve a forma ao forno
  Enquanto não corar deixe a forma no forno
  Retire do forno
  Deixe esfriar

- *Troca de um pneu furado:*

  Afrouxar ligeiramente as porcas
  Suspender o carro
  Retirar as porcas e o pneu
  Colocar o pneu reserva
  Apertar as porcas
  Abaixar o carro
  Dar o aperto final nas porcas

- *Cálculo da média de um aluno:*

  Obter as notas da primeira e da segunda provas
  Calcular a média aritmética entre as duas
  Se a média for maior que 7, o aluno foi
  aprovado, senão ele foi reprovado

Esta representação é pouco usada na prática porque o uso da linguagem natural muitas vezes dá oportunidade a más interpretações, ambigüidades e imprecisões.

Por exemplo, a instrução "afrouxar ligeiramente as porcas" no algoritmo da troca de pneus está sujeita a interpretações diferentes por pessoas distintas. Uma instrução mais precisa seria: "'afrouxar a porca, girando-a de 30^o no sentido anti-horário".

2.2.2 - Fluxograma Convencional

É uma representação gráfica de algoritmos onde formas geométricas diferentes implicam ações (instruções, comandos) distintos. Tal propriedade facilita o entendimento das idéias contidas nos algoritmos e justifica sua popularidade.

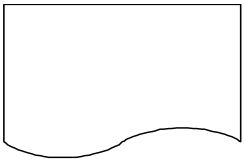
Esta forma é aproximadamente intermediária à descrição narrativa e ao pseudocódigo (subitem seguinte), pois é mais precisa que a primeira e, no entanto, não se preocupa com detalhes de implementação do programa, como o tipo das variáveis usadas (veremos adiante).

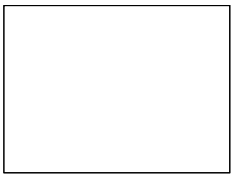
Há vários padrões que definem as formas geométricas das figuras que devem ser usadas para representar cada um dos diversos tipos de instruções; contudo, nenhum deles se sobressai com relação aos demais no que diz respeito à aceitação por parte dos usuários.

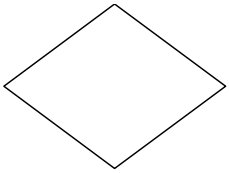
Uma notação simplificada de fluxogramas é a seguinte:


= Início e final do fluxograma


= Operação de entrada de dados

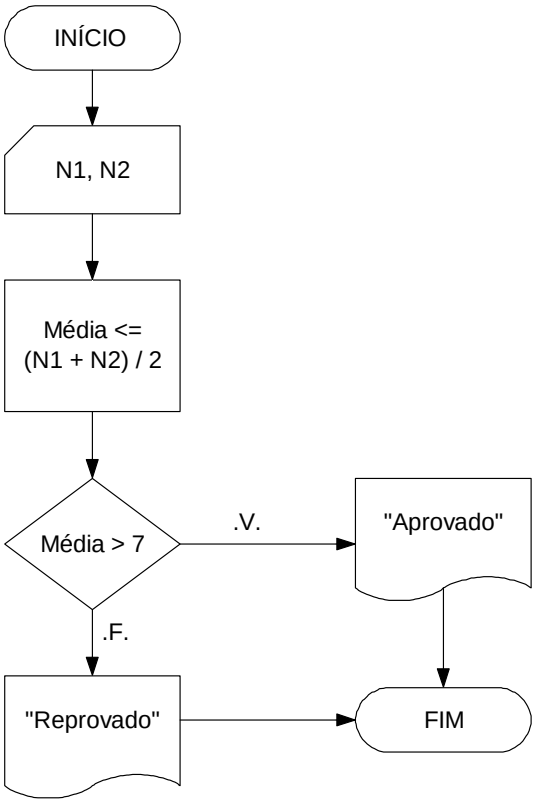

= Operação de saída de dados


= Operação de atribuição e chamada ou retomo de subalgoritmo


= decisão

De modo geral, um fluxograma se resume a um único símbolo inicial, por onde a execução do algoritmo começa, e um ou mais símbolos finais, que são pontos onde a execução do algoritmo se encerra. Partindo do símbolo inicial, há sempre um único caminho orientado a ser seguido, representando a existência de uma única seqüência de execução das instruções. Isto pode ser melhor visualizado pelo fato de que, apesar de vários caminhos poderem convergir para uma mesma figura do diagrama, há sempre um único caminho saindo desta. Exceções a esta regra são os símbolos finais, dos quais não há nenhum fluxo saindo, e os símbolos de decisão, de onde pode haver mais de um caminho de saída (usualmente dois caminhos), representando uma bifurcação no fluxo.

A figura abaixo mostra a representação do algoritmo de cálculo da média de um aluno sob a forma de um fluxograma.



2.2.3 - Pseudocódigo

Esta forma de representação de algoritmos é rica em detalhes, como a definição dos tipos das variáveis usadas no algoritmo e, por assemelhar-se bastante à forma em que os programas são escritos, encontra muita aceitação.

Na verdade, esta representação é suficientemente geral para permitir que a tradução de um algoritmo nela representado para uma linguagem de programação específica seja praticamente direta.

A forma geral da representação de um algoritmo na forma de pseudocódigo é a seguinte:

Bibliotecas

<declaração_de_variáveis> //Variáveis globais
<subalgoritmos>

Nome-do-algoritmo()

Início
    <declaração_de_variáveis> //Variáveis Locais
    <corpo_do_algoritmo>
Fim.

onde:

- **Bibliotecas** são funções escritas pelos fabricantes do compilador e já estão pré-compiladas.

- **<Nome_do_algoritmo>** é um nome simbólico dado ao algoritmo com a finalidade de distingui-lo dos demais.
- **<declaração-de-variáveis>** consiste em uma porção opcional onde são declaradas as variáveis globais usadas no algoritmo principal e, eventualmente, nos subalgoritmos com variáveis locais.
- **<subalgoritmos>** consiste de uma porção opcional do pseudocódigo onde são definidos os subalgoritmos (veremos adiante).
- **Início** e **Fim** são respectivamente as palavras que delimitam o início e o término do conjunto de instruções do corpo do algoritmo.

2.3 - SOLUÇÃO DE PROBLEMAS ATRAVÉS DO COMPUTADOR HIPOTÉTICO

Como exemplo, abaixo temos a representação do algoritmo do cálculo da média de um aluno, na forma de um pseudocódigo. As expressões, operadores e instruções, bem como a forma correta de colocá-las (sintaxe), serão vistas na unidade 4.

```
Média ()
Início
        Real Nl, N2, Média;
        Leia (“%R”, &N1);
        Leia (“%R”, &N2);
        Média ← (N1 + N2) / 2;
        Se (Média > 7) Então
                Escreva ("Aprovado");
        Senão
                Escreva ("Reprovado");
Fim
```

2.4 - ERROS

Quando se fala em erro em um algoritmo ou programa é porque as instruções não estão numa seqüência correta ou ainda, por tratar de informações que não existam ou que existam mas não são do tipo correto.
Os erros são ocorrências comuns que você deverá permanentemente estar preocupado, não só pela sua ocorrência mas, também, pela resolução da mesma.
Os tipos de erros mais comuns que ocorrem num programa são, geralmente, erros de sintaxe e erros de lógica. Analisaremos esses tipos através do exemplo anterior.

```
Média()
    Real Nl, N2, Média;
    Média ← (N1 + N2) / 2;
    Leia (%R, &N1);
    Leia (“%R”, n2);
    Se (Média < 7) Então
        Escreva ("Aprovado");
    Senão
        Escreva ("Reprovado");
Fim
```

Perceba que algumas instruções foram alteradas para simular erros, vamos relacioná-los:

- As linhas 3, 4 e 5 estão invertidas não se pode calcular a média sem antes saber quais são as notas.

- Na linha 4 o primeiro argumento – a *string de controle de formato* –, indicando que o dado deve ser um Real, deve se encontrar **entre** aspas duplas.
- Na linha 5 temos uma instrução onde o programa pede informações que serão armazenadas em variáveis. Perceba que a variável n2 não existe. Este tipo de erro é bastante comum. O correto seria N2, pois a linguagem C é dita como "case sensitive", ou seja, ela diferencia maiúsculas e minúsculas. Além disto o segundo argumento de **Leia** começa com um e-comercial (**&,** ampersand, em inglês) – chamado em C de *operador de endereço*.
- Na linha 6, onde acontece uma comparação lógica, o programa escreverá "aprovado" se a média for menor que 7. Na realidade a instrução deverá ser média > 7, ou seja, se a expressão for verdadeira então o programa escreverá "aprovado", caso contrário (se a média for <= 7 então o programa escreverá "reprovado".

EXERCÍCIOS:

1. O que é algoritmo?

2. Qual a diferença entre algoritmo e programa?

3. Quais são as formas de representação de algoritmos? Descreva-as.

4. Elabore um algoritmo na forma narrativa para descrever os processos desde quando você acorda até o momento de ir trabalhar ou estudar.

5. Elabore um algoritmo (em todas as representações) para somar dois números.

6. Descreva o significado de Erros em um programa ou algoritmo. Exemplifique a ocorrência de um erro do algoritmo anterior.

7. Uma pessoa precisa trocar o pneu furado de um carro. Quais as ações necessárias para realizar essa tarefa e em qual seqüência?

8. Qual a seqüência para se obter a resposta da operação matemática "5 multiplicado por 6 mais 2, dividido por 3" em uma calculadora simples?

9. Qual a seqüência de ações para se abrir uma porta?

10. Complete os termos faltantes da seqüência: 1, 1, 2, 3, 5, 8, 13, ..., ..., ...

11. Criar um algoritmo para colocar um carro em movimento.

12. Imagine que uma pessoa decida ir de táxi a uma reunião de negócios. Monte um algoritmo com a seqüência de ações para que ela chegue ao prédio onde vai ocorrer a reunião.
    a) Entrar no prédio da reunião.
    b) Sair do táxi.
    c) Acenar para que o táxi pare.
    d) Perguntar o preço da corrida.
    e) Informar o destino ao motorista
    f) Esperar o táxi.
    g) Pagar a corrida.
    h) Entrar no táxi

13. Monte um algoritmo com a seqüência de ações para fazer uma vitamina com um mamão, uma banana, uma maçã, um pouco de leite e açúcar.

14. Monte um algoritmo com as ações para encontrar o nome de João Ferreira Neto em uma lista telefônica.

15. Monte um algoritmo com as ações para retirar U$ 100,00 de um caixa automático de banco.

16. Monte um algoritmo com as ações para fazer uma macarronada com molho de tomate (em lata).

UNIDADE 3 - FERRAMENTAS DE PROGRAMAÇÃO

3.1 - EDITOR

Um programa de computador nada mais é do que um grupo de comandos logicamente dispostos para executarem um determinada tarefa. Esses comandos são gravados em um *arquivo-texto* que é processado ao comando do usuário e passa então a executar cada um dos comandos que lá estão gravados.

3.2 - INTERPRETADOR

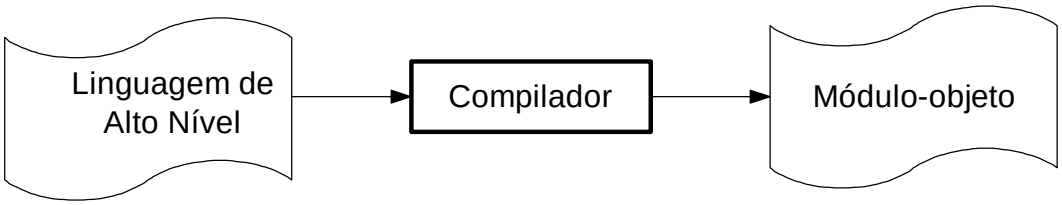
O interpretador é considerado um tradutor que gera código-objeto. A partir de um programa-fonte, escrito em linguagem de alto nível, o interpretador, no momento da execução do programa, traduz cada instrução e a executa em seguida.

A maior desvantagem da utilização de interpretadores é o tempo gasto na tradução das instruções de um programa toda vez que este for executado, já que não existe a geração de um código executável. A vantagem é permitir a implementação de tipos de dados dinâmicos, ou seja, que podem mudar de tipo durante a execução do programa, aumentando, assim, sua flexibilidade.

Algumas linguagens tipicamente interpretadas são o Basic, dbase e APL.

3.3 - COMPILADOR

O compilador é o utilitário responsável por gerar, a partir de um programa escrito em uma linguagem de alto nível, um programa em linguagem de máquina não executável (módulo-objeto).



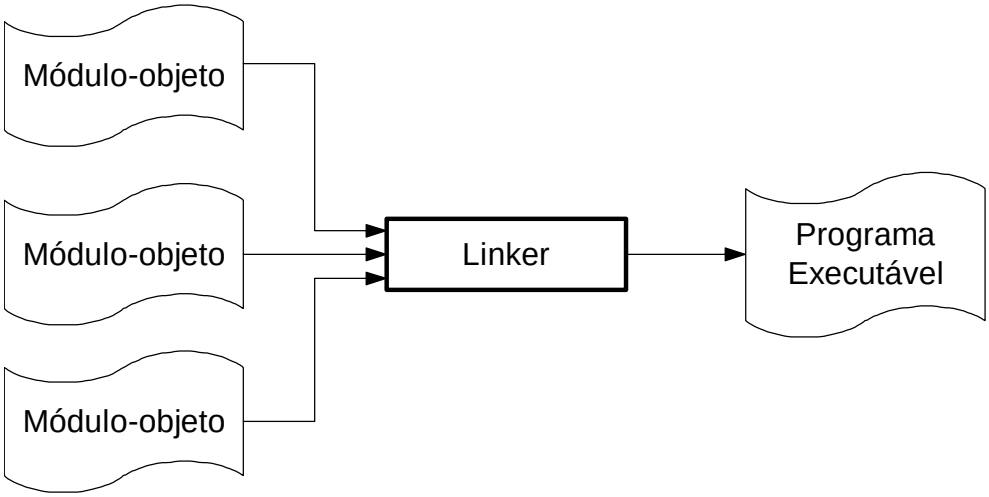
As linguagens de alto nível, corno Pascal, Fortran, Cobol, não têm nenhuma relação direta com a máquina, ficando essa preocupação exclusivamente com o compilador. Os programadores de alto nível têm, apenas, que se preocupar com o desenvolvimento de suas aplicações, não tendo que se envolver com detalhes sobre a arquitetura do processador. Assim, os programas-fonte podem ser transportados entre computadores de diversos fabricantes, desde que existam regras de definição para a linguagem. Isso permite o desenvolvimento de aplicações independentes do equipamento.

O compilador converte um programa escrito em linguagem de alto nível para um programa escrito em linguagem objeto (módulo objeto).

Um compilador é um utilitário que opera de modo integrado aos componentes do sistema de programação disponíveis, sob a supervisão do sistema operacional. Podemos visualizar, então, o compilador como uma interface entre o sistema operacional e o usuário, de maneira que é possível acessar diversos serviços do sistema sem a necessidade da utilização de linguagem de controle ou de outros utilitários.

3.4 - LINK EDIÇÃO

O linker (ligador), também chamado de *linkage editor (editor de ligação)*, é o utilitário responsável por gerar, a partir de um ou mais módulos-objeto, um único programa executável (Figura). Suas funções básicas são resolver todas as referências simbólicas existentes entre os módulos e reservar memória para a execução do programa.



Para resolver todas as referências a símbolos, o linker também pode pesquisar em bibliotecas do sistema ou do próprio usuário. Bibliotecas são arquivos que contêm diversos módulos-objeto e/ou definições de símbolos. Outra função importante do linker é determinar uma região de memória na qual o programa será carregado para ser executado. Esta operação é denominada *relocação.*

Em sistemas operacionais antigos, a relocação era realizada somente uma vez, na etapa de linkedição. Todos os endereços simbólicos do programa são traduzidos para endereços físicos (binding), e o programa executável é gerado, podendo ser carregado a partir de uma posição prefixada na memória (código absoluto). Nesse tipo de relocação, o programa poderá ser carregado, apenas, a partir de uma única posição na memória.

3.5 - DEPURADOR DE PROGRAMA

O desenvolvimento de programas está sujeito a erros de lógica, independente de metodologias utilizadas pelo programador. A depuração é um dos estágios desse desenvolvimento, e a utilização de ferramentas adequadas é essencial para acelerar o processo de correção dos programas.

O depurador (debugger) é o utilitário que permite ao usuário controlar toda a execução de um programa a fim de detectar erros na sua estrutura. Este utilitário oferece ao usuário recursos como:

- acompanhar a execução de um programa instrução por instrução;
- possibilitar a alteração e visualização do conteúdo de variáveis;

- implementar pontos de parada dentro do programa (breakpoint), de forma que, durante a execução, o programa pare nesses pontos;
- especificar que, toda vez que o conteúdo de uma variável for modificado, o programa envie uma mensagem (watchpoint).

3.6 - AMBIENTE INTEGRADO

Um computador, visto, somente, como um gabinete composto de circuitos eletrônicos, cabos e fontes de alimentação (hardware), não tem nenhuma utilidade. É através de programas (software) que o computador consegue armazenar dados em discos, imprimir relatórios, gerar gráficos, realizar cálculos entre outras funções. O hardware é o responsável pela execução das instruções de um programa, com a finalidade de se realizar alguma tarefa.

Uma operação efetuada pelo software pode ser implementada em hardware, enquanto uma instrução executada pelo hardware pode ser simulada via software. Tanto o hardware quanto o software são logicamente equivalentes, se interagindo de uma forma única para o usuário.
O surgimento do sistema operacional, tornou a interação entre o usuário e computador mais simples, confiável e eficiente. A partir desse acontecimento, não existia mais necessidade de o programador se envolver com a complexidade do hardware para poder trabalhar; ou seja, a parte física do computador tornou-se transparente para o usuário.

É através do sistema operacional que futuros programadores (como você) perceberá a aplicação da integridade do sistema computacional. Você poderá desenvolver um programa e interpretá-lo ou ainda, dependendo da linguagem, poderá compilar, linkar e executar o programa.

3.7 - BIBLIOTECAS

Durante o desenvolvimento de programas, suas versões codificadas podem ser mantidas numa biblioteca de declarações-fonte. O ponto principal é compreender como as coisas acontecem. Tendo-se o conhecimento dos conceitos subjacentes, é relativamente fácil entender como criar ou obter acesso às bibliotecas em quase todos os sistemas.

Bibliotecas são um conjunto de sub-rotinas (programas ou funções) que podem ser utilizados em qualquer programa.

O uso de bibliotecas dependem da linguagem. O FORTRAN, por exemplo, suporta algumas sub-rotinas científicas. O programador que quiser usar uma dessas sub-rotinas científicas deve especificar com clareza ao linkage editor onde a biblioteca de sub-rotinas pode ser encontrada.

EXERCÍCIOS:

1. Para que serve um editor?

2. O que é, e como funciona um interpretador?

3. O que é, e como funciona um compilador?

4. O que é um linker e quais suas funções?

5. O que é um depurador de programas e quais são seus recursos ?
6. O que são bibliotecas e qual a sua principal vantagem?

UNIDADE 4 - INTRODUÇÃO À LINGUAGEM DE PROGRAMAÇÃO

4.1 - TIPOS DE DADOS PRIMITIVOS

Todo o trabalho realizado por um computador é baseado na manipulação das informações contidas em sua memória. *Grosso* modo, estas informações poderão ser classificadas em dois tipos:

- As **instruções,** que comandam o funcionamento da máquina e determinam a maneira como devem ser tratados os dados. As instruções são específicas para cada modelo de computador, pois são função do tipo particular de processador utilizado em sua implementação.

- Os **dados** propriamente ditos, que correspondem à porção das informações a serem processadas pelo computador.

O objetivo deste capítulo é justamente o de classificar os dados de acordo com o tipo de informação contida neles. A classificação apresentada não se aplica a nenhuma linguagem de programação específica; pelo contrário, ela sintetiza os padrões utilizados na maioria das linguagens.

Tornando ao aspecto computacional, os dados numéricos representáveis num computador são divididos em apenas duas classes: os **inteiros** e os **reais.**

4.1.1 - Dados Numéricos Inteiros

Os números **inteiros** são aqueles que não possuem componentes decimais ou fracionários, podendo ser positivos ou negativos.

Como exemplo de números **inteiros** temos:

24 - número inteiro positivo
0 - número inteiro
-12- número inteiro negativo

4.1.2 - Dados Numéricos Reais

Os dados de tipo **real** são aqueles que podem possuir componentes decimais ou fracionários, e podem também ser positivos ou negativos.

Exemplos de dados do tipo **real:**

24.01 - número real positivo com duas casas decimais
144. - número real positivo com zero casas decimais
-l3.3 - número real negativo com uma casa decimal
0.0 - número real com uma casa decimal
0. - número real com zero casas decimais

Observe que há uma diferença entre '0', que é um dado do tipo inteiro, e '0.' (ou '0.0') que é um dado do tipo real. Portanto, a simples existência do ponto decimal serve para diferenciar um dado numérico do tipo **inteiro** de um do tipo **real.**

4.1.3 - Dados Literais

O tipo de dados **literal** é constituído por uma seqüência de caracteres contendo letras, dígitos e/ou símbolos especiais. Este tipo de dados é também muitas vezes chamado de **alfanumérico**, **cadeia** (ou **cordão**) **de caracteres**, ou ainda, do inglês, **STRING**.

Usualmente, os dados literais são representados nos algoritmos pela coleção de caracteres, delimitada em seu início e término com o caractere aspas (“).

Diz-se que o dado do tipo literal possui um comprimento dado pelo número de caracteres nele contido.

Exemplos de dados do tipo literal:

“QUAL ?”	-literal de comprimento 6
‘ ’	-literal de comprimento l
“quaL ?!$”	-literal de comprimento 8
“AbCdefGHi”	-literal de comprimento 9
“l-2+3=”	-literal de comprimento 6
‘0’	-literal de comprimento l

Note que, por exemplo, “1.2” representa um dado do tipo **literal** de comprimento 3, diferindo de 1.2 que é um dado do tipo **real.**

4.1.4 - Dados Lógicos

Na linguagem C não existem dados lógicos, pois nesta linguagem, qualquer valor diferente de 0 é considerado verdadeiro.
Portanto, este tópico consta apenas a caráter de informação, pois os dados que aqui se encontram (neste tópico) , baseiam-se na linguagem Pascal.

A existência deste tipo de dado é, de certo modo, um reflexo da maneira como os computadores funcionam. Muitas vezes, estes tipos de dados são chamados de **booleanos,** devido à significativa contribuição de BOOLE à área da lógica matemática.

O tipo de dados **lógico** é usado para representar dois únicos valores lógicos possíveis: **verdadeiro** e **falso**. É comum encontrar-se em outras referências outros tipos de pares de valores lógicos como **sim/não, 1/0, true/false.**

Nos algoritmos apresentados neste texto os valores lógicos serão delimitados pelo caractere ponto(.).

Exemplo:
.V. - valor lógico verdadeiro
.F. - valor lógico falso

Exercício Proposto

1. Classifique os dados especificados abaixo de acordo com seu tipo, assinalando com **I** os dados do tipo inteiro, com **R** os reais, com **L** os literais, com **B** os lógicos (booleanos), e com **N** aqueles para os quais não é possível definir a *priori* um tipo de dado.

( ) 0	( ) -0.0001	( ) -0.0
( ) 1	( ) +0.05	( ) .V.
( ) 0.0	( ) +3257	( ) V
( ) 0.	( ) 'a'	( ) "abc"
( ) -1	( ) "+3257"	( ) F
( ) -32	( ) "+3257."	( ) 22
( ) "+36"	( ) "-0.0"	( ) 'V'
( ) +32.	( ) ".F."	( ) .F.

2. Identifique de que tipo são as constantes a seguir:
   a) 435867
   b) "RODOLFO"
   c) "8725"
   d) 0.213
   e) .F.
   f) ".V."
   g) "Limão"
   h) "23/12/03"
   i) 0.5
   j) "304958"
   k) 123456

3. Indique o(s) tipo(s) de constante(s) que pode(m) ser utilizado(s) para representar:
   a) o nome de um país;
   b) o Registro Geral (RG) de identificação de uma pessoa;
   c) o CNPJ de uma empresa;
   d) se uma pessoa é ou não maior de idade (tem mais de 18 anos);
   e) a idade de uma pessoa;
   f) o endereço de uma pessoa.

4.1.5 - Conceito e Utilidade de Variáveis

Os diversos tipos de dados são armazenadas nas memórias dos computadores. Para acessar individualmente cada uma destas informações, em princípio, seria necessário saber o tipo de dado desta informação e a posição inicial deste conjunto de bytes na memória.

Percebe-se que esta sistemática de acesso a informações na memória é bastante ilegível e difícil de se trabalhar. Para contornar esta situação criou-se o conceito de **variável,** que é uma entidade destinada a guardar uma informação.

Basicamente, uma variável possui três atributos: um **nome,** um **tipo de dado** associado à mesma e a **informação** por ela guardada.

Toda variável possui um **nome** que tem a função de diferenciá-la das demais. Cada linguagem de programação estabelece suas próprias regras de formação de nomes de variáveis.

Adotaremos neste texto as seguintes regras:
- um nome de variável deve necessariamente começar com uma letra;
- um nome de variável não deve conter nenhum símbolo especial, exceto o underline(_).

Exemplos:

SALÁRIO	-correto
1ANO	-errado (não começou com uma letra)
ANO1	-correto
A CASA	-errado (contém o caractere branco)
SAL/HORA	-errado (contém o caractere "/")
SAL_HORA	-correto
_DESCONTO	-errado (não começou com uma letra)

Obviamente é interessante adotar nomes de variáveis relacionados às funções que serão exercidas pelas mesmas dentro de um programa. Exemplificando, para guardar o salário de um funcionário de uma empresa num programa de processamento de, folha de pagamento, seria interessante utilizar uma variável chamada **SALARIO.**

Outro atributo característico de uma variável é o **tipo de dado** que ela pode armazenar. Este atributo define a natureza das informações contidas na variável. Por último, há o atributo **informação,** que nada mais é do que a informação útil contida na variável.
Uma vez definidos, os atributos **nome** e **tipo de dado** de uma variável **não** podem ser alterados e assim permanecem durante toda a sua existência, desde que o programa que a utiliza não seja modificado. Por outro lado, o atributo **informação** está constantemente sujeito a mudanças, de acordo com o fluxo de execução do programa. Por exemplo, se definirmos que uma determinada variável é chamada **SALÁRIO** e é destinada a guardar números reais, é possível que seu conteúdo seja, num dado instante, igual a 1.500,00 e posteriormente modificado para 3.152,19, de acordo com o programa executado.

Em resumo, o conceito de variável foi criado para facilitar a vida dos programadores, permitindo acessar informações na memória dos computadores por meio de um nome, em vez do endereço de uma célula de memória.

4.1.6 - Definição de Variáveis em Algoritmos

Todas as variáveis utilizadas em algoritmos devem ser definidas antes de serem utilizadas. Isto se faz necessário para permitir que o compilador reserve um espaço na memória para as mesmas.

Algumas linguagens de programação (como **CLIPPER** e **FORTRAN)** dispensam esta definição, uma vez que o espaço na memória é reservado à medida que novas variáveis são encontradas no decorrer do programa.

Nos algoritmos apresentados neste texto será adotada a seguinte convenção: todas as variáveis utilizadas em algoritmos serão definidas no inicio do mesmo, por meio de um comando de uma das formas seguintes:

<tipo_da_variável> <nome_de_variável>

- numa mesma linha poderão ser definidas uma ou mais variáveis do mesmo tipo; Para tal, deve-se separar os nomes das mesmas por vírgulas;
- variáveis de tipos diferentes devem ser declaradas em linhas diferentes;
- somente os 31 primeiros caracteres do nome da variável são reconhecidos.

A forma de utilização deste comando ficará mais clara quando da utilização da representação de algoritmos em linguagem estruturada (pseudocódigo).
Esta convenção é válida para a representação de algoritmos na forma de pseudocódigo. Em termos de fluxograma, não é usual adotar-se qualquer forma de definição de variáveis.

Exemplo de definição de variáveis:

Literal	NOME[10];
Inteiro	IDADE;
Real	SALÁRIO, DESCONTO;

No exemplo acima foram declaradas quatro variáveis:

- a variável **NOME**, capaz de armazenar dados literais de comprimento 10 (dez caracteres);
- a variável **IDADE**, capaz de armazenar um número inteiro;
- as variáveis **SALARIO** e **DESCONTO**, capazes de armazenar um número real;

Exercícios Propostos

1. Na lista seguinte, assinale com **V** os nomes de variáveis válidos e com **I** os inválidos.

( ) abc	( ) 3abc	( ) a
( ) 123a	( ) a?B	( ) acdl
( ) _	( ) Aa	( ) 1
( ) Al23	( ) _1	( ) AOl23
( ) al23	( ) _al23	( ) b3l2
( ) AB CDE	( ) etc...	( ) guarda-chuva

2. Declare as variáveis para o algoritmo de cadastramento de alunos, cujos dados são: nome, sexo, endereço, cidade, estado, CEP, telefone, data de nascimento, RG, nome do pai, nome da mãe e grau de escolaridade. Utilize pseudocódigo para representar as declarações.

3. Verifique se os identificadores a seguir são válidos. Se não forem, explique por quê.
   a. NOME-DO-ALUNO
   b. $
   c. DINHEIRO
   d. DATA/DE/NASCIMENTO
   e. NOME2
   f. 2APESSOA
   g. MAIOR_DE_IDADE?
   h. NOME_DA_PESSOA
   i. END
   j. %JUROS
   k. M239083

4. Declare as variáveis para os algoritmos dos cadastros a seguir.
   a. Passagens aéreas
   b. Fitas de vídeo de uma locadora
   c. Livros de uma biblioteca
   d. Carros de uma concessionária
   e. CDs de uma coleção de música
   f. Disciplinas de uma escola
   g. Clientes de uma loja
   h. Roupas de uma loja

4.1.7 - Expressões

O conceito de **expressão** em termos computacionais está intimamente ligado ao conceito de expressão (ou fórmula) matemática, onde um conjunto de variáveis e constantes numéricas relacionam-se por meio de operadores aritméticos compondo urna fórmula que, uma vez avaliada, resulta num valor.

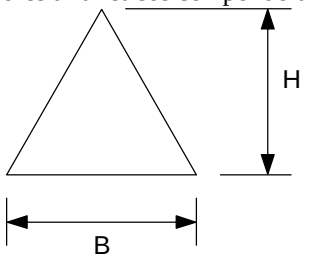


Figura: Triângulo de base **(B)** e altura **(H)**.

Por exemplo, a fórmula de cálculo da área do triângulo da Figura é dada por:

ÁREA = 0.5 x B x H

Esta fórmula utiliza três variáveis: **B** e **H,** que contêm as dimensões do triângulo, e **ÁREA**, onde é guardado o valor calculado (resultado da avaliação da expressão). Há, também, uma constante **(0.5)** e o operador de multiplicação **(x),** que aparece duas vezes na expressão.

O conceito de **expressão** aplicado à computação assume uma conotação mais ampla: uma **expressão** é uma combinação de variáveis, constantes e operadores, e que, uma vez avaliada, resulta num valor.

Expressão é uma combinação de variáveis, constantes e operadores.

4.2 - OPERADORES ARITMÉTICOS

Operadores são elementos funcionais que atuam sobre **operandos** e produzem um determinado resultado. Por exemplo, a expressão **3 + 2** relaciona dois operandos (os números 3 e 2) por meio do operador (+) que representa a operação de adição.

De acordo com o número de operandos sobre os quais os operadores atuam, os últimos podem ser classificados em:

- **Binários,** quando atuam sobre dois operandos. Ex.: os operadores das operações aritméticas básicas (soma, subtração, multiplicação e divisão);
- **Unários,** quando atuam sobre um único operando. Ex.: o sinal de na frente de um número, cuja função é inverter seu sinal.

Outra classificação dos operadores é feita considerando-se o tipo de dado de seus operandos e do valor resultante de sua avaliação. Segundo esta classificação, os operadores dividem-se em **aritméticos, lógicos** e **relacionais.** Esta divisão está diretamente relacionada com o tipo de expressão onde aparecem os operadores.

Um caso especial é o dos operadores **relacionais,** que permitem comparar pares de operandos de tipos de dados iguais, resultando sempre num valor lógico.

Mais adiante serão apresentados formalmente os operadores dos diversos tipos acima relacionados.

4.2.1 - OPERADORES ARITMÉTICOS

Os operadores aritméticos relacionados às operações aritméticas básicas estão resumidos na Tabela abaixo.

Operador	Tipo	Operação	Prioridade
+	Binário	Adição	3
-	Binário	Subtração	3
*	Binário	Multiplicação	2
/	Binário	Divisão	2
+	Unário	Manut de sinal	1
-	Unário	Inversão de sinal	1

A prioridade entre operadores define a ordem em que os mesmos devem ser avaliados dentro de uma mesma expressão. Este assunto será tratado com maior profundidade numa seção posterior.

O caractere (*) é adotado na maioria das linguagens de programação para representar a operação de multiplicação, ao invés do caractere **(x)**, devido à possibilidade da ocorrência do mesmo no nome de variáveis. Pela mesma razão, o símbolo (**) é adotado para representar a operação de exponenciação. Algumas linguagens de programação adotam o símbolo (^) (circunflexo) para esta finalidade, mas isto é pouco freqüente.

As variáveis usadas em expressões aritméticas podem somente ser do tipo inteiro ou real. Se ao menos uma das variáveis da expressão aritmética for do tipo real, então o valor resultante da avaliação da expressão é necessariamente do tipo real.

Nos exemplos seguintes, assumiremos que:
- **A, B** e **C** são variáveis do tipo inteiro;
- **X, Y** e **Z** são variáveis do tipo real.

Exemplos:

1) **A+B*C** - expressão de resultado inteiro
2) **A+B+Y** - expressão de resultado real
3) **A/B** - expressão de resultado inteiro
4) **X/Y** - expressão de resultado real

O leitor deve estranhar o fato de no terceiro exemplo ser apresentada uma expressão onde se dividem dois números inteiros e se obtém como resultado um valor inteiro. Obviamente, o resultado da divisão entre dois números inteiros não é necessariamente inteiro. Na verdade, a operação representada no terceiro item é a **divisão inteira,** onde o resto da divisão é desprezado.

4.3 - OPERADORES LÓGICOS

Expressões lógicas são aquelas que utilizam **operadores lógicos** e consequentemente resulta em um valor lógico **(.V.** ou **.F.)**.

Os operadores lógicos e suas relações de precedência são mostrados na Tabela abaixo.

Operador	Tipo	Operação	Prioridade
.OU.	Binário	Disjunção	3
.E.	Binário	Conjunção	2
.NÃO.	Unário	Negação	1

Para exemplificar o uso de operadores lógicos, a Tabela abaixo apresenta duas variáveis lógicas **A** e **B.** Uma vez que cada variável lógica possui somente dois valores possíveis, então há exatamente quatro combinações para estes valores, razão pela qual a tabela tem quatro linhas. As diversas colunas contêm os resultados das operações lógicas sobre as combinações possíveis dos valores das variáveis **A** e **B.**

A	B	A .ou. B	A .e. B
.V.	.V.	.V.	.V.
.V.	.F.	.V.	.F.
.F.	.V.	.V.	.F.
.F.	.F.	.F.	.F.

4.4 - OPERADORES RELACIONAIS

Estes operadores são somente usados quando se deseja efetuar comparações. Comparações só podem ser feitas entre objetos de mesma natureza, isto é, variáveis do mesmo tipo de dado. O resultado de uma comparação é sempre um valor lógico.

O uso de operadores relacionais possibilita o aparecimento em expressões lógicas de variáveis de outros tipos de dados que não o lógico. Abaixo é mostrada uma tabela contendo os operadores relacionais.

Operador	Comparação
==	igual
!=	diferente

<	menor
< =	menor ou igual
>	maior
> =	maior ou igual
!	Não

Exemplo: sejam **A** e **B** variáveis lógicas, **X** e **Y** variáveis reais, e **R, S** e **T** variáveis literais, com os respectivos valores:

A=.V., B =.F.,
X = 2.5, Y = 5.0,
R = "JOSÉ", S = "JOÃO" e T ="JOÃOZINHO"

A tabela seguinte contém algumas expressões lógicas contendo estas variáveis:

Expressão	Resultado
A. OU. B	.V.
A .E. B	.F.
.NÃO. A	.F.
X=Y	.F.
X = (Y/2)	.V.
R=S	.F.
S=T	.F.
R!=S	.V.
R>S	.F.
S>T	.F.
((A.OU.B).OU.(X>Y).OU.(S>T))	.V.

Algumas Normas para utilização de operadores nas expressões

Expressões que apresentam apenas um único operador podem ser avaliadas diretamente. No entanto, à medida que as mesmas vão-se tornando mais complexas com o aparecimento de mais de um operando numa mesma expres*são*, é necessária a avaliação da mesma passo a passo, tomando um operador por vez. A seqüência destes passos é definida de acordo com o formato geral da expressão, considerando-se a prioridade (precedência) de avaliação de seus operadores e a existência ou não de parênteses na mesma.

As seguintes regras são essenciais para a correta avaliação de expressões:

1. Deve-se observar a prioridade dos operadores, conforme mostrado nas Tabelas: operadores de menor prioridade devem ser avaliados primeiro. Se houver empate com relação à precedência, então a avaliação se faz considerando-se a expressão da esquerda para a direita.
2. Os parênteses usados em expressões têm o poder de "roubar" a prioridade dos demais operadores, forçando a avaliação da subexpressão em seu interior.
3. Entre os quatro grupos de operadores existentes, a saber, aritmético, lógico, literal e relacional, há uma certa prioridade de avaliação: os aritméticos e literais devem ser avaliados primeiro; a seguir, são avaliadas as subexpressões com operadores relacionais e, por último, os operadores lógicos são avaliados.

Expressões Literais

Expressões **literais** são aquelas cujo resultado da avaliação é um valor literal. Este tipo de expressão é bem menos freqüente que os anteriores. Os tipos de operadores existentes variam de uma linguagem de programação para outra, não havendo uma padronização.

Para que o assunto não passe em branco, considere-se como exemplo a operação de concatenação de **strings**: toma-se duas **strings** e acrescenta-se (concatena-se) a segunda delas ao final da primeira. Em algumas linguagens esta operação é representada Pelo símbolo (operador) (+). Por exemplo, a concatenação das **strings "REFRIGERA"** e **"DOR"** é representada por **"REFRIGERA" + "DOR"** e o resultado de sua avaliação é **"REFRIGERADOR"**.

Exemplos: suponha que **X, Y** e **Z** são variáveis reais, e **A** e **B** variáveis lógicas, com os respectivos valores:

X = 2.0, **Y** = 3.0, **Z** = 0.5, **A** = .V. e **B** = .F.

Considere as seguintes expressões e indique o resultado de suas avaliações:

1) X * Y - Z =
2) X * (Y - Z) =
3) X + Y * Z =
4) X + (Y * Z) =
5) (X + Y) * Z =
6) B .E. A .OU. X != Y/2 =
7) B .E. (A .OU. X != Y/2) =

<u>Exercícios Propostos</u>

1. Dada a declaração de variáveis

   Inteiro A, B, C;
   Real X, Y, Z;
   Literal NOME[20], RUA[20];

   Classifique as expressões seguintes de acordo com o tipo de dado do resultado de sua avaliação, em **I** (inteiro), **R** (real), **L** (literal) ou **N** (quando não for possível defini-lo):

( )A + B + C		( )A > B	
( )A + B + Z		( )A > Y	
( )NOME + RUA		( )X + Z / A	
( )X + Y / Z		( )A + X / Z	
( )A + Z / A		( )A + B / C	

2. Sendo: A = B = C = .V. e D = .F.
   a) (A E B)
   b) (A E B) OU (C)
   c) (A E B) E B
   d) (A E B) E (B OU C)
   e) (A E B) E (B OU D)
   f) (A E B) E ((A OU B) E (D OU A) E (D E A))

3. O mesmo que o nº 2 sendo, A = B = .F. e C = D = .V.
   a) b) c) d) e) f)

4. O mesmo que o nº 2 sendo, A = D = .F. e B = C = .V.
   a) b) c) d) e) f)

5. Sendo A = B = C = .V. e D = .F.
   a) !(A E B)
   b) (A E B) OU (!C)
   c) !(A OU D) E (B)
   d) (A OU B) E (!D)
   e) (A OU B) E (!(B E D))
   f) ((A E B) E C) E (!(A OU B) E (!D OU A) E (D E !A))

6. O mesmo que o nº 5 sendo, A = B = .F. e C = D = .V.
   a) b) c) d) e) f)

7. O mesmo que o nº 5 sendo, A = D = .F. e C = B = .V.
   a) b) c) d) e) f)

8. Sendo A = 20, B = 30, C = .V. e D = .F.
   a) (A = = B) OU (C E D)
   b) (A = = B) E (C E D)
   c) (!(A = = B)) OU (C E D)
   d) (!(A = = B)) OU (!C E D)
   e) (A > =B)
   f) (A = = B)
   g) (A < B)
   h) (A < = B) E (C E D)
   i) (A ! = B)

UNIDADE 5 - ESTRUTURAS DE CONTROLE

5.1 - ESTRUTURAS SEQÜÊNCIAIS

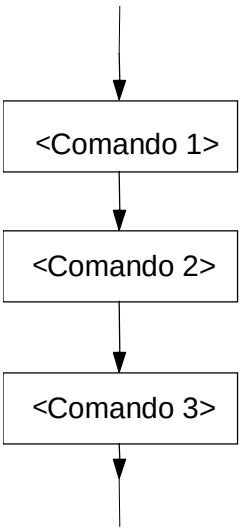
As estruturas seqüências são todas as instruções de um algoritmo ou programa para realizar determinadas tarefa. Na realidade, uma estrutura seqüencial é um conjunto de instruções ordenadas logicamente.

Na estrutura seqüencial os comandos de um algoritmo são executados numa seqüência pré-estabelecida. Cada comando é executado somente após o término do comando anterior.

Em termos de fluxogramas, a estrutura seqüencial é caracterizada por um único fluxo de execução (um único caminho orientado) no diagrama. Em pseudocódigos, a estrutura seqüencial caracteriza-se por um conjunto de comandos dispostos ordenadamente. Como exemplos de aplicação desta estrutura de controle tem-se os algoritmos do capítulo anterior, onde não há estruturas de decisão ou de repetição.

Uma estrutura seqüencial é aquela em que os comandos vão sendo executados numa seqüência pré-estabelecida, um após o outro.

A Figura abaixo exemplifica um trecho seqüencial de um algoritmo.



(a)

(b) <Comando 1>;
<Comando 2>;
<Comando 3>;

Figura: Trecho seqüencial de um algoritmo:
(a) fluxograma;
(b) pseudoc6digo.

Para aprimorar nossos conhecimentos afim de montarmos algoritmos estruturados logicamente, aprofundaremos nos conceitos de instruções primitivas e suas derivadas, ou seja, sabermos o que significa e como se faz cada instrução. Tais conceitos serão de fundamental importância para elaboração de nossos futuros programas.

5.1.1 - Instruções Primitivas

Como o próprio nome diz, ou **instruções primitivas** são os comandos básicos que efetuam tarefas essenciais para a operação dos computadores, como entrada e saída de dados (comunicação com o usuário e com os dispositivos periféricos), e movimentação dos mesmos na memória. Estes tipos de instrução estão presentes na absoluta maioria das linguagens de programação. De fato, um programa que não utiliza nenhuma instrução primitiva - como as que serão definidas neste capítulo - é incapaz de se comunicar com o mundo exterior e, portanto, não tem utilidade alguma.

Antes de passar à descrição das instruções primitivas, é necessária a definição de alguns termos que serão utilizados mais à frente. Além dos dispositivos de entrada e de saída (visto no capítulo 1) temos ainda:

- **sintaxe** é a forma como os comandos devem ser escritos, a fim de que possam ser entendidos pelo tradutor de programas. A violação das regras sintáticas é considerada um erro sujeito à pena do não-reconhecimento do comando por parte do tradutor;
- **semântica** é o significado, ou seja, o conjunto de ações que serão exercidas pelo computador durante a execução do referido comando.

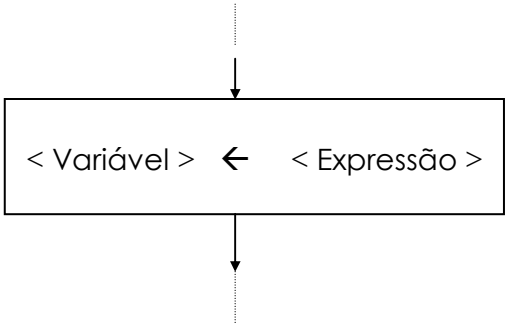
Daqui em diante, todos os comandos novos serão apresentados por meio de sua sintaxe e sua semântica, isto é, a forma como devem ser escritos e a(s) ação(ões) que executam.

5.1.2 - Instrução Primitiva de Atribuição

A instrução primitiva de atribuição, ou simplesmente atribuição, é a principal maneira de se armazenar uma informação numa variável.

Sua sintaxe é: <nome_de_variável> ← <expressão>

No fluxograma, os comandos de atribuição são representados como na Figura.



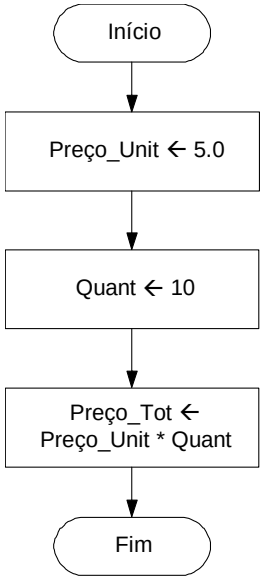
O modo de funcionamento (semântica) de uma atribuição consiste :

1. na avaliação da expressão, e
2. no armazenamento do valor resultante na posição de memória correspondente à variável que aparece à esquerda do comando.

A expressão presente no comando de atribuição pode ser de qualquer tipo de variável.

Uma implicação bastante séria, para qual a atenção deve ser dirigida, é a necessidade da compatibilidade entre o tipo de dado resultante da avaliação da expressão e o tipo de dado da variável, no sentido em que esta deve ser capaz de armazenar o resultado da expressão. Mais explicitamente, se uma expressão resulta num valor lógico, então a variável deve ser também do tipo lógico. O mesmo acontece no caso de dados literais e reais. Uma exceção é o caso em que a variável é do tipo real e a expressão resulta num valor inteiro. Nesta situação, o resultado (dado do tipo inteiro) é convertido para o tipo real e posteriormente armazenado na **variável**.

A Figura abaixo mostra um exemplo de algoritmo onde algumas atribuições são feitas: os valores 5.0 e 10 são atribuídos às variáveis **PREÇO_UNIT** e **QUANT,** respectivamente; posteriormente, o resultado do produto entre as duas anteriores é armazenado em **PREÇO_TOT**.



(a)

(b)

```
EXEMPLO1()
{

        Real     PREÇO_UNIT, PREÇO_TOT;
        Inteiro  QUANT;

        PREÇO_UNIT ← 5.0;
        QUANT ← 10;
        PREÇO_TOT ← PREÇO-UNIT * QUANT;
}
```

Figura : Exemplo de aplicação de comandos de atribuição:
(a) fluxograma;
(b) pseudocódigo.

5.1.3 - Instrução Primitiva de Saída de Dados

O algoritmo da Figura não é prático, pois, apesar do computador ter calculado um determinado valor ou armazenado o mesmo na variável **PREÇO_TOT,** este valor ficou retido na memória do computador e não foi mostrado ao **usuário**, o maior interessado no mesmo.

> ***As instruções primitivas de saída de dados são o meio pelo qual informações contidas na memória dos computadores são colocadas nos dispositivos de saída, para que o usuário possa apreciá-las.***

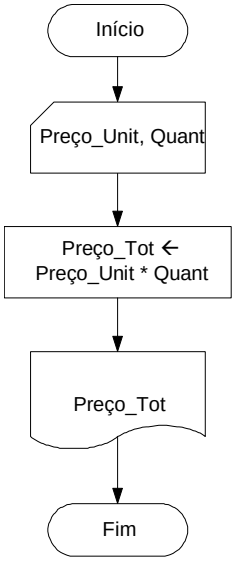
Há duas sintaxes possíveis para esta instrução:

Escreva ("%<tipo_variável>", <variável>);
ou
Escreva ("%<tipo_variável_1> %<tipo_variável_2>", <variável_1>, <variável_2>); → as variáveis precisam estar na ordem em que "aparecerem" no comando de saída.
ou
Escreva ("<literal>");

Repare que a representação no fluxograma dispensa o uso da palavra reservada **Escreva**, uma vez que a mesma já está embutida na forma geométrica da figura.

A semântica da instrução primitiva de saída de dados é muito simples: os argumentos do comando são enviados para o dispositivo de saída. No caso de uma lista de variáveis, o conteúdo de cada uma delas é pesquisado na posição de memória correspondente à variável e depois enviado para o dispositivo de saída. No caso de argumentos do tipo **string,** estes são enviados diretamente ao referido dispositivo.

Há, ainda, a possibilidade de se misturar nomes de variáveis com literais na lista de um mesmo comando. O efeito obtido é bastante útil e interessante: a lista é lida da esquerda para a direita e cada elemento da mesma é tratado separadamente; se um nome de variável for encontrado, então a informação da mesma é pega da memória e colocada no dispositivo de saída; no caso de um literal, o mesmo é escrito diretamente no dispositivo de saída.



(a)

(b)

```
EXEMPLO2()
{
        Real      PREÇO_UNIT, PREÇO_TOT;
        Inteiro   QUANT;
        Leia ("%R", &PREÇO_UNIT);
        Leia ("%I", &QUANT);
        PREÇO_TOT ← PREÇO-UNIT * QUANT;
        Escreva ("O preço total é %.2R", PREÇO_TOT);
}
```

Figura : Exemplo de aplicação de comandos de entrada, atribuição e saída de dados:
(a) fluxograma;
(b) pseudocódigo.

5.1.3.1 – Constantes de Barra Invertida

O C utiliza, para facilitar a tarefa de programar, vários códigos chamados códigos de barra invertida. Estes são caracteres que podem ser usados como qualquer outro. A lista completa dos códigos de barra invertida é dada a seguir.

Código	Significado
\b	Retrocesso ("back")
\f	Alimentação de formulário ("form feed")
\n	Nova linha ("new line")
\r	Retorno de carro ("carriage return")
\t	Tabulação horizontal ("tab")
\"	Aspas
\'	Apóstrofo
\0	Nulo (0 em decimal)
\\	Barra Invertida
\v	Tabulação Vertical
\a	Sinal sonoro ("beep")
\N	Constante octal (N é o valor da constante)
\xN	Constante hexadecimal (N é o valor da constante)

5.1.4 - Instrução Primitiva de Entrada de Dados

O algoritmo da figura acima ainda carece de uma melhoria essencial. Toda vez que ele é executado, o mesmo valor é calculado, já que os valores das variáveis **PREÇO_UNIT** e **QUANT** permanecem inalterados. Seria interessante que estes valores pudessem ser fornecidos ao computador pelo usuário do programa toda vez que o programa fosse executado, para que o usuário tivesse um maior controle sobre o valor calculado. A **instrução primitiva de entrada de dados** foi criada para suprir esta necessidade.

Sua sintaxe é: **Leia** ("%<tipo_variavel>", &<variável>);

A função **Leia** recebe a entrada do dispositivo padrão, que normalmente é o teclado. Esta função tem dois argumentos:

- O primeiro argumento, a *string de controle de formato*, indica o tipo de dado que deve ser fornecido pelo usuário. Nesse contexto, o % é considerado por **Leia** (e por **Escreva**) um caractere de escape (como o \) e a combinação %<tipo_variável> é uma seqüência de escape.
- O segundo argumento de **Leia** começa com um e-comercial (**&**, ampersand, em inglês) – chamado em C de *operador de endereço* – seguido do nome da variável. O e-comercial, quando combinado com o nome da variável, diz ao **Leia** o local na memória onde a variável está armazenada. O computador então armazena o valor informado pelo usuário naquele local.

Da mesma forma que **Escreva**, daqui em diante **Leia** será tratada como uma palavra-reservada e não mais poderá ser usada como nome de **variável** em algoritmos.

A figura anterior mostra como uma instrução de entrada de dados é representada em fluxogramas. Esta representação dispensa o uso da palavra-reservada **Leia,** pelo fato da mesma já estar de certo modo embutida na forma geométrica da figura.

A semântica da instrução de entrada (ou leitura) de dados é, de certa forma, inversa à da instrução de escrita: os dados são fornecidos ao computador por meio de um dispositivo de entrada e armazenados nas posições de memória das variáveis cujos nomes aparecem na **lista_de_variáveis**.

Obs.: Perceba que após os comandos tem-se um **";"** (ponto e vírgula) significando o fim do comando. Desta forma, pode-se escrever mais de um comando na mesma linha.
Ex.: printf("Entre com o valor: "); scanf(X);

Exercício Resolvido

1. Escreva um algoritmo (fluxograma e pseudocódigo) para calcular a média entre dois números quaisquer.

Solução:A idéia principal do algoritmo está centrada na expressão matemática utilizada no cálculo da média **(M)** entre dois números, **N1** e **N2**, dada por:

M = (N1 + N2) / 2

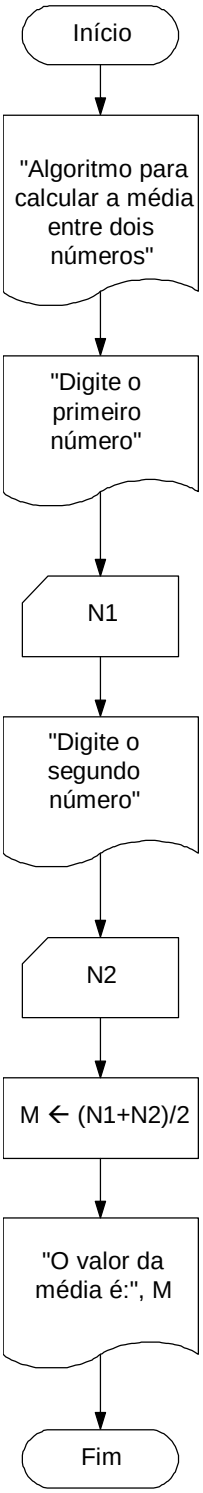
Para que o valor de **M** possa ser calculado pelo algoritmo, é necessário que os valores de **N1** e **N2** tenham sido fornecidos ao mesmo com antecedência. Portanto, a primeira etapa do algoritmo consiste da obtenção (leitura) dos valores de **N1** e **N2** e armazenamento dos mesmos em posições distintas de memória (variáveis).

Na seqüência, o valor da média deve ser calculado por meio de uma expressão apropriada e atribuído a uma terceira variável **(M)**.

Por fim, deve-se relatar ao usuário o valor calculado por meio de uma instrução primitiva de saída de dados.

O fluxograma do algoritmo descrito é mostrado a seguir. Note que ele está enriquecido com instruções para informar sua finalidade, os dados que devem ser fornecidos ao usuário e o significado do valor calculado.

A transformação do fluxograma em pseudocódigo exige a disponibilidade de algumas informações adicionais concernentes ao tipo das variáveis utilizadas. Como o algoritmo opera apenas com dados numéricos, certamente as variáveis utilizadas serão do tipo inteiro ou real. Como se deseja calcular a média entre dois números quaisquer, então as variáveis **N1** e **N2** devem ser capazes de armazenar números com ou sem parte fracionária e, portanto, é necessário que estas sejam do tipo real. Como o valor médio entre dois números reais é um número que pode ou não ter parte fracionária, então a variável **M** também deve ser do tipo real.



De posse dessa informação, pode-se escrever o pseudocódigo do algoritmo em questão, a partir de seu fluxograma.

```
Media()
{
        Real N1, N2, M;
        Escreva ("Algoritmo para calcular a média entre dois números");
        Escreva ("Digite o primeiro número: ");
        Leia ("%R", &N1);
        Escreva ("Digite o segundo número: ");
        Leia ("%R", &N2);
        M ← (N1 +  N2) / 2;
        Escreva ("O valor da média é: %R", M);
}
```

Exercícios Propostos

1. Escreva um algoritmo para calcular o valor de **y** como função de **x,** segundo a função **y(x) = 3x + 2,** num domínio real.
2. Escreva um algoritmo para calcular o consumo médio de um automóvel (medido em **Km/l**), dado que são conhecidos a distância total percorrida e o volume de combustível consumido para percorrê-la (medido em litros).
3. Para as instruções propostas a seguir, expresse um algoritmo que pode ser usado em sua solução na forma de um fluxograma e pseudocódigo.
   Ao final da execução do trecho abaixo, quais os valores de A, B e C?

   A ← 1; B ← 2; C ← 1;
   A ← B+1;
   B ← B-1;
   C ← A+3;
   C ← B-3;
   A ← C+1;

   A=_____ B=_____ C=_____

4. O que será mostrado no vídeo durante a execução deste trecho de programa?
   x ← 1;
   z ← 2;
   a ← x + z;
   b ← x - z;
   escreva("Valor de A: %R", a);
   escreva("Valor de A-1: %R", a-1);

5.2 - ESTRUTURAS DE SELEÇÃO

Neste tipo de estrutura o fluxo de instruções a ser seguido é escolhido em função do resultado da avaliação de uma ou mais condições.

Uma condição é uma expressão lógica.

A classificação das estruturas de decisão é feita de acordo com o número de condições que devem ser testadas para que se decida qual o caminho a ser seguido. Segundo esta classificação, têm-se dois tipos de estruturas de decisão:

5.2.1 Estruturas de Decisão do Tipo Se

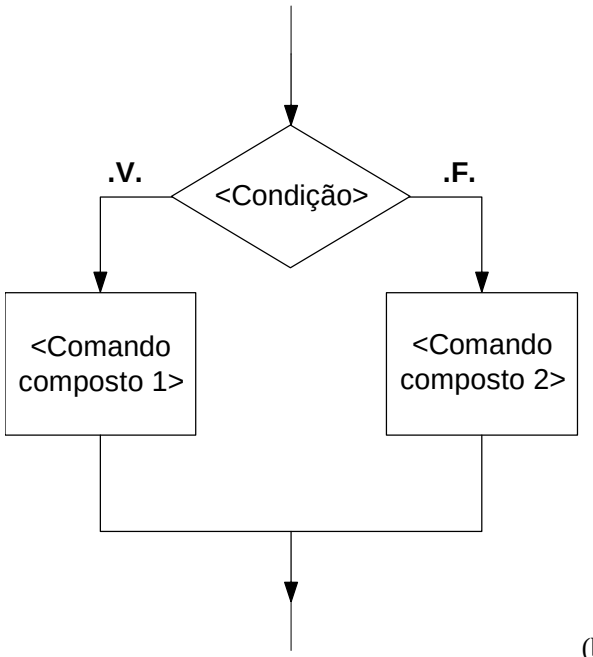
Nesta estrutura uma única condição (expressão lógica) é avaliada. Se o resultado desta avaliação for verdadeiro (.V.), então um determinado conjunto de instruções (comando composto) é executado. Caso contrário, ou seja, quando o resultado da avaliação for falso (.F.), um comando diferente é executado.

Em termos de fluxogramas, uma construção do tipo Se pode ser encarada como uma bifurcação onde há dois caminhos que podem ser seguidos (Figura abaixo). A execução do algoritmo prosseguirá necessariamente por um deles. Esta escolha é feita em função do resultado da expressão: um dos caminhos é rotulado com (.V.) e será seguido quando a condição for verdadeira; o outro é rotulado com (.F.) e será seguido quando a condição for falsa.

A sintaxe da estrutura de decisão do tipo SE é:

Se (<condição>**)** {
    <Comando_composto_l>;
} **Senão** {
    <Comando_composto_ 2>; (a)
}

Um comando composto é um conjunto de zero ou mais comandos (ou instruções) simples, como atribuições e instruções primitivas de entrada ou saída de dados, ou alguma das construções que ainda serão apresentadas neste capítulo.



(b)

Figura: Sintaxe da estrutura de decisão **Se-Então-Senão-Fim_se:**

A semântica desta construção é a seguinte: a condição é avaliada. Se o resultado for verdadeiro, então o **comando_composto_l** é executado. Ao término de sua execução o fluxo do algoritmo prossegue pela instrução seguinte à construção, ou seja, o primeiro comando após o **Fim_se**. Nos casos em que a condição é

avaliada como falsa, o **comando_composto_2** é executado e, ao término do mesmo, o fluxo de execução prossegue pela primeira instrução seguinte ao **Fim_se**.

Há casos particulares e muito comuns desta construção, onde o comando composto 2 é um conjunto vazio de instruções. Neste caso, a porção relativa ao **Senão** pode ser omitida, resumindo a sintaxe da construção à forma mostrada na Figura seguinte.

Se (<condição>)
{
    <Comando_composto_l>;
}

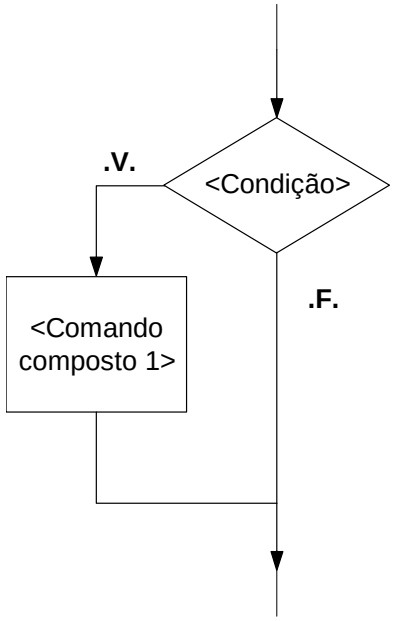


Figura: Sintaxe da estrutura de decisão **Se-Então-Fim_se**

A semântica desta construção é a seguinte: no caso da condição ser verdadeira, o **comando_composto_l** é executado e, após seu término, o fluxo de execução prossegue pela próxima instrução após o **Fim_se**. Quando a condição é falsa, o fluxo de execução prossegue normalmente pela primeira instrução após o **Fim_se**.

A Figura seguinte exemplifica o uso da construção **Se-Então-Senão-Fim_se** num algoritmo para determinar se uma pessoa é maior ou menor de idade.

```
Exemplo()
{
    Inteiro IDADE;
    Leia ("%I",&IDADE);
    Se (IDADE >= 18)
        Escreva ("Maior de idade");
    Senão
        Escreva ("Menor de idade");
}
```

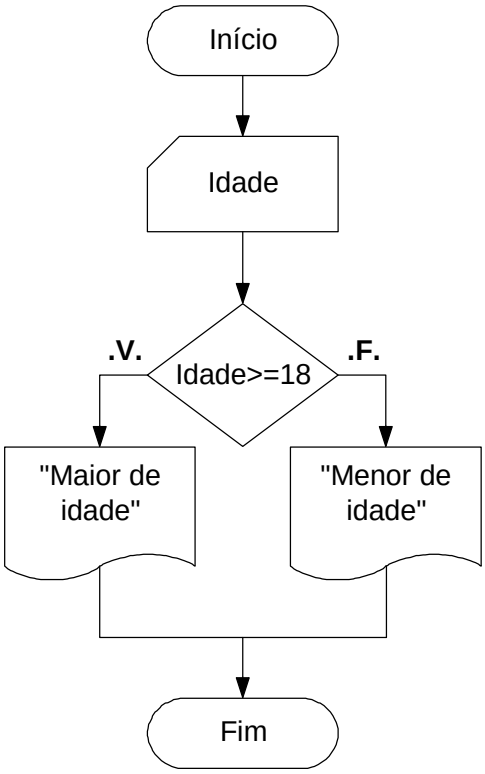


Figura: Exemplo de aplicação da estrutura de decisão **Se-Então-Senão-Fim_se.**

Estruturas **Se/Senão** *aninhadas* verificam vários casos inserindo umas estruturas **Se/Senão** em outras. Por exemplo, a instrução em pseudocódigo a seguir imprimirá **A** para graus de exame maiores ou iguais a **90**, **B** para graus maiores que ou iguais a **80**, **C** para graus maiores que ou iguais a **70**, **D** para graus maiores que ou iguais a **60** e **F** para todos os outros graus.

```
Se (grau >= 90)
        Escreva (“A”);
Senão
        Se (grau >= 80)
                Escreva (“B”);
        Senão
                Se (grau >= 70)
                        Escreva (“C”);
                Senão
                        Se (grau >= 60)
                                Escreva (“D”);
                        Senão
                                Escreva (“F”);
```

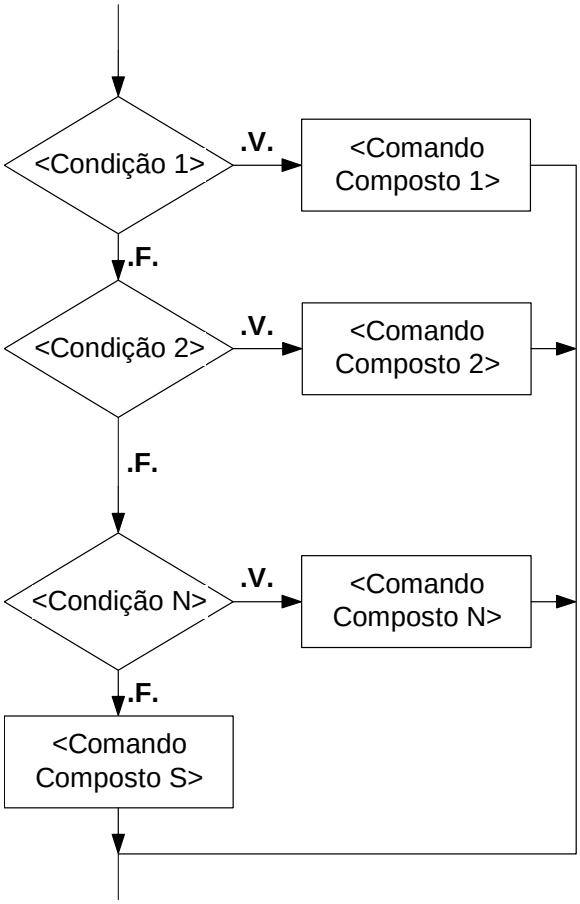
Observação: A estrutura se seleção **Se** deve conter apenas uma instrução em seu corpo. Para incluir várias instruções no corpo de um **Se**, coloque o conjunto de instruções entre chaves (**{** e **}**). Um conjunto de instruções dentro de um par de chaves é chamado uma *instrução composta*.

5.2.2 - Estruturas de Decisão do Tipo Escolha

Este tipo de estrutura é uma generalização da estrutura **Se**, onde somente uma condição era avaliada e dois caminhos podiam ser seguidos. Na estrutura de decisão do tipo **Escolha** pode haver uma ou mais condições a serem testadas e um comando composto diferente associado a cada uma destas.

As estruturas de decisão permitem escolher qual o caminho a ser seguido num algoritmo em função de uma ou mais condições.

A sintaxe da construção de **Escolha** é mostrada na Figura a seguir:



Escolha (<variável>) **{**
    Caso <Valor_1> : <Comando_composto_1>; Pare;
    Caso <Valor_2> : <Comando_composto_2>; Pare;
    . . . . .
    Caso <Valor_N > : <Comando_composto_N>; Pare;
    Senão <Comando_composto_S>;
}

Figura: Sintaxe usada para a condição **Escolha**.

Seu funcionamento é o seguinte: ao entrar-se numa construção do tipo **Escolha**, a **Condição_l** é testada: se for verdadeira, o **comando_composto_l** é executado e, após seu término, o fluxo de execução prossegue pela primeira instrução após o final da construção **(Fim escolha)**; se a **condição_l** for falsa, a **condição_2** é testada: se esta for verdadeira, o **comando_composto_2** é executado e, ao seu término, a execução prossegue normalmente pela instrução seguinte ao **Fim_escolha** (**}**). O mesmo raciocínio é estendido a todas as

condições da construção. No caso em que todas as condições são avaliadas como falsas, o **comando_composto_S** (correspondente ao **Senão** da construção) é executado.

Um exemplo de aplicação desta construção é mostrado a seguir, baseado num algoritmo que verifica se o usuário digitou o número 0, 1, 2 ou outro valor.

```
Exemplo()
{
   Inteiro NÚMERO;
    Leia ("%I",&NÚMERO);
    Escolha (NÚMERO) {
          Caso 0 : Escreva ("Opção escolhida 0"); Pare;
          Caso 1 : Escreva ("Opção escolhida 1"); Pare;
          Caso 2 : Escreva ("Opção escolhida 2"); Pare;
          Senão  Escreva ("Opção inválida");
    }
}
```

Um caso particular desta construção é aquele em que o **comando_composto_S** não contém nenhuma instrução. Isto ocorre nas situações em que não se deseja efetuar nenhuma ação quando todas as condições testadas são falsas. Assim, pode-se dispensar o uso do **Senão** na construção, corno acontece também na construção **Se**.

5.3 - ESTRUTURAS DE REPETIÇÃO

Uma estrutura de repetição permite ao programador especificar que uma ação deve ser repetida enquanto uma determinada condição for verdadeira.

Para	Sabemos o número de repetições
Enquanto	Não sabemos o número exato de repetições (FLAG)
Faça-Enquanto	Quando não sabemos o número de repetições (FLAG). Vamos executar o conjunto de instruções dentro da estrutura pelo menos uma vez.

5.3.1 ESTRUTURA PARA

A estrutura de repetição Para manipula automaticamente todos os detalhes da repetição controlada por contador.

Sintaxe: **Para(**<inicialização>**;**<condição>**;**<incremento>**) {**
```
            Instrução_1;
            Instrução_N;
      }
```

No exemplo de pseudocódigo a seguir, trata-se de um programa para escrever os números de 1 a 10.

```
Exemplo()
{  Inteiro contador;
   Para(contador = 1; contador <= 10; contador←contador+1) {
       Escreva("%d", contador);
   }
}
```

O programa funcionada da seguinte maneira. Quando a estrutura Para começa a ser executada, a variável de controle **contador** é inicializada com o valor **1**. A seguir, a seguir a condição de continuição do laço **contador <= 10** é examinada. Como o valor inicial de **contador** é **1**, a condição é satisfeita, portanto a instrução **Escreva** imprime o valor de **contador**, ou seja, **1**.

A variável de controle **contador** é então incrementada pela expressão **contador → contador + 1** e o laço começa novamente com seu teste de continuação. Como a variável de controle é agora **2**, o valor final não é excedido, e o programa realiza a instrução **Escreva** mais uma vez. O processo continua até que a variável de controle **contador** seja incrementada até o seu valor final de **11** – isso faz com que o teste de continuação do laço não seja verdadeiro e a repetição termine. O programa continua com a primeira instrução depois da estrutura **Para**.

5.3.1 ESTRUTURA ENQUANTO

É uma estrutura de repetição semelhante ao Para-Faça. A diferença básica se refere a quando a repetição irá parar. Na estrutura Para-Faça, sabemos a princípio, quando vamos terminar a estrutura de repetição, ou seja, o número de repetições é conhecido. O valor é dado no problema ou é lido no teclado antes do início da repetição. Na estrutura de repetição podemos ter dois casos:

a) Laços contados (onde o número de repetições é conhecido)
b) Laços condicionais (onde o número de repetições não é conhecido a priori)

Diferenças básicas:

- A variável de controle é iniciada antes do **Enquanto**;
- A variável de controle é incrementada dentro do **Enquanto**;

Sintaxe:

Enquanto (<***condição***>**) {**
    <Comando_Composto>;
    :
    :
}

Exemplo:

PARA	**ENQUANTO**

Leia ("%I",&N);
Para (i = 0; i<N; i ← i + 1) {
  Leia ("%I",&Valor);
  Soma ← Soma + Valor;
}
	Leia ("%I",&N);
	i ← 1;
	Enquanto (i <= N) {
	  Leia ("%I",&Valor);
	  Soma ← Soma + Valor;
	  i ← i + 1;
	}

Onde <**Comando_Composto**> é um comando vazio, um comando simples ou um bloco de comandos.
A ***condição*** pode ser qualquer expressão, e verdadeiro é qualquer valor não-zero.

O laço se repete quando a condição for verdadeira.

Quando a condição for falsa, o controle do programa passa para a linha após o "Fim_Enquanto" (**}**).

Como os laços **Para**, os laços **Enquanto** verificam a condição de teste no início do laço, o que significa que o código do laço pode não ser executado. Isso elimina a necessidade de se efetuar um teste condicional antes do laço.

Cuidados:

- Como os valores em ponto flutuante (Reais) podem ser valores aproximados, controlar a contagem de laços com variáveis de ponto flutuante pode resultar em valores impreciso de contadores e exames incorretos da condição de terminação.
- Uma vez dentro do corpo do laço, a execução somente abandonará o mesmo quando a condição for falsa. O usuário deste tipo de construção deve estar atento à necessidade de que em algum momento a condição deverá ser avaliada como falsa. Caso contrário, o programa permanecerá indefinidamente no interior do laço, o que é conhecido como laço ou looping infinito.

5.3.3 ESTRUTURA FAÇA-ENQUANTO

Ao contrário dos laços **Para** e **Enquanto-**, que testam a condição do laço no começo, o laço **Faça-enquanto** verifica a condição ao final do laço. Isso significa que um laço **Faça-Enquanto** sempre será executado ao menos uma vez. A forma geral do laço **Faça-Enquanto** é:

Faça {
    <*Comando_Composto*>;
} Enquanto (<*condição*>**);**

Observação: É importante observar que ao final do comando **Faça-Enquanto**, é necessária a utilização do ponto-e-vírgula para o seu término.

O laço **Faça-Enquanto** repete até que a condição se torne falsa. O seguinte laço **Faça-Enquanto** lerá números do teclado até que encontre um número maior que zero e menor ou igual a 100.

Faça {
    Leia ("%I",&num);
Enquanto ((num < 0) **.OU.** (num > 100))**;**

Talvez o uso mais comum do laço **Faça-Enquanto** seja em uma rotina de seleção por menu. Quando o usuário entra com uma resposta válida, ela é retornada como o valor da função. Respostas inválidas provocam uma repetição do laço.

Exemplo: Fazer um algoritmo para ler dois valores e executar uma das seguintes operações matemáticas, de acordo coma escolha do usuário: 1. Soma, 2. Subtração, 3. Multiplicação, 4. Divisão.

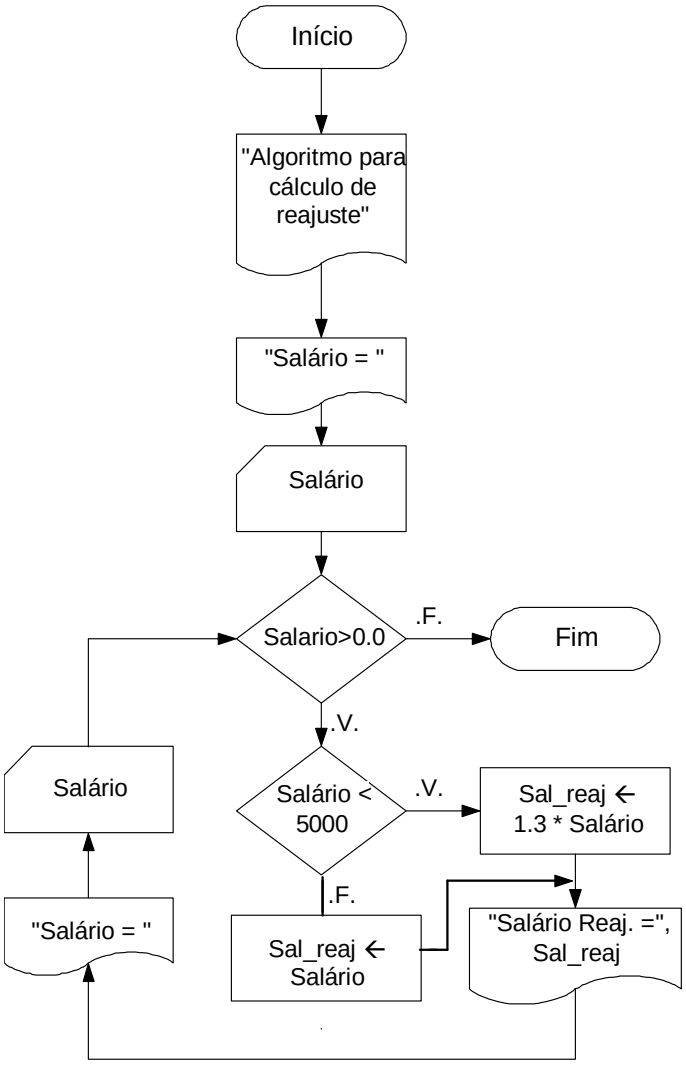
```
Múltipla_Escolha()
{
    Real A, B, X;
    Inteiro Op;
    Escreva ("Entre com o valor de A: ");
    Leia("%R",&A);
    Escreva ("Entre com o valor de B: ");
    Leia("%R",&B);
```

```
        Faça
            Escreva (“1. X = A + B”);
            Escreva (“2. X = A - B”);
            Escreva (“3. X = A * B”);
            Escreva (“4. X = A / B”);
            Escreva (“Opção (1 – 4): ”);
            Leia(“%I”,&Op);
        Enquanto ((Op < 1) .OU. (Op > 4))
        Se (Op = 1) {
            X = A + B;
            Escreva (“X = %I”, A + B);
        } Senão
            Se (Op = 2) {
                X = A - B;
                Escreva (“X = %I”, A - B);
            } Senão
                Se (Op = 3) {
                    X = A * B;
                    Escreva (“X = %I”, A * B);
                } Senão {
                    X = A / B;
                    Escreva (“X = %I”, A / B);
                }
            }
        }
}
```

Exemplo de fluxograma de modo que seja possível o cálculo do salário reajustado de um número indeterminado de funcionários segundo tal critério de reajuste.

Na solução é importante notar que:

- a primeira ação tomada pelo algoritmo é a leitura do valor do salário do primeiro funcionário, que ocorre antes mesmo de se entrar no laço principal do algoritmo. Dessa forma evita-se que o algoritmo entre no corpo do laço condicional se o primeiro valor de **SALÁRIO** já for menor ou igual a zero;
- a condição utilizada no teste da condição de continuidade do laço é **SALÁRIO 0.0**, de modo que, quando um valor menor ou igual a zero for especificado para SALARIO, a condição será avaliada como **.F.** e o laço será abandonado;
- corpo do laço consiste no cálculo do salário reajustado propriamente dito, da escrita do valor calculado e da leitura do valor do salário do próximo funcionário, com base no qual o laço atuará na próxima iteração.
- na estrutura **Se**, se não utilizasse o Senão sal_reaj ← salario, e se a condição do Se fosse falsa, seria mostrado o salário reajustado do funcionário anterior.



5.3.4 Exercícios

1. Diga se cada uma das afirmações seguintes é verdadeira ou falsa. Se for falsa, explique por quê.

    a) Quando a função **Escreva** é chamada, ela sempre começa a imprimir no início de uma nova linha.

    b) Os comentários fazem com que o computador imprima na tela o texto situado entre **/*** e ***/** quando o programa é executado.

    c) A seqüência de escape **\n**, quando usada em uma string de controle de formato de **Escreva**, faz com que o cursor se posicione no início da próxima linha na tela.

    d) Todas as variáveis devem ser declaradas antes de serem usadas.

    e) Todas as variáveis devem receber a atribuição de um tipo ao serem declaradas.

f) O C considera idênticas as variáveis **numero** e **NuMeRo**.

g) As declarações podem aparecer em qualquer lugar do corpo de uma função.

h) Todos os argumentos após a string de controle de formato em uma função **Escreva** devem ser precedidos por um e-comercial (**&**).

i) O operador resto (**%**) só pode ser usado com operadores inteiros.

j) Os operadores aritméticos *****, /, **%** e **-** possuem o mesmo nível de procedência.

2. Escreva uma única instrução ou linha em C que realize cada um dos pedidos seguintes:

   a) Imprima a mensagem **"Entre com dois números"**.

   b) Atribua o produto das variáveis **b** e **c** à variável **a**.

   c) Informe que o programa realiza um exemplo de cálculo de folha de pagamento (use um texto que ajude a documentar o programa).

   d) Forneça ao programa três valores inteiros digitados no teclado e coloque esses valores nas variáveis inteiras **a**, **b** e **c**.

3. Elabore um programa em que o usuário digita um caracter. Se esse caracter for "*" deve surgir uma mensagem "Você digitou o caracter *", caso contrário surgirá uma outra "Digitou o caracter <*caracter que o usuário digitou*>".

4. Elabore um programa em que o usuário digita um caracter. Se esse caracter for 'N' ou 'n' deve surgir a mensagem "Escolheu a opção NÃO", caso contrário surgirá uma outra "Escolheu outra opção".

5. Elabore um programa que informe se um número inserido pelo usuário é par ou ímpar.

6. Elabore um programa que ordene por ordem crescente dois números inseridos pelo usuário (diferentes).

7. Elabore um programa que ordene por ordem decrescente três números inseridos pelo usuário (diferentes).

8. Elabore um programa que peça ao usuário para fornecer dois números inteiros, obtenha-os do usuário e imprima o maior deles seguido das palavras "é maior". Se os números forem iguais, imprima a mensagem "Estes números ao iguais".

9. Elabore um programa que receba três números inteiros **diferentes** digitados no teclado e imprima a soma, a média, o produto, o menor e o maior desses números.

10. Escreva um programa que leia cinco números inteiros e então determine e imprima o maior e o menor inteiro do grupo.

11. Escreva um programa que leia dois inteiros e então determine e imprima se o primeiro é múltiplo do segundo.

12. Escreva um algoritmo para determinar se um número é maior, menor ou igual a 0 (zero) utilizando a estrutura de seleção IF.

13. Faça um algoritmo que leia o nome do aluno e suas duas notas. A partir das informações lidas:
    - calcule e mostre a média do aluno;

- verifique qual será o conceito atribuído ao aluno:
  - se a média for menor que 5: E
  - se a média for maior ou igual que 5 e menor que 6: D
  - se a média for maior ou igual que 6 e menor que 7: C
  - se a média for maior ou igual que 7 e menor que 9: B
  - se a média for maior ou igual que 9 e menor ou igual a 10: A

Faça um algoritmo usando a estrutura **Escolha** e outro usando **Se**.

14. Faça a questão anterior para **n** alunos. Considere que o algoritmo repita até que o nome do aluno seja igual a "FIM".

15. Faça um algoritmo para gerar os 10 primeiros termos da seqüência de Fibonacci:
Obs. A partir do 3o termo soma-se os dois anteriores.

1, l, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

16. Escreva um algoritmo para calcular o reajuste salarial de uma empresa que possui 100 funcionários, de acordo com os seguintes critérios:
- os funcionários com salário inferior a 10.000,00 devem ter um reajuste de 55%;
- os funcionários com salário entre 10.000,00 (inclusive) e 25.000,00 (exclusive) devem ter um reajuste de 30%; e
- os funcionários com salário superior a 25.000,00 devem ter um reajuste de 20%.

17. Faça um algoritmo para gerar os números pares entre 20 e 50. Faça outro algoritmo utilizando a estrutura Para.

18. Uma empresa com 30 funcionários decide presentear seus funcionários com um bônus de Natal, cujo valor é definido do seguinte modo:
- os funcionários do sexo masculino com tempo de casa superior a 15 anos terão direito a um bônus de 20% de seu salário;
- as funcionárias com tempo de casa superior a 10 anos terão direito a um bônus de 25% de seu salário; e
- os demais funcionários terão direito a um bônus de 5.000,00.

Elabore um algoritmo para calcular o valor do bônus concedido a cada funcionário e o impacto de tal atitude no orçamento da empresa (ou seja, o montante total dos bônus concedidos).

19. Faça um programa que leia 10 números e obtenha:
- a soma dos números;
- a média dos números;
- o maior número; e
- o menor número.

20. Faça um algoritmo que solicite 2 números e uma operação matemática elementar.

21. Fazer um algoritmo onde dados 30 números, imprima a média destes números. Verifique se todos são positivos.

22. Fazer um algoritmo onde Dada a idade, dizer se é adulto, jovem ou velho.

23. Fazer um algoritmo para identificar o maior número dentre 3 números informados pelo usuário.

24. Faça um algoritmo onde, dados 2 números, apresente-os ordenados.

25. Calcular a idade média de uma turma de alunos onde não é conhecido a priori o número de alunos. Neste exemplo utilizaremos FLAG que indicará o término do número de alunos. Assumiremos que esta FLAG é um número negativo.

26. Fazer um algoritmo para determinar o número de pessoas do sexo feminino de uma turma de alunos, onde para sexo=1 é Feminino e sexo=2 é masculino. A FLAG de finalização é –1. Imprima ao final o número de pessoas do sexo feminino.

27. 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 escreva a idade média deste grupo de indivíduos.

28. Supondo que a população de um país A seja da ordem de 90.000.000 de habitantes com uma taxa anual de crescimento de 3% e que a população de um país B seja, aproximadamente, de 200.000.000 de habitantes com uma taxa anual de crescimento de 1.5%, fazer um programa que calcule e escreva o número de anos necessários para que a população do país A ultrapasse ou se iguale à população do país B, mantidas essas taxas de crescimento.

29. Tem-se um conjunto de dados contendo a altura e o sexo (masculino=1, feminino=2) de N pessoas. Fazer um programa que calcule e escreva:

- a maior e a menor altura do grupo;
- a média de altura das mulheres;
- o número de homens;
- % de mulheres.

Adotar como FLAG altura < 0.

30. 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);
- cor dos olhos (azuis, verdes, castanhos);
- cor dos cabelos (louros, castanhos, pretos);
- idade em anos.

Para cada habitante, foi perfurado um cartão com esses dados, e o último cartão, que não corresponde a ninguém, conterá o valor da idade igual a –1. Fazer um programa que determine e escreva:

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

31. Um comerciante deseja fazer o levantamento do lucro das mercadorias que ele comercializa. Pra isto, mandou digitar uma linha para cada mercadoria com o nome, preço de compra e preço de venda das mesmas. Fazer um programa que:

Determine e escreva quantas mercadorias proporcionam:

Lucro <	10%	
10% ≤	Lucro ≥	20%
Lucro >	20%	

Determine e escreva o valor total de compra e de venda de todas as mercadorias assim como o lucro total.

Observação: Adotar um *flag* adequado.

32. A UDESC deseja fazer um levantamento a respeito de seu concurso vestibular. Para cada curso, é fornecido o seguinte conjunto de valores:

- código do curso;
- número de vagas;
- número de candidatos do sexo masculino;
- número de candidatos do sexo feminino.

O último conjunto, para indicar fim de dados, contém o código do curso igual a zero. Fazer um programa que:

- calcule e escreva, para cada curso, o número de candidatos por vaga e a porcentagem de candidatos do sexo feminino (escreva também o código correspondente do curso);
- determine o maior número de candidatos por vagas e escreva esse número juntamente com o código do curso correspondente (supor que não haja empate);
- calcule e escreva o total de candidatos.

33. Escreva um programa que leia uma string, conte quantos caracteres desta string são iguais a ‘a’ e substitua os que forem iguais à ‘a’ por ‘b’. O programa deve imprimir o número de caracteres modificados e a string modificada.

34. Faça um programa que inverta uma string: leia a string com gets e armazene-a invertida numa outra string. Use a estrutura Para-Faça para percorrer a string até o seu final.

UNIDADE 6 – VARIÁVEIS INDEXADAS

6.1 - Introdução

Existem diversas situações na área de processamento de dados que existe a necessidade de se armazenar um grande conjunto de dados na memória RAM do computador. Muitas dessas situações os dados estão relacionados.

Por Exemplo:

1. Ler todos os nomes dos alunos da turma de ICC e ordena-los alfabeticamente.
2. Ler todas as notas dos alunos desta turma e classifica-los imprimindo os que tiveram média acima de sete.
3. Etc.

Nestes casos é inviável utilizarmos uma variável para cada aluno ou nota, sendo assim, as linguagens de programação incluem um mecanismo chamado **variável indexada** que permite realizar este armazenamento com uma única variável. O termo *indexada* provém da maneira como esta individualização é feita: por meio de **índices**.

Variáveis indexadas são denominadas matrizes que se dividem em duas partes:

- Matrizes Unidimensionais : também chamadas vetores (um único índice é usado);
- Matrizes Multidimensionais : duas ou mais dimensões (possui dois ou mais índices).

6.2 - Operações Básicas com Variáveis Indexadas

Do mesmo modo que acontece com variáveis simples, também é possível operar com variáveis indexadas. Contudo, não é possível operar diretamente com o conjunto completo, mas com cada um de seus componentes isoladamente. O acesso individual a cada componente de um conjunto é realizado pela especificação de sua posição no mesmo por meio de um ou mais índices (no caso de uma matriz).

Portanto, não é possível operar diretamente sobre conjuntos como um todo, mas apenas sobre seus componentes, um por vez. Por exemplo, para somar dois vetores é necessário somar cada um de seus componentes dois a dois. Da mesma forma, as operações de atribuição, leitura e escrita de conjuntos devem ser feitas elemento a elemento.

6.2.1 - Atribuição

No caso de variáveis indexadas, além do nome da variável deve-se necessariamente fornecer também o(s) índice(s) da componente do conjunto onde será armazenado o resultado da avaliação da expressão.

Naturalmente, também é possível usar conjuntos no meio de expressões, como se faz com variáveis simples, desde que seja especificado o elemento do conjunto que servirá como operando.

Exemplos:

NUMEROS[2] ← 10;

SALARIOS[3] ← 100.00;

NOMES[10] ← “JOÃO”;

MATRIZ[5][8] ← - 3.6;

6.2.2 - Leitura

A leitura de um conjunto é feita passo a passo, um componente por vez, usando a mesma sintaxe da instrução primitiva de entrada de dados (**Leia** nome_variável). Mais uma vez, além do nome do conjunto, deve ser explicitada a posição do componente lido.

Exemplo:

```
Exemplo_622()
{
        Inteiro Numeros[10], i;
        Para (i ← 0; i < 10; i ← i + 1) {
             Escreva(“%I - Número:  ”, i+1);
             Leia (“%I”,&Numeros[ i ]);
        }
}
```

Uma observação importante a ser feita é o uso da construção **Para-Faça** a fim de efetuar a operação de leitura dez vezes, em cada uma delas lendo um determinado componente do vetor. De fato, o uso desta construção é muito comum quando se opera com conjuntos, devido à necessidade de se realizar uma mesma operação com os diversos componentes dos mesmos. Na verdade, são raras as situações em que se deseja operar isoladamente com um único componente do conjunto.

6.2.3 - Escrita

A escrita de um conjunto obedece à mesma sintaxe da instrução primitiva de saída de dados (<**Escreva** nome_variável>). Mais uma vez, convém lembrar que, além do nome do conjunto, deve-se também especificar por meio de seu(s) índice(s) qual(is) o(s) componente(s) do vetor a ser(em) escrito(s).

Exemplo 01:

```
Exemplo_623()
{
        Inteiro Numeros[10], i;
        Para (i ← 0; i < 10; i ← i + 1) {
             Escreva("%I - Número:  ", i+1);
             Leia ("%I",&Numeros[ i ]);
             Escreva ("%I",Numeros[ i ]);
        }
}
```

Exemplo 02:

```
Exemplo_624()
{
        Real Numeros[10], Soma;
        Inteiro i;
        Para (i ← 0; i < 10; i ← i + 1) {
             Escreva("%I - Número:  ", i+1);
             Leia ("%I",&Numeros[ i ]);
             Soma ← Soma + Números[i];
        }
        Escreva ("Soma = %R", Soma);
}
```

6.3 - Matrizes Unidimensionais (Vetores)

Como outras variáveis, as matrizes devem ser explicitamente declaradas para que o compilador possa alocar espaço para elas na memória. A quantidade de armazenamento necessário para guardar uma matriz está diretamente relacionada com seu tamanho e tipo. Para uma matriz unidimensional, o tamanho total em bytes é calculado por: **total em bytes = sizeof(tipo) * tamanho da matriz**

Matrizes unidimensionais são, essencialmente, listas de informações do mesmo tipo, que são armazenadas em posições contíguas da memória em uma ordem de índice.

Obs.: Um detalhe importantíssimo da linguagem C no tratamento destas variáveis é que o índice de acesso a um dos valores de uma variável com N elementos vai de 0 (zero) até (N – 1), diferente da maioria das outras linguagens que vão de 1 a N. Em função deste *range* de variação é muito comum as linhas de programa com "Para_Faça" com índices que variam de 0, enquanto for menor que o tamanho N.

Exemplo 01:

Ler três números inteiros, achar a média e imprimir os números maiores que a média.

Solução Tradicional:

```
Algoritmo <Programa_Imprime>
Início
        Inteiro A1, A2, A3;
        Real Media;

        Ler A1;
        Ler A2;
        Ler A3;
        Media ← (A1 + A2 + A3) / 3;
        Se (A1 > Media) então
             Escreva A1;
        Fim_Se
        Se (A2 > Media) então
             Escreva A2;
        Fim_Se
        Se (A3 > Media) então
             Escreva A3;
        Fim_Se
Fim
```

Modelo de Memória

A1	A2	A3	Media
5	7	8	6.6

Obs.: Imagine o mesmo exercício para 100 valores.

Exemplo 02:

```
Algoritmo <Programa_10>
Início
        Real X1, X2, X3, ..., X10, Media;

        Ler X1;
        Ler X2;
        Ler X3;
              :
              :
        Ler X10;
        Media ← (X1 + X2 + X3 + X4 + X5 + X6 + X7 + X8 + X9 + X10) / 10;
        Escreve Media;
        Escreve X1;
        Escreve X2;
              :
              :
        Escreve X10;
Fim
```

O uso de variáveis indexadas nos dá a possibilidade de combinar o nome de uma variável com um índice numérico. O nome da variável associado com o valor numérico está relacionado a uma posição de memória.

Exemplo:

Variável Real Aux[10] – significa que na memória foi reservado 10 posições e se pode ler e atribuir 10 valores a esta variável que são diferenciados por seus índices.

1ª Entrada de Dados

```
Aux[0] = 1;
Aux[1] = 5;
Aux[2] = 3;
      :
      :
Aux[9] = 45;
```

2ª Entrada de Dados (Mudança de Valores)

```
Aux[0] = 15;
Aux[1] = 25;
Aux[9] = 2;
```

Modelo de Memória

Índice		***0***	***1***	***2***	***3***	***...***	***9***
Conteúdo	1ª	1	5	3	?	?	45
	2ª	15	25	3	?	?	2

Um vetor é definido da seguinte forma:

<tipo variável> <nome_variável>[Nº de itens];

Exemplos: Inteiro X[5];
           Real R[3];

Modelo de Memória

X[0]	X[1]	X[2]	X[3]	X[4]	R[0]	R[1]	R[2]
?	?	?	?	?	?	?	?

Vantagens:

- Com única definição de variável pode-se armazenar vários valores na memória do computador;
- O acesso pode ser feito diretamente ou através de um índice;
- Facilidade de adicionar grandes quantidades de valores na memória;
- Simplificação do código para leitura e escrita de valores.

Exemplos:

```
R[0] = 5;
R[2] = 9;
```

Ou

```
Para (i ← 0; i < 5; i ← i + 1) Faça
    X[ i ] ← 10;
Fim_Para    // Ao final todos os elementos de X[0 – 5] estão com valores 10.
```

Modelo de Memória

X[0]	X[1]	X[2]	X[3]	X[4]	R[0]	R[1]	R[2]
10	10	10	10	10	5	?	9

```
Para (i ← 0; i < 3; i ← i + 1) Faça
    Ler R[ i ];
Fim Para
```

Modelo de Memória ao final do Para-Faça

R[0]	R[1]	R[2]
5	7	8

```
Para (i ← 0; i < 3; i ← i + 1) Faça
    Escreva R[ i ];
Fim Para
```

Aplicando estes Conceitos ao Exercício Anterior

Exemplo 01

```
Algoritmo <Imprime_vetor>
Inicio
        Inteiro   A[100], i;
        Real   Media, Soma;

        Soma ← 0;
        Para (i ← 0; i < 100; i ← i + 1) Faça
            Ler A[ i ];
            Soma ← Soma + A[ i ];
        Fim_Para;

        Media ← Soma / 100;
        Para (i ← 0; i ← 100; i ← i + 1) Faça
            Se (A[ i ] > Media) então
                Escreva A[ i ];
            Fim_se
        Fim_Para
Fim
```

6.3.1 – Métodos de Pesquisa Seqüencial

A pesquisa **seqüencial** ou **linear** é o método mais objetivo para se encontrar um elemento particular num conjunto (em geral, um vetor) não classificado, isto é, cujos elementos não estão ordenados segundo algum critério.

Esta técnica envolve a simples verificação de cada componente do conjunto seqüencialmente (uma após a outra) até que o elemento desejado seja encontrado (neste caso diz-se que a pesquisa foi **bem-sucedida**) ou que todos os elementos do conjunto (vetor) tenham sido verificados sem que o elemento procurado tenha sido encontrado (pesquisa **malsucedida**).

Exemplo: Ler um nome e verificar se o mesmo se encontra num vetor de 100 nomes utilizando a pesquisa seqüencial.

```
Algoritmo <Pesq_Seq>
Inicio
        Literal Valor[30], Nomes [100][30];
        Inteiro i, Achou;
            Para (i ← 0; i < 100; i ← i + 1) faça
            Escreva (“Nome:  ”);
            Leia Nomes[ i ];
        Fim_Para
```

```
        Escreva("Nome a ser pesquisado:  ");
        Leia Valor;
        i ← 0;
        Achou ← 0;
        Enquanto (i < 100 .E. !Achou) faça
            Se (Nomes[ i ] = Valor) então
                Achou ← 1;
            Senão
                i ← i + 1;
            Fim_Se
        Fim_Enquanto
        Se (Achou) então
            Escreva ("O nome %L foi encontrado", Valor);
        Senão
            Escreva("O nome %L não foi encontrado", Valor);
        Fim_se
Fim
```

6.3.2 – Método de Pesquisa Binária

Quando os elementos de um vetor estão previamente classificados segundo algum critério, então pesquisas muito mais eficientes podem ser conduzidas. Entre elas destaca-se o método de **pesquisa binária**.

Seu funcionamento é o seguinte: inicialmente o vetor é classificado (por exemplo, em ordem crescente). O elemento que divide o vetor ao meio (ao menos aproximadamente) com relação ao seu número de componentes é localizado e comparado com o valor procurado. Se ele for igual ao valor procurado a pesquisa é dita bem-sucedida e interrompida. No caso dele ser maior que o valor procurado, repete-se o processo na primeira metade do vetor. No caso do elemento central do vetor ser menor que o valor procurado, repete-se o processo na segunda metade do vetor. Este procedimento é continuado até que o elemento desejado seja localizado (pesquisa bem-sucedida), ou então, até que não reste mais um trecho do vetor a ser pesquisado (pesquisa malsucedida).

Exemplo:

```
Algoritmo<Pesq_Bin>
Início
        Literal Valor[30], Nomes[100][30];
        Inteiro Meio, Alto, Baixo, Achou;

        Para (i ← 0; i < 100; i ← i + 1) faça
            Escreva ("Nome:  ");
            Leia Nomes[ i ];
        Fim_Para
        Escreva("Nome a ser pesquisado:  ");
        Leia Valor;
        Achou ← 0;
        Baixo ← 0;
        Alto ← 99;
```

```
        Enquanto (Baixo <= Alto .E. !Achou) Faça
            Meio ← (Baixo + Alto) / 2;
            Se (Valor < Nomes[Meio]) então
                Alto ← Meio – 1;
            Senão
                Se (Valor > Nomes[Meio]) então
                    Baixo ← Meio + 1;
                Senão
                    Achou ← 1;
                Fim_Se
            Fim_se
        Fim_Enquanto
        Se (Achou) então
            Escreva (“O nome %L foi encontrado”, Valor);
        Senão
            Escreva(“O nome %L não foi encontrado”, Valor);
        Fim_se
Fim
```

O método de pesquisa binária tem a desvantagem de exigir que o vetor seja previamente ordenado para seu correto funcionamento, o que não acontece no caso da pesquisa seqüencial. Por outro lado, o método da pesquisa binária é **em média** mais rápido que o método de pesquisa seqüencial.

6.3.3 – Método da Bolha de Classificação

Este método não é o mais eficiente, mas é um dos mais populares devido à sua simplicidade.

A filosofia básica deste método consiste em “varrer” o vetor, comparando os elementos vizinhos entre si. Caso estejam fora de ordem, os mesmos trocam de posição entre si. Procede-se assim até o final do vetor. Na primeira “varredura” verifica-se que o último elemento do vetor já está no seu devido lugar (no caso de ordenação crescente, ele é o maior de todos). A segunda “varredura” é análoga à primeira e vai até o penúltimo elemento. Este processo é repetido até que seja feito um número de varreduras igual ao número de elementos a serem ordenados menos um. Ao final do processo o vetor está classificado segundo o critério escolhido.

Exemplo: Classificação de um vetor de dez elementos reais em ordem crescente.

```
Algoritmo<Bolha>
Início
        Real Numeros[10], aux;
        Inteiro i, j;

        Para (i ← 0; i < 10; i ← i + 1) faça
            Escreva (“Número:  ”);
            Leia Numeros[ i ];
        Fim_Para
        j ← 9;
        Enquanto (j > 0) faça
            Para (i ← 0; i < j; i ← i + 1) faça
                Se (Numeros[ i ] > Numeros[i + 1]) então
                    Aux ← Numeros[ i ];
                    Numeros[ i ] ← Numeros [i + 1];
                    Numeros[i + 1] ← Aux;
                Fim_se
            Fim_Para
            j ← j – 1;
        Fim_Enquanto
        Escreva (“Vetor Ordenado:  ”);
        Para (i ← 0; i < 10; i ← i + 1) faça
            Escreva (“%R”, Numeros[ i ]);
        Fim_Para
Fim
```

6.3.4 Exercícios

6.3.4.1 – Faça um algoritmo para ler, somar e imprimir 10 valores (utilizar 3 estruturas Para).

6.3.4.2 – Escreva um novo algoritmo usando no máximo duas estruturas Para-Faça.

6.3.4.3 – Faça um algoritmo para ler n valores. Após todos os elementos lidos calcular a soma, a média e ao final imprimir os valores, ou seja, quando o usuário entrar com o valor 0 (zero = Flag).

6.3.4.4 – Dado um vetor com 10 elementos (lidos do teclado), verifique se o valor "25" pertence a este conjunto de dados. Se pertencer imprima a posição do vetor em que está armazenada.

6.3.4.5 – Dado o vetor inteiro [100]:

a) preenche-lo com valor -1;
b) preenche-lo com o valor 2, 4, 6, 8, ..., 200;
c) preenche-lo com o valor 1, 2, 3, 5, 8, 13, ...

6.3.4.6 – Ler o nome e idade de 25 alunos. Calcular a média e imprimir o nome e idade dos alunos com idade inferior a média.

6.3.4.7 – Faça um algoritmo para ler um conjunto de dados inteiros até que seja encontrado um valor negativo ou igual a zero. Após a inserção de todos os valores, calcule a soma e a média dos valores lidos (exceto o último) e ao final imprima todos os valores.

6.3.4.8 – Dados os vetores abaixo:

A =

0	1	2	3	4	5	6	7
8	7	6	5	4	3	2	1

B =

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8

Fazer um algoritmo para somar os dois vetores e armazenar o resultado em um vetor C.

6.3.4.9 – Elabore um algoritmo que leia "N" números reais (N < = 1000) e imprima-os em ordem inversa.

6.3.4.10 – A UDESC – SBS promoveu um vestibular onde se inscreveram 1.000 alunos. Para cada aluno é dado o código e suas respostas (10). Também é fornecido um gabarito com as respostas corretas (10 questões – A, B, C, D). Faça um programa para ler o gabarito, o nome e as respostas de cada candidato a partir do gabarito, calcule a nota de cada um e armazene este valor. Imprima ao final os códigos aprovados com média superior a 5.0.

6.3.4.11 – Ler 30 números inteiros e armazena-los em um vetor. Conte o número de valores que são iguais a três e imprima a posição do vetor em que estão armazenados.

6.3.4.12 – Ler para um conjunto de alunos o nome, código, nota e freqüência.

a) Imprima um relatório com os alunos aprovados com nota maior ou igual a sete;
b) Imprima um relatório com os alunos reprovados por falta (freqüência menor que 75%);
c) Imprima um relatório dos alunos reprovados por nota.

Estas opções de relatório devem ser através de um menu de opções.

6.3.4.13 – Ler duas variáveis unidimensionais 'A' e 'B', contando, cada uma, 25 elementos inteiros. Intercale os elementos destes dois conjuntos formando uma nova variável de 50 elementos.

A =

0	1	2	3	4	5	...	24
100	200	500	300	250	15	...	351

B =

0	1	2	3	4	5	...	24
1	2	3	4	5	6	...	8

C =

0	1	2	3	4	...	49	49
100	1	200	2	500	...	351	8

6.3.4.14 – Dado um vetor com 20 posições com valores inteiros lidos do teclado, verificar se o último valor armazenado (Vetor[19]) está repetido entre as posições 0 ... 18 do mesmo vetor. Se estiver, quantas vezes?

6.3.4.15 – Ler o nome e o salário de n funcionários e armazena-los em vetores. Ao final da leitura, imprimir o nome e o salário de todos os funcionários.

6.3.4.16 – Idêntico ao anterior, sendo que foi concedido um aumento de 30% para todos os funcionários.

6.3.4.17 – Faça um algoritmo para ler o número de matrículas e a idade de 10 alunos. Calcule e imprima a média e a soma de idades. Ao final, imprima o número de matrícula e a idade de cada aluno.

6.3.4.18 – Faça um algoritmo para ler as fichas de um conjunto de N funcionários. O valor N é digitado pelo operador do programa. As fichas contém os seguintes dados:

- Nome;
- Idade;
- Salário.

Calcular um novo salário para os funcionários com salário inferior a R$ 500,00 com um aumento de 20%. Ao final imprima o nome, idade, e salário de todos os funcionários.

6.3.4.19 – Faça um algoritmo para ler o número de matrícula e a idade de N alunos. Calcule e imprima a média e a soma das idades. Também imprima o número de matrícula do aluno que possuir maior idade, bem como sua idade. Ao final, imprima o número de matrículas e a idade de todos os alunos.
Relatórios:
1 – Nome dos alunos por ordem crescente de idade;
2 – Por ordem crescente de idade.

6.3.4.20 – Faça um algoritmo para ler N valores (onde N é lido do teclado) para cada conjunto de dados é lido o número do empregado, o salário, a idade e o sexo (0 para feminino e 1 para masculino). Calcule ao final da leitura dos dados:

- A soma dos salários;
- Aumento de 10% para todos.

Imprima:

- A soma dos salários;
- A média dos salários;
- O número, novo salário, idade e o sexo para cada funcionário;
- O maior e o menor salário com o número do funcionário que o recebe.

6.3.4.21 – Um professor tem uma turma de 20 alunos e deseja calcular e imprimir a nota de cada aluno seguido da média da turma.
Aluno 1 = 10 Média da Turma = 10.0
Aluno 2 = 5 Média da Turma = 7.5
Faça um algoritmo para o problema acima, usando as estruturas Enquanto-Faça e Para-Faça.

6.3.4.22 – Faça um algoritmo para calcular o número de alunos que tiraram nota acima da média da turma. As notas serão fornecidas via teclado e a turma tem 10 alunos.

6.3.4.23 – Fazer um algoritmo para calcular o número de alunos que tiraram nota acima da nota média da turma. Imprimir o número de matrícula e anota destes alunos. As notas e os números são fornecidos via teclado e a turma tem 40 alunos.

6.3.4.24 – Um conjunto de dados com os dias do ano e a temperatura média do dia é fornecida. Elabore um algoritmo que calcule e escreva:

- A menor temperatura ocorrida no ano;
- A maior temperatura ocorrida no ano;
- A temperatura média anual;
- O número de dias em que a temperatura foi inferior à média anual, também imprima o dia (1 ... 365) e a média de temperatura do dia.

6.3.4.25 – Desenvolver um programa para ler do teclado um conjunto de fichas de funcionários de uma empresa chamada Casa Velha. Cada ficha tem as informações de idade, salário e nome. A flag para término das fichas é o salário = 0. Ao final da leitura das fichas, o programa deve apresentar um menu de relatórios com os seguintes itens:
1 – Imprimir o nome de todos os funcionários;
2 – Imprimir o nome e o salário de todos os funcionários;
3 – Imprimir o nome, o salário e o novo salário considerado um aumento de 30% para todos;
4 – Imprimir o nome e o salário das pessoas que possuem o maior e o menor salário;
5 – Imprimir o nome da pessoa mais jovem e a posição que ela ocupa no vetor;
6 – Fim.
Obs.: o programa deve continuar a apresentar o menu na tela após cada relatório, sendo que o programa deve finalizar somente após se teclar a opção 6.

6.3.4.26 – Faça um algoritmo para ler a idade de 6 alunos de uma estrutura e imprimir as idades em ordem crescente.

6.3.4.27 – Escrever um algoritmo para ler uma variável unidimensional A de 25 elementos e inverter estes elementos armazenando-a num vetor B.

A =

0	1	2	3	4	...	23	24
10	20	30	40	50	...	80	90

B =

0	1	...	20	21	22	23	24
90	80	...	50	40	30	20	10

6.3.4.28 – Ler 50 valores inteiros, multiplicar todos os valores por uma constante lida do teclado. Imprimir a posição de todos os valores que são iguais a 20 e os que tem valor negativo.

6.3.4.29 – Ler um conjunto de valores inteiros de 10 posições e armazenar em um vetor. Ordena-los em forma crescente.

6.3.4.30 – Idem ao anterior mas em ordem decrescente.

6.3.4.31 – Escrever um algoritmo que faça reserva de passagens aéreas de uma companhia. Além da leitura do número dos vôos e da quantidade de lugares disponíveis, ler vários pedidos de reserva, constituídos do número da carteira de identidade do cliente e do número do vôo desejado. Para cada cliente, verificar se há disponibilidade no vôo desejado. Em caso afirmativo, imprimir o número da identidade do cliente e o número do vôo, atualizando o número de lugares disponíveis. Caso contrário, avisar ao cliente da inexistência de lugares. Indicando o fim dos pedidos de reserva, existe um passageiro cujo número da carteira de identidade é 99999. Considerar fixo e igual a 37 o número de vôos da companhia.

6.3.4.32 – São lidos dois vetores: O 1º com a descrição das mercadorias e o 2º com a quantidade de mercadorias existentes. A seguir, são lidos os pedidos dos clientes que contém o código do cliente, o nome do produto e a quantidade de produtos. O número de mercadorias disponíveis é igual a 100. O código do cliente que finaliza os produtos é igual a 0. Fazer um algoritmo para:

- Ser lido e listado o estoque inicial;
- Seja verificado, para cada pedido, se ele pode ser integralmente atendido; Em caso negativo, seja impresso o código do cliente e a mensagem: "Não temos a mercadoria em estoque suficiente";
- Seja atualizado o estoque após cada pedido;
- Seja listado o estoque final.

6.3.4.33 – Faça um algoritmo para ler 50 valores inteiros. Faça um menu de opções e execute o programa de acordo com a opção escolhida.
1 – Soma e impressão de todos os valores;
2 – Produto de todos os valores;
3 – Impressão dos valores ordenados por ordem decrescente;
4 – Fim.
Obs.: O menu deve ser apresentado até que seja digitado como opção de escolha o valor 4 (Fim).

6.3.4.34 – Construir um algoritmo que, dado um vetor A de 20 elementos, calcule:

$$S = (A_0 - A_{19})^2 + (A_1 - A_{18})^2 + ... + (A_9 - A_{10})^2$$

6.3.4.35 – A UDESC – SBS deseja saber se existem alunos cursando, simultaneamente, as disciplinas de ICC e CDI. Existem disponíveis, na unidade de entrada, os números de matrícula dos alunos de ICC (no máximo 100 alunos) e de CDI (no máximo 140 alunos). Cada conjunto dos números de matrícula dos alunos de uma disciplina tem a matrícula fictícia 999999 no final. Formular um algoritmo que imprima o número de matrícula dos alunos que estão cursando estas disciplinas simultaneamente. Trata-se, então, de verificação da ocorrência de um elemento de um conjunto em outro conjunto. Assim, após a leitura dos dados, estarão montadas as seguintes variáveis compostas unidimensionais ICC e CDI, contendo, respectivamente, os números de matrícula dos alunos que estão cursando estas duas disciplinas.

6.3.4.36 – Fazer um algoritmo para corrigir provas de múltipla escolha. Cada prova tem 10 questões e cada questão vale 1 ponto. O primeiro conjunto de dados a ser lido será o gabarito para a correção da prova. Os outros dados serão, os números dos alunos e suas respectivas respostas, e o último número, do aluno fictício, será 9999. O algoritmo deverá calcular e imprimir:

a) para cada aluno, o seu número e a sua nota;
b) a porcentagem de aprovação, sabendo-se que anota mínima de aprovação é 7;
c) a nota que teve maior freqüência absoluta, ou seja, a nota que apareceu maior número de vezes (supondo a inexistência de empates).

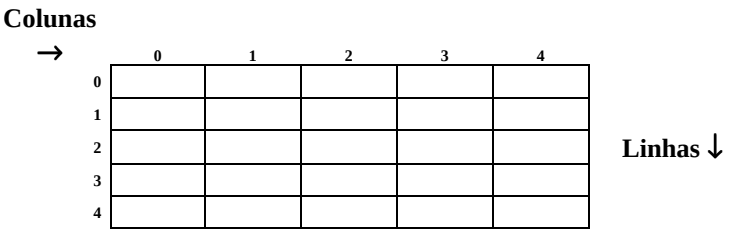
6.4 – Matrizes Multidimesionais (Matrizes)

6.4.1 - Introdução

O recurso de se colocarem várias constantes em uma só variável não é exclusivo do vetor; a matriz é uma variável que também apresenta essa característica.

A forma mais simples de matriz multidimensional é a matriz bidimensional – uma matriz de matrizes unidimensionais.

Matriz A



A – Tem dimensão 5 x 5

6.4.2 - Declarando a Matriz

Como acontece com outras variáveis, quando uma matriz é declarada, o computador reserva uma parte da memória para ela. No caso da matriz, essa parte é dividida em determinada quantidade de colunas e linhas (indicada na declaração) e identificada por um par de números (um referente à linha e outro à coluna).

Uma matriz só pode armazenar constantes de um único tipo.

É conveniente prestar bastante atenção à declaração, pois muitas linguagens de computador usam vírgulas para separar as dimensões da matriz; C, em contraste, coloca cada dimensão no seu próprio conjunto de colchetes.

Definição:

<tipo variável> <nome_variável> <[Dim 1][Dim 2]>;

onde Dim1 e Dim 2 são as dimensões da matriz (linha x coluna).

Exemplo:

Inteiro A[5][5]; //uma matriz inteira com 5 linhas e 5 colunas.

Matrizes bidimensionais são armazenadas em uma matriz linha-coluna, onde o primeiro índice indica a linha, e o segundo, a coluna.

Isso significa que o índice mais a direita varia mais rapidamente do que o mais à esquerda quando acessamos os elementos da matriz na ordem em que eles estão realmente armazenados na memória.

6.4.3 - Inserindo valores em uma matriz

No caso de variáveis indexadas, além do nome da variável deve-se necessariamente fornecer também o(s) índice(s) do componente do conjunto onde será armazenado o resultado da avaliação da expressão.
Naturalmente, também é possível usar conjuntos no meio de expressões, como se faz com variáveis simples, desde que seja especificado o elemento do conjunto que servirá como operando.

Exemplo (especificando o elemento do conjunto):

```
A[0][0] ← 5;
A[0][1] ← 6;
A[0][4] ← 10;
A[1][0] ← 10;
A[3][4] ← 20;
```

Colunas →

Linhas ↓	0	1	2	3	4
0	5	6			10
1	10				
2					
3					20
4					

Exemplo (inserindo constantes - zerando uma matriz):

```
Algoritmo<Zera_Matriz>
Inicio
    Inteiro A[5][5], i, j;

    Para (i ← 0; i < 5; i ← i + 1) faça
        Para (j ← 0; j < 5; j ← j + 1) faça
            A[i][j] ← 0;
        Fim_Para
    Fim_Para
Fim
```

	0	1	2	3	4
0	0	0	0	0	0
1	0	0	0	0	0
2	0	0	0	0	0
3	0	0	0	0	0
4	0	0	0	0	0

Exemplo (inserindo constantes):

```
Algoritmo<um_a_doze>
Inicio
    Inteiro num[3][4], t, i;

    Para (t ← 0; t < 3; t ← t + 1) faça
        Para (i ← 0; i < 4; i ← i + 1) faça
            num[t][i] ← (t * 4) + i + 1;
        Fim_Para
    Fim_Para
//agora escreva-os...
    Para (t ← 0; t < 3; t ← t + 1) faça
        Para (i ← 0; i < 4; i ← i + 1) faça
            Escreva (“%I”, num[t][i]);
        Fim_Para
        Escreva(“\n”); //troca de linha
    Fim_Para
Fim
```

num [t] [i]

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

Exemplo (Lendo e escrevendo os valores lidos de uma matriz 4 x 4):

```
Algoritmo<leitura_e_escrita>
Inicio
    Inteiro mat[4][4], i, j;

    Para (i ← 0; i < 4; i ← i + 1) faça
        Para (j ← 0; j < 4; j ← j + 1) faça
            Leia mat[i][j];         //Leia ("%I", mat[i][j]);
        Fim_Para
    Fim_Para
//agora escreva-os...
    Para (i ← 0; i < 4; i ← i + 1) faça
        Para (j ← 0; j < 4; j ← j + 1) faça
            Escreva ("%I", num[i][j]);
        Fim_Para
        Escreva("\n");              //troca de linha
    Fim_Para
Fim
```

6.4.4 - Cálculo do Tamanho

No caso de uma matriz bidimensional, a seguinte fórmula fornece o número de bytes de memória necessários para armazena-la:

Bytes = tamanho do 1º índice * tamanho do 2º índice * sizeof(tipo base)

Portanto, assumindo inteiros de dois bytes, uma matriz de inteiros com dimensões 10 x 5 teria 10 * 5 * 2, ou 100 bytes alocados.

No exemplo para preencher uma matriz com 12 constantes em ordem crescente, por se tratar de uma matriz 3 x 4, então o espaço alocado em memória é: 3 * 4 * 2 = 24 bytes.

6.4.4 - Exercícios

OBS.: Em todas as questões onde constar um número máximo a ser inserido, indica que esta variável DEVE ser validada.
Ex.: M < = 50 - onde não pode ser inserido um valor maior que 50.

6.4.4.1 - Seja a seguinte variável composta bidimensional A:

0	175	225	10	9000	3.7	4.75
1	9.8	100	363	432	156	18
2	40	301	30.2	6381	1	0
3	402	4211	7213	992	442	7321
4	21	3	2	1	9000	2000
	0	1	2	3	4	5

a) Quantos elementos fazem parte do conjunto?
b) Qual o conteúdo do elemento identificado por A[4][5]?
c) Qual o conteúdo de X após a execução do comando X = A[3][2] + A[4][1]?
d) O que aconteceria caso fosse referenciado o elemento A[5][2] no algoritmo?
e) Somar os elementos da quarta coluna (A[0][3] + A[1][3] + A[2][3] + A[3][3] + A[4][3]) - mediante algoritmo;
f) Somar os elementos da terceira linha (A[2][0] + A[2][1] + A[2][2] + A[2][3] + A[2][4] + A[2][5]) - mediante algoritmo;

6.4.4.2 - Dada a variável bidimensional B de 100 linhas por 200 colunas, escrever o trecho de programa que calcula o somatório dos elementos da 40ª coluna.

6.4.4.3 - Com a mesma variável composta do exercício anterior, escrever o trecho de programa que calcula o somatório dos elementos da 30ª linha.

6.4.4.4 - Faça o programa para preencher os elementos de uma matriz (5 x 3) com o valor 6. Imprimir a matriz.

6.4.4.5 - Faça um programa para ler uma matriz (3 x 4) e multiplique os elementos pares por 3 (três) e os elementos ímpares por 2 (dois). Imprimir a matriz resultado.

6.4.4.6 - Faça o programa para preencher os elementos de uma matriz quadrada (6 x 6) com o valor 0 quando o valor da linha é igual ao valor da coluna, e com 1 para os demais elementos. Imprimir a matriz.

6.4.4.7 - Faça um programa para ler uma matriz de dimensões (M x N) e copiar os elementos para uma outra matriz trocando a seqüência de leitura na Segunda Matriz.
Exemplo:

A

4	2	3	1
6	6	8	7

B

7	8	6	6
1	3	2	4

6.4.4.8 - Fazer um programa para ler as variáveis compostas A e B e fazer a soma dos elementos gerando a variável composta C; portanto, nada mais é do que a soma de matrizes o conjunto C = A + B;

6.4.4.9 - Fazer um programa que:

a) Leia as duas variáveis compostas bidimensionais de dimensão M x N (M < 20, N < = 30). O valor de M e N estão perfurados no primeiro cartão. Os cartões seguintes contêm as linhas de cada uma das variáveis.
b) Calcule e imprima a soma dessas variáveis.

6.4.4.10 - Fazer um programa que:
a) Leia uma matriz A, de dimensão M x N (M < = 20, N < = 50). Os valores de M e N estão perfurados no primeiro cartão. Os cartões seguintes contêm cada um dos elementos de uma linha da matriz;
b) Determine a matriz transposta de A;
c) Imprima a matriz A e a sua transposta.

A

9	16	34
32	11	17

Transposta de A

9	32
16	11
34	17

6.4.4.11 - Fazer um programa que:
a) Leia uma matriz inteira A de M x N, onde os elementos de cada linha estão perfurados em um cartão (M < = 20, N < = 10);
b) Imprima a matriz lida;
c) Calcule e imprima uma matriz modificada B (M x N + 1), sendo que os elementos da (N + 1)-ésima coluna são formados com o produto dos elementos da mesma linha.

Exemplo:

A

2	3
4	5

B

2	3	6
4	5	20

6.4.4.12 - Uma biblioteca possui oito departamentos. Cada departamento contém 40 estantes capazes de conter cada uma, 150 livros. Supondo que o livro padrão tenha 200 páginas de 35 linhas por 60 colunas de caracteres, declarar uma variável composta capaz de conter todos os caracteres presentes nos livros da biblioteca.

6.4.4.13 - Um grupo de pessoas respondeu a um questionário composto de 10 perguntas. Cada pergunta contém cinco opções ou respostas possíveis, codificadas de 1 a 5. Cada pergunta é respondida com a escolha de apenas uma opção dentre as cinco possíveis. As respostas de cada pessoa foram perfuradas em cartões, cada cartão contendo o nome da pessoa e as 10 respostas dadas. O último cartão, utilizado como flag, contém o nome da pessoa igual a “VAZIO”. Fazer um programa para ler e imprimir os dados lidos e calcular e imprimir o número de pessoas que responderam a cada uma das cinco opções de cada pergunta.

6.4.4.14 - Fazer um programa que leia e imprima uma variável composta bidimensional cujo conteúdo é a população dos 10 (dez) municípios mais populosos de cada um dos 25 estados brasileiros.

	0	1	...	9
0				
...				
24				

População [i][j]
População do j-ésimo município do i-ésimo estado

Determinar e imprimir o número do município mais populoso e o número do estado a que pertence. Considerando que a primeira coluna contém sempre a população da capital do estado, calcular a média da população das capitais dos 25 estados.

6.4.4.15 - Fazer um programa que:
a) Leia uma matriz quadrada real A, de dimensão M x M (M < = 20). O valor de M está perfurado no primeiro cartão. Os cartões seguintes contêm cada um, os elementos de uma linha da matriz;

b) Verifique se a matriz é simétrica, ou seja, se A[i][j] = A[j][i] para i, j < = M;
c) Imprima a palavra "SIMÉTRICA", se a matriz A for de simetria, e "NÃO SIMÉTRICA", caso contrário.

6.4.4.16 - A variável composta X, de N linhas por 4 colunas, contém informação sobre alunos da universidade. Os elementos da primeira, segunda, terceira e quarta colunas são respectivamente, o número de matrícula, sexo (0 ou 1), número do curso e a média geral do curso. Fazer um programa que:
a) Leia o número N de alunos (N < = 2000);
b) Leia as informações sobre os alunos;
c) Determine e imprima o número da matrícula do aluno de sexo 1, curso 153 que obteve a melhor média.

Supor a inexistência de empate.

6.4.4.17 - Fazer um programa que efetue um produto matricial. Seja (M x N) e B (N x M) as matrizes fatores, sendo M < = 40 e N < = 70. Deverão ser impressas as matrizes A, B e a matriz produto obtida.

6.4.4.18 - A tabela dada, a seguir, contém vários itens que estocados em vários armazéns de uma companhia. É fornecido, também, o custo de cada um dos produtos armazenados.

	Produto 1 (unid.)	Produto 2 (unid.)	Produto 3 (unid.)
Armazém 1	1200	3700	3737
Armazém 2	1400	4210	4224
Armazém 3	2000	2240	2444

Custo (R$)	260	420	330

Fazer um programa que:
a) Leia o estoque inicial;
b) Determine e imprima quantos itens estão armazenados em cada armazéns;
c) Qual o armazém que possui a maior quantidade de produto 2 armazenado.
d) O custo total de cada produto em cada armazém; estoque em cada armazém; cada produto em todos os armazéns.

6.4.4.19 - Ler uma matriz e imprimir os elementos da diagonal principal.

6.4.4.20 - Leia uma matriz M x N (M < = 10 e N < = 20), e em seguida ordene-a em ordem crescente. Ao final, imprima esta matriz, ordenada.

6.4.4.21 - Ler uma matriz e somar o valor de uma constante, lida do teclado, à matriz triangular superior. Imprimir a matriz resultante.

6.4.4.22 - Calcular o produto de uma matriz A x B. Obs.: Verifique as dimensões.

6.4.4.23 - Ler uma matriz A[10][10], somar os elementos de cada uma das linhas e armazenar em um vetor V[10]. Imprimir a matriz A, o vetor V e a soma dos elementos do vetor.
Obs.: Realizar toda a impressão ao final.

6.4.4.24 - Ler uma matriz A e calcular a soma dos elementos da diagonal principal.

6.4.4.25 - Fazer um programa que:
a) Leia duas matrizes quadradas A e B com M < = 10;
b) Apresente um menu ao usuário que possibilite a escolhas das seguintes opções: soma (em uma matriz C), multiplicação (em uma matriz C) e ordenação (da matriz A);
c) Realize a opção desejada pelo usuário e imprima.

UNIDADE 7 – STRINGS E CARACTERES

7.1 Definição

Os caracteres são peças fundamentais dos programas-fonte. Cada programa é composto de uma seqüência de caracteres que – ao serem agrupados de uma forma significativa – são interpretados pelo computador como uma série de instruções usadas para realizar uma tarefa. Um programa pode conter *constantes de caracteres*. Uma constante de caractere é um valor inteiro representado como um caractere entre aspas simples. O valor da constante de caracteres é o valor inteiro do caractere no conjunto de caracteres do equipamento. Por exemplo, **'z'** representa o valor inteiro do **z**, e **'\n'** representa o valor inteiro de uma nova linha.
Uma string é uma série de caracteres tratada como uma unidade simples. Uma string pode incluir letras, dígitos e vários *caracteres especiais* ***como*** **+, -, *, /, $** e outros.
Ou seja, strings são vetores de caracteres. Devemos apenas ficar atentos para o fato de que as strings têm o seu último elemento como um '\0' (nulo). A declaração geral para uma string é:

char nome_da_string [tamanho];

7.1.1 Observações Importantes

- Ao declarar um vetor de caracteres para conter uma string, o vetor deve ser suficientemente grande para armazenar a string e seu caractere **NULL** de terminação.
- Devemos lembrar que o tamanho da string deve incluir o '\0' final. Por essa razão é necessário declarar matrizes de caracteres como sendo um caractere mais longo que a maior string que elas devem guardar.
- Uma string pode ser atribuída a um vetor de caracteres usando **scanf**. Porém, a função **scanf()** lerá os caracteres até que **um espaço ou um indicador de nova linha ou de fim de linha** seja encontrado. Para realizar a leitura de strings que envolvem sentenças (frases) utilizamos o comando gets() explicado logo a seguir.

7.2 Funções para Manipulação de Strings

As funções apresentadas nestas seções estão no arquivo cabeçalho **string.h** (exceto função gets).

A biblioteca padrão do C possui diversas funções que manipulam strings. Estas funções são úteis pois não se pode, por exemplo, igualar duas strings:

```
string1=string2;        /* NAO faca isto */
```

Logo, as strings devem ser igualadas elemento a elemento.

Quando vamos fazer programas que tratam de string muitas vezes podemos fazer bom proveito do fato de que uma string termina com '\0' (isto é, o número inteiro 0). Veja, por exemplo, o programa abaixo que serve para igualar duas strings (isto é, copia os caracteres de uma string para o vetor da outra):

```
#include <stdio.h>
void main ()
{
      int cont;
      char str1[100],str2[100];

      ....          /* Aqui o programa le str1 que sera copiada para str2 */
```

```
        Para (cont=0; str1[cont]; cont++)
              str2[cont]=str1[cont];
        str2[cont]='\0';
....          /* Aqui o programa continua */

}
```

A condição no loop **Para** acima é baseada no fato de que a string que está sendo copiada termina em '\0'. Quando o elemento encontrado em **str1[count]** é o '\0', o valor retornado para o teste condicional é falso (nulo). Desta forma a expressão que vinha sendo verdadeira (não zero) continuamente, torna-se falsa.

7.2.1 Get String (gets)

A função **gets()** lê os caracteres do dispositivo padrão de entrada (teclado) para o seu argumento – um vetor do tipo **char** – até que um caractere de nova linha ou o indicador de fim de arquivo seja encontrado. Um caractere **NULL** (**'\0'**) será adicionado ao vetor quando a leitura terminar. Sua forma geral é:

gets (nome_da_string);

O programa abaixo demonstra o funcionamento da função **gets()**:

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

Como o primeiro argumento da função **printf()** é uma string também é válido fazer:

```
printf (string);  /* isto simplesmente imprimirá a string. */
```

7.2.2 String Copy (strcpy)

Sua forma geral é:

strcpy (string_destino,string_origem);

A função **strcpy()** copia a string-origem para a string-destino. A string-destino deve ser grande o suficiente para armazenar a string origem e seu caractere **NULL** de terminação que também é copiado.

Exemplo:

```
#include <stdio.h>
#include <string.h>
void main ()
{
        char str1[100],str2[100],str3[100];
        printf ("Entre com uma string: ");
        gets (str1);
```

```
        strcpy (str2,str1); /* Copia str1 em str2 */
        strcpy (str3,"Voce digitou a string "); /* Copia "Voce digitou a string" em str3 */
        printf ("\n\n%s%s",str3,str2);
}
```

7.2.3 String Concatenate (strcat)

A função **strcat()** tem a seguinte forma geral:

strcat (string_destino,string_origem);

Ela anexa seu segundo argumento – uma string – ao seu primeiro argumento – um vetor de caracteres contendo uma string. O primeiro caractere do Segundo argumento substitui o **NULL** (**'\0'**) que termina a string no primeiro argumento. O programador deve se assegurar de que o vetor usado para armazenar a primeira string, a segunda string e o caractere **NULL** de terminação (copiado da segunda string).
A string de origem permanecerá inalterada e será anexada ao fim da string de destino.

Exemplo:

```
#include <stdio.h>
#include <string.h>
void main ()
{
        char str1[100],str2[100];
        printf ("Entre com uma string: ");
        gets (str1);
        strcpy (str2,"Voce digitou a string ");
        strcat (str2,str1);  /* str2 armazenara' Voce digitou a string + o conteudo de str1 */
        printf ("\n\n%s",str2);
}
```

7.2.4 String Length (strlen)

Sua forma geral é:

strlen (string);

A função **strlen()** retorna o comprimento da string fornecida. O terminador nulo não é contado. Isto quer dizer que, de fato, o comprimento do vetor da string deve ser um a mais que o inteiro retornado por **strlen()**. Um exemplo do seu uso:

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

```
        printf ("\n\nA string que voce digitou tem tamanho %d",size);
}
```

7.2.5 String Compare (strcmp)

Sua forma geral é:

strcmp (string1,string2);

A função **strcmp()** compara a primeira string com a segunda, caractere por caractere. A função retorna 0 (zero) se as strings forem iguais, um valor negativo se a primeira string for menor do que a segunda e um valor positivo se a primeira string for maior do que a segunda.
Observação: Lembre-se de que **strcmp()** retorna falso se as strings são iguais. Assegure-se de usar o operador ! para reverter a condição (Exemplo 02), se você estiver testando igualdade.

Exemplo 01:

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

Exemplo 02:

```
#include<stdio.h>
#include<conio.h>
void main()
{
        char s1[80], s2[80];
        gets(s1);
        gets(s2);
        printf(“Comprimentos: %d %d\n”, strlen(s1), strlen(s2));
        if(!strcmp(s1,s2)) printf(“As strings são iguais”);
        strcat(s1, s2);
        printf(“%s\n”, s1);
        strcpy(s1, “Isso é um teste.\n”
        printf(s1);
}
```

7.2.6 String Lower (strlwr)

Protótipo:

*char strlwr(char *str);*

A sua forma geral é:

strlwr(string);

A função **strlwr()** converte a string apontada por *str* em letras minúsculas. Ela retorna *str*.

7.2.7 String Upper (strupr)

Protótipo:

*char *strupr(char *str);*

A sua forma geral é:

strupr(string);

A função **strupr()** converte a string apontada por *str* em letras maiúsculas. Ela retorna *str*.

7.2.8 String Reverse (strrev)

Protótipo:

*char *strrev(char *str);*

A sua forma geral é:

strrev(string);

A função **strrev()** inverte todos os caracteres, exceto o terminador nulo, na string apontada por *str*. Ela retorna *str*.

7.2.9 String Set (strset)

Protótipo:

*char *strset(char *str, int ch);*

A sua forma geral é:

strset(string, 'caracter');

A função **strset()** configura todos os caracteres na string apontada por *str* para o valor de *ch*. Ela retorna *str*.

7.3 Funções para Manipulação de Caracteres

A biblioteca de manipulação de caracteres ctype.h inclui várias funções que realizam testes úteis e manipulações de dados de caracteres. Cada função recebe UM caractere – representado como um **int** – ou **EOF** como argumento.

7.3.1 Função tolower

A função tolower converte uma letra maiúscula em uma letra minúscula e retorna a letra minúscula. Se o argumento não for uma letra maiúscula, **tolower** retorna o argumento inalterado.

7.3.2 Função toupper

A função toupper converte uma letra minúscula em uma letra maiúscula e retorna a letra maiúscula. Se o argumento não for uma letra minúscula, **toupper** retorna o argumento inalterado.

7.3.3 Outras Funções

isdigit(int c)	Retorna um valor verdadeiro se **c** for um dígito e 0 (falso) em caso contrário
isalpha(int c)	Retorna um valor verdadeiro se **c** for uma letra e 0 em caso contrário
isalnum(int c)	Retorna um valor verdadeiro se **c** for um dígito ou uma letra e 0 em caso contrário
isxdigit(int c)	Retorna um valor verdadeiro se **c** for um caractere de dígito hexadecimal e 0 em caso contrário
islower(int c)	Retorna um valor verdadeiro se **c** for uma letra minúscula e 0 em caso contrário
isupper(int c)	Retorna um valor verdadeiro se **c** for uma letra maiúscula e 0 em caso contrário
isspace(int c)	Retorna um valor verdadeiro se **c** for um caractere de espaço em branco – nova linha (‘\n’), espaço (‘ ’), avanço de folha (‘\f’), carriage return (‘\r’), tabulação horizontal (‘\r’) ou tabulação vertical (‘\v’) – e 0 em caso contrário
iscntrl(int c)	Retorna um valor verdadeiro se **c** for um caractere de controle e 0 em caso contrário
ispunct(int c)	Retorna um valor verdadeiro se **c** for um caractere imprimível diferente de um espaço, um dígito ou uma letra e 0 em caso contrário
isprint(int c)	Retorna um valor verdadeiro se **c** for um caractere imprimível incluindo espaço (‘ ’) e 0 em caso contrário
isgraph(int c)	Retorna um valor verdadeiro se **c** for um caractere imprimível diferente de espaço (‘ ’) e 0 em caso contrário

7.4 Limpeza do Buffer do Teclado

O teclado de um computador funciona da seguinte forma: conforme as teclas são pressionadas, os códigos das mesmas são armazenados, de forma seqüencial, no *buffer* do teclado que possui capacidade para armazenamento de **16 teclas**; sempre que possível, a CPU do computador retira do *buffer* a tecla que está **armazenada a mais tempo**; no caso da CPU estar ocupada com outras tarefas, e no caso de serem digitas mais teclas do que a capacidade do *buffer*, um sinal sonoro é enviado informando que o *buffer* está cheio (16 teclas armazenadas).

Mediante o seguinte programa:

```
include <stdio.h>
void main(){
  int numero;
  char palavra[9];
  int i=-1;
```

```
  printf("\nEntre com uma palavra(maximo 8 caracteres) <ENTER>: ");
  do{
    i++,
    palavra[i]=getchar();
  }while ((palavra[i] != 10)&&(i<7));
  palavra[i+1] = '\0';
  printf("\nA palavra digitada foi %s",palavra);
  printf("\nEntre com um numero <ENTER>: ");
  scanf("%d",&numero);
  printf("\nO numero digitado foi %d",numero);
}
```

Caso haja a inserção de uma palavra com mais de 8 caracteres, que é o tamanho máximo de **palavra**, o texto digitado "a mais" ficará guardado no *buffer do teclado* e conseqüentemente será utilizado para atribuir um valor a **numero** no comando **scanf()** logo em seguida, gerando um erro.
Para que este erro não ocorra, é necessária a utilização do comando **fflush(stdin)** para que seja feita esta "limpeza".
Com o término normal do programa ou quando os *buffers* estão cheios, os mesmos são automaticamente esvaziados, porém, no programa anterior, precisamos realizar esta operação antes do comando de leitura de **numero**. Então inserimos a linha "fflush(stdin);" antes da linha de leitura.

Explanação do comando:

Inicialmente é necessário entender o conceito de Streams:

O sistema de arquivos de C é projetado para trabalhar com uma ampla variedade de dispositivos, incluindo terminais, acionadores de disco e acionadores de fita.
Embora cada um dos dispositivos seja muito diferente, o sistema de arquivo com buffer transforma-os em um dispositivo lógico chamado de stream. Todas as streams comportam-se de forma semelhante. Pelo fato de as streams serem amplamente independentes do dispositivo, a mesma função pode escrever em um arquivo em disco ou em algum outro dispositivo, como o console.

As Streams Padrão

Sempre que um programa em C começa a execução, três streams são abertas automaticamente. Elas são a entrada padrão (**stdin**), a saída padrão (**stdout**) e a saída padrão de erro (**stderr**). Normalmente, essas streams referem-se ao console, mas podem ser redirecionadas pelo sistema operacional para algum outro dispositivo em ambientes que suportam redirecionamentos de E/S.
Em geral, **stdin** é utilizada para ler do console e **stdout** e **stderr** , para escrever no console.
Tenha em mente que **stdin**, **stdout** e **stderr** não são variáveis no sentido normal e não podem receber nenhum valor. Além disso, da mesma maneira que são criados automaticamente no início do seu programa, os ponteiros são fechados automaticamente no final; você não deve tentar fecha-los.

O comando fflush

Biblioteca: stdio.h
Tipo: int flush(FILE *stream);

Se stream aponta para um arquivo de entrada, o conteúdo do buffer de entrada é esvaziado. Caso a stream seja associada a um arquivo aberto para escrita, uma chamada a fflush() escreve fisicamente no arquivo o conteúdo do buffer de saída. Nos dois casos o arquivo continua aberto.

Exemplo Corrigido:

```
include <stdio.h>
void main(){
```

```
  int numero;
  char palavra[9];
  int i=-1;
  printf("\nEntre com uma palavra(maximo 8 caracteres) <ENTER>: ");
  do{
    i++,
    palavra[i]=getchar();
  }while ((palavra[i] != 10)&&(i<7));
  palavra[i+1] = '\0';
  fflush(stdin);
  printf("\nA palavra digitada foi %s",palavra);
  printf("\nEntre com um numero <ENTER>: ");
  scanf("%d",&numero);
  printf("\nO numero digitado foi %d",numero);
}
```

UNIDADE 8 - FUNÇÕES

8.1 - Introdução

A maioria dos programas de computador que resolvem problemas do mundo real é muito maior do que os algoritmos que vimos até agora. A experiência tem mostrado que a melhor maneira de desenvolver e manter um programa grande é construí-lo a partir de pequenas partes ou *módulos*, sendo cada uma delas mas fácil de manipular do que o programa original.
Os módulos em C são chamados *funções*. Geralmente, os programas em C são escritos combinando novas funções que o programador escreve com funções "prontas" disponíveis na *biblioteca padrão do C* (*C standard library*).

8.2 - Funcionamento

O programador pode escrever funções para definir tarefas específicas e que podem ser utilizadas em muitos locais dos programas. Algumas vezes elas são chamadas *funções definidas pelo programador*. As instruções reais que definem a função são escritas apenas uma vez e são escondidas das outras funções.
As funções são *ativadas* (*chamadas* ou *invocadas*) por uma *chamada de função*. A chamada de função especifica o nome da função e fornece informações (como *argumentos*) de que a referida função necessita para realizar a tarefa designada.

FUNÇÕES

Conjunto de comandos agrupados em um bloco que recebe um nome e através deste pode ser ativado.

- Melhor maneira de desenvolver e manter um programa grande é construí-lo a partir de pequenas partes ou *módulos*.
- Os módulos em C são chamados de *funções*.
- As funções permitem ao programador modularizar um programa.
- Todas as variáveis declaradas em definições de funções são variáveis locais.
- Variável local: Uma variável local tem validade apenas dentro do bloco onde é declarada, isto significa que podem ser apenas acessadas e modificadas dentro de um bloco. O espaço de memória alocado para esta variável é criado quando a execução do bloco é iniciada e destruído quando encerrado, assim variáveis de mesmo nome mas declaradas em blocos distintos, são para todos os efeitos, variáveis distintas.
- Variáveis Globais: É dita global, se for declarada fora do bloco de uma função. Uma variável global tem validade no escopo (por **escopo de uma variável** entende-se o bloco de código onde esta variável é válida) de todas as funções, isto é, pode ser acessada e modificada por qualquer função. O espaço de memória alocado para esta variável é criado no momento de sua declaração e destruído apenas quando o programa é encerrado.

8.2.1 Porque usar funções?

- Para permitir o reaproveitamento de código já construído (por você ou por outros programadores);
- Para evitar que um trecho de código que seja repetido várias vezes dentro de um mesmo programa;
- Para permitir a alteração de um trecho de código de uma forma mais rápida. Com o uso de uma função é preciso alterar apenas **dentro** da função que se deseja;

- Para que os blocos do programa não fiquem grandes demais e, por conseqüência, mais difíceis de entender;
- Para facilitar a leitura do programa-fonte.
- Para separar o programa em partes(blocos) que possam ser logicamente compreendidos de forma isolada.

Exemplo: Achar o maior entre dois valores reais digitados pelo usuário:

Corpo da função **antes** do programa principal (SEM PROTOTIPAÇÃO)

```
#include <conio.h>
#include <stdio.h>
float max(float a, float b){        // funcao max()
      if(a > b){
            return(a);
      }else{
            return(b);
      }
}

void main(){                  // programa principal
      float num1,num2,m;
      clrscr();
      puts("*** Valor maximo de dois reais ***");
      puts("Digite dois numeros:");
      scanf("%f %f",&num1,&num2);
      m = max(num1,num2);           // chamada da funcao
      printf("O maior valor e': %f",m);
      getch();
}
```

Corpo da função **depois** do programa principal (COM PROTOTIPAÇÃO)

```
#include <conio.h>
#include <stdio.h>
void main(){            // programa principal
      float max(float,float);   // prototipo da funcao max()
      float num1, num2;
      clrscr();
      puts("*** Valor maximo de dois reais ***");
      puts("Digite dois numeros");
      scanf("%f %f",&num1,&num2);
      printf("O maior valor e': %f",max(num1,num2));  // chamada a funcao
      getch();
}

float max(float a, float b){  // funcao max()
      if(a > b){
            return(a);
      }else{
            return(b);
      }
}
```

8.2.2 Erros Comuns de Programação

- Omitir o tipo-do-valor-de-retorno em uma definição de função causa um erro de sintaxe se o protótipo da função especificar um tipo de retorno diferente de **int**.
- Esquecer de retornar um valor de uma função que deve fazer isso pode levar a erros inesperados. O padrão ANSI estabelece que o resultado dessa omissão não é definido.
- Retornar um valor de uma função cujo tipo de retorno foi declarado **void** causa um erro de sintaxe.
- Declarar parâmetros da função do mesmo tipo como **float x, y** em vez de **float x, float y**. A declaração de parâmetros **float x, y** tornaria na realidade **y** um parâmetros do tipo **int** porque **int** é o default.
- Colocar um ponto-e-vírgula após o parêntese direito ao encerrar a lista de parâmetros de uma definição de função é um erro de sintaxe.
- Esquecer o ponto-e-vírgula no final de um protótipo de função causa um erro de sintaxe.
- Definir um parâmetro de função novamente como variável local dentro da função é um erro de sintaxe.
- Definir uma função dentro de outra função é um erro de sintaxe.

8.2.3 Funções Matemáticas

Para evitar "reinventar a roda", aconselha-se que o acadêmico tenha um bom conhecimento das principais funções da ferramenta/linguagem utilizada.
O padrão ANSI/ISO C/C++ define 22 funções matemáticas que se dividem nas seguintes categorias:

- Funções trigonométricas
- Funções hiperbólicas
- Funções exponenciais e logarítmicas
- Diversas

Todas as funções matemáticas requerem que o cabeçalho **<math.h>** seja incluído em qualquer programa que as utiliza. Além de declarar as funções matemáticas, esse cabeçalho define três macros (será visto o conceito de macro no capítulo 9) chamadas **EDOM, ERANGE** e **HUGE_VAL**.
As funções da biblioteca matemática permitem ao programador realizar determinados cálculos matemáticos comuns.

Funções Matemáticas definidas pelo padrão ANSI/ISO C/C++	
double acos(double arg)	double frexp(double num, int *exp)
double asin(double arg)	double ldexp(double num, int exp)
double atan(double arg)	double log(double num)
double atan2(double y, double x)	double log10(double num)
double ceil(double num)	double modf(double num, double *i)
double cos(double arg)	double pow(double base, double exp)
double cosh(double arg)	double sin(double arg)
double exp(double arg)	double sinh(double arg)
double fabs(double num)	double sqrt(double num)
double floor(double num)	double tan(double arg)
double fmod(double x, double y)	double tanh(double arg)

8.2.3 Exercícios:

1. Escreva uma algoritmo utilizando "função"que retorne o menor número entre três números de ponto flutuante.

2. Escreva um programa que determine e imprima todos os números perfeitos entre 1 e 10.000, mediante utilização de função. Imprima os fatores de cada número encontrado para confirmar que ele é realmente perfeito.

Sugestão: Escreva uma função **perfeito** que determine se o parâmetro **numero** é um número perfeito.

3. Mostre o valor de x após a execução de cada uma das seguintes instruções:
    a) x = fabs(7.5)
    b) x = floor(7.5)
    c) x = fabs(0.0)
    d) x = ceil(0.0)
    e) x = fabs(-6.4)
    f) x = ceil(-6.4)
    g) x = ceil(-fabs(-8+floor(-5.5)))

4. Escreva um programa que leia vários números e use a instrução a seguir para arredondar cada um desses números para o inteiro mais próximo. Para cada número processado, imprima o número original e o número arredondado.

y = floor (x + .5);

Sugestão:

- Os computadores estão desempenhando um papel cada vez maior em educação. Escreva um programa que ajudará os alunos da escola do primeiro grau a aprender a multiplicar. Use **rand** para produzir dois inteiros positivos de um dígito. O programa deve então imprimir uma pergunta do tipo "**Quanto é 6 vezes 7?".** O aluno deve digitar a resposta. Seu programa deve imprimir **"Muito Bem!"** e fazer outra pergunta de multiplicação. Se a resposta estiver errada, o programa deve imprimir **"Não. Tente novamente, por favor."** e então deixar que o aluno fique tentando acertar a mesma pergunta repetidamente até por fim conseguir.
- Modifique o programa anterior de forma que sejam impressos vários comentários para cada resposta correta e incorreta, como se segue:
    - Comentários para uma resposta correta:
        - Muito Bem!
        - Excelente!
        - Bom Trabalho!
        - Certo. Continue assim!

    - Comentários para uma resposta incorreta
        - Não. Tente novamente, por favor.
        - Errado. Tente mais uma vez.
        - Não desista!
        - Não. Continue tentando.

- Modifique o programa anterior para que ele calcule a porcentagem de resposta corretas.

8.3 DEFINIÇÃO DE FUNÇÕES

- Se a função não retorna nenhum valor para o programa que a chamou devemos definir o retorno como **void**, ou seja, um retorno ausente.
- Se nenhum tipo de retorno for especificado o compilador entenderá que o retorno será tipo **int**.
- Em C existe apenas **UM** valor de retorno para funções, ou seja, não se pode retornar dois ou mais valores.
- Porém, isto não é uma limitação séria, pois o uso de ponteiros contorna o problema.
- O comando **return()** tem duas finalidades:
    - Determina o fim lógico da rotina, e
    - Valor de retorno da função.

8.3.1 CHAMANDO FUNÇÕES

- **Por Valor**: é feita uma cópia dos argumentos e a mesma é passada para a função chamada. As modificações na cópia não afetam o valor original de uma variável na função que realizou a chamada. Deve ser usada sempre que a função chamada não precisa modificar o valor da variável original da função chamadora.
- **Por Referência**: A função chamadora permite realmente que a função chamada modifique o valor original da variável. Só deve ser usada com funções confiáveis que precisem modificar a variável original.

8.3.2 REGRAS DE ESCOPO

- Um **escopo** de um identificador é a parte do programa na qual ele pode ser referenciado. Por exemplo, quando declaramos uma variável local em um bloco, podem ser feitas referências a ela apenas naquele bloco ou em blocos ali aninhados.
- Os quatro escopos de um identificador são escopo de: função, arquivo, bloco e protótipo de função.
    - **Função**: Os **rótulos** são os únicos identificadores com este escopo, pois eles podem ser usados em qualquer lugar de uma função na qual aparecem, mas não pode ser feita qualquer referência a eles fora do corpo da função.
    - **Arquivo**: Para identificadores declarados fora de qualquer função. Tal identificador é "conhecido" por todas as funções, desde o local onde é declarado até o final do arquivo. Ex.: Variáveis globais, definições de funções e protótipos de funções colocados fora de uma função.
    - **Bloco**: Para identificadores declarados dentro de um bloco. As variáveis locais declaradas no início de uma função possuem este escopo, assim como os parâmetros, que são considerados variáveis locais por ela.
    - **Protótipo de função**: Os únicos identificadores são os utilizados na lista de parâmetros de um protótipo de função.

8.3.3 CLASSES DE ARMAZENAMENTO

Uma classe de armazenamento de um identificador ajuda a determinar seu tempo de armazenamento, escopo e linkage.

- **Tempo de Armazenamento**: período durante o qual aquele identificador permanece na memória.
- **Escopo**: Onde se pode fazer referência àquele identificador dentro do programa.
- **Linkage**: Determina para um programa com vários arquivos-fonte, se um identificador é conhecido apenas no arquivo-fonte atual ou em qualquer arquivo-fonte com as declarações adequadas.

As quatro classes de armazenamento dos identificadores podem ser divididas em dois tempos de armazenamento: **Tempo de Armazenamento Automático** e **Tempo de Armazenamento Estático**.

- **Tempo de Armazenamento Automático**: São criadas quando o bloco no qual são declaradas é acionado. Elas existem enquanto o bloco estiver ativo e são destruídas quando o bloco é deixado para trás. Palavras-chave: **auto** e **register**.
    - **Auto**: As variáveis locais possuem tempo de armazenamento automático por default, portanto a palavra-chave **auto** raramente é usada.
    - **Register**: Pode ser colocado antes de uma declaração de variável automática para indicar ao compilador que a variável seja conservada em um dos registradores de hardware de alta velocidade do computador. Freqüentemente usado com contadores ou para cálculo de totais.
- **Tempo de Armazenamento Estático**: Existem desde o instante em que o programa começou a ser executado. Entretanto, muito embora as variáveis e os nomes das funções existam quando o programa começa a ser executado, isso não significa que esses identificadores podem ser usados em todo o programa. Palavras-chave: **extern** e **static**.
    - **Extern**: Variáveis globais e nomes das funções são pertencentes desta classe por default.
    - **Static**: As variáveis locais declaradas com **static** são conhecidas apenas na função na qual são definidas, mas, diferentemente das variáveis automáticas, as variáveis locais **static** conservam seus valores quando a função é encerrada. Na próxima vez em que a função for chamada, a variável local **static** conservará o valor que tinha quando a função foi executada pela última vez.

8.3.4 PASSANDO MATRIZES PARA FUNÇÕES

A passagem de matrizes, como argumentos, para funções, é uma exceção à convenção de passagem de parâmetros com chamada por valor.
Quando uma matriz é usada como argumento para uma função, apenas o endereço da matriz é passado, não uma cópia da matriz inteira. Na verdade, é passado o endereço do primeiro elemento da matriz. Por ser passado o endereço inicial da matriz, a função chamada sabe precisamente onde a matriz está armazenada, portanto, quando a função chamada modifica os elementos da matriz em seu corpo de função, os elementos reais estão sendo modificados em suas posições originais.

8.3.4.1 Passando Matrizes Unidimensionais para Funções

➢ Na **passagem** de vetores para funções:

Nome_da_função (Nome_do_vetor);

Onde,

Nome_da_função é o nome da função que se está chamando; e
Nome_do_vetor é o nome da matriz unidimensional que queremos passar. APENAS o nome do vetor, SEM índices.

➢ Na **declaração** de funções que recebem vetores:

Tipo_função Nome_função (Tipo_vetor Nome_vetor[]);

Onde,

Tipo_função é o tipo de **retorno** da função;
Nome_função é o nome da função;
Tipo_vetor é o tipo de elementos do vetor; e

Nome_vetor[ ] é o nome do vetor. Declara-se um ÍNDICE VAZIO para indicar que estamos recebendo um vetor.

Atenção: Ao contrário de variáveis comuns, o conteúdo de um vetor **PODE SER MODIFICADO** pela função chamada, como visto anteriormente. Isto ocorre porque a passagem de matrizes para funções é feita de modo especial dito passagem por referência ou endereço.

8.3.4.2 Passando Matrizes Multidimensionais para Funções

- Na **passagem** de matrizes para funções:

Semelhante a passagem de vetores: chama-se a função passando o nome da matriz, **SEM ÍNDICES**.

Nome_da_função (Nome_da_matriz);

Onde,

Nome_da_função é o nome da função que se está chamando; e
Nome_da_matriz é o nome da matriz multidimensional que queremos passar. APENAS o nome do vetor, SEM índices.

- Na **declaração** de funções que recebem matrizes:

Tipo_função Nome_função (Tipo_matriz Nome_matriz[tam_1][tam_2]...[tam_n]);

Onde,

Tipo_função é o tipo de **retorno** da função;
Nome_função é o nome da função;
Tipo_matriz é o tipo de elementos da matriz; e
Nome_matriz[ ] é o nome da matriz. É preciso declarar os índices contendo os tamanhos de cada dimensão da matriz.

Exemplo: O programa abaixo demonstra a diferença entre passagem de parâmetros por valor e por referência mediante a utilização de matrizes.

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

#define N 5

void imprime_vetor(int []); //prototipação das funções, pelo fato delas estarem localizadas abaixo do main.
void modifica_vetor(int []);
void modifica_elemento(int);


void main()
{
	int a[N] = {0, 1, 2, 3, 4}; //inicialização do vetor

	printf("\nVetor Original: \n\n");
	imprime_vetor(a); //chama a função para a impressão do vetor, passando ele como argumento
```

```
	modifica_vetor(a); //chama a função para modificar os elementos do vetor
	printf("\nValores modificados do vetor: \n\n");
	imprime_vetor(a); //imprime os elementos agora modificados
	printf("\n\nO valor de a[3] é %d\n", a[3]);
	modifica_elemento(a[3]); // “modifica” um elemento isolado do vetor, porém este é passado por valor
	printf("\n\nO valor de a[3] é %d\n", a[3]); /*imprime o elemento não modificado, pois foi feita uma
passagem por valor, que faz uma cópia deste valor e atribuí a variável e que é tida como local na função
modifica_elemento*/
}

void imprime_vetor(int b[]) //recebe um vetor inteiro b
{
	for(int i=0; i<N; i++)
		printf("%3d  ", b[i]); //imprime cada elemento isolado
	printf("\n");
}

void modifica_vetor(int b[]) //recebe um vetor inteiro b
{
	for(int i=0; i<N; i++)
		b[i] *= 2; //modifica cada elemento multiplicando ele mesmo por 2
}

void modifica_elemento(int e) //recebe uma cópia do elemento passado como argumento para a função.
{
	printf("\n\nValor em Modifica_Elemento é %d", e*=2); //tenta modificar o elemento
}
```

8.4 A instrução return

A instrução return tem duas utilizações importantes. Primeiro, provoca uma saída imediata da função. Isto é, faz com que a execução do programa retorne para o código de chamada. Segundo, pode ser utilizada para retornar um valor.

8.4.1 Retornando de uma função

Uma função termina a execução e retorna para o chamador de uma de duas maneiras. A primeira é quando a última instrução na função executou e, conceitualmente, a chave de término (}) da função é encontrada. Por exemplo, essa função aceita um endereço para uma string como um parâmetro e exibe a string de trás para frente:

```
void inverte(char texto[])
{
	for(int i = strlen(texto)-1; i >= 0;  i--) printf(“%c”, texto[i]);
}
```

Uma vez que a string foi exibida, não há nada para inverte() fazer, então ela retorna para o lugar de onde foi chamada.
Realmente, não muitas funções utilizam esse método padrão de terminar sua execução. A maioria das funções conta com a instrução **return** para interromper a execução porque um valor deve ser retornado ou fazer o código de uma função mais simples e mais eficiente.

8.4.2 Retornando valores

Todas as funções, exceto aquelas do tipo **void**, retornam um valor. Esse valor é especificado pela instrução **return**. Em C++, uma função não-**void** *deve* utilizar uma instrução **return** que retorna um valor. Contanto que uma função não seja declarada como **void**, você pode utiliza-la como um operando em uma expressão. Porém, como uma regra geral, uma chamada de função não pode estar no lado esquerdo de uma atribuição. Uma instrução como

*troca(x, y) = 100/ /*instrução incorreta */*

está errada. O compilador irá sinaliza-la como um erro e não compilará um programa que a contiver.
Se uma função for declarada como **void**, não pode ser utilizada em qualquer expressão.
Quando vocês escreve programas, suas funções serão de três tipos. O primeiro tipo é simplesmente computacional. Essas funções são especificamente projetadas para realizar operações em seus argumentos e retornar um valor com base nessa operação. Uma função computacional é uma função "pura". Os exemplos são as funções de biblioteca padrão **sqrt()** e **sin()**, que calculam a raiz quadrada e o seno de seus argumentos.
O segundo tipo de função manipula as informações e retorna um valor que simplesmente indica o sucesso ou falha dessa manipulação. Um exemplo é a função de biblioteca **fwrite()**, que grava as informações em um arquivo de disco. Se a operação de gravação for bem-sucedida, **fwrite()** retorna o número de itens gravados com sucesso. Se um erro ocorrer, **fwrite()** retornará um número que não é igual ao número de itens que foi solicitado para gravar.
O último tipo de função não tem nenhum valor explícito de retorno. Em essência, a função é estritamente procedural e não produz nenhum valor. Um exemplo é **srand()**, que inicializa a função aleatória de gerador de número **rand()**. Às vezes, as funções que realmente não produzem um resultado interessante freqüentemente retornam algo de qualquer jeito..

8.4.3 O que main() retorna?

A função **main()** retorna um inteiro para o processo de chamada, que é geralmente o sistema operacional. Retornar um valor de **main()** é equivalente a chamar **exit()** com o mesmo valor. Um valor de retorno de 0 indica que o programa terminou normalmente. Todos os outros valores indicam que algum erro ocorreu com a saída do programa.

8.5 RECURSIVIDADE

Definição

Recursão é o processo pelo qual uma função chama a si mesma repetidamente um número finito de vezes.
Função Recursiva é uma função que chama a si mesma, ou diretamente ou indiretamente por meio de outra função.

Características

Os métodos recursivos para solução de problemas possuem vários elementos em comum. Uma função recursiva é chamada para resolver o problema. Na realidade, a função só sabe resolver o(s) caso(s) mais simples, ou o(s) chamado(s) caso(s) básico(s).
Se a função for chamada em um problema mais complexo, ela divide o problema em duas partes teóricas:
- um aparte que a função sabe resolver;
- e outra que ela não sabe.

Funcionamento

Para tornar viável a recursão, a segunda parte deve ser parecida com o problema original, mas ser uma versão um pouco mais simples ou menor do que ele.
Por esse novo problema ser parecido com o problema original, a função chama (lança) uma cópia de si mesma para lidar com o problema menor.
A etapa de recursão é executada enquanto a chamada original para a função ainda não tiver sido concluída. Ela pode levar a outras chamadas recursivas, à medida que a função continuar a dividir cada problema em duas partes conceituais.

Desvantagens

A maioria das funções recursivas não minimiza significativamente o tamanho do código nem melhora a utilização da memória. Além disso, as versões recursivas da maioria das rotinas podem ser executadas um pouco mais lentamente que suas equivalentes iterativas devido às repetidas chamadas à função (seqüência de overhead). Isso pode custar caro tanto em tempo de processador como em espaço de memória.

Vantagens

A principal vantagem das funções recursivas é que você pode usa-las para criar versões mais claras e simples de vários algoritmos.

Observações

Aos escrever funções recursivas, você deve ter um comando if em algum lugar para forçar a função a retornar sem que a chamada recursiva seja executada. Se você não o fizer, a função nunca retornará quando chamada.

Recursão x Iteração

RECURSÃO	ITERAÇÃO
Usa estrutura de seleção	Usa estrutura de repetição
Obtém repetição por intermédio de chamadas repetidas de funções	Usa explicitamente uma estrutura de repetição
Termina quando um caso básico é reconhecido	Termina quando uma condição de continuação do loop se torna falsa
Permanece produzindo versões mais simples do problema original até chegar no caso básico	Permanece modificando um contador até que ele assuma um valor que faça a condição de continuação do loop se tornar falsa
Ocorre um loop infinito se a etapa de recursão não reduzir gradualmente o problema de forma que ele convirja para o caso básico	Ocorre um loop infinito se o teste de continuação do loop nunca se tornar falso

8.6 Exercícios

1 Crie um função `float round(float r)` que faça o arredondamento de números reais: Por exemplo: `5 = round(5.4), 7 = round(6.5)`.

2 Crie uma função `int sim_nao(void)` que espera o usuário pressionar as teclas `[s]` ou `[n]` retornando 1 ou 0 respectivamente. Se o usuário pressionar qualquer outra tecla um som (de advertência) de *50 Hz* por *250 ms* é emitido.

3. Faça uma função que determine se três números `a, b, c` formam um triângulo ou não. A função deve ter a seguinte declaração `int triângulo(float a, float b, float c)` onde o valor de retorno tem o seguinte significado:

```
0: não forma triângulo,
1: triângulo qualquer,
2: triângulo isósceles,
3: triângulo equilátero.
```

4. Faça uma função que determine se um determinado numero é primo ou não. A função deve ter a seguinte declaração `int primo(int N)` onde `N` é o valor a ser testado. A função deve retornar `1` se `N` é primo e `0` caso contrário.

5. Transforme o programa do fibonacci em uma função `int fib(int n)` que retorna o n-ésimo numero de Fibonacci.

6. O maior divisor comum dos inteiros positivos *a* e *b*, que abreviamos como *mdc(a, b)*, é o maior número *m* tal que *m* é divisor tanto de *a* quanto de *b*. Por exemplo: *4 = mdc(20,16)*, *7 = mdc(21,7)*. O valor de *m* pode ser calculado com o seguinte algoritmo recursivo, de *Euclides*:

```
se a > b então mdc(a, b) é igual a
b se resto(a, b) é 0
mdc(b, resto(a, b)) caso contrário.
```

Faça uma função *recursiva* para o cálculo do máximo divisor comum de dois números.

7. Caso já não tenha feito assim, transforme a função *iterativa* do exercício **5** em uma função *recursiva*.

UNIDADE 9 – O PRÉ-PROCESSADOR

Neste tópico serão apresentadas as explanações de apenas algumas diretivas, por serem muito utilizadas no decorrer do semestre e por estarem presentes no escopo da disciplina de ICC.

9.1 Definição

Você pode incluir diversas instruções do compilador no código-fonte de um programa em C. Elas são chamadas de *diretivas do pré-processador* e, embora não seja realmente parte da linguagem de programação C, expandem o escopo do ambiente de programação em C.
O pré-processamento ocorre antes de um programa ser compilado, ou seja, isto quer dizer que as diretivas de compilação são comandos que não são compilados. Portanto, o pré-processador modifica o programa fonte, entregando para o compilador um programa modificado. Algumas ações possíveis são: inclusão de outros arquivos no arquivo que está sendo compilado, definição de *constantes simbólicas* e *macros*, *compilação condicional* do código do programa e *execução condicional das diretivas do pré-processador*. Todas as diretivas do pré-processador começam com #, e somente os caracteres de espaço em branco podem aparecer antes de uma diretiva de pré-processador em uma linha. Além disso, cada diretiva do pré-processador deve estar na sua própria linha.

Não há ponto-e-vírgula após a diretiva de compilação. Esta é uma característica importante de todas as diretivas de compilação e não somente da diretiva #include.

9.2 Diretivas

Como definido pelo padrão C ANSI, o pré-processador de C contém as seguintes diretivas:

#if	#else	#include	#line
#ifdef	#elif	#define	#error
#ifndef	#endif	#undef	#pragma

9.2.1 Diretiva #include

Faz com que seja incluída uma cópia de um arquivo especificado no lugar da diretiva. As duas formas diretiva #include são:

#include <nome_do_arquivo>
#include “nome_do_arquivo”

A diferença entre essas formas é o local no qual o pré-processador procura pelo arquivo a ser incluído. Se o nome do arquivo está envolvido por chaves angulares (sinais de maior e menor), o arquivo será procurada de forma definida pelo criador do compilador. Freqüentemente, isso significa procurar em algum diretório especialmente criado para arquivos de inclusão. Se o nome do arquivo está entre aspas, o arquivo é procurado de uma maneira definida pela implementação. Para muitas implementações, isso significa uma busca no diretório de trabalho atual. Se o arquivo não for encontrado, a busca será repetida como se o nome do arquivo estivesse envolvido por chaves angulares.

A maioria dos programadores tipicamente usa chaves angulares para incluir os arquivos de cabeçalho padrão. O uso de aspas é reservado geralmente para a inclusão de arquivos do projeto. No entanto, não existe nenhuma regra que exija este uso.

9.2.2 Diretiva #define

A diretiva #define cria *constantes simbólicas* – constantes representadas por símbolos – e *macros* – operações definidas como símbolos. Ou seja, ela define um identificador e uma string que o substituirá toda vez que for encontrado no arquivo fonte. O padrão C ANSI refere-se ao identificador como um *nome de macro* e ao processo de substituição como *substituição de macro*. A sua forma geral é:

#define *nome_macro string*

Quando essa linha aparece em um arquivo, todas as ocorrências subseqüentes de *nome_macro* serão substituídas automaticamente por *string* antes de o programa ser compialdo. Por exemplo,

#define PI 3.14159

substitui todas as ocorrências subseqüentes da constante simbólica PI pela constante numérica 3.14159. As constantes simbólicas permitem que o programador crie um nome para a constante e use o nome ao longo de todo o programa. Se a constante precisar ser modificada no programa, ela pó ser modificada dentro da diretiva #define – e quando o programa voltar a ser compilado, todas as ocorrências da constante no programa serão modificadas automaticamente.
Observação: Tudo à direita do nome da constante simbólica substitui a constante simbólica.
Por exemplo,

#define PI = 3.14159

faz com que o pré-processador substitua todas as ocorrências de PI por = 3.14159. Isso é a causa de muitos erros lógicos e de sintaxe sutis. Redefinir a constante simbólica com um novo valor também configura um erro.

Uma vez com que um nome de macro tenha sido definido, ele pode ser usado como parte da definição de outros nomes de macro. Por exemplo, este código define os valores UM, DOIS e TRÊS:

```
#define     UM      1
#define     DOIS    UM+UM
#define     TRES    UM+DOIS
```

Substituição de macro é simplesmente a transposição de sua string associada. Portanto, se você quisesse definir uma mensagem de erro padrão, poderia escrever algo como isto:

```
#define MSG_E “erro padrão na entrada\n”

printf(MSG_E);
```

O compilador substituirá a string “erro padrão na entrada\n” quando o identificador **MSG_E** for encontrado. Para o compilador, o comando com o **printf()** aparecerá, na realidade, como

```
printf(“erro padrão na entrada\n”);
```

Nenhuma substituição de texto ocorre se o identificador está dentro de uma string entre aspas. Por exemplo,

```
#define XYZ isso é um teste

printf("XYZ");
```

não escreve **isso é um teste**, mas **XYZ**.

Se a string for maior que uma linha, você pode continuá-la na próxima colocando uma barra invertida no final da linha, como mostrado aqui:

```
#define STRING_LONGA "isso é uma string muito longa\
que é usada como um exemplo"
```

Os programadores C geralmente **usam letras maiúsculas para identificadores definidos**. Essa convenção ajuda qualquer um que esteja lendo o programa a saber de relance que uma substituição de macro irá ocorrer. Além disso, é melhor colocar todos os #define's no início do arquivo ou em um arquivo de cabeçalho separado em lugar de espalha-los pelo programa.
Substituição de macro é usada mais freqüentemente para definir nomes para "número mágicos" que aparecem em um programa. Por exemplo, você pode ter um programa que defina uma matriz e tenha diversas rotinas que acessam essa matriz. Em lugar de fixar o tamanho da matriz com uma constante, você poderia definir um tamanho e usar esse nome sempre que o tamanho da matriz for necessário. Dessa forma, você só precisa fazer uma alteração e recompilar para alterar o tamanho da matriz. Por exemplo,

```
#define TAMANHO_MAX 100

/* ... */

float vetor[TAMANHO_MAX];

/* ... */

for (i=0; i<TAMANHO_MAX; i++) printf("%f", vetor[i]);
```

Como TAMANHO_MAX define o tamanho da variável **vetor**, se o tamanho desta precisar ser modificado no futuro, você só precisa modificar a definição de TAMANHO_MAX. Todas as referências subseqüentes a ela serão automaticamente atualizadas quando você recompilar seu programa.

9.2.2.3 Definindo Macros Semelhantes a Funções

A diretiva #define possui outro recurso poderoso: o nome da macro pode ter argumentos. Cada vez que o nome da macro é encontrado, os argumentos usados na sua definição são substituídos pelos argumentos reais encontrados no programa. As macros podem ser definidas com ou sem argumentos. Uma macro sem argumentos é processadas como uma constante simbólica.
Considere a seguinte definição de macro com um argumento para calcular a área de um círculo:

```
#define AREA_CIRCULO(x) ( PI * (x) * (x) )
```

Sempre que AREA_CIRCULO(x) aparecer no arquivo, o valor de **x** substitui **x** no texto de substituição, a constante simbólica PI é substituída por seu valor (definido anteriormente) e a macro é expandida no programa. Por exemplo, a instrução

```
area = AREA_CIRCULO(4);
```

é expandida em

```
area = ( 3.14159 * (4) * (4) );
```

Como a expressão consiste apenas em constantes, durante a compilação, o valor da expressão é calculado e atribuído à variável **area**. Os parênteses em torno de cada **x** no texto de substituição impõem a ordem adequada de cálculo quando o argumento da macro for uma expressão. Por exemplo, a instrução

```
area = AREA_CIRCULO(c + 2);
```

é expandida em

```
area = ( 3.14159 * (c + 2) * (c + 2) );
```

que é calculada corretamente porque os parênteses impõem a ordem adequada de cálculo. Se os parênteses fossem omitidos, a expansão da macro seria

```
area = 3.14159 * c + 2 * c + 2;
```

que é calculada erradamente como

```
area = ( 3.14159 * c ) + (2 * c) + 2;
```

devido a regra de precedência de operadores.
Para efetuar este cálculo poderia ter sido definida uma função, porém o overhead de uma chamada de função. As vantagens da macro AREA_CIRCULO são que as macros inserem código diretamente no programa – evitando o overhead das funções – e o programa permanece legível porque o cálculo de AREA_CIRCULO é definido separadamente e recebe um nome significativo. Uma desvantagem é que seu argumento é calculado duas vezes.

A seguir está uma definição de macro com 2 argumentos para a área de um retângulo:

```
#define AREA_RETANGULO(x, y) ( (x) * (y) )
```

Sempre que AREA_RETANGULO(x, y) aparecer no programa, os valores de **x** e **y** são substituídos no texto de substituição da macro e a macro é expandida no lugar de seu nome. Por exemplo a instrução

```
areaRet = AREA_RETANGULO(a + 4, b + 7);
```

é expandida em

```
areaRet = ( (a + 4) * (b + 7) );
```

O valor da expressão é calculado e atribuído a variável areaRet.

ANEXO I – Operadores

Os operadores de incremento e decremento são unários que alteram a variável sobre a qual estão aplicados. O que eles faze é incrementar ou decrementar a variável sobre a qual estão aplicados, de 1.

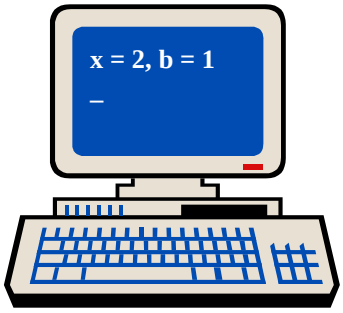
Então:	São equivalentes a:
x++;	x = x + 1;
x--;	x = x – 1;

Estes operadores podem ser pré-fixados ou pós-fixados. A diferença é que quando são pré-fixados eles incrementam e retornam o valor da variável já incrementada. Quando são pós-fixados eles retornam o valor da variável sem o incremento e depois incrementam a variável.

x++	→	incrementa, devolvendo o valor NÃO atualizado
++x	→	incrementa devolvendo o valor atualizado
x--	→	decrementa devolvendo o valor NÃO atualizado
--x	→	decrementa devolvendo o valor atualizado

Exemplo:

```
Incrementa()
{
        Inteiro x, b;
        x = 1;
        b = x++;
        Escreva("x = %I, b = %I\n", x, b);
}
```



Exemplo:

```
x = 23;
y = x++;
```
→ Teremos, no final y = 23 e x = 24

Em

```
x = 23;
y = ++x;
```
→ Teremos, no final y = 24 e x = 24

ANEXO II – Expressões

Expressões que podem ser abreviadas

O C admite as seguintes equivalências, que podem ser usadas para simplificar expressões ou facilitar o entendimento de um programa:

Expressão Original	Expressão equivalente
x = x + k;	x+=k;
x = x – k;	x–=k;
x = x * k;	x*=k;
x = x/k;	x/=k;
x = x%k;	x%=k;
Etc…	

Modeladores

Um modelador é aplicado a uma expressão. Ele "força" a mesma a ser de um tipo especificado. A sua forma geral é: *(tipo) expressão*

Exemplo:

```
Modelador()
{
        Inteiro num;
        Real f;
        num = 10;
        f = (Real)num/7;
        Escreva("%R", f);
}
```

Se não se tivesse usado o modelador neste exemplo,o C faria uma divisão inteira entre 10 e 7. O resultado seria 1 (um) e este depois seria convertido para Real, mas continuaria a ser 1.0. Com o modelador tem-se o resultado correto.

Exercícios:

1. Considere as expressões que se seguem e transforme-as em expressões lineares para poderem ser compreendidas pelo compilador C.

a) $a = \dfrac{b + \dfrac{1}{c} + c^2}{\dfrac{b}{c} \times \dfrac{2+d}{r}}$

b) $b = a * \sqrt{\dfrac{4+r}{3 \times a} + \dfrac{c^2}{2+t}} + \dfrac{1}{c}$

Notas:
- A função sqrt() devolve a raiz quadrada de um número passado como parâmetro.
- A função pow() devolve o valor de um dado número elevada a um dado expoente, ambos passados como parâmetro. Ex.: pox(x, 2) = x^2

2. Diga o resultado das variáveis depois da seguinte seqüência de operações:

a)	x	y	z
Inteiro x, y, z;			
x = y = 10;			
z = ++x;			
x = -x;			
y++;			
x = x + y – (z--);			

b)	x	y	z
Inteiro x, y, a=14, b=3;			
Real z;			
x = a / b;			
y = a % b;			
z = y / x;			

c)	v	x	y	z
Inteiro v=0, x=1, y=2, z=3;				
v += x + y;				
x *= y = z +1;				
z %= v + v + v;				
v += x += y += 2;				

3. A expressão lógica (-5 || 0) && (3 > = 2) && (1 != 0) || (3 < 0) é:
   a) Verdadeira
   b) Falsa
   c) Inválida, pois sua sintaxe está errada
   d) Nem verdadeira, nem falsa
   e) Nenhuma das opções anteriores

ANEXO III – As Instruções Break e Continue

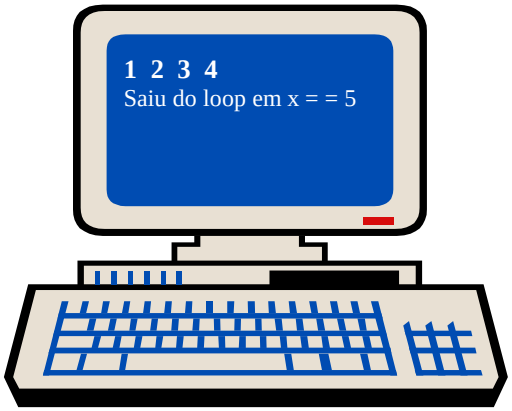
As instruções break e continue são usados para alterar o fluxo de controle.

A Instrução Break

A instrução break quando executada em uma estrutura de repetição (Para-Faça, Enquanto-Faça e Faça-Enquanto) faz com que aconteça a saída imediata daquela estrutura. Já no caso da estrutura Escolha, esta instrução quebra a execução deste comando (switch).
A execução do programa continua com a primeira instrução depois da estrutura que está sendo interrompida. Os usos comuns da instrução break são para sair prematuramente de um loop, ou para saltar sobre o restante de uma estrutura Escolha (switch).

Exemplo:

```
Inst_break()
{
        Inteiro x;
        Para(x = 1; x< = 10; x++){
              Se (x = = 5) então
                    break;
              Escreva (“%I”, x);
        }
        Escreva (“\nSaiu do loop em x = =  %I\n”, x);
}
```

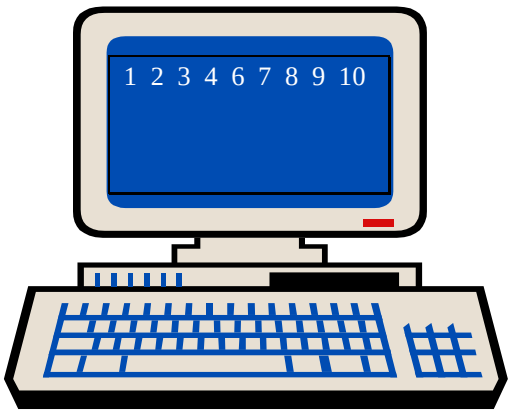


A instrução Continue

O comando continue pode ser visto como sendo o oposto do break e só funciona dentro de uma estrutura de repetição (Para-Faça, Enquanto-Faça ou Faça-Enquanto).
Este comando ignora (salta sobre) as instruções restantes no corpo daquela estrutura e realiza a próxima iteração do loop. em estruturas Enquanto-Faça e Faça-Enquanto, o teste de continuação do loop é realizado imediatamente após a instrução continue ser executada. Na estrutura Para-Faça, a expressão de incremento é executada e depois o teste de continuação do loop é realizado.
Na estrutura Enquanto-Faça, caso a expressão de incremento encontre-se após o comando continue, a mesma não é executada antes de a condição do loop ser testada e o Enquanto-Faça não é executado da mesma maneira que o Para-Faça.

Exemplo:

```
Inst_Continue()
{
        Inteiro x;
        Para(x = 1; x < = 10; x++){
              Se (x = = 5) então
                    continue;
              Escreva(“%I”, x);
```



```
    }
}
```

Exercícios

1. Sendo num uma variável inteira, o que imprime o trecho de código a seguir?

```
num = 1;
Escolha(num){
    Caso 1:
        Escreva("O número é 1");
    Caso 2:
        Escreva("O número é 2");
    Default:
        Escreva("O número é diferente de 1 e 2");
}
```

2. Os dois blocos de código a seguir produzem o mesmo resultado? Faça o mapa de memória.

a)

```
Para(i = 0; i < 3; i++){
    Para(j = 0; j < 3; j++)
        Escreva("i + j = %I\n", i+j);
}
```

b)

```
Para(i = 0, j =0; i <3; i++){
    Para(   ; j < 3; j++)
        Escreva("i + j = %I\n", i+j);
}
```

3. Qual a saída produzida pelo fragmento de código a seguir:

```
Inteiro x;
Para(x = 35; x > 0; x/=3)
    Escreva("%I", x);
```

4. Os fragmentos de código a seguir são equivalentes entre si? Faça o mapa de memória.

a)

```
Inteiro x = 10;
Enquanto (--x > 9)
    Escreva("%I", x);
```

b)

```
Inteiro x = 10;
Faça{
    Escreva("%I", x);
}Enquanto(--x > 9);
```

ANEXO IV – TABELA ASCII

As tabelas mostradas neste apêndice representam os 256 códigos usados nos computadores da família IBM. Esta tabela refere-se ao *American Standard Code for Information Interchange* (código padrão americano para troca de informações), que é um conjunto de números representando caracteres ou instruções de controle, usados para troca de informações entre computadores entre si, entre periféricos (teclado, monitor, impressora) e outros dispositivos. Estes códigos tem tamanho de 1 byte com valores de `00h` a `FFh` (0 a 255 decimal). Podemos dividir estes códigos em três conjuntos: controle, padrão e estendido.

Os primeiros 32 códigos de `00h` até `1Fh` (0 a 31 decimal), formam o **conjunto de controle** ASCII. Estes códigos são usados para controlar dispositivos, por exemplo uma impressora ou o monitor de vídeo. O código `0Ch` (*form feed*) recebido por ima impressora gera um avanço de uma página. O código `0Dh` (*carriage return*) é enviado pelo teclado quando a tecla ENTER é pressionada. Embora exista um padrão, alguns poucos dispositivos tratam diferentemente estes códigos e é necessário consultar o manual para saber exatamente como o equipamento lida com o código. Em alguns casos o código também pode representar um caracter imprimível. Por exemplo o código 01h representa o caracter ☺ (*happy face*).

Os 96 códigos seguintes de `20h` a `7Fh` (32 a 127 decimal) formam o **conjunto padrão** ASCII. Todos os computadores lidam da mesma forma com estes códigos. Eles representam os caracteres usados na manipulação de textos: códigos-fonte, documentos, mensagens de correio eletrônico, etc. São constituídos das letras do alfabeto latino (minúsculo e maiúsculo) e alguns símbolos usuais.

Os restantes 128 códigos de `80h` até `FFh` (128 a 255 decimal) formam o **conjunto estendido** ASCII. Estes códigos também representam caracteres imprimíveis porem cada fabricante decide como e quais símbolos usar. Nesta parte do código estão definidos os caracteres especiais: é, ç, ã, ü ...

Vale ressaltar que o caractere resultante a partir de sua inserção mediante a tecla Alt + valor decimal, poderá variar de acordo com o tipo de fonte utilizado.

Dec.	Hex.	Controle
0	00h	NUL (*Null*)
1	01h	SOH (*Start of Heading*)
2	02h	STX (*Start of Text*)
3	03h	ETX (*End of Text*)
4	04h	EOT (*End of Transmision*)
5	05h	ENQ (*Enquiry*)
6	06h	ACK (*Acknowledge*)
7	07h	BEL (*Bell*)
8	08h	BS (*Backspace*)
9	09h	HT (*Horizontal Tab*)
10	0Ah	LF (*Line Feed*)
11	0Bh	VT (*Vertical Tab*)
12	0Ch	FF (*Form Feed*)
13	0Dh	CR (*Carriage Return*)
14	0Eh	SO (*Shift Out*)
15	0Fh	SI (*Shift In*)
16	10h	DLE (*Data Link Escape*)
17	11h	DC1 (*Device control 1*)
18	12h	DC2 (*Device control 2*)
19	13h	DC3 (*Device control 3*)
20	14h	DC4 (*Device control 4*)
21	15h	NAK (*Negative Acknowledge*)
22	16h	SYN (*Synchronous Idle*)
23	17h	ETB (*End Transmission Block*)
24	18h	CAN (*Cancel*)
25	19h	EM (*End of Media*)
26	1Ah	SUB (*Substitute*)
27	1Bh	ESC (*Escape*)
28	1Ch	FS (*File Separator*)
29	1Dh	GS (*Group Separator*)
30	1Eh	RS (*Record Separator*)
31	1Fh	US (*Unit Separator*)

Caracter	Dec.	Hex.
<espaço>	32	20h
!	33	21h
"	34	22h
#	35	23h
$	36	24h
%	37	25h
&	38	26h
'	39	27h
(	40	28h
)	41	29h
*	42	2Ah
+	43	2Bh
,	44	2Ch
-	45	2Dh
.	46	2Eh
/	47	2Fh
0	48	30h
1	49	31h
2	50	32h
3	51	33h
4	52	34h
5	53	35h
6	54	36h
7	55	37h
8	56	38h
9	57	39h
:	58	3Ah
;	59	3Bh
<	60	3Ch
=	61	3Dh
>	62	3Eh
?	63	3Fh
@	64	40h
A	65	41h
B	66	42h
C	67	43h

Caracter	Dec.	Hex.
D	68	44h
E	69	45h
F	70	46h
G	71	47h
H	72	48h
I	73	49h
J	74	4Ah
K	75	4Bh
L	76	4Ch
M	77	4Dh
N	78	4Eh
O	79	4Fh
P	80	50h
Q	81	51h
R	82	52h
S	83	53h
T	84	54h
U	85	55h
V	86	56h
W	87	57h
X	88	58h
Y	89	59h
Z	90	5Ah
[	91	5Bh
\	92	5Ch
]	93	5Dh
^	94	5Eh
_	95	5Fh
`	96	60h
a	97	61h
b	98	62h
c	99	63h
d	100	64h
e	101	65h
f	102	66h
g	103	67h

Caracter	Dec.	Hex.
h	104	68h
i	105	69h
j	106	6Ah
k	107	6Bh
l	108	6Ch
m	109	6Dh
n	110	6Eh
o	111	6Fh
p	112	70h
q	113	71h
r	114	72h
s	115	73h
t	116	74h
u	117	75h
v	118	76h
w	119	77h
x	120	78h
y	121	79h
z	122	7Ah
{	123	7Bh
|	124	7Ch
}	125	7Dh
~	126	7Eh
<delete>	127	7Fh
Ç	128	80h
ü	129	81h
é	130	82h
â	131	83h
ä	132	84h
à	133	85h
å	134	86h
ç	135	87h
ê	136	88h
ë	137	89h
è	138	8Ah
ï	139	8Bh

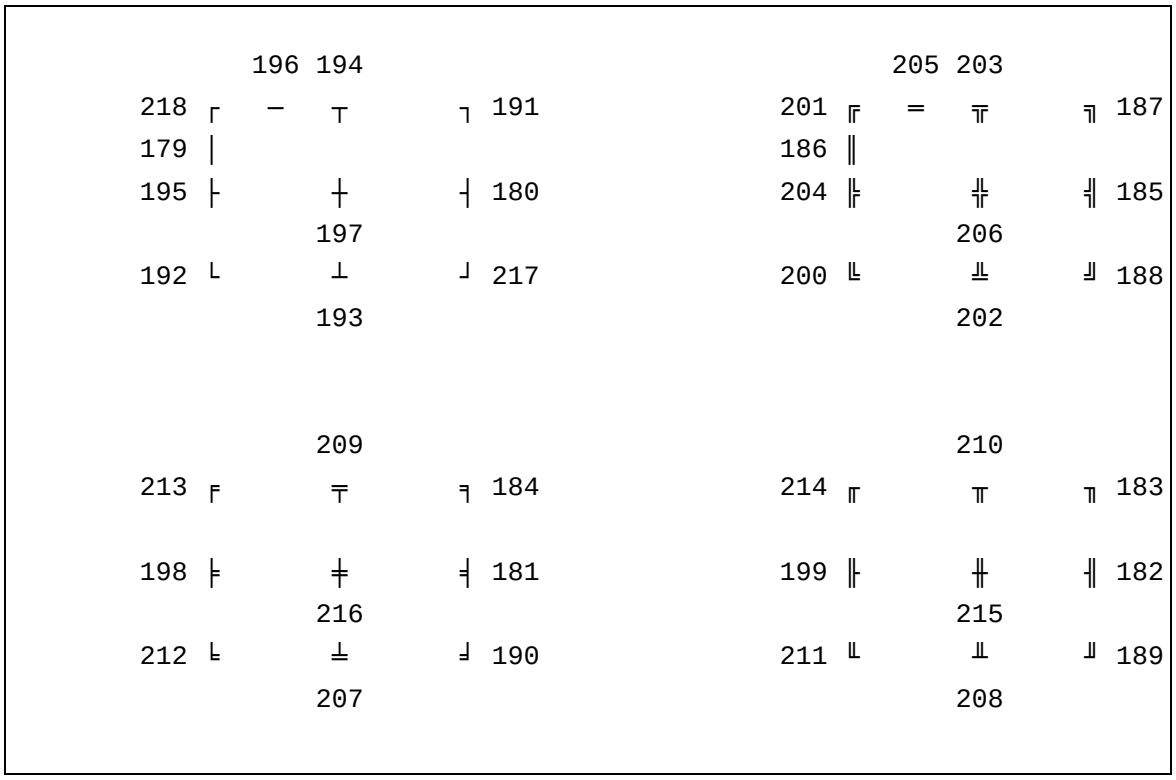
Caracter	Dec.	Hex.
î	140	8Ch
ì	141	8Dh
Ä	142	8Eh
Å	143	8Fh
É	144	90h
æ	145	91h
Æ	146	92h
ô	147	93h
ö	148	94h
ò	149	95h
û	150	96h
ù	151	97h
ÿ	152	98h
Ö	153	99h
Ü	154	9Ah
¢	155	9Bh
£	156	9Ch
¥	157	9Dh
₧	158	9Eh
ƒ	159	9Fh
á	160	A0h
í	161	A1h
ó	162	A2h
ú	163	A3h
ñ	164	A4h
Ñ	165	A5h
ª	166	A6h
º	167	A7h
¿	168	A8h
⌐	169	A9h
¬	170	AAh
½	171	ABh
¼	172	ACh
¡	173	ADh
«	174	AEh
»	175	AFh
░	176	B0h

Caracter	Dec.	Hex.
▒	177	B1h
▓	178	B2h
│	179	B3h
┤	180	B4h
Á	181	B5h
Â	182	B6h
À	183	B7h
©	184	B8h
╣	185	B9h
║	186	BAh
╗	187	BBh
╝	188	BCh
╜	189	BDh
╛	190	BEh
┐	191	BFh
└	192	C0h
┴	193	C1h
┬	194	C2h
├	195	C3h
─	196	C4h
┼	197	C5h
ã	198	C6h
Ã	199	C7h
╚	200	C8h
╔	201	C9h
╩	202	CAh
╦	203	CBh
╠	204	CCh
═	205	CDh
╬	206	CEh
╧	207	CFh
╨	208	DOh
╤	209	D1h
╥	210	D2h
╙	211	D3h
╘	212	D4h
╒	213	D5h

Caracter	Dec.	Hex.
╓	214	D6h
╫	215	D7h
╪	216	D8h
┘	217	D9h
┌	218	DAh
█	219	DBh
▄	220	DCh
▌	221	DDh
▐	222	DEh
▀	223	DFh
α	224	E0h
ß	225	E1h
Γ	226	E2h
π	227	E3h
Σ	228	E4h
σ	229	E5h
µ	230	E6h
τ	231	E7h
Φ	232	E8h
Θ	233	E9h
Ω	234	EAh
δ	235	EBh
∞	236	ECh
φ	237	EDh
∈	238	EEh
∩	239	EFh
≡	240	F0h
±	241	F1h
≥	242	F2h
≤	243	F3h
⌠	244	F4h
⌡	245	F5h
÷	246	F6h
≈	247	F7h
°	248	F8h
∙	249	F9h
·	250	FAh

Caracter	Dec.	Hex.
√	251	FBh
n	252	FCh
2	253	FDh
▪	254	FEh
	255	FFh

Entre os caracteres da tabela ASCII estendidos os mais úteis estão, talvez, os caracteres de desenho de quadro em linhas simples e duplas: os caracteres de B3h até DAh (179 a 218 decimal). Como a visualização deste conjunto é difícil, o desenho abaixo pode auxiliar nesta tarefa:



Caracteres de desenho de quadro e seus respectivos códigos ASII.

ANEXO V – Tipos de Dados e Valores Máximos Permitidos

A seguir estão listados os tipos de dados permitidos e seus valores máximos num compilador típico para um hardwarede 16 bits:

Tipo	***Num de Bits***	***Intervalo***	
		Início	***Fim***
char	8	-128	127
unsigned char	8	0	255
signed char	8	-128	127
int	16	-32.768	32.767
unsigned int	16	0	65.535
signed int	16	-32.768	32.767
short int	16	-32.768	32.767
unsigned short int	16	0	65.535
signed short int	16	-32.768	32.767
long int	32	-2.147.483.648	2.147.483.647
signed long int	32	-2.147.483.648	2.147.483.647
unsigned long int	32	0	4.294.967.295
float	32	3,4E-38	3,4E+38
double	64	1,7E-308	1,7E+308
long double	80	3,4E-4932	3,4E+4932

O tipo **long double** é o tipo de ponto flutuante com maior precisão. É importante observar que os intervalos de ponto flutuante, na tabela acima, estão indicados em faixa de *expoente*, mas os números podem assumir valores tanto positivos quanto negativos.