Introdução a Algoritmos e Programação

Profa. Dra. Ana Cristina dos Santos

Sumário

1. Introdução

1.1 Lógica

A lógica de programação é necessária para as pessoas que desejam trabalhar com desenvolvimento de sistemas e programa, ela permite definir a sequência lógica para o desenvolvimento.

Lógica de programação é a técnica de encadear pensamentos para atingir determinado objetivo.

1.2 Sequência Lógica

Estes pensamentos podem ser descritos como uma sequência de instruções, que devem ser seguidas para cumprir uma determinada tarefa.

Sequência Lógica são passos executados até atingir um objetivo ou solução de um problema.

1.3 Instruções

Na linguagem comum, entende-se por instruções "um conjunto de regras ou normas definidas ou normas para a realização ou emprego de algo". Em computação, porém, a instrução é a informação que indica a um computador uma ação elementar a executar.

Convém ressaltar que uma ordem isolada não permite realizar o processo completo, para isso é necessário um conjunto de instruções colocadas em ordem sequencial lógica. Por exemplo, se quisermos fazer uma omelete de batatas, precisaremos colocar em prática uma série de instruções: descascar batatas, bater ovos, fritar as batatas, etc.

É evidente que essas instruções têm que ser executadas em uma ordem adequada, ou seja, em uma sequência lógica, não se pode descascar as batatas depois de fritá-las.

Dessa maneira, não possui muito sentido uma instrução isolada. Para obtermos o resultado, precisamos colocar um conjunto de todas as instruções na ordem correta.

1.4 Algoritmo

1.4.1 Definição

"O conceito central da programação e da Ciência da Computação é o conceito de algoritmos, isto é, programar é basicamente construir algoritmos."

"Um algoritmo é uma sequência finita de instruções sem ambiguidades e cuja execução, em tempo finito, resolve um problema."

Algoritmo não é a solução de um problema, pois se assim fosse cada problema teria um único algoritmo. Algoritmo é um caminho para a solução de um problema, e em geral, os caminhos que levam a uma solução são muitos.

O aprendizado de algoritmos é adquirido através de muitos exercícios. Algoritmos só são aprendidos construindo-os e testando-os.

Um algoritmo deve possuir as seguintes características:

- Ter início e fim.
- Ser escrito em termos de ações ou comandos bem definidos.
- Deve ser fácil de interpretar e codificar, ou seja, ele deve ser o intermediário entre a linguagem falada e a linguagem de programação.
- Ter uma sequência lógica.

1.4.2 Exemplo

- **Algoritmo para fazer ligação telefônica através de um telefone fixo.**

Início

1. Tirar o fone do gancho;
2. Esperar até ouvir o sinal de linha;
3. Teclar o número desejado;
4. Se der o sinal de chamar:
   4.1. Conversar;
   4.2. Desligar;
5. Se não der o sinal de chamar:
   5.1. Desligar;
   5.2. Repetir desde o passo 1

Fim.

- **Algoritmo para trocar lâmpada**

Início

1. Pegar uma escada;
2. Posicionar a escada debaixo da lâmpada;
3. Pegar uma lâmpada nova;
4. Subir na escada;
5. Retirar a lâmpada velha;
6. Colocar a lâmpada nova;
7. Descer na escada;
8. Acionar o interruptor;
9. Se a lâmpada não acender:
   9.1. Repetir desde o passo 3;

Fim.

1.4.3 Entrada, Processamento e Saída

Para criar um algoritmo é necessário dividir o problema proposto em três fases fundamentais: entrada, processamento e saída.

Entrada: são os dados necessários para resolução do problema proposto. Nem sempre é preciso solicitar informações para resolver um problema.

Processamento: são os procedimentos utilizados para chegar ao resultado final.

Saída: são os dados processados apresentando o resultado para o problema proposto.

Exemplo: Considere o seguinte problema: São fornecidas as notas do 1º e 2º bimestre de um aluno, calcule a média final do semestre de uma determinada disciplina. Mostre a média calculada e se o aluno foi aprovado ou reprovado.

Em que, média = (P1 +2* P2) / 3

O aluno é aprovado quando a média é maior ou igual a 6.0, caso contrário é reprovado.

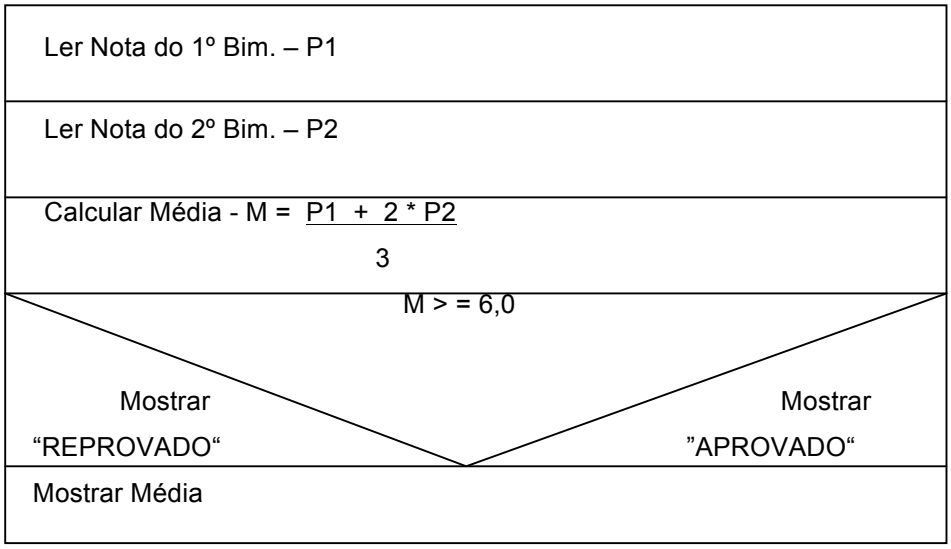
Para montar o algoritmo proposto, deve-se questionar:

- Quais são os dados de entrada?
  - R: As notas do 1º e 2º bimestre: P1 e P2
- Qual será o processamento a ser utilizado?
  - R: Efetuar o cálculo (P1 +2* P2)/3 e comparar a 6.0
- Quais são os dados de entrada?
  - R: A média calculada e o resultado "aprovado" ou "reprovado".

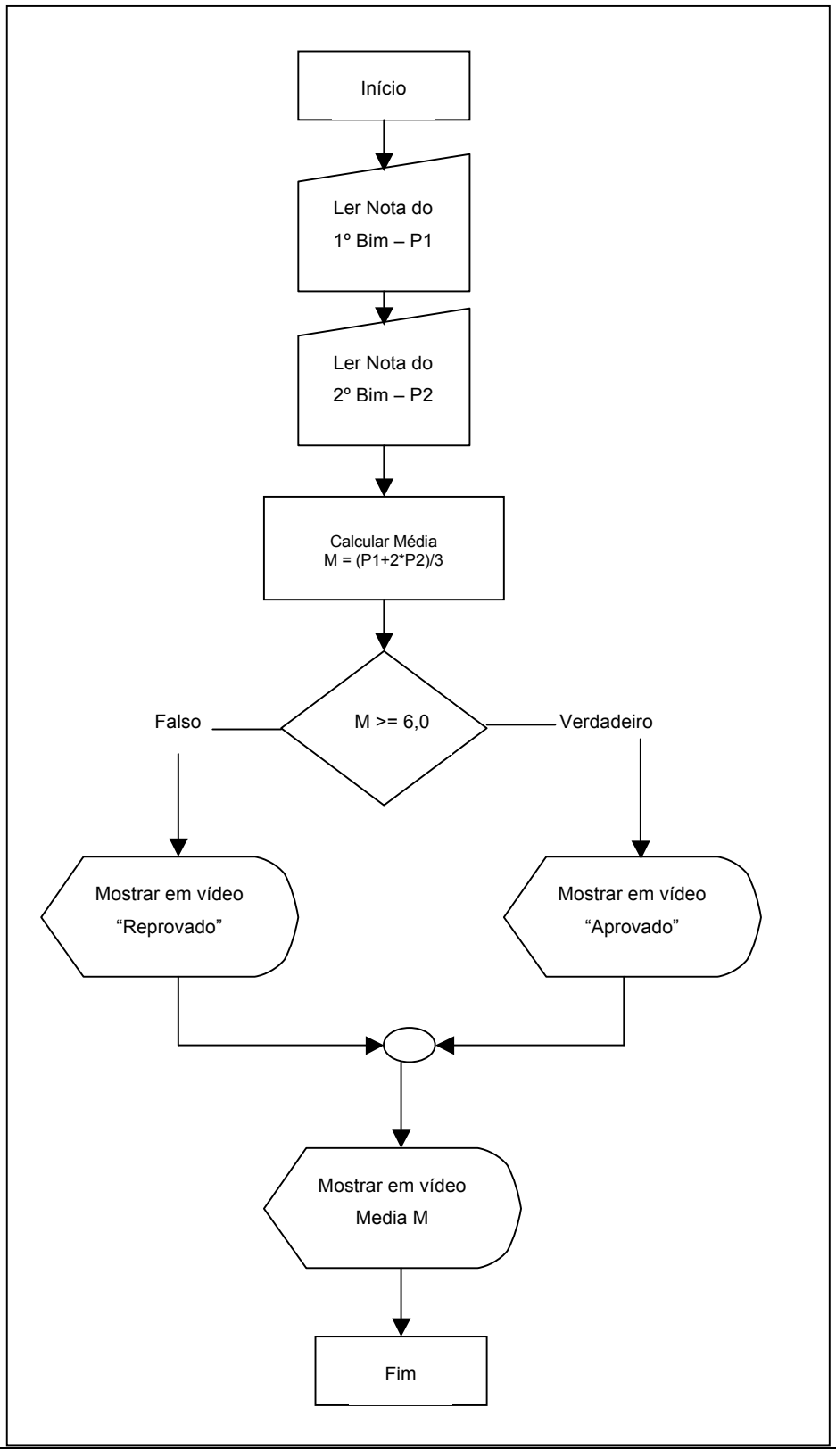
1.4.4 Representação de Algoritmos

Para definir algoritmos existe uma série de representações. Pode-se representar um algoritmo:

- Em uma língua (português, inglês): apresenta um inconveniente em relação à ambigüidade de alguns termos.
- Em uma linguagem de programação: limita-se as instruções da linguagem utilizada e dificulta conversão para uma outra.
- Em gráficos:
  - Diagramas de Nassi-Shneiderman ou Chapin



- Fluxograma


Início
Ler Nota do
1º Bim – P1
Ler Nota do
2º Bim – P2
Calcular Média
M = (P1+2*P2)/3
M >= 6,0
Falso
Verdadeiro
Mostrar em vídeo
“Reprovado”
Mostrar em vídeo
“Aprovado”
Mostrar em vídeo
Media M
Fim

- Em pseudo-código: Não apresenta os inconvenientes de ambiguidade de uma língua, nem os rigores de uma linguagem de programação. Esta representação é feita através de um português estruturado com "frases" correspondentes às estruturas básicas de programação.

1.4.5 Regras para Representação

Neste curso representaremos os algoritmos através de **PSEUDOCÓDIGO**.

Inicialmente veremos as regras básicas da representação de um algoritmo através de pseudocódigo:

- Definir um NOME para o algoritmo;
- Descrever o OBJETIVO do algoritmo;
- Descrever a ENTRADA DE DADOS para o algoritmo;
- Descrever a SAÍDA DE DADOS do algoritmo;
- O algoritmo deve iniciar com a palavra: **INÍCIO;**
- Para indicar o fim do algoritmo utilizamos a palavra: **FIM;**
- A palavra que indica ação a ser executada no algoritmo deve estar sublinhada, tais como: **início**, **fim**, **leia**, **escreva**, **se**, **então**, **senão**, **enquanto**, **faça**, **até que**, **para**;
- Usar somente um verbo por frase no modo imperativo;
- Usar frases simples e curtas;
- Alinhar as instruções de acordo com o nível a que pertençam com a finalidade de destacar a estrutura na qual estão contidos (identação);
- Incorporar comentários no algoritmo.

A seguir veremos um exemplo de algoritmo representado em pseudocódigo:

Nome: CÁLCULO MÉDIA.

Objetivo: Calcular a média de um aluno e apresentar se ele foi aprovado ou reprovado.

Entrada de Dados: Notas do 1° e 2° bimestre (P1 e P2).

Saída de Dados: Média e uma mensagem indicando aprovado ou reprovado.

```
Início
        LEIA a nota do 1° Bimestre – P1
        LEIA a nota do 2° Bimestre - P2
        EFETUE o cálculo da média: Média <-(P1 + 2*P2)/3
        SE média >= 6.0
        ENTÃO
                IMPRIMA "Aprovado"
        SENÃO
             IMPRIMA "Reprovado"
        FIM-SE
        IMPRIMA valor da média
Fim.
```

1.5 Exercícios

1. Identifique os dados de entrada, processamento e saída no algoritmo abaixo.

```
Início

    LEIA o código da peça;

    LEIA o valor da peça;

    LEIA a quantidade de peças;

    EFETUE o cálculo: Total da Peça ←(Quantidade * Valor da peça)

    IMPRIMA o valor total da peça

Fim.
```

2. Faça um algoritmo para somar dois números e multiplicar o resultado pelo primeiro número.

3. Elabore um algoritmo que mova três discos de uma Torre de Hanói, que consiste em três hastes (a - b - c), uma das quais servem de suporte para três discos de tamanhos diferentes (1 - 2 - 3), os menores sobre os maiores. Pode-se mover um disco de cada vez para qualquer haste, contanto que nunca seja colocado um disco maior sobre um menor. O objetivo é transferir os três discos para outra haste.

4. Três jesuítas e três canibais precisam atravessar um rio; para tal dispõem de um barco com capacidade para duas pessoas. Por medidas de segurança, não se deve permitir que em alguma margem a quantidade de jesuítas seja inferior à de canibais. Qual a solução para efetuar a travessia com segurança? Elabore um algoritmo mostrando a resposta, indicando as ações que concretizam a solução deste problema.

2. Linguagem de Programação

2.1 Programas

Programas são sequências de instruções que descrevem as tarefas a serem realizadas para alcançar a solução de um determinado problema, e devem ser escritos em uma linguagem de programação para que possam ser executados em um computador. Portanto, um programa é um algoritmo construído segundo as regras de uma linguagem de programação.

2.2 Técnicas de Programação

Na elaboração de programas é necessário utilizar um método sistemático de programação que permita a obtenção de programas confiáveis, flexíveis e eficientes. Quando existe um problema e utiliza-se um computador para resolvê-lo inevitavelmente tem-se que passar pelas seguintes etapas:

- Análise do problema:

Deve-se estudar minuciosamente o problema, eliminando possíveis ambiguidades e assegurando o completo entendimento das especificações de entrada e saída.

- Projeto do sistema ou programa - algoritmo e estruturas de dados: constitui-se no desenvolvimento e descrição do algoritmo.

No desenvolvimento do algoritmo identifica-se as partes ou etapas na estratégia de resolução do problema, elaborando inicialmente um esboço da resolução. Em seguida, detalha-se cada etapa, refinando o processo de resolução, até chegar a uma sequência de operações básicas sobre os tipos de dados considerados.

- Implementação (codificação) e teste do programa:

A implementação de um algoritmo em uma linguagem de programação pode ser uma tarefa simples ou trabalhosa, dependendo principalmente das características da linguagem escolhida e dos tipos de dados nela definidos.

O teste de um programa tem por finalidade demonstrar que o algoritmo realmente resolve o problema proposto.

2.3 Linguagem de Programação (LP)

2.3.1 Motivação para se estudar uma LP

1. Maior habilidade em resolver problemas: uma maior compreensão de uma LP pode aumentar nossa habilidade em pensar como atacar os problemas. Esta habilidade será melhorada quando se dominam os vários modelos de LP.
2. Melhor uso de uma LP: compreensão das funções e implementação das estruturas de uma LP se leva a usar a LP de modo a extrair o máximo de sua funcionalidade de maneira eficiente..
3. Melhor escolha de uma LP: adequação ao problema, escolhendo a LP certa para um projeto particular reduz-se a quantidade de código necessária.
4. Maior facilidade em aprender novas LPs: conceitos chaves comuns às LPs.
5. Melhor projeto de LPs: linguagens de interfaces de sistemas, extensão de LP via operadores e tipos de dados.

2.3.2 Definição

Uma LP é uma linguagem destinada a ser usada por uma **pessoa** para expressar um **processo** através do qual um **computador** pode resolver um **problema**.

Em outras palavras, uma LP faz a ligação entre o pensamento humano (muitas vezes de natureza não estruturada) e a precisão requerida para o processamento pela máquina.

Os quatros componentes de uma LP são:

- **Computador**: a máquina que executará o processo descrito através do programa;
- **Pessoa**: o programador que serve como a origem da comunicação;
- **Processo**: a atividade que está sendo descrita através do programa;
- **Problema:** o sistema atual ou ambiente onde o problema surgiu.

2.3.3 Histórico

Segue-se com um pouco da história de algumas linguagens de programação que introduziram conceitos importantes para as futuras LPs e que ainda estão em uso. Essas linguagens estão classificadas em três períodos, de acordo com a época em que surgiram.

2.3.3.1 Período: 1955 - 1965

- **FORTRAN (FORmula TRANslation)**

A linguagem Fortran, desenvolvida em 1956 por John Backus, foi proposta visando a resolução de problemas científicos, para isto utilizando a notação algébrica. Foi desenvolvida inicialmente para uma máquina específica o IBM 704. É, ainda hoje, uma linguagem muito utilizada no meio técnico-científico, tendo sido aprimorada ao longo do tempo, constituindo as diversas versões disponíveis. Um dos motivos pelos quais o FORTRAN é ainda muito utilizado é a disponibilidade de uma vasta biblioteca de software contendo rotinas frequentemente utilizadas, tais como rotinas para cálculo de funções trigonométricas, equações algébricas polinomiais, etc, o que permite uma redução dos custos e tempo de desenvolvimento dos programas.

- **COBOL (COmmon Business Oriented Language)**

A linguagem COBOL, desenvolvida em 1959 pelo Departamento de Defesa dos EUA e fabricantes de computadores, é padrão para as aplicações comerciais e a linguagem muito utilizada ainda hoje. O código é "English-like" e é excelente para a manipulação de arquivos.

- **ALGOL 60 (ALGorithmic Oriented Language)**

Linguagem algébrica de origem européia, desenvolvida pelo comitê Internacional popular, destinada à resolução de problemas científicos. Influenciou o projeto de quase todas as linguagens projetadas a partir de 1960. Foi projetada independente da implementação, o que permite uma maior criatividade, porém de implementação mais difícil. É pouco usada em aplicações comerciais devido à ausência de facilidades de E/S na descrição e pelo pouco interesse de vendedores. Além disso, tornou-se padrão para a publicação de algoritmos.

- **LISP (LISt Processing)**

Linguagem funcional criada em 1960, por John McCartly do grupo de IA do MIT, para dar suporte à pesquisa em Inteligência Artificial. Foi inicialmente desenvolvida para o IBM 704. Existem muitos dialetos, pois LISP nunca foi padronizada. Porém, em 1981 surgiu o Common LISP que é um padrão informal. Os programas em LISP são listas.

- **APL (A Programming Language)**

Foi desenvolvida por volta de 1960 por Kenneth Iverson - Harvard, IBM. Utiliza notação matemática, com operadores poderosos, possuindo muitos operadores e muitos caracteres o que gera grande dificuldade de implementação. Tem uma notação compacta e é utilizada em aplicações matemáticas. Segue o modelo funcional e tem como principal estrutura de dados o ARRAY, com diversos operadores sobre esta estrutura.

- **BASIC (Beginners All-purpose Symbolic Instruction Code)**

A linguagem BASIC, desenvolvida em meados dos anos 60 por John Kemeny e Thomas Kurtz no Dartmouth College, teve como objetivo ensinar alunos de graduação a usarem um ambiente interativo de programação, através de uma LP de fácil aprendizado. Com o surgimento dos microcomputadores de baixo custo, no início dos anos 70, o BASIC tornou-se muito popular, embora não tenha contribuído muito tecnologicamente.

2.3.3.2 Período: 1965 - 1971 (LP's baseadas em ALGOL)

- **PL/I (Programming Language I)**

Desenvolvida em meados dos anos 60 pela IBM com o objetivo de incorporar características das LPs existentes numa única LP de propósito geral. Assim PL/I inclui:

- estrutura de bloco, de controle e recursividade do ALGOL 60;
- subprogramas e E/S formatadas do FORTRAN;
- manipulação de arquivos e registros do COBOL;
- alocação dinâmica de memória e estruturas encadeadas do LISP;
- operações matriciais do APL.

É uma linguagem difícil de aprender e implementar devido a sua grande complexidade.

- **SIMULA 67**

Linguagem baseada em Algol 60, criada no início dos anos 60 por Ole Johan Dahl e Kristan Nygaard, na Noruega. É destinada à descrição de sistemas e programação de simulações.

- **ALGOL 68**

É muito diferente do Algol 60. Linguagem de propósito geral que foi projetada para a comunicação de algoritmos, para sua execução eficiente em vários computadores e para ajudar seu ensino a estudantes. Porém é de difícil descrição, o que resultou em uma baixa popularidade.

- **PASCAL**

Desenvolvida por Niklaus Wirth em 1969, é uma linguagem de fácil aprendizado e implementação, suporta programação estruturada e é adequada para o ensino de programação. Em meados dos anos 80 também passou a ser usada para a programação em microcomputadores. Influenciou praticamente todas as linguagens mais recentes.

2.3.3.3 Linguagens dos anos 80 (criadas na década de 70)

- **PROLOG (PROgramming in LOGic)**

Linguagem desenvolvida em 1972 em Marseille na França. É destinada a aplicações de Inteligência Artificial e se baseia em lógica formal. Um uso comum do Prolog é como um tipo de banco de dados inteligente.

- **SMALL TALK**

Criada por Alan Kay da Xerox - Palo Alto no início dos anos 70. Esta linguagem apresenta um ambiente de programação com menus pop-up, windows e mouse (modelo para Apple Macintosh). Segue o modelo orientado a objetos, possuindo o conceito de classe do SIMULA 67 mais encapsulamento, herança e instanciação.

- **C**

Desenvolvida no laboratório Bell no início dos anos 70, visando a implementação do UNIX. Possui um padrão feito por Kernighan e Ritchie em 1978. Tem facilidades para a programação em "baixo nível" e gera código eficiente. O C tem instruções de controle adequadas e facilidades de estruturação de dados para permitir seu uso em muitas áreas de aplicação. Também tem um rico conjunto de operadores, o que permite um código compacto, porém de baixa legibilidade. É excelente para construir programas portáveis.

- **MODULA 2**

Criada por Niklaus Wirth no final dos anos 70, é uma linguagem de propósito geral, baseada em melhorias no Pascal. É boa para projetos de desenvolvimento de software de grande porte. Além disso foi usada para ensinar programação.

- **ADA**

Foi desenvolvida no início dos anos 70 pelo Departamento de Defesa dos Estados Unidos. É dedicada aos "embedded systems" (operam como parte de um sistema maior) e se baseia no Pascal. Teve um padrão em 1983. Além disso, usa conceitos de classe do Simula 67, adota o tratamento de exceções de PL/I e provê facilidades para processamento concorrente. Foi projetada para apoiar aplicações numéricas, programação de sistemas e aplicações que envolvem considerações de tempo real e concorrência. Seu nome se deve a ADA Augusta, 1a programadora, colaboradora de Charles Babbage - século 19.

2.3.3.4 Linguagens dos anos 2000 (criadas na década de 90)

- **C++**

C++ é uma linguagem de programação criada por Bjarne Stroustrup no início da década de 1980. Com base em C e em Simula, Hubbard(2003) acrescenta que é atualmente uma das linguagens mais populares para programação orientada a objetos. Foi padronizada em 1998 pelo American National Standards Institute (ANSI) e pela International Standards Organization (ISO). Possui o mecanismo classe/objeto, permite herança simples e herança múltipla e sobrecarga de operadores e funções.

- **Java**

Java é uma linguagem de programação originalmente desenvolvido por James Gosling em Sun Microsystems (que é agora uma subsidiária da Oracle Corporation) E lançado em 1995 como um componente essencial da Sun Microsystems Plataforma Java. A linguagem deriva muito de sua sintaxe a partir de C e C + +, mas tem uma simples modelo de objeto e menos de baixo nível instalações. As aplicações Java são normalmente compilados para bytecode (arquivo de classe) Que pode rodar em qualquer Java Virtual Machine (JVM), independentemente da arquitetura de computadores. é um propósito geral, simultâneo , com base na classe de linguagem orientada a objetos que foi especificamente projetado para ter como dependências de execução possível. Java é atualmente uma das linguagens de programação mais popular em uso, e é amplamente utilizada a partir de aplicação de software para aplicações web.

2.3.4 Especificação de uma LP

A descrição de uma linguagem de programação envolve dois aspectos: sintaxe e semântica.

A **sintaxe** é o conjunto de regras que determina quais construções são corretas para formação dos programas e quais não são. Em outras palavras, preocupa-se com a "forma" dos programas (como expressões, comandos, declarações, etc, são colocados juntos para formar programas).

A **semântica** é a descrição de como as construções sintaticamente corretas são interpretadas ou executadas. Em outras palavras, preocupa-se com o "significado" dos programas (como o programa vai se comportar quando executado no computador).

Exemplo: a = b (em C)

- comando de atribuição correto (sintaxe)
- substitua valor de a com o valor atual de b (semântica)

Exemplo: a := b (em C)

- comando de atribuição incorreto (erro de sintaxe)

Exemplo: a = b (em C), porém com variáveis com tipos diferentes (inteiro ou caractere)

- substitua valor de a com o valor atual de b (erro de semântica)

2.3.5 Tradução de uma LP

O objetivo de qualquer linguagem é a comunicação entre duas partes, um emissor e um receptor. Em LP, a comunicação ocorre entre um programador e um programa tradutor. Existem dois métodos para se traduzir um programa escrito em uma

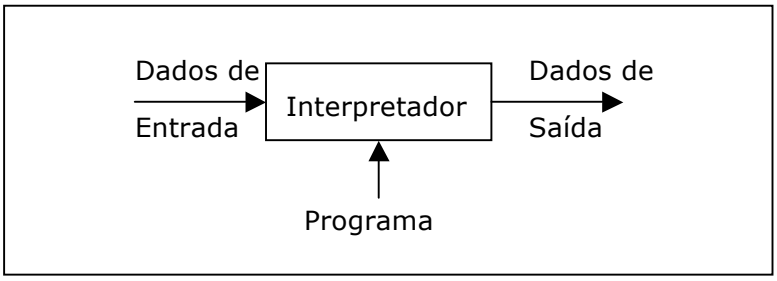
determinada linguagem de programação para a linguagem de máquina: **Interpretação** e **Compilação**.

2.3.5.1 Interpretação

Um interpretador traduz o programa fonte um comando por vez e chama uma rotina para executar esse comando. Mais precisamente, um interpretador é um programa que executa repetidamente a seguinte sequência:

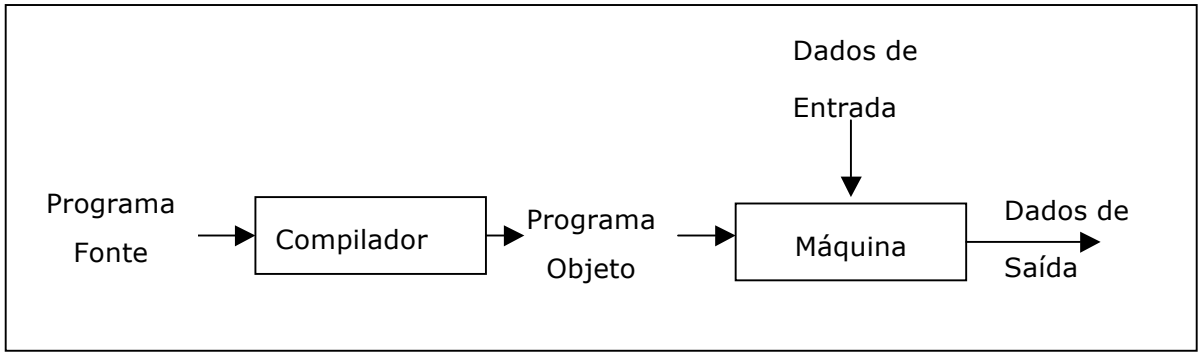
- pega a próxima instrução;
- determina as ações a serem executadas;
- executa estas ações.

O interpretador precisa estar presente todas as vezes que se deseja executar um programa e o trabalho de checagem da sintaxe e tradução deverá ser repetido. Se uma parte do programa necessitar ser executada muitas vezes, o processo é feito o mesmo número de vezes. Na prática as linguagens interpretadas servem para a realização de uma prototipagem rápida.



2.3.5.2 Compilação

Um compilador traduz o programa fonte inteiro, produzindo um outro programa equivalente, em linguagem executável. A vantagem é que o compilador precisa traduzir um comando apenas uma única vez, não importando quantas vezes ele será executado. Na prática o compilador é usado para gerar o código definitivo (eficiente) de um programa.



2.3.6 Paradigmas de LP

Os paradigmas, ou também chamados modelos, de linguagens de programação são:

- Paradigma Imperativo
- Paradigma Funcional
- Paradigma Lógico
- Paradigma Orientado a Objetos

2.3.6.1 O Paradigma Imperativo de LP

O modelo **Imperativo ou Procedural** é baseado na perspectiva do computador: a execução sequencial de comandos e o uso de dados são conceitos baseados no modo como os computadores executam programas no nível de linguagem de máquina.

Este modelo representa o estilo de programação convencional onde os programas são decompostos em passos de processamento que executam operações complexas. Rotinas são usadas como unidades de modularização para definir tais passos de processamento. As LPs imperativas são de fácil tradução. Linguagens Imperativas: FORTRAN, COBOL, ALGOL 60, APL, BASIC, PL/I, SIMULA 67, ALGOL 68, PASCAL, C, MODULA 2, ADA.

2.3.6.2 O Paradigma Funcional de LP

O modelo **Funcional** focaliza o processo de resolução do problema. A visão funcional resulta num programa que descreve as operações que devem ser efetuadas para resolver o problema. Linguagem Funcional: LISP

2.3.6.3 O Paradigma Lógico de LP

O modelo **Lógico** está relacionado à perspectiva da pessoa: ele encara o problema de uma perspectiva lógica. Um programa lógico é equivalente à descrição do problema expressa de maneira formal, similar à maneira que o ser humano raciocinaria sobre ele. Linguagens Lógicas: PROLOG

2.3.6.4 O Paradigma Orientado a Objeto (OO) de LP

O modelo **Orientado a Objeto** está centrado no problema. Um programa OO é equivalente a objetos que mandam mensagens entre si. Os objetos do programa equivalem aos objetos da vida real (problema). A abordagem OO é importante para resolver muitos tipos de problemas através de simulação.

A primeira linguagem OO foi Simula, desenvolvida em 1966 e depois refinada em Smalltalk. Existem algumas linguagens híbridas: Modelo Imperativo mais características de Orientação a Objetos (OO). Ex: C++.

No modelo OO a entidade fundamental é o objeto. Objetos trocam mensagens entre si e os problemas são resolvidos por objetos enviando mensagens uns para os outros.

Linguagens Orientadas a Objeto: SMALL TALK, Java

3. Tipos de Dados e Operações Básicas

3.1 Variáveis

Os algoritmos manipulam dados, e como são organizados esses dados para processá-los corretamente?

O computador possui uma área de armazenamento conhecida como memória. Todas as informações existentes no computador estão ou na memória primária (memória RAM), ou na memória secundária (discos, fitas, CD-ROM etc). Nós iremos trabalhar, neste momento, somente com a memória primária, especificamente com as informações armazenadas na RAM (memória de acesso aleatório).

A memória do computador pode ser entendida como uma sequência finita de caixas, que num dado momento, guarda algum tipo de informação, como número, uma letra, uma palavra, uma frase etc, não importa, basta saber que lá sempre existe alguma informação.

O computador, para poder trabalhar como alguma destas informações, precisa saber onde em que local da memória o dado está localizado. Fisicamente, cada caixa, ou cada posição de memória, possui um endereço, ou seja, um número, que indica onde cada informação está localizada. Este número é representado através da notação hexadecimal, tendo o tamanho de quatro, ou mais bytes. Por exemplo:

Endereço Físico	**Informação**
3000: B712	"João"
2000: 12EC	12345
3000: 0004	'H'

Como pode ser observado, o endereçamento das posições de memória através de números hexadecimais é perfeitamente compreendido pela máquina, mas para nós humanos torna-se uma tarefa complicada. Pensando nisto, as linguagens de computador facilitaram o manuseio, por parte dos usuários, das posições de memória da máquina, permitindo que, ao invés de trabalhar diretamente com números hexadecimais, fosse possível dar nomes diferentes a cada posição de memória. Tais nomes seriam de livre escolha do usuário. Com este recurso, os usuários ficaram livres dos endereços físicos

(números hexadecimais) e passaram a trabalhar com endereços lógicos (nomes dados pelos próprios usuários). Desta forma, o exemplo acima, poderia ser alterado para ter o seguinte aspecto:

Endereço Lógico	Informação
Nome	"João"
Número	12345
letra	'H'

Como tínhamos falado, os endereços lógicos são como caixas, que num dado instante guardam algum tipo de informação. Mas é importante saber que o conteúdo desta caixa não é algo fixo, permanente, na verdade, uma caixa pode conter diversas informações, ou seja, como no exemplo acima, a caixa (endereço lógico) rotulada de "Nome" num dado momento contém a informação "João", mas em um outro momento, poderá conter uma outra informação, por exemplo "Pedro".

Com isto queremos dizer que o conteúdo de uma destas caixas (endereço lógico) pode variar, isto é podem sofrer alterações em seu conteúdo. Tendo este conceito em mente, a partir de agora iremos chamar de forma genérica, as caixas ou endereços lógicos, de variáveis.

Desta forma podemos dizer que uma variável é uma posição de memória, representada por um nome simbólico (atribuído pelo usuário), a qual contém, num dado instante, uma informação.

3.2 Nome de Variáveis

Uma variável é formada por uma letra, "_ " underline ou então por uma letra seguida de letras ou dígitos, em qualquer número. Não é permitido o uso de espaços em branco ou de qualquer outro caractere, que não seja letra ou dígito, na formação de um identificador.

Na formação do nome da variável de um nome significativo, para que se possa ter ideia do seu conteúdo sem abri-la. Se utilizar palavras para compor o nome da variável utilize o "_ " underline para separar as palavras.

3.3 Conteúdo de uma Variável

O conteúdo de uma variável é o valor do dado que está armazenado em um endereço da memória. O conteúdo de uma variável pode ser alterado durante a execução do programa. Considerando o exemplo acima, o conteúdo das variáveis é: a palavra "João", o número 12.345 e o caractere 'H'.

Nota-se que os dados são tipos de informações diferentes e que são armazenadas na memória de formas diferentes. Um caractere é representado na memória por um *byte*, uma palavra, que é uma sequência de caracteres, é representada por **n** bytes, onde n é número de caracteres que compõe a palavra. Devido a essa diferença, quando desenvolve um algoritmo é necessário especificar quais são as variáveis necessárias e que tipo de dado elas vão armazenar.

3.4 Tipos de Dados

Os dados são representados pelas informações a serem processadas por um computador. Os tipos de dados podem ser:

- **Primitivos** ➔ Pré-definidos
    - Inteiro
    - Real
    - Caractere
    - Lógico
- **Compostos** ➔ Definidos pelo programador
    - Cadeia de Caracteres (String)
    - Matrizes (Arrays)
    - Registros

Nesse curso inicial veremos os tipos de dados simples ou primitivos.

Quando é necessária a utilização de variáveis para resolver um problema, deve-se especificar essa necessidade no algoritmo e também o tipo de informação que uma variável pode conter, isto é, se uma dada posição de memória armazenará um número ou uma letra etc.

Cada linguagem apresenta seus tipos pré-definidos, em algoritmos representados em pseudocódigo utilizaremos os seguintes tipos:

Nome do Tipo	*Descrição*
inteiro	Representa números sem casas decimais
real	Representa os números com casas decimais
caractere	Representa um dos caracteres, da tabela ASCII.
cadeia de caractere ou String	Conjunto de caracteres.
booleano	Valor lógico. Assume somente dois valores: TRUE (Verdade) ou FALSE (Falso).

3.5 Declaração de Variáveis

Para indicar o uso de uma variável no algoritmo faremos da seguinte forma:

- Após a definição da saída, definir as variáveis informando seu nome e que tipo de dado ela vai armazenar. Obedecendo a seguinte ordem:

SINTAXE:

<TIPO DA VARIÁVEL> <NOME DA VARIÁVEL> <COMENTÁRIO >

Exemplo:

```
Nome: Exemplo 1
Objetivo:
Dados de Entrada:
Saída:
Variáveis: inteiro      X          <resultado do acréscimo>
           caracter     resposta   <resposta do usuário para executar
                                        o programa 'S' ou 'N'>
           real         preco      <preço do produto a ser  reajustado>
Início
  .
  .
  .
Fim
```

3.6 Operador de atribuição

Quando definimos uma variável é natural atribuirmos a ela uma informação. Uma das formas de colocar um valor dentro de uma variável, consequentemente colocar este dado na memória do computador, é através da atribuição direta, do valor desejado que a variável armazena. Para isto utilizaremos o símbolo ← , que significa: recebe, ou seja, a posição, de memória que uma variável representa, receberá uma informação, a qual será armazenada no interior desta variável.

SINTAXE:

<NOME_DA_VARIÁVEL> ← <VALOR> ou

<NOME_DA_VARIÁVEL> ← <EXPRESSÃO> ou

<NOME_DA_VARIÁVEL> ← <VARIÁVEL>

Exemplo:

Nome: **Exemplo 2**
Objetivo: Mostrar a atribuição de variáveis
Dados de Entrada:
Saída:
Variáveis:
inteiro X <armazena resultado das atribuições>
Y <armazena um valor constante>

Início
Y ← 50 <Após a execução desse comando, a variável Y recebe o valor 50 >

X ← 10 <Após a execução desse comando, a variável X recebe o valor 10 >

X ← Y <Após a execução desse comando, a variável X recebe o valor armazenado na variável Y>

X ← Y + 2 <Após a execução desse comando, a variável X recebe o valor armazenado na variável Y mais dois.>

Fim

Observações:

- As variáveis somente poderão receber valores de tipos de dados definidos em sua declaração.
- Não é possível fazer atribuições entre variáveis de tipos diferentes.

3.7 Operadores matemáticos

Para formar uma expressão matemática utiliza-se os operadores matemáticos. Os operadores matemáticos são representados da seguinte forma:

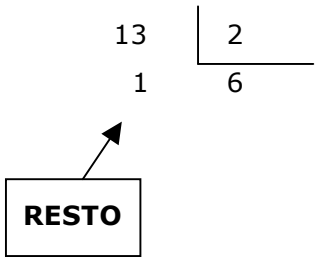
Operação	Símbolo
Adição	**+**
Subtração	**-**
Multiplicação	*****
Divisão	**/**
Resto da divisão	**% (RESTO)**

Nas expressões aritméticas, as operações guardam entre si uma relação de prioridade, tal como na matemática.

Prioridade	Operação
1º	Potenciação, radiciação
2º	Multiplicação, divisão
3º	Adição, subtração

- **Observações:**
  - Utiliza-se parênteses para indicar a prioridade das operações.
  - O operador Resto da Divisão (%) somente é utilizado com variáveis do tipo inteiro.

Exemplo:



13 / 2 = 6

13 % 2 = 1

Exemplo:

Nome: **Exemplo 3**
Objetivo: Mostrar a atribuição de expressões a variáveis
Dados de Entrada:
Saída:
Variáveis: inteiro X <armazena resultado das atribuições>
Y <armazena um valor constante>

```
Início
        Y ← 2
        X ← 10
        X ← Y + 2
        X ← ((Y + 3 ) * 4) - 5
        <Após a execução desse comando, a variável X recebe o valor 15 >
        X ← (X % 2 )
        <Após a execução desse comando, a variável X recebe o valor 1 >
Fim
```

3.8 Instrução para Leitura

Outra forma de atribuir valor a uma variável é através da entrada de dados para isso utilizaremos a instrução LEIA.

SINTAXE:

```
LEIA <NOME_DA_VARIÁVEL>
```

Exemplo:

```
Nome: Exemplo 4
Objetivo: Mostrar a leitura de variáveis
Dados de Entrada:
Saída:
Variáveis: inteiro      X  <armazena valor digitado pelo usuário>
                        Y  <armazena um valor constante>
Início
        LEIA X
        <Após a execução desse comando, a variável X recebe o valor
        digitado pelo usuário >
        LEIA Y
        <Após a execução desse comando, a variável Y recebe o valor
        digitado pelo usuário >
Fim
```

3.9 Instrução para Impressão

As variáveis armazenam valores que serão processados durante o programa e que, em geral, pode ser a saída do programa. Para mostrar o valor de uma variável utilizaremos a instrução IMPRIMA.

SINTAXE:

IMPRIMA <NOME_DA_VARIÁVEL>

Exemplo:

Nome: **Exemplo 5**
Objetivo: Mostrar a leitura de variáveis
Dados de Entrada:
Saída:
Variáveis:
inteiro X <armazena valor digitado pelo usuário>
Y <armazena um valor >

Início

LEIA X
LEIA Y

X ← Y* 2

IMPRIMA X
<Após a execução desse comando, o valor da variável X será apresentada em vídeo. >

IMPRIMA " Oi ! Isso é um teste"
<Após a execução desse comando, a mensagem "Oi ! Isso é um teste" será apresentada em vídeo. >

Fim

3.10Variáveis do tipo Caractere e Cadeia de Caracteres

As definições de variáveis como sendo do tipo caractere e cadeia de caracteres, possuem algumas curiosidades que merecem um cuidado especial.

3.10.1 Uso das aspas simples (')

Quando estivermos fazendo a atribuição de um valor para uma variável do tipo caractere temos que ter o cuidado de colocar o valor (dado) entre aspas simples ('), pois esta é a forma de informar que a informação é caractere e não uma variável numérica.

3.10.2 Uso das aspas dupla (")

Quando estivermos fazendo a atribuição de um valor para uma variável do tipo cadeia de caracteres temos que ter o cuidado de colocar o valor (dado) entre aspas duplas ("), pois esta é a forma de informar que a informação é cadeia de caracteres e não uma variável numérica.

Exemplo:

```
Nome: Exemplo 6
Objetivo: Mostrar a leitura de variáveis
Dados de Entrada:
Saída:
Variáveis: Caracter     L       <armazena um caracter>
           Cadeia       Nome    <armazena uma sequência de caracteres>

Início
        LEIA        L
        <A variável L recebe um caractere digitado pelo usuário>

        LEIA Nome
        <A variável Nome recebe uma palavra digitada pelo usuário>

        IMPRIMA  L
        <Apresenta no vídeo o valor da variável L >

        IMPRIMA  'L'
        <Apresenta no vídeo o caractere 'L '>

        IMPRIMA  "Nome"
        <Apresenta no vídeo a palavra "Nome">

        IMPRIMA  Nome
        <Apresenta no vídeo o valor da variável Nome>

Fim
```

3.10.3 Manipulação de Cadeias de Caracteres ou Strings

A manipulação de cadeias (strings) não é tão simples assim, pois a cadeia (string) é armazenada como um vetor de caracteres. Desta forma, é necessário compreender o conceito de vetores que será visto mais adiante, quando se abordará também cadeias de caracteres.

3.11 Exercícios

1) Dar o tipo de cada uma das constantes

a) 613 b) 613,0

c) –613 d) "613"

e) $-3{,}012 * 10^{15}$ f) $17*10^{12}$

g) $-28{,}3 * 10^{-23}$ h) "Fim de Questão"

2) Indique qual será o resultado obtido das seguintes expressões:

a) 1 / 2

b) 1 % 2

c) (200 / 10) % 4

d) 5*2 + 3

e) 3,0 * 5,0 +1

f) 1 / 4 + 2

g) 29,0 / 7 + 4

3) Qual a diferença existente nas atribuições seguintes?

a) Letra ← 'A'
Nome ← "João"

b) Letra ← A
Nome ← João

4) No algoritmo seguinte existe algum erro? Onde?

Nome: Exercício 4

Objetivo: Verificar erros de variáveis
Dados de Entrada:
Saída:
Variáveis

CADEIA	Maria
INTEIRO	inteiro
CARACTER	letra
REAL	Maria

Início

idade ← 23
idade ← letra
letra ← ABC
letra ← A

```
        letra  ← 2
Fim
```

5) Indique o valor da variável X do tipo inteiro em cada expressão aritmética. Considere A=10, B=5, C=6, D=7, E=2, F=3, em que A, B, C, D, E, F são variáveis inteiras.

a) X ← A+B

b) X ← A + B * C

c) X ← A + B * C / F – E

d) X ← A % B + C % E + C % F + C % B + D / C

e) X ← A + A % C + C

f) X ← (A * B * C * D)/ (E * F)

6) Faça um algoritmo para ler as seguintes informações de uma pessoa: Nome, Idade, Sexo, Peso, Altura, Profissão, Cidade, Estado, CEP, Telefone.

7) Ler dois valores para as variáveis A e B, efetuar a troca dos valores de forma que a variável A passe a possuir o valor da variável B e que a variável B passe a possuir o valor da variável A Apresentar os valores trocados.

8) Leia as seguintes informações de um funcionário: Nome, idade cargo e o seu salário bruto e considere:

a) O salário bruto teve um reajuste de 38%.
b) O funcionário receberá uma gratificação de 20% sobre salário bruto reajustado.
c) O Salário total é descontado em 15% sobre salário bruto reajustado.

Faça um algoritmo para:

- Imprimir Nome, idade e cargo.
- Imprimir o salário bruto.
- Imprimir o salário líquido.

9) Faça um algoritmo para ler a base e a altura de um triângulo. Em seguida, escreva a área do mesmo: Área = (Base * Altura) / 2

10) Escreva um algoritmo para calcular o volume de um objeto. O algoritmo deve ler a massa e a densidade do objeto. A massa será dada em gramas, a densidade será dada em gramas por centímetros cúbicos. A relação entre a massa, densidade e o volume de um objeto é dada por : Densidade = Massa/Volume

Seu algoritmo deve escrever o resultado em centímetros cúbicos.

11) Escreva um algoritmo para calcular a massa de um bloco de alumínio. O algoritmo deve ler as dimensões do bloco, isto é, comprimento, largura e altura, em centímetros. A densidade do alumínio é 2.7 g/cm^3. O algoritmo deve escrever a massa em gramas.

12) Determinar a área e o comprimento de um círculo conhecido seu raio.

Área : $\pi * r^2$.

Comprimento : $2 * \pi * r$, em que o valor de $\pi \cong 3.14$ e r = raio.

13) Escreva um algoritmo que calcule o volume de um cilindro circular, dados o raio e altura do mesmo.

$V = \pi * r^2 * h$, onde $\pi \cong 3.14$, r = raio e h = altura.

14) Uma empresa tem para um determinado funcionário uma ficha contendo o nome, número de horas trabalhadas e o n^0 de dependentes de um funcionário.

Considerando que:

a) A empresa paga 12 reais por hora e 40 reais por dependentes.

b) Sobre o salário são feitos descontos de 8,5% para o INSS e 5% para IR.

Faça um algoritmo para ler o nome, número de horas trabalhadas e número de dependentes de um funcionário. Após a leitura, escreva qual o nome, salário bruto, os valores descontados para cada tipo de imposto e finalmente qual o salário líquido do funcionário.

15) O preço de um automóvel é calculado pela soma do preço de fábrica com o preço dos impostos (45% do preço de fábrica) e a percentagem do revendedor (28% do preço de fábrica). Faça um algoritmo que leia o nome do automóvel e o preço de fábrica e imprima o nome do automóvel e o preço final.

16) Faça um algoritmo que, tendo como dados de entrada a distância total (em Km) percorrida por um automóvel e a quantidade de combustível em litros consumida para percorrê-la, calcule o consumo médio de combustível (km / litro).

17) Faça um algoritmo que, tendo como dados de entrada o primeiro elemento de um PG (progressão geométrica), a razão e o número n de elementos, calcule e informe:

- o enésimo elemento da PG;
- o produto de elementos desta PG, sabendo que: $Pn = a_1^n * r^{(n*(n-1)/2)}$

em que: a_1 é o primeiro elemento da PG;

Pn é o produto dos n-ésimos elementos;

r é a razão da PG.

18) Faça um algoritmo que, tendo como dados de entrada o nome do funcionário, o salário hora e o número de horas trabalhadas, calcule e informe o salário líquido sabendo-se que:

- salário líquido = salário bruto - desconto do INSS.
- salário bruto = número de horas trabalhadas * salário hora;
- desconto do INSS = 9.5 % do salário bruto.

19) Faça um algoritmo que, tem como entrada os dados: o ano atual, o nome do usuário e sua idade. Calcular e informar: o ano de seu nascimento, a idade de seus filhos e os respectivos anos de nascimentos, sabendo-se que:

- a idade do filho mais velho é encontrada pela extração da raiz quadrada da idade do pai;
- o segundo filho é um ano e meio mais novo que seu irmão.

20) Faça um algoritmo que, tendo como dados de entrada os três comprimentos das arestas de uma caixa "retangular" (x, y, z), calcule e informe:

- a quantidade de tinta necessária para pintá-la externamente;
- o custo total = mão de obra + material;

Sabendo-se que :

- para cada metro quadrado são necessários 3 litros de tinta;
- para cada 10 metros quadrados é necessário um novo rolo de tinta que custa R$ 5,00 cada;
- a mão de obra cobra R$20,00 por metro quadrado;
- cada lata de tinta contém 5 litros e custa R$ 45,00

21) Faça um algoritmo que, tendo como dados de entrada dois pontos quaisquer do plano, $P(x_1,y_1)$ e $Q(x_2,y_2)$, calcule e informe a distância entre eles. A fórmula para efetuar tal cálculo é:

$$d = \sqrt{\left((x_2 - x_1)^2 + (y_2 - y_1)^2\right)}.$$

- x_1,y_1 = coordenadas do ponto P;
- x_2,y_2 = coordenadas do ponto Q.

4. Tipos de Dados e Operações Básicas na Linguagem C

4.1 Introdução

A principal finalidade de qualquer programa de computador é a manipulação de dados. Assim, uma importante característica a ser estudada em uma linguagem de programação é a forma com que ela trabalha com as informações. Variáveis e constantes são os objetos de dados básicos manipulados em um programa. As declarações listam as variáveis a serem usadas e definem os tipos que as mesmas devem ter e talvez seus valores iniciais. Os operadores especificam o que deve ser feito com as variáveis. As expressões combinam variáveis e constantes para produzir novos valores.

4.2 Ciclo de Desenvolvimento de um Programa

Resumidamente tem-se abaixo um fluxograma das principais etapas p/ o desenvolvimento de um programa.

1 - Determinação e entendimento do problema :

É sem dúvida a principal etapa deste ciclo, pois um problema mal entendido ou mal interpretado, levará à soluções erradas. Portanto deve-se entender exatamente o problema a ser solucionado, esquematizando a solução por meio um algoritmo, representado por algumas ferramentas como, por exemplo, o fluxograma ou pseudocódigo.

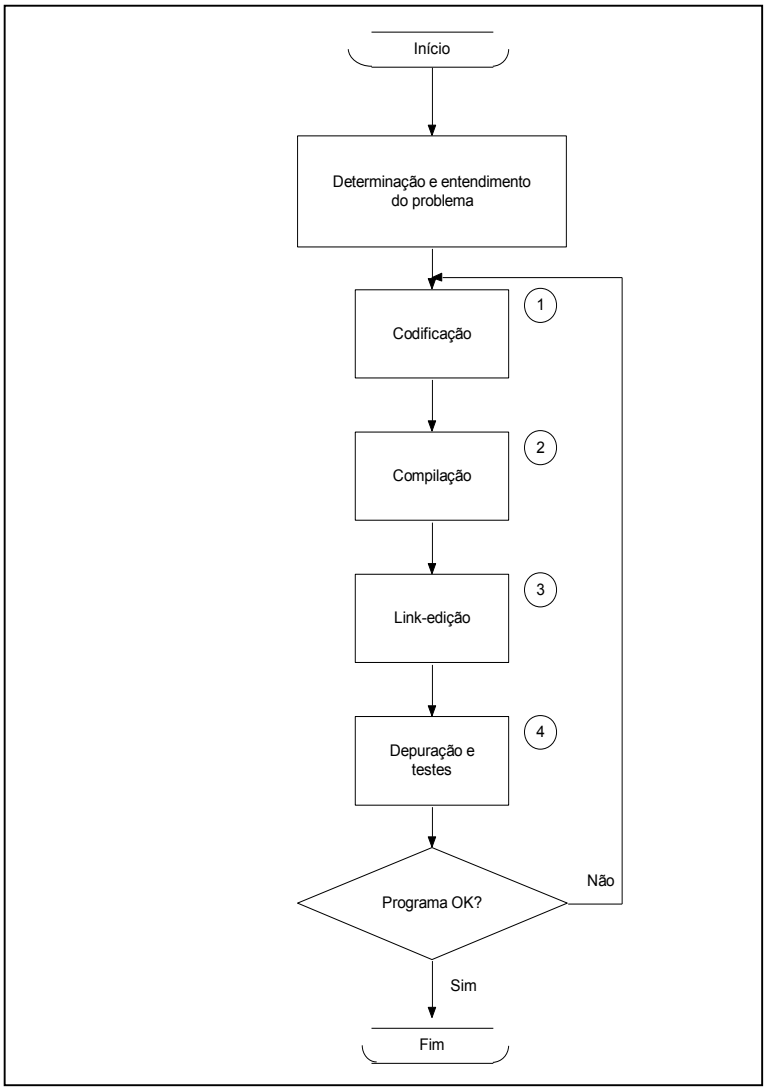
1- Codificação: Uma vez esquematizado a maneira como se pretende solucionar o problema, devemos iniciar a codificação do programa que nada mais é que escrever o programa em uma linguagem e usando regras que o compilador da linguagem entenda, gerando desta maneira um CÓDIGO FONTE (arquivo.c). Para isto, pode-se utilizar a maioria dos editores de texto. Em nosso caso, estaremos usando um editor de texto do próprio ambiente de programação.

2- Compilação: É o processo através do qual um compilador lê o programa inteiro codificado e converte-o em um CÓDIGO OBJETO (arquivo.obj), que é uma tradução do CÓDIGO FONTE do programa em uma forma que o computador possa executar diretamente. O CÓDIGO OBJETO é também conhecido como código binário ou código de máquina.

3- Link-Edição: Todo compilador C vem com uma biblioteca padrão de funções que realizam as tarefas necessárias mais comuns, como exemplo escrever no vídeo (*printf()*), ler um caractere do teclado (*getch()*), entre outras. O padrão ANSI C (*American*

National Standards Institute) especifica o conjunto mínimo de funções que estará contido na biblioteca. No entanto seu compilador provavelmente conterá outras funções que podem não estar incluídas em um compilador de outro fabricante. O processo de link-edição é combinar o CÓDIGO OBJETO gerado pelo compilador que você escreveu, com o código objeto já encontrado na biblioteca padrão quando da chamada de uma função desta biblioteca, por exemplo *printf*() e *getch*(). O resultado deste processo é o CÓDIGO EXECUTÁVEL (arquivo.exe , no DOS).

4- Depuração e Testes: Uma vez gerado o código executável passa-se para a etapa de verificação de funcionamento do programa, entrando em detalhes como valores de variáveis, fluxo dos dados, etc..., isto é na realidade o que chamamos de Depuração, e o ambiente a ser utilizado possui uma ferramenta poderosa para tal, o DEBUG. Uma vez toda que toda a parte lógica do programa está certa, parte-se para os testes. Se nesta etapa qualquer problema é encontrado, devemos iniciar todo o processo, passando novamente pelas etapas 1, 2, 3 e 4.



4.3 Estrutura básica de um programa em C

Antes de explorar alguns detalhes de estrutura é necessário conhecer a estrutura básica de um programa na linguagem C.

O nosso programa inicialmente apresentará as seguintes partes:

```
/* Cabeçalho do programa */
// Programa: Modelo Básico
// Objetivo:
// Dados de entrada:
// Dados de saída:

/* Declaração das Bibliotecas */
#include <stdio.h>                  //funções para entrada e saída de dados


/*Definição das Constantes */
#define INSS  0.08


/* Função principal */

int main (void )
{                                   //início da função main

   /*Declaração das Variáveis da função principal */

   /*Instruções */

    return 0;                       //termina a execução da função main
}                                   //fim da função main
```

Análise de cada parte do programa:

- **Cabeçalho do Programa:** É uma descrição das informações principais do programa que ajudem a elucidar o funcionamento do mesmo, tais como nome, data, nome do programador, dados de entrada, saída, objetivo, entre outros.

- **Declaração das Bibliotecas, Definições de Pré-Processamento ou Arquivos-Cabeçalhos:** Existem funções pré-definidas em C e para utilizá-las em um programa é necessário declarar a biblioteca a qual a função está definida.

  Quando é iniciada a compilação, a primeira tarefa do compilador é executar automaticamente o pré-processador, que consiste em ler o seu código fonte procurando por linha que iniciam com # e expande o seu programa fonte com o conteúdo do arquivo indicado. Os sinais de < e > ao redor do nome do arquivo, indicam ao pré-processador para procurar no diretório padrão onde foi instalado o compilador (usualmente abaixo do diretório *include*). Quando é utilizado o sinal " o pré-compilador procura no diretório corrente.

- **Definição das Constantes:** uma constante simbólica é um nome que substitui uma sequência de caracteres. Ela permite que um nome apareça no

lugar de uma constante numérica, uma constante caractere ou uma constante *string*. Quando um programa é compilado, cada ocorrência de uma constante simbólica é substituída pela sua sequência de caracteres correspondente.

- **Função Principal (main):** É a função por onde é iniciado a execução de um programa em C. Todo programa em C deve possuir a função principal. Para delimitar a área de uma função usa-se chaves (**{ }**) . Entre as chaves está o corpo da função, chamado também de bloco.
- **Corpo da Função Principal:** O corpo da função principal é dividido em duas partes:
  - **Declaração das Variáveis:** Nesta parte deve-se declarar as variáveis que serão utilizadas na função.
  - **Instruções:** Após definir as variáveis, inicia-se a sequência de comandos que o programa deve executar.

4.4 Comentários

Comentários são textos que se introduz no meio do programa fonte com a intenção de torná-lo mais claro. Tudo que estiver entre os símbolos **/*** e ***/** são considerados comentários, o qual não tem sentido para o **compilador.**

As linhas de comentários são extremamente importantes para garantir uma boa compreensão do programa e, automaticamente, sua manutenção. Nunca poupe comentários. Você sempre vai agradecer quando precisar mexer novamente no programa.

No programa, após duas barras **//** até o final da linha, o compilador considera como comentário.

Exemplo:

```
/* Isto é um comentário */                  → ignora toda a frase
teste = 1;        //  Isto é um teste       → ignora  a frase "Isto é um teste"
```

4.5 Palavras Reservadas em C

As palavras reservadas da linguagem são palavras que fazem parte da sua estrutura e têm significados pré-determinados. Elas não podem ser redefinidas e não podem ser utilizadas como identificadores de variáveis, funções, etc.

auto	continue	enum	if	short	switch	volatile
break	default	extern	int	signed	typedef	while
case	Do	float	long	sizeof	union	
char	Double	for	register	static	unsigned	
const	Else	goto	return	struct	void	

4.6 Variáveis

4.6.1 Nome de Variáveis

Cada linguagem possui suas regras para dar nome a uma variável.

Identificadores são os nomes escolhidos para representar constantes, variáveis, tipos, funções. Estes nomes obedecem as seguintes regras:

a) O identificador deve ter como primeiro caractere uma letra ou underscore ' _ ';

b) Após a primeira letra o identificador só pode conter: letras, números ou underscore ' _ ';

c) O identificador não pode conter espaços e caracteres especiais.

d) Não pode ser uma palavra reservada.

e) É uma prática tradicional do C, usar letras minúsculas para nomes de variáveis e maiúsculas para nomes de constantes. Isto facilita na hora da leitura do código;

f) Quando se escreve código usando nomes de variáveis em português, evita-se possíveis conflitos com nomes de rotinas encontrados nas diversas bibliotecas, que são em sua maioria absoluta, palavras em inglês.

Deve-se ressaltar que a linguagem C é "Case Sensitive", isto é, maiúsculas e minúsculas fazem diferença. Caso se declare uma variável com o nome **soma** ela será diferente de **Soma**, **SOMA**, **SoMa** ou **sOmA**.

Da mesma maneira, os comandos do C **if** e **for**, por exemplo, só podem ser escritos em minúsculas pois senão o compilador não irá interpretá-los como sendo comandos, mas sim como variáveis. mas o uso de palavras reservadas não é uma coisa recomendável de se fazer pois pode gerar confusão.

Observações:

- Utilize sempre nomes que lembrem a finalidade do que está sendo identificado.

- Embora seja permitido em algumas versões de compiladores, não é conveniente usar acentuação em nomes de identificadores.

Exemplos:

Meu_Nome	válido
MEU_NOME	válido
_Linha	válido
Exemplo23	válido
2teste	não válido, começa com número
Exemplo 23	não válido, tem um espaço
X ...Y	não válido
#max	não válido

4.6.2 Constantes

Uma constante é um identificador que recebe um nome e é usada para guardar um valor que **não** pode ser modificado durante a execução do programa.

A constante precisa ser inicialmente declarada, e isso é feito com a diretiva ***#define***, normalmente no início do programa ou em arquivos de cabeçalho (.h).

As variáveis constantes devem ser definidas de acordo com a sintaxe seguinte:

SINTAXE:

#define <*nome_da_constante* > <*valor_da_constante*>

As constantes podem ser de qualquer tipo de dados básicos. O valor da constante é que define o tipo da constante, ou seja, o modo como cada constante é representada depende do seu tipo.

Exemplo:

```
// Programa : Exemplo 1
// Objetivo: Mostrar a declaração de constantes  e seus tipos
// Dados de entrada:
// Dados de saída:

#define FALSO           0
#define MAX             1000
#define  ERRO           "Ocorreu um  erro !!!"
#define VERDADEIRO  'V'
#define PI              3.1416

int main (void )
{
```

```
    ...
    return 0;
}
```

4.6.3 Variáveis

Uma variável é uma posição de memória que recebe um nome e é usada para guardar um valor que pode ser modificado durante a execução do programa.

Para ser utilizada, a variável precisa ser inicialmente declarada. A forma geral para declaração de variáveis é:

SINTAXE:

```
<tipo de dado>  variável_1, variável_2,  ...  , variável_n;
    Ou,
    <tipo de dado>      variável_1,
                        variável_2,
                        variável_n;
    ou ainda;
    <tipo de dado>      variável_1;
    <tipo de dado>      variável_2;
```

O tipo da variável é o tipo de dado que a variável armazena, antes de apresentar alguns exemplos serão apresentados os tipos de dados em C.

4.7 Tipos de Dados

Inicialmente serão utilizados dados do tipo simples, ou seja, os tipos de dados já definidos pela linguagem.

Em C, existem cinco tipos de dados: Inteiro (int), Real (float), Duplo (double), caractere (char) e sem valor (void).

Tipo	Palavra chave	Tamanho (em bytes)	Intervalo
Caractere	char	1	-128 a 127
Inteiro	int	2	-32.768 a 32.767
Ponto flutuante com precisão simples	float	4	3.4 E-38 a 3.4+38
Ponto flutuante com precisão dupla	double	8	1.7 E-308 a 1.7E+308
Sem valor	void	0	sem valor

Com exceção do tipo **void**, os tipos básicos de dados poderão ter **modificadores** precedendo-os. Um modificador é usado para alterar o significado do tipo base para adequá-los melhor às necessidades do programa, aumentando a capacidade de armazenamento das informações.

Os modificadores são: **signed (com sinal), unsigned (sem sinal), long (longo), short (curto)**. Assim, têm-se os seguintes tipos de dados.

Tipo	Palavra chave	Tamanho (em bytes)	Intervalo
Caractere	**char**	1	-128 a 127
Caractere com sinal	signed char	1	-128 a 127
Caractere sem sinal	unsigned char	1	0 a 255
Inteiro	**int**	2	-32.768 a 32.767
Inteiro com sinal	signed int	2	-32.768 a 32.767
Inteiro sem sinal	unsigned int	2	0 a 65.535
Inteiro curto	short int	2	-32.768 a 32.767
Inteiro curto com sinal	signed short int	2	-32.768 a 32.767
Inteiro curto sem sinal	unsigned short int	2	0 a 65.535
Inteiro longo	long int	4	-2.147.483.648 a 2.147.483.647
Inteiro longo com sinal	signed long int	4	-2.147.483.648 a 2.147.483.647
Inteiro longo sem sinal	unsigned long int	4	0 a 4.294.967.295
Ponto flutuante com precisão simples	**float**	4	3.4 E-38 a 3.4E+38
Ponto flutuante com precisão dupla	**double**	8	1.7 E-308 a 1.7E+308
Ponto flutuante com precisão dupla longo	long double	16	3.4E-4932 a 1.1E+4932

Observações:

- O tamanho de cada tipo pode ser diferente em computadores ou compiladores C diferentes, portanto, se for necessário saber exatamente o tamanho, é importante testá-lo no ambiente em uso. Veja, posteriormente, a utilização do operador ***sizeof*** para determinar o tamanho de um tipo de dados.
- Na tabela tem-se o tipo de dados *char* que é o tipo utilizado para caracteres, no entanto, sua faixa de dados está indicada com números. A explicação é que a linguagem C converte os números em caracteres através da tabela ASCII.

Exemplo:

```
// Programa : Exemplo 2
// Objetivo: Mostrar a declaração de variáveis e seus tipos
// Dados de entrada:
// Dados de saída:
```

```
int main ( void)
{
        int     X;          // resultado do acréscimo
        char    resposta;   // resposta para executar o programa 'S' ou 'N'
        float   preco;      // preço do produto a ser reajustado

    ...return 0;
}
```

4.8 Operador de Atribuição

O operador utilizado para atribuição na linguagem C é o **=** . A variável que recebe o valor é especificada a esquerda do operador e o valor atribuído é especificado a direita do operador.

SINTAXE:

<*nome_da_variável*> = <*valor*> ;
ou
<*nome_da_variável*> = <*expressão*> ;
ou
<*nome_da_variável*> = <*variável*> ;

Exemplo:

```
// Programa : Exemplo 3
// Objetivo: Mostrar as atribuições de variáveis em C

int main (void)
{
        int     x;          // armazena resultado das atribuições
        int     y;          // armazena um valor constante

    y = 50;                 // a variável y recebe o valor 50
    x = 10;                 // a variável x recebe o valor 10
    x = y;                  // a variável x recebe o valor armazenado em y
    x = y + 2;              // x recebe o valor armazenado em y mais dois
    return 0;
}
```

4.9 Inicialização de Variáveis

É possível combinar uma declaração de variável com o operador de atribuição para que a variável tenha um valor ao mesmo tempo de sua declaração. Isto é chamado de inicialização de variáveis.

Isto é importante, pois quando o C cria uma variável ele não a inicializa. Isto significa que até que um primeiro valor seja atribuído à nova variável ela tem um valor indefinido e que não pode ser utilizado para nada. Nunca presuma que uma variável declarada vale zero ou qualquer outro valor.

SINTAXE:

<*tipo da variável*> <*nome_da_variável*> = constante;

Exemplo:

```
// Programa : Exemplo 4
// Objetivo: Mostrar a inicialização de variáveis em C

int main (void)
{
        int         x = 5;
        char        ch = 'A';
        float       altura = 1.60;

...
        return 0;
}
```

Ressalte-se que, em C, uma variável tem que ser declarada no início de um bloco de código.

4.10 Expressões Aritméticas

Para a construção de programas todas as expressões aritméticas devem ser linearizadas, ou seja, colocadas em linhas. É importante também ressalvar o uso dos operadores correspondentes da aritmética tradicional para a computacional.

Exemplo:

$$\left[\frac{2}{3}+(5-3)\right]+1= \qquad \text{tradicional}$$

$$(2/3+(5-3))+1= \qquad \text{computacional}$$

4.10.1 Operadores Aritméticos

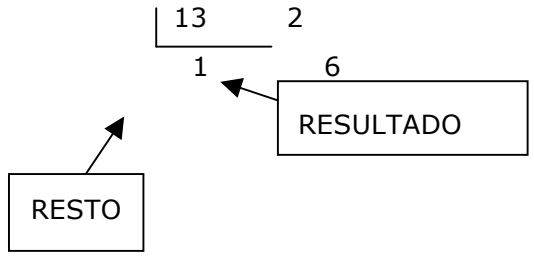
Para formar uma expressão matemática utiliza-se os operadores matemáticos. Os operadores matemáticos são representados da seguinte forma:

Operação	Símbolo
Adição	**+**
Subtração	**-**
Multiplicação	*****
Divisão	**/**
Resto da divisão	**% (RESTO)**

Observações:

- Utilizam-se parênteses para indicar a prioridade das operações.
- O operador Resto da Divisão (%) é utilizado somente com variáveis do tipo inteiro
- O operador / (divisão) quando aplicado a variáveis inteiras, nos fornece o resultado da divisão inteira; quando aplicado a variáveis em ponto flutuante nos fornece o resultado da divisão "real".

Exemplos:



13 / 2 = 6

13 % 2 = 1

```
// Programa : Exemplo 5
// Objetivo: Mostrar a atribuição de expressões a variáveis em C

int main (void)
{
   int       x, y;

       y = 2;
       x = 10;
       x = y + 2;
       x = ((y+3)*4) - 5;          // x recebe o valor 15
       x = (x % 2);                // x recebe o resto da divisão, igual a 1
       return 0;
}
```

```
// Programa : Exemplo 6
// Objetivo: Mostrar a atribuição de expressões a variáveis em C

int main (void)
{
   int       a = 17,  b = 3;
   int       x, y;
   float     z = 17., z1, z2;

       x = a / b;                  // o quociente da divisão:  x é 5
```

```
        y = a % b;                  // o resto da divisão: y é 2
        z1 = z / b;                 // o quociente da divisão: z1 é 5.666667
        z2 = a / b;                 // o quociente da divisão: z2 é 5.0
        return 0;
}
```

Observação:

- Note que, na linha do exemplo 3a correspondente a z2, primeiramente é feita uma divisão inteira (pois os dois operandos são inteiros). Somente após efetuada a divisão é que o resultado é atribuído a uma variável *float*.

4.10.2 Funções matemáticas pré-definidas

Um programa, escrito em linguagem C, pode fazer uso das seguintes funções matemáticas já definidas na biblioteca **math.h .**

Função	Operação básica matemática
abs(x)	Retorna valor absoluto de x
sin (x)	Retorna o seno de x (x deve ser do tipo double)
tan (x)	Retorna o arco da tangente x (x deve ser do tipo double)
cos(x)	Retorna o cosseno de x (x deve ser do tipo double)
pow (x, y)	Retorna o x elevado a y (x^y) (x e y devem ser double)
sqrt (x)	Retorna a raiz quadrada de x

Exemplo:

```
// Programa : Exemplo 7
// Objetivo: Mostrar as funções matemáticas básicas em C

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

int main (void)
{
   double      a , b, c, d ;

        a = fabs (-3.5);
       b = log (a);
       c = sqrt (b);
      d = pow ( a, b);
      printf (" a = %6.2f ,  b =  %6.2f ,  c = %6.2f ,  d = %6.2f ", a, b, c, d);
       return 0;
}
```

4.11 Entrada e Saída de Dados em C

Neste momento no curso veremos somente as funções que realizam entrada e saída de dados E/S em console, ou seja, realizaremos somente operações de E/S que ocorram no teclado e na tela do seu computador.

4.11.1 A Função printf( )

A função printf(),da biblioteca **stdio.h,** é uma função que permite efetuar a saída de informações no vídeo.

SINTAXE:

```
printf ( " expressão de controle " ) ;

ou

printf ( "expressão de controle " , lista de argumentos ) ;
```

- Na expressão de controle tem-se uma descrição de tudo que a função vai colocar na tela. Ela mostra não apenas os caracteres que devem ser colocados na tela, mas também quais as variáveis e suas respectivas posições. Isto é feito usando-se os códigos de controle, que usam a notação %.

  Na expressão de controle indica-se quais, de qual tipo e em que posição estão as variáveis a serem apresentadas. Portanto, a expressão de controle pode conter: texto, especificadores ou códigos de formato, códigos de barra invertida, por exemplo.

- A lista de argumentos representa os dados de saída individuais, que podem ser constantes, variáveis, expressões ou funções.

- É muito importante que, para cada código de controle, tenha-se um argumento na lista de argumentos. Apresenta-se a seguir alguns dos códigos % e alguns códigos de barra invertida:

Código de formato	Significado
% c	caracter
% d *ou* % i	inteiro
% f	Ponto flutuante (*float*)
% s	Cadeia de caracter
% o	octal
% u	Decimal sem sinal
% x	hexadecimal
% e	Notação científica
% %	Coloca na tela um %

Código para caracteres especiais	Formato
\b	Retrocesso (backspace).
\f	Alimentação de formulário.
\n	Nova linha.
\r	Retorno de Carro.
\t	Tabulação horizontal
\v	Tabulação vertical.
\0	Nulo.
\\	Barra invertida.
\´ (\")	Aspas simples (dupla)

Exemplos:

```
// Programa : Exemplo 8
// Objetivo: Mostrar a função de saída printf( )

#include <stdio.h>

int main (void)
{
    int num;

        num = 1;
        printf ("Eu sou um simples");
        printf ("computador. \n");
        printf ("Meu numero favorito eh %d porque ele eh o primeiro. \n", num);
        return 0;
}
```

A execução do programa acima resultará em:

_____

Eu sou um simples computador.

Meu numero favorito eh 1 porque ele eh o primeiro.

_____

```
// Programa : Exemplo 9
// Objetivo: Mostrar a função de saída printf( )

#include <stdio.h>

int main (void)
{
        printf ("%s esta a %d milhoes de milhas \n do Sol.", "Venus", 67);
        return 0;
}
```

A execução do programa acima resultará em:

```
Venus esta a 67 milhoes de milhas
 do Sol.
```

```
// Programa : Exemplo 10
// Objetivo: Mostrar a função de saída printf( )

#include <stdio.h>

int main (void)
{
        printf ("Teste %% %%\n");
        printf ("%f\n",40.345);
        printf ("Um caractere %c e um inteiro %d\n",'D',120);
        printf ("%s%d%%","Juros de ",10);
        return 0;
}
```

A execução do programa acima resultará em:

```
Teste % %
40.345
Um caractere D e um inteiro 120
Juros de 10%
```

Na função *printf( )* é possível estabelecer o tamanho mínimo para a impressão de um campo. Veja o exemplo seguinte.

```
// Programa : Exemplo 11
// Objetivo: Mostrar a função de saída printf( )

#include <stdio.h>

int main (void)
{
        printf ("Os alunos são %2d. \n", 350);
        printf ("Os alunos são %4d. \n", 350);
        printf ("Os alunos são %5d. \n", 350);
        return 0;
}
```

A execução do programa acima resultará em:

```
Os alunos são 350.
Os alunos são _350.
Os alunos são _ _350.
```

Pode-se ainda usar tamanho de campos com números em ponto flutuante para obter precisão e arredondamento, bem como alinhar à direita ou à esquerda.

Veja os exemplos seguintes.

```
// Programa : Exemplo 12
// Objetivo: Mostrar a função de saída printf( )

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

int main (void)
{
        printf ("%4.2f\n", 3456.78);
        printf ("%3.2f\n", 3456.78);
        printf ("%3.1f\n", 3456.78);
        printf ("%10.3f\n", 3456.78);
        system (“Pause”);
        return 0;
}
```

A execução do programa acima resultará em:

```
_____________________________
        3456.78
        3456.78
        3456.7
        - - 3456.780
_____________________________
```

```
// Programa : Exemplo 13
// Objetivo: Mostrar a função de saída printf( )

#include <stdio.h>

int main (void)
{
        printf ("%.2f %.2f %.2f\n", 8.0, 15.3, 584.13);
        printf ("%.2f %.2f %.2f\n", 834.0, 1500.55, 4890.21);
        return 0;
}
```

A execução do programa acima resultará em:

```
_____________________________
        8.00 15.30 584.13
        834.00 1500.55 4890.21
_____________________________
```

```
// Programa : Exemplo 14
// Objetivo: Mostrar a função de saída printf( )

#include <stdio.h>

int main (void)
{
        printf ("%10.2f %10.2f %10.2f\n", 8.0, 15.3, 584.13);
        printf ("%10.2f %10.2f %10.2f\n", 834.0, 1500.55, 4890.21);
```

```
        return 0;
}
```

A execução do programa acima resultará em:

```
---8.00          15.30          584.13
834.00           1500.55         4890.21
```

4.11.2 A Função scanf( )

Para executar um comando de entrada de dados, utiliza-se a função ***scanf( )*** da biblioteca **stdio.h** .

SINTAXE:

```
scanf ( " % t " , & NOME_DA_VARIÁVEL ) ;
```

- O NOME_DA_VARIÁVEL indica em qual variável o valor lido será armazenado
- O símbolo '**&**' indica que o valor lido deve ser armazenado no endereço referente a variável especificada, ou seja, qual a localização, na memória do micro, da variável definida.
- **%t** é um conversor que indica o tipo de informação que deve ser lida. Deve-se substituir a letra **t** pelo caractere de acordo com a tabela:

Código do formato	Significado
%s	Lê uma cadeia de caracteres (*string*)
%d	Lê um número inteiro da base 10 (decimal)
%o	Lê um número inteiro da base 8 (octal)
%x	Lê um número inteiro da base 16 (hexadecimal)
%f	Lê um número de ponto flutuante (real)

Exemplos:

```
// Programa : Exemplo 15
// Objetivo: Mostrar a função de entrada scanf( )

#include <stdio.h>

int main (void)
{
   int valor;

      printf ("Digite um valor inteiro e tecle Enter : ");
      scanf ("%i", &valor);
      printf ("Valor em Decimal = %d \n", valor);
      printf ("Valor em Hexadecimal = %x \n", valor);
      printf ("Valor em Octal = %o \n", valor);
      printf ("Valor em caractere (ASCII) = %c \n", valor);
      return 0;
}
```

A execução do programa anterior resultará em:

```
_____________________________
            Digite um valor inteiro e tecle Enter: 65
            Valor em Decimal = 65
            Valor em Hexadecimal = 41
            Valor em Octal = 101
            Valor em caractere (ASCII) = A
_____________________________
```

```
// Programa : Exemplo 16
// Objetivo: Mostrar a função de entrada scanf( )

#include <stdio.h>

int main (void)
{
   char a;

      printf ("Digite um caractere e veja-o em decimal,  ");
      printf ("octal e hexadecimal. \n");
      scanf ("%c", &a);
      printf ("\n%c=%d decimal, %o octal e %x hexadecimal \n", a, a, a, a);
      return 0;
}
```

A execução do programa acima resultará em:

```
_____________________________
            Digite um caractere e veja-o em decimal, octal e hexadecimal.
            m

            m=109 decimal, 155 octal e 6d hexadecimal
_____________________________
```

O programa a seguir faz com que *scanf()* atribua o primeiro dígito da entrada à variável *um_digit*, os próximos dois dígitos à variável *dois_digit* e os últimos três à variável *tres_digit*.

```
// Programa : Exemplo 17
// Objetivo: Mostrar a função de entrada scanf( )

#include <stdio.h>

int main (void)
{
    int um_digit, dois_digit, tres_digit;

        printf ("Digite um valor de 6 digitos e tecle Enter : ");
        scanf ("%1d%2d%3d", &um_digit, &dois_digit, &tres_digit);
        printf ("Valores individuais são %d %d %d \n", um_digit, dois_digit,
tres_digit);
        return 0;
}
```

A execução do programa acima resultará em:

```
_____________________________
            Digite um valor de 6 digitos e tecle Enter
            123456
            Valores individuais são 1  23  456
_____________________________
```

```
// Programa : Exemplo 18
// Objetivo: Mostrar a função de entrada scanf( )

#include <stdio.h>

int main (void)
{
    int dia, mes, ano;

        printf("Digite a data de hoje no formato dd-mm-aaaa ");
        scanf("%d-%d-%d", &dia, &mes, &ano);
        printf("Data: %d-%d-%d \n", dia, mes, ano);
        return 0;
}
```

A execução do programa acima resultará em:

```
_____________________________________________
            Digite a data de hoje no formato  dd-mm-aaaa
12-03-2001
Data: 12-3-2001
_____________________________________
```

4.12 Variáveis do tipo Caractere

4.12.1 As funções getche( ) e getch ( )

A biblioteca **conio.h** oferece funções que lêem um caractere no instante em que é digitado, sem esperar **[enter].** Para isto as funções mais usadas, quando se está trabalhando em ambiente DOS ou Windows, são *getch()* e *getche().*

Ambas retornam o caractere pressionado. *getche()* imprime o caractere na tela antes de retorná-lo e *getch()* apenas retorna o caractere pressionado sem imprimi-lo na tela.

Geralmente estas funções não estão disponíveis em ambiente Unix (compiladores cc e gcc) e podem ser substituídas pela função *scanf()*, porém sem as mesmas funcionalidades.

Exemplos:

```
// Programa : Exemplo 19
// Objetivo: Mostrar a função de entrada getch ( )

#include <stdio.h>
#include <conio.h>
/* Este programa usa conio.h . Se você não tiver a conio, ele não funcionará no Unix */
int main (void)
{
      char Ch;
      Ch=getch();
      printf ("Voce pressionou a tecla %c", Ch);
}
```

Equivalente para o ambiente Unix do programa acima, sem usar getch():

```
// Programa : Exemplo 20
// Programador : Fulano de Tal
// Data :
// Objetivo: Mostrar a função de entrada scanf( )
#include <stdio.h>

int main (void)
{
      char Ch;
      scanf(" %c ", &Ch);
      printf ("Voce pressionou a tecla %c", Ch);
}
```

4.13 Variáveis do tipo Cadeia de Caracteres

Apesar de na Linguagem C não haver um tipo de dados *string*, pode-se trabalhar com uma cadeia de caracteres.

Na linguagem C uma *string* é um vetor de caracteres terminado com um caractere nulo. O caractere nulo é um caractere com valor inteiro igual a zero (código ASCII igual a 0). O terminador nulo também pode ser escrito usando a convenção de barra invertida do C como sendo '\0'.

Embora o assunto vetores seja discutido posteriormente, será visto aqui os fundamentos necessários para que seja possível utilizar as *strings*. Para declarar uma *string*, podemos usar o seguinte formato geral:

SINTAXE:

```
char <nome_da_string> [TAMANHO];
```

Isto declara um vetor de caracteres (uma *string*) com número de posições igual a tamanho. Note que, como temos que reservar um caractere para ser o terminador nulo, declaramos o comprimento da *string* como sendo, no mínimo, um caractere maior que a maior *string* que se pretende armazenar. Supor que se declare uma *string* de 7 posições e se coloque a palavra João nela. Temos:

J	**O**	**A**	**O**	**\0**	**...**	**...**

No caso acima, as duas células não usadas têm valores indeterminados. Isto acontece porque o C não inicializa variável, cabendo ao programador esta tarefa. Portanto, as únicas células que são inicializadas são as que contêm os caracteres 'J', 'o', 'a', 'o' e '\0' .

A função *scanf()*, não é muito adequada para pegar do teclado um sequência de caracteres, pois um simples espaço a mesma enxerga como um terminador de *string* e a partir daí não pega mais nada do teclado. Além disso, você precisa pressionar **[enter]** depois da sua entrada para que a função termine a leitura. Para solucionar este problema, deve-se usar a função *gets()*, que será vista a seguir.

4.13.1 A Função gets( )

Caso se deseje ler uma *string* fornecida pelo usuário pode-se usar a função gets(). Características:

- Arquivo de cabeçalho a ser incluído: *stdio.h*
- Lê uma *string* de caracteres entrada pelo teclado e coloca-o no endereço apontado por seu argumento.
- Pode-se digitar caracteres até que seja pressionada a tecla **[Enter]**, quando então *gets()* retorna.
- Quando pressionada a tecla Enter é acrescentado um terminador nulo ao final da string.
- Pode-se corrigir erros de digitação usando a tecla <backspace>.

Exemplo:

```
// Programa : Exemplo 21
// Objetivo: Mostrar a função de entrada gets( )
#include <stdio.h>

int main (void)
{
   char cadeia[100];

      printf ("Digite uma cadeia de caracteres ( string) : ");
      gets (cadeia);
      printf ("\n Voce digitou %s", cadeia);
}
```

Observação:

➔ Neste programa, o tamanho máximo da *string* que você pode entrar é uma string de 99 caracteres. Se você entrar com uma *string* de comprimento maior, o programa irá aceitar, mas os resultados podem ser desastrosos. Será visto porque posteriormente.

➔ A função fflush(stdin), normalmente é usada quando se faz várias chamadas da função gets ( ). O objetivo é limpar o buffer, ou área de armazenamento temporário.

4.14 Exercícios

1) Crie seu diretório de desenvolvimento, invoque o ambiente de desenvolvimento do DevC++ e digite os programas exemplos que foram apresentados neste capítulo, execute os mesmos e analise os resultados.

2) Um dos alunos preparou o seguinte programa e o apresentou para ser avaliado. Ajude-o.

```
// Programa : Exercício 1
// Objetivo: Encontrar erros

#include <stdio.h>

 main {}
(

        printf ( Existem %d semanas no ano.  , 56);

)
```

R: _____

3) O programa seguinte tem vários erros em tempo de compilação. Primeiro, analise estes erros e, depois, execute-o e observe as mensagens apresentadas por seu compilador.

```
// Programa : Exercício 2
// Objetivo: Encontrar erros

#include <stdio.h>

int Main ( )
{
    int a=1; b=2, c=3;

        printf (" Os numeros sao:  %d %d %d \n , a, b, c, d);
        return 0;
}
```

R: _____

4) Qual será a saída do programa abaixo?

```
// Programa : Exercício 3
// Objetivo: Analisar a saida

#include <stdio.h>

int main (void)
{

        printf (" %s\n%s\n%s ", "um", "dois", "tres");
        return 0;
}
```

R: _____

5) Qual será a impressão obtida por cada uma das instruções seguintes? .

```
// Programa : Exercício 4a
// Objetivo: Escrever a saída

#include <stdio.h>

int main (void)
{

                printf (" Bom dia !  Shirley. ");
                printf (" Voce já tomou cafe ? \n");
                return 0;
}
```

R: _____

```
// Programa : Exercício 4b
// Objetivo: Escrever a saída

#include <stdio.h>

int main (void)
{

                printf (" A solucao não existe!\nNao insista ");
                return 0;
}
```

R: _____

```
// Programa : Exercício 4c
// Objetivo: Escrever a saída

#include <stdio.h>

int main (void)
{

        printf (" Duas linhas de saida\nou uma ? ");
        return 0;
}
```

R: _____

6) Identifique o tipo das seguintes constantes:

a. ' \ r
b. 2130
c. –123
d. 33.28
e. 0x42
f. 0101
g. 2.0e30
h. ' \xDC '
i ' \ " '
j. ' \ \ '
k ' F '
l. 0
m. ' \0 '

7) O que é uma variável em C?

_____

_____

8) Quais os 5 tipos básicos de variáveis em C ?

_____

9) Quais dos seguintes nomes são válidos para variáveis em C ?

a. 3ab
b. _sim
c. n_a_o
d. 00FIM
e. int
f. A123

g. x**x
h. __A
i. y-2
j. OOFIM
k. \meu
l. *y2

10) Quais das seguintes instruções, para declaração de variáveis, são corretas?

a. int a;
b. float b;
c. double float c
d. unsigned char d;
e. long float e;

11) O tipo **float** ocupa o mesmo espaço que _____ variáveis do tipo char.

12) Tipos de variáveis **long int** podem conceber números não maiores que o dobro da maior variável do tipo **int**. Verdadeiro ( ) Falso ( )

13) Escreva um programa que imprima na tela:

um

dois

três

14) Escreva em C os exercícios a partir do 6º. do capítulo anterior.

5. Estruturas Lógicas de Seleção

5.1 Introdução

Neste capítulo iniciaremos com as instruções estruturadas que se diferenciam das simples, pelo fato de possuírem outras internamente, formando uma *estrutura de controle ou estruturas lógicas.*

As estruturas de controle ou estruturas lógicas são: *sequenciais, de seleção e de repetição.* Neste capítulo iniciaremos com as estruturas sequenciais. Porém antes de iniciar com estas estruturas, veremos os operadores relacionais e lógicos que são necessários nesta fase.

5.2 Expressões Lógicas

Expressão lógica é uma expressão cujos operadores são os operadores lógicos ou relacionais e cujos operandos são relações e/ou variáveis do tipo lógico.

5.2.1 Operadores Relacionais

Uma relação é uma **comparação** realizada entre valores **do mesmo tipo**, assim os operadores relacionais são utilizados para comparar dados do mesmo tipo. Os valores a serem comparados podem ser constantes, variáveis ou expressões aritméticas.

Os resultados obtidos dos operadores relacionais são sempre valores lógicos (VERDADEIRO ou FALSO).

Operadores Relacionais		**Utilização**
Algoritmo	**Linguagem C**	
=	= =	Igual a
>	>	Maior que
<	<	Menor que
≠ ou < >	! =	Diferente de
≥	> =	Maior ou igual a
≤	< =	Menor ou igual a

Exemplos em Algoritmo:

2 + 5 > 4 ➔ Verdadeiro	3 ≠ 3 ➔ Falso
2 * 4 = 64/8 8 = 8 ➔ Verdadeiro	15 % 4 < 19 % 6 3 < 1 ➔ Falso
3 * 5 / 4 ≤ pow(3,2)/0,5 15 / 4 ≤ 9/0,5 3 ≤ 18 ➔ Verdadeiro	2+ 8 % 7 ≥ 3 * 6 - 15 2 + 1 ≥ 18 - 15 3 ≥ 3 ➔ Verdadeiro

Exemplo em linguagem C :

1. cont <= 100
2. sqrt(a+b+c) > 0.005
3. resposta = = 0
4. total >= minimo
5. ch1 < 'T'
6. letra != 'x'

As 4 primeiras expressões envolvem operandos numéricos.

A 5ª expressão envolve uma variável do tipo caractere ch1. Esta expressão será VERDADEIRA se o caractere representado por ch1 vier antes de T no conjunto de caracteres, isto é, se o valor numérico usado para codificar o caractere for menor que o valor numérico usado para codificar a letra T.

A 6ª expressão utiliza uma variável do tipo caractere letra. Essa expressão será VERDADEIRA se o caractere representado por letra for qualquer outro menos x.

Em linguagem C todas as expressões lógicas e relacionais produzem resultado 1, se o resultado da expressão for VERDADEIRO, e 0, se o resultado for FALSO.

Exemplos em linguagem C:

2 + 5 > 4 ➔ 1	3 ≠ 3 ➔ 0
2 * 4 = 64/8 8 = 8 ➔ 1	15 % 4 < 19 % 6 3 < 1 ➔ 0
3 * 5 / 4 ≤ pow(3,2)/0,5 15 / 4 ≤ 9/0,5 3 ≤ 18 ➔ 1	2+ 8 % 7 ≥ 3 * 6 - 15 2 + 1 ≥ 18 - 15 3 ≥ 3 ➔ 1

5.2.2 Operadores Lógicos

Os operadores lógicos servem para combinar resultados de expressões, retornando se o resultado final é VERDADEIRO (TRUE) ou FALSO (FALSE).

Operadores Lógicos		**Características**
Algoritmo	**Linguagem C**	
E	**&&**	A operação E resulta em VERDADEIRO ⇔ se todos os operandos forem VERDADEIRO
OU	**| |**	A operação OU resulta em VERDADEIRO ⇔ Quando pelo menos um dos operandos for VERDADEIRO
NÃO	**!**	A operação NÃO inverte o valor da expressão ou condição, se VERDADEIRO inverte para FALSO e vice-versa.

Os operadores lógicos realizam as operações da Álgebra Booleana. Os operadores são os seguintes:

a) E

b) OU

c) NÃO

Tabela Verdade:

p	**q**	**p E q**	**p OU q**
falso	falso	Falso	falso
falso	verdadeiro	falso	verdadeiro
verdadeiro	falso	falso	verdadeiro
verdadeiro	verdadeiro	verdadeiro	verdadeiro

Exemplos:

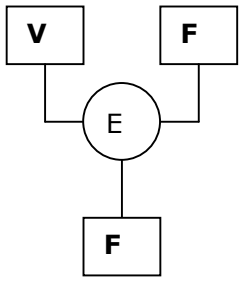
1. Se chover E relampejar, eu choro de raiva. Quando eu choro de raiva?

   A proposição só será verdadeira (ou seja, eu choro de raiva) quando os termos chover e relampejar forem simultaneamente verdadeiros.

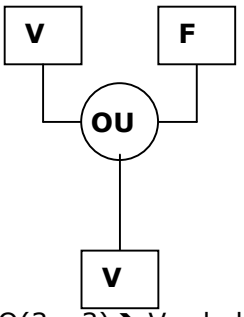
2. Se chover OU relampejar, eu choro de raiva. Quando eu choro de raiva?

Com o operador lógico OU, as possibilidades de "eu chorar de raiva" se tornam maiores. Assim, a proposição será verdadeira em três situações: somente chovendo, somente relampejando, chovendo e relampejando.

2 + 5 > 4 E 3 ≠ 3 ➔ Falso



3. 2+5 > 4 OU 3 ≠ 3 ➔ Verdadeiro



4. NÃO(3 ≠ 3)➔ Verdadeiro

5.2.3 Prioridades entre Operadores em Algoritmo

Assim como acontece entre os operadores aritméticos, também existe uma relação de prioridade entre os operadores lógicos.

Nas expressões lógicas, vários níveis de parênteses podem ser utilizados com a finalidade de estabelecer uma nova ordem de prioridade de execução.

Prioridade	Operador
1ª	aritmético
2ª	relacional
3ª	NÃO
4ª	E
5ª	OU

Exemplos:

1. (A = 1 **E** (B + C ≠ 0 **OU** K ≤ 2)) é diferente de

   (A = 1 **E** B + C ≠ 0 **OU** K ≤ 2)

2. (**NÃO** (total ≥ 2 **E** A ≠ B) **OU** teste) é diferente de

   (**NÃO** total ≥ 2 **E** A ≠ B **OU** teste)

5.2.4 Operadores de Incremento e Decremento em Linguagem C

Os operadores de incremento e decremento são operadores unários, ou seja, são operadores que atuam em um único operando para produzir um novo valor.

O operador de incremento (++) soma 1 ao seu operando e o operador de decremento(--) subtrai 1. Normalmente usamos o operador de incremento.

Operador de Incremento e Decremento	Significado
x ++	x = x + 1
x --	x = x - 1

O aspecto não usual é que os operadores ++ e -- podem ser usados tanto como operadores pré-fixados (antes da variável, como em ++x) ou pós-fixados (após a variável, como em x++).

Em ambos os casos a variável x é incrementada, porém a expressão ++x incrementa x **antes** de usar o valor do operando, enquanto x++ incrementa x **após** seu valor ser usado.

Exemplo:

```
x = 10;
y = ++x;
```

Neste caso, y será igual a 11, uma vez que x é incrementado primeiro e

teremos, no final, y=11 e x=11.

Entretanto,

```
x = 10;
y =  x ++;
```

Neste caso, é atribuído 10 a y e depois incrementado.

teremos no final, y=10 e x=11.

Em ambos os casos, x se torna igual a 11, a diferença está em quando isso acontece.

Exemplo: Vamos analisar as duas expressões seguintes:

k = 3 * n ++;	Primeiro, n é multiplicado por 3, depois, o resultado é atribuído a k finalmente, n é incrementado de 1.
k = 3 * ++ n;	Primeiro, n é incrementado de 1, depois, n é multiplicado por 3 finalmente, o resultado é atribuído a k.

5.2.5 Prioridades entre Operadores em C

A seguir temos a tabela de precedência dos operadores em C.

Prioridade	Categoria	Operadores
1ª	Lógico NOT, incremento e decremento	**! ++ --**
2ª	Aritmético: multiplicação, divisão e resto	*** / %**
3ª	Aritmético: adição e subtração	**+ -**
4ª	Relacional: maior, maior e igual, menor e menor e igual	**> >= < <=**
5ª	Relacional: igual e diferente	**= = !=**
6ª	Lógico: AND	**&&**
7ª	Lógico: OR	**||**

5.3 Estrutura Sequencial

A estrutura sequencial de um algoritmo corresponde ao fato de que o conjunto de ações primitivas será executado em uma sequência linear de cima para baixo e da esquerda para a direita, ou seja, na mesma ordem em que aparecem escritas no texto do algoritmo.

O algoritmo seguinte ilustra o modelo básico da estrutura sequencial.

Nome: **Modelo Básico**
Objetivo: Mostrar o modelo básico de um algoritmo sequencial
Dados de Entrada:
Dados de Saída:
Variáveis:

Início

instrução 1
instrução 2

```
.
.
.
instrução n
Fim
```

Na estrutura sequencial em linguagem C os comandos são executados em uma sequência na ordem em que aparecem escritos no texto do programa.

```
instrução_1;
instrução_2;
instrução_3;
...
instrução_n;
```

Exemplo: Calcular a média aritmética semestral entre 2 notas bimestrais quaisquer fornecidas por um aluno.

```
Nome: Exemplo 1
Objetivo: Calcular a média aritmética semestral entre 2 notas bimestrais
Dados de Entrada: duas notas bimestrais: n1, n2
Saída: média aritmética semestral: media
Variáveis:
        real        n1   <armazena a primeira nota do bimestre>
                    n2   < armazena a segunda nota do bimestre >
                    media   < armazena a média semestral >
Início
        LEIA n1
        LEIA n2
        media ← (n1 + n2)/2
        IMPRIMA  media
Fim
```

Em linguagem C o algoritmo acima seria:

```
// Programa: Exemplo 1
// Objetivo: Calcular a média aritmética semestral entre 2 notas bimestrais

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

int main(void )
{
      float          n1,          //armazena a primeira nota do bimestre
                     n2,          //armazena a segunda nota do bimestre
                     media;       // armazena a média semestral

        printf ("Entre com a nota do 1° bimestre: ");
        scanf("%f", &n1);
        printf ("Entre com a nota do 2° bimestre: ");
```

```
        scanf("%f", &n2);
        media = (n1 + n2)/2;
        printf ("A media semestral e :  %.1f\n", media);
        return 0;
}
```

5.4 **Estrutura de Seleção ou Condicional**

Uma estrutura de seleção ou condicional permite a escolha de um grupo de ações ou instruções a ser executado quando determinadas **condições**, representadas por **expressões lógicas** ou **relacionais**, são ou não satisfeitas.

5.4.1 Seleção Simples

Quando precisamos testar uma condição antes de executar uma instrução, usamos a seleção simples.

SINTAXE no algoritmo:

```
SE  <condição>
ENTÃO
            instrução 1
            instrução 2
               .
               .
               .
            instrução n
FIM-SE
```

A <condição> é uma expressão lógica que gera um resultado VERDADEIRO ou FALSO.

Se a <condição> resultar em VERDADEIRO, a instrução especificada na cláusula **ENTÃO** será executada.

Caso contrário, ou seja, se a <condição> resultar em FALSO nenhuma instrução será executada e encerra-se a seleção simples **FIM-SE**. A próxima instrução a ser executada será a que estiver especificada após o **FIM-SE**.

Se a <condição> resultar em VERDADEIRO, então será executada a sequência de instruções especificadas no "bloco verdade", caso contrário, isto é, <condição> resultar em FALSO, nenhuma instrução será executada.

Fazemos esse tipo de operação na linguagem C com a instrução *if,* vejamos sua sintaxe:

SINTAXE em C:

```
if  ( condição )
{
                    instrução 1;
                    instrução 2;
                    .
                    .
                    .
                    instrução n;
}
```

Exemplo: Calcular a média aritmética semestral entre 2 notas bimestrais quaisquer fornecidas por um aluno, mostrando se ele foi aprovado. O aluno é aprovado quando sua média é maior ou igual a 6.0.

Nome: Exemplo 2
Objetivo: Calcular a média aritmética semestral entre 2 notas bimestrais e mostrar se o aluno foi aprovado.
Dados de Entrada: duas notas bimestrais: n1, n2
Saída: média aritmética semestral: media e mensagem de aprovação(quando o aluno for aprovado).
Variáveis:

```
        real        n1   <armazena a primeira nota do bimestre>
                    n2   < armazena a segunda nota do bimestre >
                    media   < armazena a média semestral >
```

Início

```
LEIA n1
LEIA n2
media ← (n1 + n2)/2

SE media ≥ 6.0 ENTÃO
   IMPRIMA  "Você foi aprovado !"
   IMPRIMA  "Parabéns !"
FIM-SE

1. IMPRIMA  media
```

Fim

```
// Programa: Exemplo 2
// Objetivo: Calcular a média aritmética semestral entre 2 notas bimestrais
//                                   e mostrar se o aluno foi aprovado
#include <stdio.h>
#include <conio.h>

int main(void )
{
      float                n1,          //armazena a primeira nota do bimestre
                           n2,          //armazena a segunda nota do bimestre
                           media;       // armazena a média semestral

        printf ("Entre com a nota do 1° bimestre: ");
        scanf("%f", &n1);
        printf ("\n Entre com a nota do 2° bimestre: ");
        scanf("%f", &n2);

        media = (n1 + n2)/2;

        if (media >= 6.0)
        {
                    printf ("\n Voce foi aprovado !! ");
                    printf ("\n Parabéns !");
        }
        printf ("\n A media semestral e :  %.1f", media);
        return 0;
}
```

5.4.2 Seleção Composta

Quando tivermos situações em que duas alternativas dependem de uma mesma condição, ou seja, temos uma ação para, quando a <condição> resultar em VERDADEIRO e outra para, quando a <condição> resultar em FALSO, usamos a estrutura de seleção composta.

Observamos que a existência do bloco verdade continua, sendo que este será executado caso a <condição> seja VERDADEIRA. Porém, caso o resultado da <condição> seja FALSO, a instrução após a cláusula **SENÃO** será executada.

Na seleção composta apenas um dos blocos será executado: o bloco verdade (bloco após a cláusula **ENTÃO**) ou o bloco falso (bloco após a cláusula **SENÃO**).

SINTAXE em algoritmo:

```
SE  <condição>
ENTÃO
            instrução 1A
            instrução 2A
               .
```

```
            .
        instrução nA

SENÃO
        instrução 1B
        instrução 2B
            .
            .
        instrução nB
FIM-SE
```

SINTAXE em C:

```
if  ( condição )
{       //início do bloco verdade
                instrução 1A;
                instrução 2A;
                ...
                instrução nA;
} //fim do bloco verdade
else
{ //início do bloco falso
                instrução 1B;
                instrução 2B;
                ...
                instrução nB;
} //fim do bloco falso
```

Exemplo: Calcular a média aritmética semestral entre 2 notas bimestrais quaisquer fornecidas por um aluno, mostrando se ele foi aprovado ou reprovado uma mensagem para cada situação. O aluno é aprovado quando sua média é maior ou igual a 6.0, caso contrário é reprovado.

```
Nome: Exemplo 3
Objetivo: Calcular a média aritmética semestral entre 2 notas bimestrais e mostrar se o
aluno foi aprovado ou reprovado
Dados de Entrada: duas notas bimestrais: n1, n2
Saída: média aritmética semestral: media e mensagem de aprovação ou reprovação.
Variáveis:
        real        n1   <armazena a primeira nota do bimestre>
                    n2   < armazena a segunda nota do bimestre >
                    media   < armazena a média semestral >
Início
    LEIA n1
    LEIA n2
    media ← (n1 + n2)/2
    SE media ≥ 6.0
      ENTÃO
            IMPRIMA  "Você foi aprovado !"
            IMPRIMA  "Parabéns !"
      SENÃO
```

```
                    IMPRIMA  "Você foi reprovado !"
                    IMPRIMA  "Estude mais !"
            FIM-SE

          IMPRIMA  media

    Fim
```

A tradução da estrutura condicional do algoritmo acima para a linguagem C está apresentada no trecho de código do programa abaixo :

```
void main(void)
{
        ...
        if (media >= 6.0)
        {
                printf ("\n Voce foi aprovado ! ");
                printf ("Parabens ! ");
        }
        else
        {
                printf ("\n Voce foi reprovado ! ");
                printf ("Estude mais ! ");
         }
        .
}
```

O código apresentado acima está longe de ser completo, o objetivo deste exemplo é destacar a estrutura da seleção composta em linguagem C. Fica, portanto, como exercício a tradução completa do algoritmo Exemplo 3 para a linguagem C.

5.4.3 Seleção Encadeada ou Aninhada

A seleção encadeada ou aninhada é o agrupamento de várias seleções (internas) a uma seleção.

As instruções que compõem um bloco verdade ou falso podem ser também uma estrutura de seleção, isso significa que podemos testar mais que uma condição.

SINTAXE em algoritmo:

```
SE <condição1>
ENTÃO
        instrução 1A
        instrução 2A
          .
        instrução nA
SENÃO
 SE <condição2>
    ENTÃO
        instrução 1B
        instrução 2B
          .
        instrução nB
     SENÃO
        SE <condição3>
        ENTÃO
              instrução 1C
              instrução 2C
                .
              instrução nC

        SENÃO
              instrução 1D
              instrução 2D
                .
              instrução nD
        FIM-SE
     FIM-SE
FIM-SE
```

Estrutura de seleção encadeada ou alinhada em linguagem C aninhada é o agrupamento de várias seleções (internas) a uma seleção. O único cuidado que devemos ter é o de saber exatamente a qual if um determinado else está ligado.

```
if  ( condição1 )
{ //início do bloco verdade 1
            instrução 1A
            instrução 2A
               .
            instrução nA
} // fim do bloco verdade 1
else
{
            if  ( condição2 )
            { // início do bloco verdade 2
                    instrução 1B
                    instrução 2B
                       .
                    instrução nB
            }  //fim do bloco verdade 2
            else
            {
                    if  ( condição3 )
                    {  // início do bloco verdade 3
                            instrução 1C
                            instrução 2C
                               .
                            instrução nC
                    }  //fim do bloco verdade 3
                    else
                    {  // início do bloco falso
                            instrução 1D
                            instrução 2D
                               .
                            instrução nD
                    }  //fim do bloco falso
              }
```

```
}
```

Exemplo: Calcular a média aritmética semestral entre 2 notas bimestrais quaisquer fornecidas por um aluno, mostrando se ele foi aprovado, reprovado ou ficou de exame. O aluno é aprovado quando sua média é maior ou igual a 6.0. Quando a média é maior igual 2.0 e menor que 6.0 o aluno ficou de exame, caso contrário é reprovado.

```
Nome: Exemplo 4
Objetivo: Calcular a média aritmética semestral entre 2 notas bimestrais e mostrar se o
aluno foi aprovado, reprovado ou ficou de exame.
Dados de Entrada: duas notas bimestrais: n1, n2
Saída: média aritmética semestral: media e mensagem de aprovação ou reprovação.
Variáveis:
        real    n1   <armazena a primeira nota do bimestre>
                n2   < armazena a segunda nota do bimestre >
                media   < armazena a média semestral >

Início

    LEIA n1
    LEIA n2

    media ← (n1 + n2)/2

    SE media ≥ 6.0
        ENTÃO
                IMPRIMA  "Você foi aprovado !"
                IMPRIMA  "Parabéns !"
        SENÃO
                SE media ≥ 2.0
                    ENTÃO
                        IMPRIMA  "Você ficou de exame !"
                        IMPRIMA  "Aproveite essa chance!"
                    SENÃO
                        IMPRIMA  "Você foi reprovado !"
                        IMPRIMA  "Estude mais !"
                    FIM-SE
  FIM-SE

  IMPRIMA  media

Fim
```

Exemplo: Ler três valores para os lados de um triângulo, considerando os lados como: A, B e C. Verificar se os lados fornecidos formam um triângulo. Se for esta condição verdadeira, deverá ser indicado qual tipo de triângulo foi formado: isósceles, escaleno ou equilátero.

Resolução:

Devemos saber primeiramente qual a definição de um triângulo.

Triângulo é uma forma geométrica (polígono) composta de três lados, onde cada lado é menor que a soma dos outros dois lados.

Perceba que isto é uma regra (uma condição) e deverá ser considerada. É um triângulo quando A < B + C, quando B < A + C e quando C < A + B.

Tendo certeza que os valores informados para os três lados formam um triângulo, serão então, analisados os valores para se estabelecer qual tipo de triângulo será formado: isósceles, escaleno ou equilátero.

- isósceles quando possui dois lados iguais e um diferente, sendo A=B ou A=C ou B=C;
- escaleno quando possui todos os lados diferentes, sendo A < > B e B < > C; e
- equilátero quando possui todos os lados iguais, sendo A=B e B=C.

Etapas principais do algoritmo:

- Ler três valores para os lados de um triângulo: A, B e C;
- Verificar se cada lado é menor que a soma dos outros dois lados

Se sim, saber se A=B e se B=C, sendo verdade o triângulo é equilátero

Se não, verificar se A=B ou se A=C ou se B=C, sendo verdade o triângulo é isósceles, caso contrário o triângulo é escaleno.

- Caso os lados fornecidos não caracterizem um triângulo, avisar a ocorrência.

Nome: Exemplo 5
Objetivo: Verificar se os 3 valores dados formam um triângulo e, se forem, verificar se compõem um triângulo equilátero, isósceles ou escaleno. Informar se não compuserem nenhum triângulo.
Dados de Entrada: Três valores A, B e C
Saída: Mensagem com o tipo de triângulo formado: "triângulo equilátero", "triângulo isósceles", "triângulo escaleno" ou uma mensagem "Estes valores dados não formam um triângulo"
Variáveis:

inteiro	A	<armazena o primeiro lado do suposto triângulo>
	B	< armazena o segundo lado do suposto triângulo >
	C	< armazena o terceiro lado do suposto triângulo >

Início

```
LEIA A
LEIA B
LEIA C

SE ( (A < B + C) E  (B < A + C) E (C < A + B))
        ENTÃO  SE ( (A = B) E  (B = C))
                    ENTÃO
                        IMPRIMA  "Triângulo Equilátero"
                    SENÃO
                        SE ( (A = B) OU  (A = C) OU (B = C))
                            ENTÃO
                                IMPRIMA  " Triângulo Isósceles "
                            SENÃO
                                IMPRIMA  " Triângulo Escaleno "
                            FIM-SE
                    FIM-SE
        SENÃO
                IMPRIMA  " Estes valores não formam um triângulo "
    FIM-SE
Fim
```

```
void main(void)
{
        ...
    if (a < b + c  &&  b < a + c  &&  c < a + b)
    {
        if (a == b  &&  b == c) {
                        printf ("Triangulo equilatero \n");
        }
        else
        {
                if (a == b  ||  a == c  ||  c == b)
                {
                        printf ("Triangulo isosceles \n");
                }
                else
                {
                        printf ("Triangulo escaleno \n");
                }
        }
      }
      else
```

```
        {
                printf ("Os valores fornecidos não formam um triangulo \n");
        }
                ...
}
```

5.4.4 ***Seleção de Múltipla Escolha***

Algumas vezes, em um problema necessitamos escolher uma entre várias alternativas, apesar da seleção encadeada ou aninhada ser uma opção para este tipo de construção, ela não é uma maneira elegante de construção, além de muitas vezes dificultar o entendimento de sua lógica. Neste caso, utilizamos a seleção de múltipla escolha.

SINTAXE em algoritmo:

```
ESCOLHA  <variável>
        CASO valor1 :
                instrução 1.1
                instrução 1.2
                ...
                instrução 1.n

        CASO valor2 :
                instrução 2.1
                instrução 2.2
                ...
                instrução 2.n


                .
        CASO valorn :
                instrução n.1
                instrução n.2
                ...
                instrução n.n
        CASO CONTRÁRIO :
                instrução p.1
                instrução p.2
                ...
                instrução p.n
FIM-ESCOLHA
```

Caso o conteúdo de <variável> seja igual ao valor de uma <opção>, então a instrução correspondente será executada. Caso contrário, serão inspecionados os outros casos até ser encontrada uma igualdade ou terminarem os casos. Para executarmos uma instrução que se verifica com todos os outros valores, exceto os discriminados caso a caso, incluímos outra situação **CASO CONTRÁRIO**.

A estrutura de seleção de múltipla escolha que, na linguagem C é chamada de **switch.** Nesta instrução **switch**, a variável é testada sucessivamente contra uma lista de **inteiros ou constantes do tipo caracter**. Quando uma associação é encontrada, a instrução ou bloco de instruções, associada com a constante, são executadas.

A declaração **default** é opcional e será executada apenas se a variável, que está sendo testada, não for igual a nenhuma das constantes. Se não estiver presente, nenhuma ação será realizada se todas as correspondências falharem.

O comando **break**, faz com que o **switch** seja interrompido assim que uma das declarações seja executada. Mas ele não é essencial ao comando **switch**. Se após a execução da declaração não houver um **break**, o programa continuará executando. Isto pode ser útil em algumas situações, mas recomenda-se cuidado.

SINTAXE em C:

```
switch  ( variável )
        {
        case valor1:
        instrução 1.1;
        instrução 1.2;
        ...
        instrução 1.n;
        break;
        case valor2:
        instrução 2.1;
        instrução 2.2;
        ...
        instrução 2.n;
        break;
            . . .
        case valorn:
        instrução n.1;
        instrução n.2;
```

```
            ...
            instrução n.n;
            break;
            default:
            instrução p.1;
            instrução p.2;
            ...
            instrução p.n;
}
```

Obs: A variável e os valores valor1, valor2, ..., valorn devem ser do mesmo tipo.

Exemplo:

Determinar se a uma letra digitada pelo usuário é vogal ou consoante.

```
Nome: Exemplo 6
Objetivo: Verificar se a letra recebida como entrada é Vogal ou Consoante
Dados de Entrada: letra
Saída: Mensagem de classificação da letra.
Variáveis:
  caracter          letra  <armazena a letra digitada>

Início

        LEIA letra
        ESCOLHA letra
          CASO 'a' :
                IMPRIMA "A letra é uma vogal"
          CASO 'e' :
                IMPRIMA "A letra é uma vogal"
          CASO 'i' :
                IMPRIMA "A letra é uma vogal"
          CASO 'o' :
                IMPRIMA "A letra é uma vogal"
          CASO 'u' :
                IMPRIMA "A letra é uma vogal"
             CASO CONTRÁRIO:
                IMPRIMA "A letra é uma consoante"

          FIM-ESCOLHA
Fim
```

```
//Programa: Exemplo 6
//Objetivo: Verificar se a letra recebida como entrada é Vogal ou Consoante

#include <stdio.h>

void main(void)
{
    char            letra;  //armazena a letra digitada

      printf("Entre com uma letra ...(minuscula por favor) :");
      scanf("%c",&letra);
    switch (letra)
    {
        case 'a' :
                printf("A letra é uma vogal"); break;
        case 'e' :
              printf( "A letra é uma vogal"); break;
        case 'i' :
              printf( "A letra é uma vogal");break;
        case 'o' :
              printf( "A letra é uma vogal"); break;
        case 'u' :
              printf( "A letra é uma vogal");
        default:
              printf( "A letra é uma consoante");
    }
}
```

Exemplo: Dado a idade de um nadador, classifique-o nas categorias:

Categoria	Idade
Fraldinha	Até 4 anos
Infantil	5 a 10 anos
Juvenil	11 a 17 anos
Sênior	Maiores de 18, inclusive

Primeiro vamos mostrar como o exemplo ficaria com uma estrutura de seleção encadeada:

```
Nome: Exemplo 7
Objetivo: Verificar a classificação de um nadador de acordo com sua categoria.
Dados de Entrada: Idade
Saída: Mensagem com sua categoria, de acordo com a idade
Variáveis:
  inteiro              Idade   <armazena a idade de um nadador>

Início

LEIA Idade

   SE ( Idade ≥ 0 E Idade ≤ 4 )
         ENTÃO
               IMPRIMA  "Fraldinha"
      SENÃO
               SE (Idade ≥ 5 E Idade ≤ 10)
                     ENTÃO
                           IMPRIMA  " Infantil"
                  SENÃO
                           SE (Idade ≥ 11 E Idade ≤ 17)
                              ENTÃO
                                 IMPRIMA  " Juvenil"
                              SENÃO
                                 IMPRIMA  " Sênior "
                               FIM-SE
                     FIM-SE
    FIM-SE

Fim
```

Agora, reescrevendo o algoritmo usando uma estrutura de seleção de múltiplas escolhas temos:

```
Nome: Exemplo 8
Objetivo: Verificar a classificação de um nadador de acordo com sua categoria.
Dados de Entrada: Idade
Saída: Mensagem com sua categoria, de acordo com a idade
Variáveis:
  inteiro              Idade  <armazena a idade de um nadador>

Início

 LEIA Idade

   ESCOLHA Idade
         CASO  0 :   IMPRIMA  "Fraldinha"
         CASO  1 :   IMPRIMA  "Fraldinha"
         CASO  2 :   IMPRIMA  "Fraldinha"
         CASO  3 :   IMPRIMA  "Fraldinha"
         CASO  4 :   IMPRIMA  "Fraldinha"
         CASO  5 : IMPRIMA  "Infantil"
         CASO  6 : IMPRIMA  "Infantil"
         CASO  7 : IMPRIMA  "Infantil"
         CASO  8 : IMPRIMA  "Infantil"
         CASO  9 : IMPRIMA  "Infantil"
         CASO  10 : IMPRIMA  "Infantil"
         CASO  11 : IMPRIMA  "Juvenil"
         CASO  12 : IMPRIMA  "Juvenil"
```

```
            CASO 13 : IMPRIMA "Juvenil"
            CASO 14 : IMPRIMA "Juvenil"
            CASO 15 : IMPRIMA "Juvenil"
            CASO 16 : IMPRIMA "Juvenil"
            CASO 17 : IMPRIMA "Juvenil"
            CASO CONTRÁRIO:
                  IMPRIMA " Sênior "
      FIM-ESCOLHA
Fim
```

```
// Programa: Exemplo 8
// Objetivo: Verificar a classificação de um nadador de acordo com sua categoria.
// Dados de Entrada: Idade
// Saída: Mensagem com sua categoria, de acordo com a idade

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

int main(void )
{
      inteiro                  idade;          //armazena a idade do nadador

        printf ("Entre com a idade do nadador = ");
        scanf("%d", &idade);
        switch(idade)
        {
            case 0:
            case 1:
            case 2:
            case 3:
            case 4:
                        printf ("Categoria Fraldinha");
                        break;
            case 5:
            case 6:
            case 7:
            case 8:
            case 9:
            case 10:
                        printf ("Categoria Infantil");
                        break;
            case 11:
            case 12:
            case 13:
            case 14:
            case 15:
            case 16:
            case 17:
                        printf ("Categoria Juvenil");
                        break;
            default:
                        printf ("Categoria Sênior");
                }
      return 0;
}
```

5.5 Exercícios

1) Tendo as variáveis **salário**, **IR** e **salliq,** e considerando os valores abaixo. Informe se as expressões são verdadeiras ou falsas.

Salário	**IR**	**salliq**	**Expressão**	**V ou F**
100,00	0,00	100,00	(salliq ≥ 100,00)	
200,00	10,00	190,00	(salliq < 190)	
300,00	15,00	285,00	(salliq = salario - IR)	

2) Sabendo que A = 3, B = 7 e C = 4, informe se as expressões são verdadeiras ou falsas.

a) (A + C) > B ( )

b) B ≥ (A + 2) ( )

c) C = (B - A) ( )

d) (B + A) ≤ C ( )

e) (C+ A) > B ( )

3) Sabendo que A=5, B=4 e C=3 e D=6, informe se as expressões são verdadeiras ou falsas.

a) (A > C) **E** (C ≤ D) ( )

b) 10 > (A + B) **OU** (A + B) = (C + D) ( )

c) (A ≥ C) **E** (D ≥ A) ( )

4) Determine o resultado da expressão lógica seguinte, analisando-a passo a passo.

NÃO(5 ≠ 10/2 **OU** VERDADEIRO **E** 2 -5 > 5-2 **OU** VERDADEIRO)

5) Mostre o valor de X, sabendo que A=5, B=4 e C=3 e D=6

a)

SE **NÃO**(D > 5)
ENTÃO X ← (A + B) * D
SENÃO X ← (A - B) * C
FIM-SE

Valor X :_____

b)

SE (A > 2) **E** (B < 7)
ENTÃO X ← (A + 2) * (B - 2)
SENÃO X ← (A + B) / D * (C + D)
FIM-SE

Valor X :_____

c)

SE NÃO(A > 2) **OU NÃO** (B > 5)
ENTÃO X ← (A + B)
SENÃO X ← (A / B)
FIM-SE

Valor X :_____

6) Diga o resultado das variáveis x, y e z depois da seguinte trecho de um programa em C:

```
int     x, y, z;

x = y = 10;
z = ++x;
x = -x;
y++;
x = x + y - (z--);
```

a. x = 11, y = 11, z = 11
b. x = -11, y = 11, z = 10
c. x = -10, y = 11, z = 10
d. x = -10, y = 10, z = 10

7) Diga o resultado das variáveis x, y e z depois da seguinte sequência de operações:

```
int       x, y;
int             a = 14, b = 3;
float     z;

x = a/b;
y = a%b;
z = y/x;
```

a. x = 4.66666, y = 2, z = 0.4286
b. x = 5, y =2, z= 0.4
c. x = 5, y = 2, z = 0.
d. x = 4, y = 2, z = 0.5
e. x = 4, y =2, z = 0.

8) A operação lógica (-5 || 0) && (3 >= 2) && (1 != 0) || (3 < 0) é:

a. Verdadeira
b. Falsa

9) Quais os valores de a, b e c após a execução do código abaixo?

```
int a = 10, b = 20, c;
c = a+++b;
```

a. a = 11, b = 20, c =30
b. a = 10 , b = 21, c = 31
c. a = 11, b = 20, c = 31
d. a = 10, b = 21, c = 30

10) Diga se as seguintes expressões serão verdadeiras ou falsas:

a) ((10 > 5) || (5 > 10))

b) (!(5 == 6) && (5 != 6) && ((2 > 1) || (5 <= 4)))

11) Indique o valor de cada uma das seguintes expressões:

```
int     i = 1, j = 2, k = 3, n = 2;
float   x = 3.3, y = 4.4;
```

(a) i < j + 3
(b) 2 * i – 7 <= j – 8
(c) –x + y >= 2.0 * y
(d) i == y
(e) i != y
(f) i + j + k == -2 * -k
(g) !(n - j)
(h) !n – j
(i) !x * !x
(j) i && j && k
(k) i || j – 3 && 0
(l) i < j || 2 >= k
(m) i < j && 2 >= k
(n) i == 2 || j == 4 || k == 5
(o) x <= 5.0 && x != 1.0 || i > j

12) A instrução **else** em uma estrutura de seleção composta, construção do tipo **if-else** em C, é executada quando:

(a) a expressão de condição seguida ao **if** for falsa;
(b) a expressão de condição seguida ao **if** for verdadeira;
(c) a expressão de condição seguida ao **else** for falsa;
(d) a expressão de condição seguida ao **else** for verdadeira.

13) Em uma estrutura de seleção simples, construção if sem o else, o que acontece se a condição seguida ao if for falsa?

(a) O controle procura pelo último **else** do programa;
(b) Nada;
(c) O controle passa para a instrução seguinte ao **if**.
(d) O corpo da instrução **if** é executada.

14) Operadores relacionais são usados para:

(a) Combinar valores;
(b) Comparar valores;
(c) Distinguir diferentes tipos de variáveis;
(d) Trocar variáveis por valores lógicos.

15) Indique o resultado da variável inteira x dos trechos de programa a seguir. Para as saídas considere os seguintes valores para as variáveis:

a = 3, b = 2, c = -5 e d = 7.
(a) Resposta_____

```
if  (! ( d > 5))
    x = (a + b) * d;
else
    x = (a – b ) / c;
```

(b) Resposta_____

```
if   ( a < 2 && b < 7)
    x = (a + 2) * ( b – 2);
else
```

```
    x = (a + b ) / d * (c + d);
```

(c) Resposta_____

```
if   ( a == 2 || b < 7)
    x = (a + 2) * ( b – 2);
else
    x = (a + b ) / d * (c + d);
```

(d) Resposta_____

```
if   ( a > 2 ||  ! b < 7)
    x = a + b – 2;
else
    x = a - b;
```

(e) Resposta_____

```
if   (! ( a > 2 ||  ! b < 7))
    x = a + b;
else
    x = a / b;
```

(f) Resposta_____

```
if   (! ( a >= 2 ||  c <= 1))
    x = (a + b) / 2;
else    x = d * c;
```

16) Tendo como entrada de dados o nome do usuário, o sexo e a idade, faça um algoritmo que apresente uma mensagem de saudação somente para usuários do sexo feminino.

17) Tendo como entrada de dados um valor numérico, faça um algoritmo que verifique se este é um número par, e apresente uma mensagem em caso afirmativo.

18) Tendo com entrada de dados um valor numérico qualquer, faça um algoritmo que verifique se este é múltiplo de 7, e em caso afirmativo apresenta uma mensagem conveniente.

19) Fazer um algoritmo que leia três valores inteiros, determine e imprima o menor deles.

20) Faça um algoritmo que calcule e informe as raízes de uma equação do 2º grau, sendo que os valores **a, b** e **c** são fornecidos pelo usuário, isto se for possível tal cálculo, em caso negativo apresentar mensagem conveniente.

21) Faça um algoritmo que leia três valores numéricos inteiros e apresente-os em ordem crescente.

22) Faça um algoritmo que leia um valor numérico inteiro e verifica se este valor é divisível por 2 e 3.

23) Tendo como dados de entrada a altura e o sexo de uma pessoa, construa um algoritmo que calcule seu peso ideal, utilizando as seguintes expressões:

- Para homens: $72.7 * h - 58$;
- Para mulheres: $62.1 * h - 44.7$

24) No correio local há somente selos de 3 e de 5 centavos. A taxa mínima para correspondência é de 8 centavos. Faça um algoritmo que determina o menor número de selos de 3 e de 5 centavos que completam o valor de uma taxa dada. Use estrutura de seleção de múltipla escolha.

25) Um posto de combustível vende três tipos de combustível : álcool, diesel e gasolina. O preço de cada litro dos combustíveis é apresentado na tabela abaixo. Faça um algoritmo que leia um caractere que representa o tipo de combustível comprado (a, d ou g) e a quantidade em litros. O programa deve imprimir o valor em reais a ser pago pelo combustível.

Combustível	**Preço por Litro**
A – Álcool	1,7997
D – Diesel	0,9798
G – Gasolina	2,1009

26) Um hospital precisa de um programa para calcular e imprimir os gastos de um paciente. A tabela de preços do hospital é a seguinte :

- Quartos :
  - Particular – R$ 160,00
  - Semi-particular – R$ 110,00
  - Coletivo – R$ 85,00
- Telefone : R$ 3,00
- Televisão : R$ 4,00

Escreva um programa que leia : o número de dias gastos no hospital; um caractere representando o tipo do quarto (P, S, C); um caractere indicando se usou ou não o telefone (S, N); outro caractere indicando se usou ou não a televisão (S, N). Então emita o seguinte relatório:

Hospital Comunitário
Número de dias no hospital : 5
Tipo de quarto : P
Diárias :..... R$ 800,00
Telefone :..... R$ 0,00
Televisão :..... R$ 4,00
Total :..... R$ 804,00

27) Elabore um algoritmo que calcule o que deve ser pago por um produto, considerando o preço normal de etiqueta e a escolha da condição de pagamento. Utilize os códigos da tabela seguinte para ler qual a condição de pagamento escolhida e efetuar o cálculo adequado.

Código	Condições de pagamento
1	À vista em dinheiro ou cheque, recebe 10% de desconto
2	À vista no cartão de crédito, recebe 5% de desconto
3	Em 2 vezes, preço normal de etiqueta sem juros
4	Em 3 vezes, preço normal de etiqueta mais juros de 10%

28) Faça um algoritmo que tendo como dados de entrada o código de região de localização do cliente, o nome do cliente, o número de peças vendidas e o nome do vendedor; calcule e informe o valor do frete, a comissão do vendedor e o lucro obtido com a venda. Sabendo-se que:

O valor do frete depende da quantidade transportada e da região;
Comissão do vendedor = 6,5 % do valor total da venda;
Lucro = Valor total venda – custo total – comissão do vendedor;
Custo por peça = R$ 7,00;
Custo total = custo por peça * número de peças vendidas;
Valor total da venda = custo total acrescido em 50%;
Valor do Frete por Região:

Código da Região	Nome da Região	Valor do frete por peça (até 1.000 peças) R$	Valor do frete por peça (acima de 1.000 peças) R$
1	Sul	1,00	10%
2	Norte	1,10	8%
3	Leste	1,15	7%
4	Oeste	1,20	11%
5	Noroeste	1,25	15%
6	Sudeste	1,30	12%
7	Centro-Oeste	1,40	18%
8	Nordeste	1,35	15%

6. Estruturas de Repetição

6.1 Introdução

A **estrutura de repetição**, ou simplesmente **laço**, permite que um grupo de instruções seja executado repetidamente um número determinado de vezes ou até que uma determinada condição se torne verdadeira ou falsa.

Neste capítulo, estudaremos as seguintes estruturas de repetição:

- a estrutura com teste no início: **ENQUANTO – FAÇA**
- a estrutura com teste no final: **FAÇA – ENQUANTO**
- a estrutura com variável de controle: **PARA – PASSO - FAÇA**
- a estrutura encadeada: constitui-se em uma combinação das anteriores

Uma característica importante a ser notada nos três primeiros tipos de estruturas de repetição é que nos dois primeiros tipos (**ENQUANTO – FAÇA** e **FAÇA – ENQUANTO**), **não sabemos previamente o número de vezes** que a repetição deverá ser executada e na terceira estrutura (**PARA – PASSO - FAÇA)**, **necessitamos saber quantas vezes** deverá ocorrer a repetição.

As estruturas de repetição na Linguagem C são:

- a estrutura com teste no início: **WHILE**
- a estrutura com teste no final: **DO-WHILE**
- a estrutura com variável de controle: **FOR**
- a estrutura encadeada: constitui-se em uma combinação das anteriores

6.2 Estrutura de Repetição com Teste no Início: ENQUANTO – FAÇA

A estrutura de repetição, com teste no início, consiste em uma estrutura de controle do fluxo lógico que permite executar diversas vezes uma mesma instrução ou um mesmo bloco de instruções no algoritmo (bloco verdade), sempre verificando **antes** se o resultado da <condição> é VERDADEIRO.

Nesta estrutura, a <condição> é avaliada, e se o resultado for VERDADEIRO, a instrução ou o bloco verdade é executado, retornando novamente à avaliação da <condição>. A repetição da instrução ou do bloco verdade é feita enquanto a <condição> fornecer um resultado VERDADEIRO.

Quando o resultado da <condição> der FALSO, a instrução ou o bloco de instruções imediatamente posterior ao FIM-ENQUANTO será executado.

A característica principal desta estrutura é que, se o resultado da <condição> for FALSO da primeira vez, a instrução ou o bloco verdade de instruções não será executado.

SINTAXE em algoritmo:

```
ENQUANTO  <condição> FAÇA
            < início do bloco verdade>
            instrução 1
            instrução 2
               .
               .
               .
            instrução n
            <fim do bloco verdade>
FIM-ENQUANTO
```

A estrutura de repetição na linguagem C **WHILE** testa uma condição. Esta estrutura executará um determinado conjunto de instruções enquanto a condição permanece VERDADEIRA. No momento em que esta condição se torna FALSA, e o controle do programa é desviado para fora da estrutura.

SINTAXE em C:

```
while  ( condição )
            {   // início do bloco verdade
            instrução 1;
            instrução 2;
               .
               .
            instrução n;
}        //fim do bloco verdade
```

Exemplo: Elaborar um algoritmo que imprima a tabuada do número 5, usando a estrutura repetição **ENQUANTO-FAÇA**.

```
Nome: Exemplo 1
Objetivo: Obter a tabuada do número cinco, utilizando a estrutura de repetição com teste no início.
Dados de Entrada: Nenhum dado de entrada é requerido.
Saída: A tabuada do número cinco.
Variáveis:
  inteiro        cont          < armazena o contador da repetição >
                 resultado     < armazena o resultado da multiplicação>

  Início
        cont ← 1               < inicializa a variável contador >

        ENQUANTO cont ≤ 10 FAÇA                  < gera a tabuada>
                resultado ← cont * 5
                IMPRIMA cont, " x 5 = ", resultado
                cont ← cont + 1          < incrementa a variável contador >
        FIM-ENQUANTO

  Fim
```

O programa em linguagem C será:

```
// Programa: Exemplo 1
// Objetivo: Obter a tabuada do número cinco, utilizando a estrutura de
//           repetição com teste no início.
// Dados de entrada: Nenhum dado de entrada é requerido.
// Dados de saída: a tabuada do número cinco

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

int main( )
{
      int            cont,          //armazena o contador de repetição
                     resultado;     // armazena o resultado da multiplicação

       cont = 1;
       while ( cont <= 10){
                     resultado = cont * 5;
            printf ("%d X 5 = %d \n", cont, resultado);
            cont ++;
       }
       return 0;

}
```

Exemplo: Ler um número inteiro *n*, que não contém dígito 0, e escrever um número inteiro *m* que corresponde ao número *n* invertido. Por exemplo, se *n* igual a 123, a saída será *m* igual a 321.

```
Nome: Exemplo 2
Objetivo: Inverter um número inteiro sem digito 0.
Dados de Entrada: Um número inteiro e positivo n.
Saída: o número m (n invertido).
Variáveis:
        inteiro      n   <armazena o número inteiro>
                     r    < armazena um valor auxiliar  >
                     m   < armazena o número invertido >

Início

    LEIA n
    m ← 0              < inicializa a variável que conterá o inteiro invertido >
    ENQUANTO n > 0 FAÇA                  < encontra o número invertido>
                  r ← n % 10
                  m ← m * 10 + r
                  n ← n / 10
        FIM-ENQUANTO
    IMPRIMA  m

Fim
```

Observe que, neste algoritmo, o laço é repetido uma quantidade de vezes igual ao número de dígitos de *n*. Como *n* é desconhecido até iniciar a execução do algoritmo, não somos capazes de dizer, ao escrevermos o algoritmo, quantos dígitos *n* terá, logo não saberemos prever o número de repetições do laço.

Entretanto, podemos determinar o número de dígitos de *n* ao longo da execução do algoritmo e isto é feito indiretamente pelo algoritmo ao dividir *n*, sucessivamente, por 10, o que fará *n* "perder" o dígito menos significativo a cada repetição do laço e se tornar 0 em algum momento. Neste ponto, podemos encerrar o laço.

O programa em linguagem C será:

```
// Programa: Exemplo 2
// Objetivo: Inverter um número inteiro sem dígito zero
// Dados de entrada: Um número inteiro e positivo n.
// Dados de saída: O número m ( n invertido)

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

int main( )
{
    long int        n,          //armazena o número inteiro
                    r,          // armazena um valor auxiliar
                    m;          // armazena o número invertido

      printf ("Entre com um número inteiro e positivo: ");
      scanf("%ld", &n);
      m = 0;                    // inicializa a variável que conterá o inteiro invertido
```

```
        while ( n > 0){
                        r = n % 10;
                        m = m*10 + r;
                        n = n/10;
        }
        printf ("O numero invertido = %ld  \n", m);
        return 0;
}
```

Exemplo: Calcular a média aritmética semestral de cada aluno da turma 9999, que tem 50 alunos. A média aritmética semestral é calculada entre 2 notas bimestrais quaisquer fornecidas por um aluno, mostrando se ele foi aprovado, reprovado ou ficou de exame. O aluno é aprovado quando sua média é maior ou igual a 6.0. Quando a média é maior igual 2.0 e menor que 6.0 o aluno ficou de exame, caso contrário é reprovado.

```
Nome: Exemplo 3
Objetivo: Calcular a média aritmética semestral de 50 alunos entre 2 notas bimestrais e
mostrar, para cada aluno, se ele foi aprovado, reprovado ou ficou de exame.
Dados de Entrada: duas notas bimestrais: n1, n2.
Saída: média aritmética semestral: media e mensagem de aprovação, reprovação ou
exame.

Variáveis:
        real        n1          <primeira nota do bimestre>
                    n2          < segunda nota do bimestre >
                    media       < média semestral >
        inteiro     cont        < contador de número de alunos>

Início

  cont ← 0                              < inicializa a variável contador >
  ENQUANTO  cont < 50 FAÇA              < teste da condição de parada >
            LEIA n1
            LEIA n2
            media ← (n1 + n2)/2
            IMPRIMA  media
            SE media ≥ 6.0 ENTÃO
                        IMPRIMA  "Você foi aprovado !"
                        IMPRIMA  "Parabéns! "
                    SENÃO
                        SE media ≥ 2.0 ENTÃO
                                IMPRIMA  "Você ficou de exame !"
                                IMPRIMA  " Aproveite esta chance ! "
                            SENÃO
                                IMPRIMA  "Você foi reprovado !"
                                IMPRIMA  "Estude mais ! "
                        FIM-SE
                    FIM-SE
            cont ← cont + 1             < incrementa a variável contador >
    FIM-ENQUANTO

Fim
```

```
// Programa: Exemplo 3
// Objetivo: Calcular a média aritmética semestral entre 2 notas bimestrais
//e mostrar se o aluno foi aprovado, reprovado, uma mensagem
//para cada situação e para exame
// Dados de entrada: As notas bimestrais n1 e n2.
// Dados de saída: média aritmética semestral: media e mensagem de
// aprovação, exame ou reprovação.

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

void main(void )
{
      float          n1,          //armazena a primeira nota do bimestre
                     n2,          //armazena a segunda nota do bimestre
                     media;       // armazena a média semestral
       int           cont;        // contador de número de alunos

       cont = 0;
       while ( cont < 50 ) {
            printf ("Entre com a nota do 1° bimestre: ");
            scanf("%f", &n1);
            printf ("\n Entre com a nota do 2° bimestre: ");
            scanf("%f", &n2);
            media = (n1 + n2)/2;
            printf ("\n A media semestral e :  %.1f", media);
            if (media >= 6.0) {
                  printf ("\n Voce foi aprovado ! ");
                  printf ("Parabens ! ");
            }
            else
            {
               if (media >= 2.0) {
                        printf ("\n Voce ficou de exame ! ");
                     printf ("Aproveite esta chance ! ");
               }
               else
               {
                        printf ("\n Voce foi reprovado ! ");
                     printf ("Estude mais ! ");
               }
            }
            cont ++;
       }
       return  0;
}
```

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

- idade em anos
- sexo (masculino, feminino)
- cor dos olhos (azuis, verdes, castanhos)
- cor dos cabelos (louros, castanhos, pretos)

Para cada habitante são informados os quatro dados acima. A fim de indicar o final da entrada, após a sequência de dados dos habitantes, o usuário entrará com o valor -1 para a idade, o que deve ser interpretado pelo algoritmo como fim de entrada.

Encontrar a maior idade de um conjunto de indivíduos e o percentual de indivíduos do sexo feminino com idade entre 18 e 35 anos, inclusive, e olhos verdes e cabelos louros.

Resolução:

Neste exemplo não sabemos quantos habitantes serão processados, entretanto, o problema diz que a entrada encerra quando for digitado o número –1 ao invés de uma idade, o que nos possibilita utilizar uma estrutura do tipo ENQUANTO - FAÇA para ler a entrada.

Quanto a idade, observe que, ao invés de lermos o primeiro habitante separadamente para inicializarmos o valor da variável *maioridade* com a idade do primeiro habitante, usamos um número negativo. Neste caso, isso é possível, pois não existe um valor de idade negativo e, além disso, quando *maioridade* for comparada com a idade do primeiro habitante, ela sempre vai receber este valor, pois ele sempre será maior do que o valor negativo de *maioridade*.

Finalmente, o algoritmo calcula a porcentagem pedida de habitantes e escreve os resultados dentro de uma estrutura condicional SE – ENTÃO. Isto é necessário para evitar que o cálculo de porcentagem seja realizado com o valor de *totalhabitantes* igual a 0, o que seria uma instrução impossível de ser executada.

Nome: **Exemplo 4**
Objetivo: Encontrar a maior idade de um conjunto de indivíduos e o percentual de indivíduos do sexo feminino com idade entre 18 e 35 anos, inclusive, e olhos verdes e cabelos louros.
Dados de Entrada: a *idade*, o *sexo*, a cor dos *olhos* e dos *cabelos*.
Saída: a maior idade *maioridade*, e o percentual, *porcentagem*, indivíduos do sexo feminino com idade entre 18 e 35 anos

Variáveis:
inteiro idade, maioridade, habitantes, totalhabitantes, porcentagem
caracter sexo, olhos, cabelos

Início

```
maioridade ← −1                    <inicializa algumas variáveis>
habitantes ← 0
totalhabitantes ← 0

IMPRIMA "Entre com a idade do habitante ou -1 para encerrar: “
LEIA idade              <lê a idade do primeiro habitante>

ENQUANTO idade ≠ −1 FAÇA                    <calcula os resultados>
      IMPRIMA "entre com o sexo do habitante: "
      LEIA sexo
      IMPRIMA "entre com a cor dos olhos do habitante: "
      LEIA olhos
      IMPRIMA "entre com a cor dos cabelos do habitante: "
      LEIA cabelos
      SE idade > maioridade <compara a idade lida com a maior idade até então>
            ENTÃO
                  maioridade ← idade
      FIM-SE
      totalhabitantes ← totalhabitantes+1<conta o número total
                                                  de habitantes>
      SE (idade ≥ 18) E (idade ≤ 35)   E    (sexo =’F’)
                        E (olhos = ‘V’) E (cabelos = ‘L’) ENTÃO
                  habitantes ← habitantes + 1
      FIM-SE
      IMPRIMA "entre com a idade do habitante ou -1 para encerrar: "
      LEIA idade              <lê a idade do próximo habitante>
  FIM-ENQUANTO

SE totalhabitantes > 0              <escreve a maior idade e a porcentagem pedida>
      ENTÃO
            porcentagem ← habitantes*100/totalhabitantes
            IMPRIMA "a maior idade é: ", maioridade
            IMPRIMA "a porcentagem é: ", porcentagem
   FIM-SE
```

Fim

```
// Programa: Exemplo 4
// Objetivo: Encontrar a maior idade de um conjunto de indivíduos e o
//           percentual de indivíduos do sexo feminino com idade entre 18 e 35
//           anos, inclusive, e olhos verdes e cabelos louros
// Dados de entrada: a idade, o sexo, a cor dos olhos e dos cabelos.
// Dados de saída: a maior idade maioridade, e o percentual, porcentagem,
//                      indivíduos  do sexo feminino com idade entre 18 e  35 anos

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

int main(void )
{
      int               idade, maioridade, habitantes, totalhabitantes;
      float             porcentagem;
      char              sexo, olhos, cabelos;

        maioridade = -1;
        habitantes = totalhabitantes = 0;

        printf ("Entre com a idade do habitante ou –1 para encerrar: ");
        scanf("%d", &idade);

        while ( idade != -1 )
        {
            printf ("Entre com o sexo do habitante: ");
            scanf("%c", sexo);
            printf ("\n Entre com a cor dos olhos do habitante: ");
            scanf("%c", olhos);
            printf ("\n Entre com a cor dos cabelos do habitante: ");
            scanf("%c", cabelos);

            if (idade > maioridade)
                maioridade = idade;

                totalhabitantes ++;

        if ((idade >=18) && (idade <= 35) && sexo ==’F’) && (olhos ==’V’) && (cabelos ==’L’))
                                            habitantes ++;

                        printf ("Entre com a idade do habitante ou –1 para encerrar: ");
            scanf("%d", &idade);
            }

            if (totalhabitantes > 0)
            {
                        porcentagem = (float) habitantes*100/totalhabitantes;
                                printf ("A maior idade = %d\n", maioridade);
                    printf ("A porcentagem = %.2f %%", porcentagem);
          }
         return 0;

}
```

6.3 Estrutura de Repetição com Teste no Final: FAÇA-ENQUANTO

A estrutura de repetição com teste no final é semelhante à estrutura anterior, pois esta também é utilizada quando não sabemos antecipadamente o número de repetições a ser executada.

A característica principal desta estrutura é que a instrução ou o bloco verdade é executado **pelo menos uma vez**, independentemente do resultado da <condição>.

Nesta estrutura de repetição executa-se a instrução ou o bloco verdade e somente ao final desta execução a <condição> é avaliada. Se o resultado da <condição> for VERDADEIRO, então se executa novamente a instrução ou o bloco verdade, até que o resultado da <condição> seja FALSO, isto é, enquanto a <condição> for VERDADEIRO a instrução ou o bloco de instruções serão executados. Caso contrário, a instrução ou bloco de instruções imediatamente posterior à estrutura de repetição é executado.

SINTAXE:

```
FAÇA
        < início do bloco verdade>
        instrução 1
        instrução 2
            .
            .
        instrução n
        <fim do bloco verdade>
ENQUANTO <condição>
```

SINTAXE em Linguagem C:

```
do
{           // início do bloco verdade
        instrução 1;

        instrução 2;
            .
        instrução n;
}           // fim do bloco verdade
while  ( condição ) ;
```

Exemplo: Elaborar um algoritmo que imprima a tabuada no número 5, usando a estrutura repetição **FAÇA-ENQUANTO**.

```
Nome: Exemplo 5
Objetivo: Obter a tabuada do número cinco, utilizando uma estrutura de repetição com
teste no final.
Dados de Entrada: Nenhum dado de entrada é requerido.
Saída: A tabuada do número cinco.

Variáveis:
  inteiro        cont          < armazena o contador da repetição >
                 resultado     < armazena o resultado da multiplicação>

Início

  cont ← 1                                  < inicializa a variável contador >

  FAÇA                                      < gera a tabuada>
          resultado ← cont * 5
          IMPRIMA cont, " x 5 = ", resultado
          cont ← cont + 1                   < incrementa a variável contador >
    ENQUANTO cont ≤ 10

Fim
```

O programa em linguagem C será:

```
// Programa: Exemplo 5
// Objetivo: Obter a tabuada do número cinco, utilizando a estrutura de
//           repetição com teste no final.
// Dados de entrada: Nenhum dado de entrada é requerido.
// Dados de saída: a tabuada do número cinco

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

int main(void )
{
      int          cont,        //armazena o contador de repetição
                   resultado;   // armazena o resultado da multiplicação

       cont = 1;
       do
       {
              resultado = cont * 5;
              printf ("%d X 5 = %d \n", cont, resultado);
              cont ++;
       }  while ( cont <= 10);
       return 0;
}
```

Exemplo: Elaborar um algoritmo que permita fazer um levantamento do estoque de vinhos de uma adega, tendo como dados de entrada os tipos de vinho, sendo: T para tinto, B para branco e R para rose. Especifique a porcentagem de cada tipo sobre o total geral dos vinhos, a quantidade de vinhos é desconhecida, utilize como finalizador F de fim.

```
Nome: Exemplo 6
Objetivo: Fazer um levantamento do estoque de vinhos de uma adega, especificando a
porcentagem de cada tipo de vinho sobre o total geral de vinhos.
Dados de Entrada: tipo de vinho.
Saída: porcentagem de cada tipo de vinho sobre o total geral de vinhos e mensagem caso
nenhum tipo de vinho seja fornecido.

Variáveis:
  caracter      TV          < armazena o tipo de vinho>
  inteiro       CONV        < armazena o contador de vinhos >
                CT          < armazena o contador de vinho tinto>
                CB          < armazena o contador de vinho branco>
                CR          < armazena o contador de vinho rose>
  real          PT          < armazena a porcentagem de vinho tinto>
                PB          < armazena a porcentagem de vinho branco>
                PR          < armazena a porcentagem de vinho rose>

Início

  CONV ← 0          < inicializa a variável contador de vinhos>
  CT ← 0            < inicializa a variável contador de vinho tinto>
  CB ← 0            < inicializa a variável contador de vinho branco>
  CR ← 0            < inicializa a variável contador de vinho rose>

  FAÇA          < inicia a contagem dos vinhos>
                LEIA TV
                ESCOLHA TV
                                CASO ' T ':
                                  CT← CT + 1
                                  CONV← CONV + 1
                                CASO 'B ':
                                  CB← CB + 1
                                  CONV← CONV + 1
                                CASO 'R ':
                                  CR← CR + 1
                                  CONV← CONV + 1
                        FIM-ESCOLHA
      ENQUANTO TV = 'V '
    SE CONV > 0
          ENTÃO
                PT ← (CT*100)/CONV
                PB ← (CB*100)/CONV
                PR ← (CR*100)/CONV
                IMPRIMA (“Porcentagem de Vinhos Tintos = “, PT)
                IMPRIMA (“Porcentagem de Vinhos Brancos = “, PB)
                IMPRIMA (“Porcentagem de Vinhos Rosê = “, PR)
          SENÃO
                IMPRIMA “Nenhum tipo de vinho foi fornecido!”
          FIM-SE
Fim
```

O programa em linguagem C será:

```
// Programa: Nome: Exemplo 6
// Objetivo: Fazer um levantamento do estoque de vinhos de uma adega,
// especificando a porcentagem de cada tipo de vinho sobre o total geral de vinhos.
// Dados de Entrada: tipo de vinho.
// Dados de Saída: porcentagem de cada tipo de vinho sobre o total geral de vinhos e
          mensagem caso nenhum tipo de vinho seja fornecido.

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

int main(void )
{       int     CONV,                   // armazena o contador de vinhos
                CT      ,               // armazena o contador de vinho tinto
                   CB,                  // armazena o contador de vinho branco
                   CR;                  // armazena o contador de vinho rosê
        char        TV ;                // armazena o tipo de vinho
        float   PT,                     // armazena a porcentagem de vinho tinto
                PB,                     //armazena a porcentagem de vinho branco
                PR      ;               // armazena a porcentagem de vinho rosê

        CONV = 0;                               // inicializa a variável contador de vinhos
        CT = 0 ;                                // inicializa a variável contador de vinho tinto
        CB = 0 ;                                // inicializa a variável contador de vinho branco
        CR = 0 ;                                // inicializa a variável contador de vinho rosê
        do                                      // inicia a contagem dos vinhos
            {   printf (“Entre com o tipo de vinho: T -> Tinto. \n”);
                printf (“ B -> Branco e \n”);
                printf (“ R -> Rose. \n”);
                printf (“Entre com a letra F para encerrar o levantamento. \n”);
                TV = getche();
                switch(TV)
                   {    case 'T':       CT = CT + 1; CONV++;
                        break;
                        case 'B': CB = CB + 1; CONV++;
                        break;
                        case 'R': CR = CR + 1; CONV++;
                            break;
                        default:
                            printf (“\n Digite uma das letras citadas anteriormente ou  “);
                            printf (“F para finalizar, por favor!\n”);
                   }
         } while ( TV != 'F' );

         if (CONV > 0) {
                PT = (CT*100.0)/CONV;
                   PB = (CB*100)/CONV;
                   PR = (CR*100)/CONV;
                   printf (“\n Porcentagem de Vinhos Tintos = %.1f %%\n “, PT);
                   printf (“Porcentagem de Vinhos Brancos = %.1f %%\n “, PB);
                   printf (“Porcentagem de Vinhos Rosê = %.1f %%\n “, PR);
            }
            else
                        printf (“Nenhum tipo de vinho foi fornecido!”);
    return 0;
  }
```

Exemplo: Calcular a média aritmética semestral de cada aluno da turma 9999, que tem 50 alunos. A média aritmética semestral é calculada entre 2 notas bimestrais quaisquer fornecidas por um aluno, mostrando se ele foi aprovado, reprovado ou ficou de exame. O aluno é aprovado quando sua média é maior ou igual a 6.0. Quando a média é maior igual 2.0 e menor que 6.0 o aluno ficou de exame, caso contrário é reprovado.

```
Nome: Exemplo 7
Objetivo: Calcular a média aritmética semestral de 50 alunos entre 2 notas bimestrais e
mostrar, para cada aluno, se ele foi aprovado, reprovado ou ficou de exame.
Dados de Entrada: duas notas bimestrais: n1, n2.
Saída: média aritmética semestral: media e mensagem de aprovação ou reprovação.

Variáveis:
        real        n1              <primeira nota do bimestre>
                    n2              < segunda nota do bimestre >
                    media           < média semestral >
                    inteiro cont    < contador de número de alunos>

Início

  cont ← 0                  < inicializa a variável contador >

  FAÇA
            LEIA n1
            LEIA n2
            media ← (n1 + n2)/2
            IMPRIMA  media
            SE media ≥ 6.0 ENTÃO
                        IMPRIMA  "Você foi aprovado !"
                        IMPRIMA  "Parabéns! "
                  SENÃO
                        SE media ≥ 2.0 ENTÃO
                              IMPRIMA  "Você ficou de exame !"
                              IMPRIMA  " Aproveite esta chance ! "
                        SENÃO
                              IMPRIMA  "Você foi reprovado !"
                              IMPRIMA  "Estude mais ! "
                        FIM-SE
                  FIM-SE
            cont ← cont + 1       < incrementa a variável contador >
    ENQUANTO    cont ≤ 50               < teste da condição de parada >

Fim
```

O programa em linguagem C será:

```
// Programa: Exemplo 7
// Objetivo: Calcular a média aritmética semestral entre 2 notas bimestrais
//e mostrar se o aluno foi aprovado, reprovado, uma mensagem
//para cada situação e para exame
// Dados de entrada: As notas bimestrais n1 e n2.
// Dados de saída: média aritmética semestral: media e mensagem de
//                                    aprovação,  exame ou reprovação
```

```
#include <stdio.h>

int main(void )
{
      float                   n1,          //armazena a primeira nota do bimestre
                              n2,          //armazena a segunda nota do bimestre
                              media;       // armazena a média semestral
       int                    cont;        // contador de número de alunos

       cont = 0;
       do
       {
            printf ("Entre com a nota do 1° bimestre: ");
            scanf("%f", &n1);
            printf ("\n Entre com a nota do 2° bimestre: ");
            scanf("%f", &n2);
            media = (n1 + n2)/2;
            printf ("\n A media semestral e :  %.1f", media);
            if (media >= 6.0)
            {           printf ("\n Voce foi aprovado ! ");
                        printf ("Parabens ! ");
                }
                else
                {
                    if (media >= 2.0)
                    {     printf ("\n Voce ficou de exame ! ");
                         printf ("Aproveite esta chance ! ");
                          }
                          else
                          {         printf ("\n Voce foi reprovado ! ");
                                    printf ("Estude mais ! ");
                          }
            }
       cont ++;
       }while ( cont < 50 );

       return 0;
}
```

Exemplo: Dado um número inteiro e positivo, calcular o seu fatorial. Fatorial é o produto dos números naturais desde 1 até o inteiro *n*.

Resolução:

O cálculo de um fatorial é conseguido pela multiplicação sucessiva do número de termos. Sabendo que:

N ! = 1 x 2 x 3 x 4 x 5 x. ... x (N-1) x N.

0 ! = 1, por definição.

No caso de *n* = 5, o programa deverá executar as multiplicações sucessivas e acumulá-las a fim de possuir o valor 120 após 5 passos. O número de passos deverá ser controlado pela variável *i*.

```
Nome: Exemplo 8
Objetivo: Calcular o fatorial de um número inteiro.
Dados de Entrada: O valor de um número natural n.
Saída: O fatorial do número n na variável fatorial.

Variáveis:
   inteiro      n          <armazena o número natural n>
                fatorial   <armazena o valor do fatorial>
                cont       <armazena o contador>
Início

      LEIA n              <lê o numero n>

      fatorial ← 1        < inicializa a variável que irá armazenar o valor do fatorial >

      cont ← 1  < inicializa a variável contador >

      FAÇA                <calcula a fatorial>
            fatorial ← fatorial * cont
            cont ← cont + 1
      ENQUANTO cont <= n

      IMPRIMA "O fatorial de ", n, "é : ", fatorial          <imprime o fatorial>
Fim
```

O programa em linguagem C será:

```
// Programa: Exemplo 8
// Objetivo: Calcular o fatorial de um número inteiro.
// Dados de Entrada: O valor de um número natural n.
// Dados de Saída: O fatorial do número n na variável fatorial.

#include <stdio.h>

int main(void )
{
        int             n;              // armazena o número natural n
        long int        fatorial;       // armazena o valor do fatorial
           int          cont;           // armazena o contador

           printf ("Deseja calcular o fatorial do número: " );
           scanf ("%d", &n); // lê o numero n

        fatorial = 1;           // inicializa a variável que irá armazenar o valor do fatorial

        cont = 1;               // inicializa a variável contador

        do                      // calcula a fatorial
           {
                fatorial = fatorial * cont;
                cont = cont + 1;
           } while (cont <= n);

           printf ("\n O fatorial de %d é: %ld\n", n, fatorial );          //imprime o fatorial
           return 0;

}
```

6.4 Estrutura de Repetição com Variável de Controle: PARA - PASSO - FAÇA

A estrutura de repetição com variável de controle permite que uma instrução ou bloco verdade seja executado repetidamente por um número definido de vezes, pois os limites são fixos.

Então, a característica principal deste tipo de estrutura é que se **deve saber previamente o número de vezes** que a instrução ou bloco verdade deve ser repetido.

SINTAXE:

```
PARA <variável> DE < valor inicial> ATÉ <valor final> PASSO <incremento> FAÇA
        < início do bloco verdade>
        instrução 1
        instrução 2
            .
            .
            .
        instrução n
        <fim do bloco verdade>
FIM-PARA
```

O seu funcionamento segue abaixo :

1. atribui-se à <*variável*> o valor numérico <*valor inicial*>;
2. compara o valor de <*variável*> com o valor numérico <*valor final*>. Se <*variável*> for menor ou igual a <*valor final*>, então vai para o passo 3; caso contrário, executa-se a instrução ou bloco de instruções imediatamente após o **FIM-PARA**.
3. executa as instruções 1 a n;
4. incrementa-se o valor de variável de uma unidade
5. volta para o passo 2

Como se pode observar a incrementação é feita pela própria estrutura, conforme o valor do <*incremento*> fornecido pelo **PASSO**. O <*incremento*> é o valor adicionado ao valor da <*variável*> ao final de cada iteração.

Uma particularidade desta estrutura se dá quando a cláusula **PASSO** <*incremento*> não está presente: neste caso o valor assumido para o incremento é 1 (um).

Em linguagem C a estrutura de repetição **PARA-PASSO-FAÇA** é representada pelo comando **FOR** e tem a seguinte sintaxe :

```
for ( variavel = valor inicial; condição ; incremento )
{                   // início do bloco verdade
                instrução 1;

                instrução 2;
                    .
                    .
                instrução n;
}                   // fim do bloco verdade
```

O comando **FOR** é composto pela **inicialização**, por uma **condição** e por um **incremento**.

A **inicialização** é uma declaração de atribuição usada para inicializar a variável de controle do laço por um valor inicial. É sempre executada uma única vez antes do laço ser iniciado.

A **condição** é geralmente uma expressão lógica que determina quando o laço terminará pelo teste da variável de controle do laço contra algum valor.

O **incremento** define como a variável de controle do laço mudará cada vez que a repetição for realizada. Esta instrução é executada toda vez imediatamente após a execução do corpo do laço.

Podemos ver que o **FOR** executa a inicialização incondicionalmente e testa a condição. Se a condição for FALSA, a execução do programa continuará na próxima instrução depois da estrutura FOR.

Se a condição for VERDADEIRA ele executa a instrução ou bloco de instruções, faz o incremento e volta a testar a condição. Ele fica repetindo estas operações até que a condição seja FALSA.

O melhor modo de se entender o loop FOR é ver como ele funciona "por dentro". O laço ou loop FOR é equivalente a se fazer o seguinte:

```
variavel = valor inicial;
while ( condição)
{                   // início do bloco verdade
              instrução 1;
              instrução 2;
                  .
              instrução n;

              incremento;

}                   // fim do bloco verdade
```

O seu funcionamento segue abaixo:

1. o comando de inicialização é executado, a **variavel** recebe um **valor inicial**;
2. a **condição** é avaliada;
3. se a condição é verdadeira então vai para o passo 3; caso contrário, executa-se a instrução ou bloco de instruções imediatamente após o fim do bloco verdade;
4. executa as **instruções** de **1** a ***n***;
5. por último o comando de **incremento** é executado;
6. a **condição** é re-avaliada e volta-se à etapa 3;
7. a repetição será encerrada quando a **condição** = falsa.

Obs : Se inicialmente a condição = falsa então as instruções de 1 a *n* e incremento não serão executados. Se a condição não se tornar falso em alguma execução das instruções ou do incremento teremos um *loop* infinito.

Exemplo: Elaborar um algoritmo que imprima a tabuada no número 5, usando a estrutura de repetição **PARA-PASSO-FAÇA**.

```
Nome: Exemplo 9
Objetivo: Obter a tabuada do número cinco, utilizando uma estrutura de repetição com
variável de controle.
Dados de Entrada: Nenhum dado de entrada é requerido.
Saída: A tabuada do número cinco.

Variáveis:
  inteiro        cont          < armazena o contador da repetição >
                 resultado     < armazena o resultado da multiplicação>

Início

      cont ← 1                 < inicializa a variável contador >

      PARA cont DE 1 ATÉ 10 PASSO 1 FAÇA      < gera a tabuada>
            resultado ← cont * 5
            IMPRIMA cont, "x 5 =", resultado
      FIM-PARA

Fim
```

O programa em linguagem C será:

```
// Programa: Exemplo 9
// Objetivo: Obter a tabuada do número cinco, utilizando a estrutura de
//           repetição com variável de controle.
// Dados de entrada: Nenhum dado de entrada é requerido.
// Dados de saída: a tabuada do número cinco

#include <stdio.h>


int main(void )
{
        int             cont,       //armazena o contador de repetição
                        resultado;  // armazena o resultado da multiplicação

    for (cont = 1; cont <= 10; cont ++)
    {
        resultado = cont * 5;
        printf ("%d X 5 = %d \n", cont, resultado);
    }
    return 0;
}
```

Exemplo: Calcular e exibir a soma de todos os números pares desde 100 até 200, inclusive, utilizando uma instrução condicional para determinar se o número é par.

```
Nome: Exemplo 10
Objetivo: Somar os números pares de 100 a 200, utilizando uma instrução condicional
para determinar se o número é par.
Dados de Entrada: Nenhum dado de entrada é requerido.
Saída: O somatórios de todos os número pares de 100 até 200, na variável soma.

Variáveis
  Inteiro soma          < armazena o somatório de todos os números pares>
           i                    < armazena o contador da repetição >

Início

      soma ← 0                  <inicializa com zero a variável que guardará a soma>
      PARA i de 100 ATÉ 200 PASSO 1 FAÇA              <calcula a soma>
            SE ( i % 2  = 0 ) ENTÃO
                        soma ← soma + i
               FIM-SE
      FIM-PARA
      IMPRIMA "A soma dos pares de 100 a 200 é: ", soma

Fim
```

O programa em linguagem C será:

```
// Programa: Exemplo 10
// Objetivo: Somar os números pares de 100 a 200, utilizando uma instrução
             condicional para determinar se o número é par.
// Dados de Entrada: Nenhum dado de entrada é requerido.
// Dados de Saída: O somatórios de todos os número pares de 100 até 200, na variável
                   soma.
#include <stdio.h>


int main(void ) {
              int     soma,  // armazena o somatório de todos os números pares
                      i;     // armazena o contador da repetição

          soma = 0;                 //inicializa com zero a variável que guardará a soma
          for (i = 100; i<= 200; i++){// calcula a soma

           if ( i % 2  == 0 ) {
                 soma = soma + i;
           }
          }
          printf ("\n A soma dos pares de 100 a 200 é: %d\n", soma);
          return 0;
}
```

No exemplo anterior, a estrutura **PARA – FAÇA** inicia com a atribuição do valor 100 a variável *i*. Compara-se, então, o valor de *i* com 200 e, como *i* é menor que 200, executam-se pela primeira vez a instrução dentro do laço e, ao final da execução, incrementa-se *i* de uma unidade. Deste ponto em diante, o valor de *i* é comparado com 200 e, caso seja menor ou igual a 200, uma nova iteração do laço é realizada e *i* é novamente incrementada de uma unidade ao final da iteração. Este processo se repete até que *i* seja igual a 201. Desta forma, *i* guarda um valor de 100 a 200 a cada repetição da estrutura. A cada iteração, o algoritmo verifica se o valor de *i* é par ou ímpar: se o valor de *i* for par, o valor de *i* é adicionado ao valor na variável *soma*.

Observe que a variável *soma* recebe o valor 0 antes da estrutura de repetição ser executada. Esta atribuição é denominada **inicialização** da variável. A variável *soma* é utilizada como um ***acumulador***, isto é, a cada iteração da repetição, *soma* é incrementada de mais um número par, e isto é feito somando o valor atual de *soma* ao número par em *i* e atribuindo o resultado novamente à variável *soma* ← *soma* + *i*.

Pode-se resolver o problema de calcular a soma de todos os pares de 100 a 200, variando o PASSO para que o incremento seja feito de duas em duas unidades:

```
Nome: Exemplo 11
Objetivo: Somar os números pares de 100 a 200, sem utilizar um comando condicional
para determinar se o número é par.
Dados de Entrada: Nenhum dado de entrada é requerido.
Saída: O somatórios de todos os número pares de 100 até 200, na variável soma.

Variáveis
      inteiro   soma      < armazena o somatório de todos os números pares>
                i         < armazena o contador da repetição >

Início

      soma ← 0                  <inicializa com zero a variável que guardará a soma>

      PARA i de 100 ATÉ 200 PASSO 2 FAÇA        <calcula a soma>
            soma ← soma + i
      FIM-PARA

      IMPRIMA "A soma dos pares de 100 a 200 é: ", soma

Fim
```

O programa em linguagem C será:

```
// Programa: Exemplo 11
// Objetivo: Somar os números pares de 100 a 200, sem utilizar um comando
             condicional para determinar se o número é par.
// Dados de Entrada: Nenhum dado de entrada é requerido.
// Dados de Saída: O somatórios de todos os número pares de 100 até 200, na variável
                   soma.

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

void main(void )
{
        int                     soma,  // armazena o somatório de todos os números pares
                i;                     //  armazena o contador da repetição


            soma = 0;                  //inicializa com zero a variável que guardará a soma

            for (i = 100; i<= 200; i += 2)       // calcula a soma
            {
                    soma = soma + i;
            }
            printf ("\n A soma dos pares de 100 a 200 é: %d\n", soma);

            return 0;
}
```

O incremento i += 2 é um operador aritmético de atribuição e equivale a i = i + 2.

Exemplo: Calcular a média aritmética semestral de cada aluno da turma 9999, que tem 50 alunos. A média aritmética semestral é calculada entre 2 notas bimestrais quaisquer fornecidas por um aluno, mostrando se ele foi aprovado, reprovado ou ficou de exame. O aluno é aprovado quando sua média é maior ou igual a 6.0. Quando a média é maior igual 2.0 e menor que 6.0 o aluno ficou de exame, caso contrário é reprovado.

```
Nome: Exemplo 12
Objetivo: Calcular a média aritmética semestral de 50 alunos entre 2 notas bimestrais e
mostrar, para cada aluno, se ele foi aprovado, reprovado ou ficou de exame.
Dados de Entrada: duas notas bimestrais: n1, n2.
Saída: média aritmética semestral: media e mensagem de aprovação ou reprovação.

Variáveis:
        real    n1      <primeira nota do bimestre>
                n2      < segunda nota do bimestre >
                media   < média semestral >
                inteiro cont    < contador de número de alunos>

Início

  cont ← 0                      < inicializa a variável contador >

  PARA cont DE 1 ATÉ 50 PASSO 1 FAÇA
        LEIA n1
        LEIA n2
        media ← (n1 + n2)/2
        IMPRIMA  media
        SE media ≥ 6.0
                ENTÃO
                  IMPRIMA  "Você foi aprovado !"
                  IMPRIMA  "Parabéns! "
             SENÃO
                     SE media ≥ 2.0
                        ENTÃO
                          IMPRIMA  "Você ficou de exame !"
                          IMPRIMA  " Aproveite esta chance ! "
                     SENÃO
                          IMPRIMA  "Você foi reprovado !"
                          IMPRIMA  "Estude mais ! "
                     FIM-SE
                FIM-SE
    FIM-PARA

Fim
```

O programa em linguagem C será:

```
// Programa: Exemplo 12
// Objetivo: Calcular a média aritmética semestral entre 2 notas bimestrais
//           e mostrar se o aluno foi aprovado, reprovado, uma menssagem
//           para cada situação e para exame
// Dados de entrada: As notas bimestrais n1 e n2.
// Dados de saída: média aritmética semestral: media e mensagem de
//           aprovação, exame ou reprovação

#include <stdio.h>


int main(void )
{
      float                n1,          //armazena a primeira nota do bimestre
                           n2,          //armazena a segunda nota do bimestre
                           media;       // armazena a média semestral
        int                cont;        // contador de número de alunos

        for (cont = 0; cont < 50; cont++)
        {
            printf ("Entre com a nota do 1° bimestre: ");
            scanf("%f", &n1);
            printf ("\n Entre com a nota do 2° bimestre: ");
            scanf("%f", &n2);
            media = (n1 + n2)/2;
            printf ("\n A media semestral e :  %.1f", media);
            if (media >= 6.0)
            {
               printf ("\n Voce foi aprovado ! ");
               printf ("Parabens ! ");
            }
               else if (media >= 2.0) {
                           printf ("\n Voce ficou de exame ! ");
                           printf ("Aproveite esta chance ! ");
                     }
                     else {
                           printf ("\n Voce foi reprovado ! ");
                           printf ("Estude mais ! ");
                     }
               }
        return 0;
}
```

Exemplo: Uma pessoa aplicou seu capital a juros e deseja saber, trimestralmente, a posição de seu investimento inicial *c*. Chamando de *i* a taxa de juros do trimestre, escrever uma tabela que forneça, para cada trimestre, o rendimento auferido e o saldo acumulado durante o período de *x* anos, supondo que nenhuma retirada tenha sido feita.

```
Nome: Exemplo13
Objetivo: Calcular o rendimento trimestral de valor aplicado.
Dados de Entrada: O capital c, a taxa de juros i e o número de anos.
Saída: O rendimento trimestral r e o montante c tabulados.
Variáveis:
        inteiro     c       < investimento inicial >
                    i       < taxa de juros >
                x           < número de anos >
                n           < número de trimestres >
                r           < rendimentos >
                j           < contador >

Início

    LEIA c              <lê o capital inicial>
    LEIA i                      <lê a taxa de juros>
    LEIA x              <lê números de anos>
    n ← x * 4                   <calcula o número de trimestres em x anos>
    PARA j DE 1 ATÉ n PASSO 1 FAÇA    <calcula e exibe rendimento
                                            e montante >
            r ← i * c
            c ← c + r
            IMPRIMA "Rendimento do trimestre ", j, ": ", r
            IMPRIMA "Montante do trimestre ", j, ": ", c
      FIM-PARA

Fim
```

O algoritmo inicia com o cálculo do número de trimestres em *x* anos, já que o rendimento deve ser calculado por trimestre e a taxa de juros *i* também é dada em função do número de trimestres. A estrutura **PARA - PASSO - FAÇA** é repetida *n* vezes, onde *n* é o número de trimestres encontrado. A cada repetição, o rendimento *r* é calculado e o capital *c* é somado a *r* para totalizar o montante do mês. Em seguida, os valores de *r* e *c* são exibidos para o usuário.

O programa em linguagem C será:

```
// Programa: Exemplo 13
// Objetivo: Calcular o rendimento trimestral de valor aplicado.
// Dados de Entrada: O capital c, a taxa de juros i e o número de anos.
// Dados de Saída: O rendimento trimestral r e o montante c tabulados.

#include <stdio.h>


int main(void )
{
   float          c,                          // investimento inicial
                  r;                          // rendimentos
   int  i,                                    // taxa de juros
          x,                                  // número de anos
              n,                              // número de trimestres
              j;                              // contador


   printf ("\n Digite o investimento inicial: ");
   scanf ("%f", &c);                          //lê o capital inicial
   printf ("\n Digite a taxa de juros: ");
   scanf ("%d", &i);                          //lê a taxa de juros
   printf ("\n Digite o numero de anos: ");
   scanf ("%d", &x);                          //lê números de anos
   n = x * 4;                                 //calcula o número de trimestres em x anos
   for ( j = 1; j<= n; j ++)                  //calcula e exibe rendimento e montante
      {
            r = i * c;  c = c + r;
         printf ("\n Rendimento do trimestre %d e' de %.2f ", j, r);
         printf ("\n Montante do trimestre %d e' de %.2f" , j, c);
      }
      return 0;
}
```

6.5 Estrutura de Repetição Encadeada

Assim como as estruturas de seleção, as estruturas de repetição também podem conter o encadeamento de um tipo de estrutura de repetição com outro tipo de estrutura de repetição. A existência destas ocorrências vai depender do problema a ser resolvido.

Exemplo: O algoritmo seguinte gera a tabuada de 2 a 9.

```
Nome: Exemplo 14
Objetivo: Obter a tabuada do número 2 ao número 9, utilizando uma estrutura de
repetição encadeada.
Dados de Entrada: Nenhum dado de entrada é requerido.
Saída: A tabuada do número 2 ao número 9.

Variáveis:
    inteiro          bloco          < contador de bloco de tabuada >
                     titulo         < contador dos títulos >
                     tab            < contador da tabuada >

Início

    PARA bloco DE 0 ATÉ 1 PASSO 1 FAÇA
        PARA titulo DE 1 ATÉ 4 PASSO 1 FAÇA
                IMPRIMA "Tabuada do: ", titulo+4*bloco + 1
            FIM_PARA

          IMPRIMA "\n"
          PARA titulo DE 1 ATÉ 9 PASSO 1 FAÇA
                PARA tab DE (2+4*bloco) ATÉ (5+4*bloco) PASSO 1 FAÇA
                        IMPRIMA tab, "x ", titulo, " = ", tab*titulo
                  FIM-PARA
                IMPRIMA "\n"
            FIM-PARA
      FIM-PARA

Fim
```

O laço PARA-PASSO-FAÇA mais externo (o da variável *bloco*) é executado duas vezes. A primeira para imprimir o primeiro bloco de tabuadas (de 2 a 5) e a segunda vez para imprimir o segundo bloco (de 6 a 9).

O segundo laço PARA-PASSO-FAÇA imprime os títulos (o da variável *titulo*) e os laços mais internos imprimem a tabuada propriamente dita.

O exemplo 14 apresenta um algoritmo com encadeamento de estruturas de repetição. O programa abaixo apresenta a tradução em linguagem C.

O programa em linguagem C será:

```
// Programa: Exemplo 14
// Objetivo: Obter a tabuada do número 2 ao número 9, utilizando uma estrutura de
             repetição encadeada.
// Dados de entrada: Nenhum dado de entrada é requerido.
// Dados de saída: a tabuada do número dois ao nove.

#include <stdio.h>


int main(void )
{
   int          bloco,          // contador de bloco de tabuada
                titulo   ,      // contador dos títulos
                tab;            //contador da tabuada

        for (bloco = 0; bloco <= 1; bloco ++ )
        {
             for (titulo = 1; titulo <= 4; titulo ++ )
             {
                printf ("Tabuada do:%d    ", titulo + 4*bloco +1);
                }
                printf ("\n");
                for (titulo = 1; titulo <= 9; titulo ++ )
                {
                                for (tab = (2+4*bloco); tab <= (5+4*bloco) ; tab++)
                                {
                                        printf ("%3d x%3d = %3d", tab, titulo, tab*titulo);
                                }
                                printf ("\n");
                        }
          }
          return 0;
}
```

Exemplo: Calcular o fatorial de um número inteiro qualquer. Calcular outros fatoriais até que o usuário não mais deseje utilizar o algoritmo. Sendo assim, o algoritmo deverá pedir ao usuário a sua continuidade ou não.

```
Nome: Exemplo 15
Objetivo: Calcular o fatorial de um número inteiro.
Dados de Entrada: Nenhum dado de entrada é requerido.
Saída: O fatorial do número n na variável fatorial.

Variáveis:
   inteiro      n          <armazena o número natural n>
                fatorial   <armazena o valor do fatorial>
                cont       <armazena o contador>
   caracter     resp       < para confirmação da continuidade do algoritmo >

   Início

      resp ← 's'

      ENQUANTO resp = 's' FAÇA
                fatorial ← 1
                IMPRIMA " fatorial de que número: "
                LEIA n

                PARA cont DE 1 ATÉ n PASSO 1 FAÇA      <calcula a fatorial>
                          fatorial ← fatorial * cont
                     FIM-PARA
                IMPRIMA "O fatorial de ", n, "é : ", fatorial
                IMPRIMA "Deseja continuar (s/n): "
                LEIA resp
         FIM-ENQUANTO

   Fim
```

O programa em linguagem C será:

```
// Programa: Exemplo 15
// Objetivo: Calcular o fatorial de um número inteiro.
// Dados de Entrada: O valor de um número natural n.
// Dados de Saída: O fatorial do número n na variável fatorial.

#include <stdio.h>


int main(void )
{
        int             n,                      // armazena o número natural n
                       cont;                    // armazena o contador
            long int    fatorial;               // armazena o valor do fatorial
            char        resp;                   //para confirmação da continuidade do algoritmo

               resp = 'S';
            while (resp != 'N')
            {
               fatorial = 1;
               printf ("\n Fatorial de que número? “);
               scanf (“%d ", &n);
                  for (cont = 1; cont <= n; cont ++)                    //calcula a fatorial
                        fatorial = fatorial * cont;
               printf (“O fatorial de %d e’ %ld \n", n, fatorial );
                  printf ("Deseja continuar (S/N): “);
                  resp = getche();
               }
            return 0;

}
```

Exemplo: Algoritmo que testa a sua capacidade de adivinhar uma letra.

```
Nome: Exemplo 16
Objetivo: Adivinhar uma letra.
Dados de Entrada: Nenhum dado de entrada é requerido.
Saída: uma letra.
Variáveis:
      inteiro       tentativas    <número de tentativas >
      caracter      resp          < para confirmação da continuidade do algoritmo >
                    ch            < armazena um caractere aleatório >
                    c             < armazena um caractere informado pelo usuário >

   Início

      resp ← 's'

      ENQUANTO resp = 's' FAÇA

            ch ← RAND % 26 + 'a'        < para gerar letras entre a e z >
            tentativas ← 1
            IMPRIMA " Digite uma letra de a ate z: "
            LEIA c

            ENQUANTO c ≠ ch FAÇA
                  IMPRIMA c, "É incorreto. Tente novamente "
                  tentativas = tentativas + 1
                  LEIA c
            FIM-ENQUANTO

            IMPRIMA c, "É correto "
            IMPRIMA "Você acertou ", tentativas, "tentativas"
            IMPRIMA "Deseja continuar (s/n): "
            LEIA resp
         FIM-ENQUANTO

   Fim
```

Para tomar um caractere aleatório, deve-se usar uma função RAND que devolve um inteiro aleatório. A expressão RAND % 26 resultará em um número entre 0 e 25. A este número é somado o caractere 'a' para gerar letras entre 'a' e 'z'. Utilizar a função randomize ( ) para inicialização.

```
// Programa: Exemplo 16
// Objetivo: Adivinhar uma letra.
// Dados de Entrada: Nenhum dado de entrada é requerido.
// Dados de Saída: uma letra.

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

int main(void )
{
   int        tentativas;          //número de tentativas
   char       resp,                // para confirmação da continuidade do algoritmo
              ch,                        // armazena um caractere aleatório
              c;                         // armazena um caractere informado pelo usuário


              resp = 's';

           rands(time(NULL));
           while (resp == 's')
              {
                    ch = rand() % 26 + 'a';              // para gerar letras entre a e z
                    tentativas = 1;
                 printf (" \n Digite uma letra de a  ate'  z: \n");
                 while ((c=getch()) != ch)
                    {
                          printf ("%c e' incorreto. Tente novamente. \n ", c);
                             tentativas ++;
                    }
                 printf ("\n %c e' correto ", c);
                 printf ("\n Você acertou em %d tentativas", tentativas);
                 printf ("\n Deseja continuar ? (s/n): ");
                 resp = getche();
              }
              getch();
}
```

Para tomar um caractere aleatório, deve-se usar uma função rand( ), da biblioteca stdlib.h, que devolve um inteiro aleatório entre 0 e 32767. A expressão rand( ) % 26 resultará o resto da divisão de rand( ) por 26, isto é, um número entre 0 e 25. A este número é somado o caractere 'a' para gerar letras entre 'a' e 'z'.

6.6 Exercícios

1. Construa um algoritmo que leia 5 valores inteiros e positivos e, encontre o maior valor, encontre o menor valor e calcule a média aritmética dos números lidos.

2. Dado *x* inteiro e *n* natural, calcular x^n.

3. Dado *n*, imprimir os n primeiros naturais ímpares.

Exemplo: Para n = 4 a saída deverá ser 1, 3, 5, 7

4. Uma forma de encontrar o quadrado de um número positivo n é somar os n primeiros números ímpares.

Exemplo:

Para n = 3, o valor de $3^2 = 1 + 3 + 5 = 9$

Para n = 8, o valor de $8^2 = 1 + 3 + 5 + 7 + 9 + 11 + 13 + 15 = 64$

Isso pode ser traduzido pra a seguinte fórmula

$$n^2 = \sum_{i=0}^{n-1} (2i+1)$$

5. Escreva um algoritmo que leia uma sequência de números inteiros e positivos, encontre e imprima o maior e o menor número. A entrada de um número negativo indica que sequência terminou.

6. A empresa Unidos & Cansados Ltda., realizou uma pesquisa sobre a aceitação de seu produto alimentício, e necessita de um algoritmo, que tabule e informe:

➔ quantas mulheres maiores de 40 anos indicaram o produto como bom;

➔ quantas mulheres maiores de 50 anos indicaram o produto como ruim;

➔ quantos homens indicaram o produto como péssimo;

➔ total de homens que participaram da pesquisa.

➔ total de mulheres que participaram da pesquisa.

As respostas foram codificadas da seguinte maneira:

a. idade – valor numérico indicando o número de anos de vida;

b. sexo – "masculino" para homens e "feminino" para mulheres;

c. opinião com relação ao produto:

1: péssimo 2: ruim 3: regular 4: bom 5: ótimo.

d. Obs.: O programa dever terminar quando o usuário digitar s (sim)

7. Escreva um algoritmo que calcule o MMC (mínimo múltiplo comum) entre dois números naturais.

8. Escreva um algoritmo que calcule o MDC (máximo divisor comum) entre dois números naturais.

9. Escreva um algoritmo que lê um número natural, e verifica se o mesmo é primo ou não.

10. Elabore um algoritmo que imprima todos números primos existentes entre N1 e N2, em que N1 e N2 são números naturais fornecidos.

11. A série de Fibonacci é formada pela seguinte sequência: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55,...etc. Escreva um algoritmo que gere e apresente a série de Fibonacci até um número de termos definido pelo usuário. Esta série se caracteriza pela soma de um termo posterior com o seu anterior subseqüente.

12. No correio local há somente selos de 3 e de 5 centavos. A taxa mínima para correspondência é de 8 centavos. Faça um algoritmo que determina o menor número de selos de 3 e de 5 centavos que completam o valor de uma taxa dada. Use estrutura de repetição.

13. Este problema tem por objetivo multiplicar inteiros sem, obviamente, utilizar o operador (*). Estaremos assim "ensinando o computador" a multiplicar inteiros ou seja : dados n e m inteiros, determine n * m.

14. Dados m, n inteiros positivos com $n \neq 0$. Então existem únicos q e r inteiros, com $0 \leq r \leq n$ e satisfazendo m = n * q + r. O inteiro q é dito quociente da divisão inteira de m por n e r é dito o resto dessa divisão. Dados m e n inteiros, utilize a fórmula acima para "ensinar ao computador" as operações m/n e m%n.

15. Escreva um algoritmo que leia um número inteiro e positivo representando um número binário, determine o seu equivalente decimal.

Exemplo: Dado 10010 a saída será 18, pois

$$1.\ 2^{4} + 0.\ 2^{3} + 0.\ 2^{2} + 1.\ 2^{1} + 0.\ 2^{0} = 18$$

16. Escreva um algoritmo que leia um número inteiro e positivo representando um número decimal, determine o seu equivalente binário.

Exemplo: Dado 18 a saída deverá ser 10010.

17. Dizemos que um número natural *n* é *palíndromo* se*:*

1º algarismo de *n* é igual ao seu último algarismo,

2º algarismo de *n* é igual ao penúltimo algarismo e assim sucessivamente.

Exemplo: 567765 e 32423 são palíndromos

567675 não é palíndromo.

Faça um algoritmo que leia um número inteiro e positivo n, verifica se é palíndromo e imprime uma mensagem dizendo se o número lido é palíndromo ou não é palíndromo.

18. A sequência : $x_{n+1} = \frac{1}{2}\left(x_n + \frac{A}{x_n}\right), \quad x_0 = 1, n \in N$; converge para a raiz quadrada de A, sendo A>0. Calcule um valor aproximado da raiz quadrada de um número dado A, através de 5 iterações.

19. Fazer um algoritmo que calcule o valor de e^x através da série: $e^x = x^0 + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots$. De modo que o mesmo difira do valor calculado através da função EXP de, no máximo, 0.0001. O valor de x deve ser lido de uma unidade de entrada. O algoritmo deverá escrever o valor de x, o valor calculado através da série, o valor dado pela função EXP. O valor e = 2.71828182845904.

20. Prepara um algoritmo que calcule o valor de H, sendo que ele é determinado pela série

H = 1/1 + 3/2 + 7/4 + 11/6 + 15/8 + 19/10 + … + 99/50.

21. Elabore um algoritmo que determine o valor de S, em que

S = 1/1 – 2/4 + 3/9 – 4/16 + 5/25 – 6/36 + 7/49 – 8/64 + 9/81 – 10/100. Resposta: 0.645635

22. Escreva um algoritmo que calcule e escreva a soma dos dez primeiros termos da seguinte série:

F = 2/500 – 5/250 + 2/400 – 5/350 + 2/300 – 5/450 +… Resposta: -0.016513

23. Imagine a sequência (1,3,6,10,15,21,28,36 ...). Faça um programa que dado um número N calcule e escreva os N termos dessa sequência.

24. Escreva um algoritmo que, dados dois números inteiros positivos m e n, determina e escreve, entre todos os pares de números inteiro (*x,y*) tais que $x \leq m$ e $y \leq n$, um par para o qual o valor da expressão $xy - x^2 + y$ seja máximo e calcula e escreve também esse máximo.

25. Um número A (um valor maior que zero) é dito permutação de um número B se os dígitos de A formam uma permutação dos dígitos de B. Por exemplo, 5412434 é uma permutação 4321445, mas não é uma permutação de 4312455. Então, escreva um algoritmo que, dados dois números positivos n e m, que não contêm dígito 0, verifica se n é uma permutação de m. A saída do algoritmo deve ser uma mensagem com o resultado da verificação.

26. Dizemos que um número *i* é congruente módulo *m* a *j* se *i* % *m* = *j* % *m*.

Exemplo: 35 é congruente módulo 4 a 39, pois

35 % 4 = 3 = 39 % 4.

Dados *n*, *j* e *m* naturais não nulos, imprimir os n primeiros naturais congruentes a *j* módulo *m* .

7. Modularização ou Funções em Linguagem C

7.1 Introdução

Os programas que temos construído até então são muito simples, pois resolvem problemas simples e apresentam apenas os componentes mais elementares dos programas: constantes, variáveis, expressões condicionais e estruturas de controle. Entretanto, a maioria dos programas resolve problemas complicados, cuja solução pode ser vista como formada de várias sub-tarefas ou módulo, cada qual resolvendo uma parte específica do problema.

Nesta seção, veremos como escrever um programa constituído de vários módulos e como estes módulos trabalham em conjunto para resolver um determinado problema algorítmico.

Um **módulo** nada mais é do que um grupo de instruções que constitui um trecho de algoritmo com uma função bem definida e o mais independente possível das demais partes do algoritmo. Cada módulo, durante a execução do algoritmo, realiza uma tarefa específica da solução do problema e, para tal, pode contar com o auxílio de outros módulos do algoritmo. Desta forma, a execução de um algoritmo contendo vários módulos pode ser vista como um processo cooperativo.

A construção de algoritmos compostos por módulos, ou seja, a construção de algoritmos através de modularização possui uma série de vantagens:

- **Torna o algoritmo mais fácil de escrever**. O desenvolvedor pode focalizar pequenas partes de um problema complicado e escrever a solução para estas partes uma de cada vez, ao invés de tentar resolver o problema como um todo;
- **Torna o algoritmo mais fácil de ler**. O fato do algoritmo estar dividido em módulos permite que alguém, que não seja o seu autor, possa entender os seus módulos separadamente;
- **Economia de tempo, espaço e esforço.** Frequentemente, necessitamos executar uma mesma tarefa em vários lugares de um mesmo algoritmo. Uma vez que um módulo foi escrito, ele pode, como veremos mais adiante, ser "chamado" quantas vezes quisermos e de onde quisermos no algoritmo, evitando que escrevamos a mesma tarefa mais de uma vez no algoritmo, o que nos poupará tempo, espaço e esforço.

A maneira mais intuitiva de proceder a modularização de problemas é realizada através da definição de um módulo principal, que organiza e coordenação trabalho dos demais módulos, e de módulos específicos para cada uma das sub-tarefas do algoritmo. O módulo principal solicita a execução dos vários módulos em uma dada ordem, os quais, antes de iniciar a execução, recebem dados do módulo principal e devolvem o resultado de suas computações.

Até o momento nossos programas eram compostos de um único módulo. Quando um programa possui mais que uma função, no momento em que desejamos que a sub-tarefa de um determinado módulo seja realizada, fazemos a referência ao módulo em vez de escrever todo o conjunto de instruções para realizá-la.

Observe a figura 5.1, quando o módulo principal referencia-se a um dos módulos, a sequência de instruções do módulo será executada até que finalize sua sub-tarefa, quando retorna ao módulo em que foi referenciado.

Na primeira vez em que módulo principal referencia-se ao módulo X, o conjunto de instruções de X é realizado até o fim, quando retorna ao módulo principal. Na segunda vez em que o módulo principal referencia-se ao módulo X, esse deslocamento de instruções se repete.

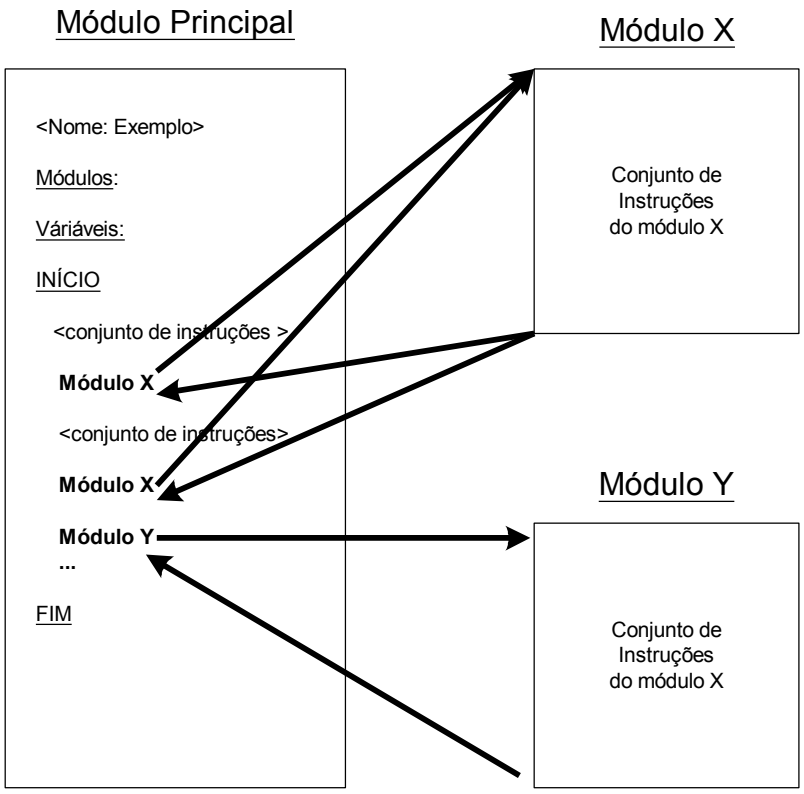


Figura 7.1 – Sequência de instruções utilizando módulos

7.2 Como Modularizar

Para organizar um algoritmo em módulos, além de compreender o problema e definir quais serão as sub-tarefas necessárias, devemos conhecer os componentes de um módulo, como criar um módulo e como solicitar a execução de um módulo.

7.3 Componentes de um módulo

Os módulos que estudaremos possuem dois componentes: corpo e interface. O **corpo** de um módulo é o conjunto de instruções que compõe o trecho de algoritmo correspondente ao módulo. Já a **interface** de um módulo pode ser vista como a descrição dos dados de entrada. O conhecimento da interface de um módulo é tudo o que é necessário para a utilização correta do módulo em um algoritmo.

A interface de um módulo é definida pelo nome do módulo e seus respectivos parâmetros. O **nome** do módulo é um identificador para referenciar o módulo. Um **parâmetro** é um tipo especial de variável que permite que o valor de um dado possa ser passado entre os módulos e qualquer para que algoritmo o use. Um parâmetro pode ser considerado como os dados de entrada para que o módulo realize sua tarefa. Os parâmetros devem receber valores quando solicitarmos a execução de um módulo.

Por exemplo, suponha que temos um para calcular o seno de um número e já existe um módulo para realizar essa tarefa, para que possamos utilizá-lo devemos conhecer sua interface, ou seja, qual o nome e quais os parâmetros do módulo.

Exemplo: seno (x)

No módulo ou subalgoritmo de cálculo do seno passamos o ângulo desejado para o calculo. Assim, x é o parâmetro.

Quando criamos um módulo, especificamos o número e tipos de parâmetros que ele necessita. Isto determina a interface do módulo. Qualquer algoritmo que use um módulo deve utilizar os parâmetros que são especificados na interface do módulo para se comunicar com ele.

7.4 Criando Módulos

Para criar um módulo precisamos construir as seguintes partes: interface e corpo. Para definir a interface precisamos identificar o tipo do módulo, o nome e os parâmetros.

Há dois tipos de módulos:

- **Função:** sempre retorna um e apenas um valor ao algoritmo que lhe chamou. Cada função tem associada ao seu valor de retorno um tipo explícito. Da mesma maneira com que os parâmetros são fixos para todas as

chamadas o retorno também é fixo. Ela pode ser vista como uma expressão que é avaliada para um único valor, sua saída, assim como uma função em Matemática.

- **Procedimento:** é um tipo de módulo usado para várias tarefas, não produzindo valores de saída, ou seja, nunca retornam valores. Em algumas linguagens não existem explicitamente (como na linguagem C).

Para definir o corpo precisamos identificar o conjunto de variáveis e instruções para realizar sua tarefa.

7.4.1 Criando Funções

Para criar uma função utilizaremos a seguinte forma geral:

SINTAXE em Algoritmo:

```
Função < tipo_de_retorno>  < nome_da_função>  ( parâmetro1;...; ...parâmetron)
  Variáveis
  <declaração das variáveis locais >
  Início

     <conjunto instruções da função>

     RETORNA  <valor_de_retorno>
  Fim

Fim-Função
```

Em que:

- *nome* é um identificador único que representa o nome da função;
- os *parâmetros* são uma lista formada pela declaração dessas variáveis especiais definida pelo tipo de dado e o nome do parâmetro da função. Essa lista de parâmetros também pode ser vazia.
- *tipo de retorno* é o tipo de dado do valor de retorno da função (REAL, INTEIRO, CARACTERE ou BOOLEANO).

  Exemplo de um lista de parâmetros: (inteiro num1, num2 ; caractere opção)

- *Conjunto de instruções* é a sequência de instruções que compõe o módulo e que sempre finaliza com um comando especial denominado a **instrução de retorno** que é da seguinte forma:

```
RETORNA <valor de retorno>
```

- *valor de retorno* é o valor de saída produzido pela função que deve ser do mesmo tipo de dado definido no tipo de retorno.

Logo após a descrição da interface podemos descrever o corpo do algoritmo da função correspondente. Para delimitar as instruções que fazem parte da função, utilizamos Fim-Função.

Exemplo: Vejamos um exemplo de uma função. Suponha que desejemos construir uma função para calcular o quadrado de um número. O número deve ser passado para a função, que calculará o seu quadrado e o devolverá para o algoritmo que a solicitou.

Vamos denominar esta função de *Quadrado*, como é mostrado a seguir:

```
Nome da Função: Quadrado
Objetivo: Função para calcular o quadrado de um número.
Dados de Entrada: número passado como parâmetro para calcular o seu quadrado.
Saída: o número ao quadrado

Função inteiro Quadrado( inteiro  num )
   Variáveis
      inteiro      res                  < declaração de variável do módulo >
   Início

      res ← num * num                   < calcula o quadrado >

      RETORNA res              < retorna o quadrado calculado >

   Fim
Fim-Função
```

Para implementar uma função em linguagem C, usamos a forma geral abaixo. Em C, não há palavras reservadas para distinguir o tipo do módulo, ou seja, se é *função* ou *procedimento*, ao especificar o tipo de dado de retorno o compilador reconhece que é uma função e que deve retornar um valor.

SINTAXE em C:

```
< tipo_de_retorno>  < nome_da_função>  ( parâmetro1;...; ...parâmetron)
{   //declaração das variáveis locais

    <conjunto instruções da função>

    return  <valor_de_retorno> ;
}
```

Uma função pode retornar ou não um valor e apenas um valor. O retorno de um valor em C é efetuado pela instrução return e é devolvido para o nome da função. A instrução return poderá ser das seguintes formas:

SINTAXE:

return (<expressão>);	// retorna o valor da expressão
return <expressão> ;	// retorna o valor da expressão
return;	// não retorna valor

Desta forma, a função *Quadrado* será implementada C da seguinte forma:

```
int Quadrado (int num)
{
   int res;
   res = num*num;
   return res;
}
```

7.4.2 Criando Procedimentos

A diferença entre funções e procedimentos é que o procedimento não retorna valor. Para criar um procedimento utilizaremos a seguinte forma geral:

SINTAXE:

Procedimento <nome_do procedimento>(parâmetro$_{1,}$... , parâmetro$_n$)
- **Variáveis**
  <declaração das variáveis locais do módulo>
  Início

    <conjunto instruções do procedimento>

  Fim

Fim-Procedimento

Em que:

- *nome* é um identificador único que representa o nome do procedimento;
- os *parâmetros* são uma lista formada pela declaração dessas variáveis especiais definida pelo tipo de dado e o nome do parâmetro da função. Essa lista de parâmetros pode possuir vários parâmetros ou ser vazia.
- *Conjunto de instruções* é a sequência de instruções que compõe o módulo. Como um procedimento não retorna valor, não é necessário utilizar a **instrução de retorno.**

Para delimitar os comandos que fazem parte do procedimento, utilizamos FIM-PROCEDIMENTO.

Exemplo: Suponha que desejemos construir um procedimento para apresentar informações do algoritmo.

```
Nome do Procedimento:  Apresentacao
Objetivo: Função para apresentar informações sobre algoritmo
Dados de Entrada:  nome do programador e objetivo
Saída: exibe na tela as informações do algoritmo passado como parâmetro

Procedimento Apresentacao ( cadeia programador, objetivo)
   Variáveis
      < nesse módulo não é necessário declarar variáveis >
   Início
      IMPRIMA "Este algoritmo foi desenvolvido por: "
      IMPRIMA  programador
      IMPRIMA  "Objetivo do algoritmo:'
      IMPRIMA  objetivo
   Fim
Fim-Procedimento
```

Como dito anteriormente, na linguagem C não é preciso identificar o tipo de módulo, é necessário identificar apenas o tipo de dado de retorno, então como serão implementados os procedimentos?

Em inglês, *void* quer dizer vazio e, é isto mesmo que o *void* é. Ele nos permite fazer funções que não retornam nada e funções que não têm parâmetros. Como os procedimentos não retornam valores, então o tipo de dado de retorno das funções será ***void*** (sem valor).

O procedimento *Apresentacao* será implementado em C da seguinte forma:

```
void Apresentacao ( cadeia programador, cadeia objetivo)
{
    printf("\n--------------------------------------------------------------------------------");
    printf("\n Este algoritmo foi desenvolvido por:  %s", programador);
    printf("\n Objetivo do algoritmo: %s",  objetivo );
    printf("\n Pressione qualquer tecla para continuar a execução do programa...");
    printf("\n--------------------------------------------------------------------------------");
}
```

Observação: Em linguagem C, não existe o tipo de dado *cadeia*, para isso definimos esse novo tipo de dado no início do programa utlizando: ***typedef char cadeia[50];***

7.4.3 Solicitando a execução de um módulo

Quando queremos utilizar os módulos que criamos para resolver uma tarefa, ou seja solicitar sua execução, devemos fazer **chamada** ao módulo. A chamada ao módulo é a forma de solicitar a execução do módulo em um determinado passo do algoritmo.

Funções e procedimentos não são diferentes apenas na forma como são implementados, mas também na forma como a solicitação da execução deles, ou simplesmente **chamada**, deve ser realizada.

A chamada de uma função é usada como um valor constante que deve ser atribuído a uma variável ou como parte de uma expressão, enquanto a chamada de um procedimento é realizada com uma instrução.

Para fazer a chamada de um módulo devemos especificar qual o nome módulo e passar valores iniciais para seus parâmetros. Esses valores iniciais devem ser do mesmo tipo de dado e especificados na mesma ordem em que foram definidos na interface do módulo.

Exemplo de chamadas do módulo *Quadrado*:

```
x ← Quadrado( 10 )      <solicita a execução do módulo quadrado para o numero 10, o
                         valor retornado pela função será atribuído à variável x >
y ← Quadrado( 20+x )    <solicita a execução do módulo quadrado para o valor
                        resultante  da soma de 20 mais o valor de x,  o valor retornado
                        pela função será atribuído à variável y>

z ← Quadrado( 5 ) + 50 <solicita a execução do módulo quadrado para o numero o 5
                        soma o valor  retornado pela função a 50 e atribui o resultado
                        à variável  z>
```

Exemplos de chamadas do módulo *Apresentacao*:

```
<solicita a execução do procedimento Apresentacao que exibirá na tela a primeira
cadeia como nome do autor e a segunda cadeia como o objetivo do algoritmo>

Apresentacao( "João", " Mostrar um exemplo do módulo" )

nome  ← "Maria"
texto   ← "Mostrar outro exemplo de chamada do módulo"

Apresentacao( nome, texto)
```

Em linguagem C, a chamada dos módulos é realizada da mesma forma, com exceção da passagem de parâmetros por referência.

7.5 Exemplo de um Algoritmo Modularizado

A partir de agora, o nosso algoritmo será composto das seguintes partes: cabeçalho, área para especificação da interface dos módulos, de declaração das variáveis, do módulo principal e da especificação do corpo dos demais módulos.

Um algoritmo para realizar a soma dos quadrados de dois números inteiros utilizando a função *Quadrado* e o procedimento *Apresentacao* seria da seguinte forma:

```
Nome do Algoritmo: soma dos quadrados
Objetivo: calcular a soma dos quadrados de dois números inteiros positivos
Dados de Entrada: dois números positivos
Saída: soma dos quadrados
<Especificação da interface dos módulos>
Módulos:
   < procedimento para exibir ao usuário as informações sobre o algoritmo >
   Procedimento Apresentacao( cadeia programador, cadeia objetivo)

   < função que calcula o quadrado de um número >
   Função inteiro Quadrado( inteiro num )

<Declaração das variáveis do módulo principal>
Variáveis:
   inteiro     x, y                    < números digitados pelo usuário >
               soma                    <quadrado>
   cadeia      nome,                   <nome do autor do algoritmo>
               descricao          <descrição objetivo do algoritmo>

< Conjunto de instruções do módulo principal >
INÍCIO
   nome ← "Fulano de Tal"
   descricao ← "Calcular a soma do quadrado de dois números"

   Apresentacao(NOME, DESCRICAO)

   FAÇA
      IMPRIMA "Entre com um número positivo: "
      LEIA x
   ENQUANTO x< 0

   FAÇA
      IMPRIMA "Entre com um número positivo: "
      LEIA y
   ENQUANTO y < 0

   soma ← Quadrado( x ) + Quadrado(y)

   IMPRIMA soma
FIM

<Especificação dos módulos>
Procedimento Apresentacao ( cadeia programador, objetivo)
   Variáveis
      < nesse módulo não é necessário declarar variáveis >
   Início
      IMPRIMA "Este algoritmo foi desenvolvido por: "
      IMPRIMA  programador
      IMPRIMA  "Objetivo do algoritmo:'
      IMPRIMA  objetivo
   Fim
Fim-Procedimento

Função inteiro Quadrado( inteiro  num )
   Variáveis
      inteiro      res                        < declaração de variável do módulo >
   Início

      res ← num * num                 < calcula o quadrado >
```

```
      RETORNA  res                    < retorna o quadrado calculado >

   Fim
Fim-Função
```

A implementação do algoritmo soma dos quadrados em linguagem C.

```
// Programa : Exemplo 1
// Objetivo: Calcular a soma dos quadrados de um número dado utilizando funções
// Dados de entrada: Dois números inteiros
// Dados de saída: A soma dos quadrados dos números lidos

//Declaração das Bibliotecas
#include <stdio.h>
#include <conio.h>

//Definição de Tipos
typedef char cadeia[50];

//Definição das Constantes
#define NOME         "Fulano de Tal"
#define DECRICAO "Calcular a soma do quadrado de dois números inteiros positivos"

//prototipação dos módulos
void Apresentacao ( cadeia programador, cadeia objetivo) ;
int Quadrado (int num);

// função principal
int main( void )
{
   int     x, y, soma;

   Apresentacao(NOME, DESCRICAO);

   do{
      printf( "\n Digite um número positivo:" );
      scanf("%d", &x);
   }while ( x < 0 );


   do{
      printf( "\n Digite um número positivo:" );
      scanf("%d", &y);
   }while ( y < 0 );

   soma =  Quadrado( x ) + Quadrado(y);
   printf( "\n Soma dos quadrados : %d", soma);
    return 0;
}

//Especificação do corpo das funções
void Apresentacao ( cadeia programador, cadeia objetivo)
{
   printf("\n--------------------------------------------------------------------------------");
   printf("\n Este algoritmo foi desenvolvido por:  %s", programador);
```

```
    printf("\n Objetivo do algoritmo: %s",  objetivo );
    printf("\n Pressione qualquer tecla para continuar a execução do programa...");
    printf("\n--------------------------------------------------------------------------------");
}

int Quadrado (int num)
{
    int res;

    res = num*num;

    return res;
}
```

7.6 Passagem de Parâmetros

Quando criamos módulos, sejam eles funções ou procedimentos, normalmente estamos delegando funções menores para eles. Como são subordinados aos algoritmos ditos chamadores então eles estão dependentes das informações que estes passem a eles. Estas informações são passadas através dos parâmetros.

Ao chamarmos um módulo, especificamos os valores que desejamos para que o módulo realize sua tarefa, neste momento estamos passando dados para os parâmetros. Essa passagem de parâmetros pode ser realizada de duas formas: por valor ou por referência.

7.6.1 Passagem por Valor

Os valores dos parâmetros passados por valor são passados por um mecanismo denominado **cópia**. O valor do parâmetro (uma constante ou o valor de uma variável ou expressão) é atribuído ao parâmetro quando da chamada do procedimento/função.

Até o momento utilizamos somente passagem por valor. Por exemplo, na chamada da função *quadrado*, *x* é um parâmetro passado por **valor**, e, assim sendo, o valor de *x* copiado para o parâmetro da função. Na chamada da função *Quadrado*(*x*), o valor da variável *x* é atribuído ao parâmetro *num* da função *Quadrado*, o parâmetro *num* assume o papel de uma variável do procedimento/função e o quadrado é calculado para esse valor. Na chamada da função *Quadrado(y)* , o parâmetro *num* recebe o valor da variável *y*, portanto o módulo retornará o quadrado da variável y.

7.6.2 Passagem por Referência

A passagem por referência ocorre quando alterações nos parâmetros, dentro da função, alteram os valores das variáveis que foram passadas para a função. Este nome vem do fato de que, neste tipo de chamada, não se passa para a função o valor das variáveis, mas sim suas referências (a função usa as referências para alterar os valores das variáveis, que foram passadas como parâmetro, fora da função).

A referência de uma variável é seu endereço de memória, quando realizamos uma passagem de parâmetro por referência, o parâmetro não recebe a cópia do valor da variável, e sim, o endereço de memória onde está armazenada essa variável. Assim, qualquer alteração no valor do parâmetro feita pelo procedimento/função acarretará uma modificação no valor da variável passada como parâmetro.

Para indicar que um parâmetro é passado por **referência**, na especificação dos parâmetros de um módulo utilizaremos a palavra-chave REF antes do tipo de dado do parâmetro.

Por exemplo, para criarmos um procedimento para realizar a troca de valores entre duas variáveis devemos passar as suas respectivas referências para que o módulo possa alterá-las.

Quando a chamada *Troca*(*num1, num2*) é executada, a variável *a* e *b* (que estão precedidas da palavra-chave REF) receberam o endereço de memória das variáveis *num1* e *num2* respectivamente. Quando as variáveis *a e b* forem alteradas no procedimento, esta atualização afetará o valor de *num1* e *num2*. Se essas variáveis fossem passadas por valor esta atualização não seria possível.

Nome do Algoritmo: Troca
Objetivo: mostrar um exemplo de passagem por referência
Dados de Entrada: valores inteiro para duas variáveis
Saída: valores das variáveis invertidos

<Especificação da interface dos módulos>
Módulos:
Procedimento Troca (REF inteiro a, b)

<Declaração das variáveis do módulo principal>
Variáveis:
real num1, num2 < números digitados pelo usuário >

< Conjunto de instruções do módulo principal >
INÍCIO

LEIA num1
LEIA num2

troca(num1, num2)

IMPRIMA num1
IMPRIMA num2

FIM

<Especificação dos módulos>

Nome do Procedimento: troca
Objetivo: Procedimento para trocar os valores de duas variáveis.
Dados de Entrada: as duas variáveis inteiras passadas por referência
Saída: As duas variáveis com valores invertidos

Procedimento Troca (REF inteiro a, REF inteiro b)
Variáveis
inteiro aux
Início
aux ← a
a ← b
b ← aux
Fim
Fim_Procedimento

Devemos observar que os parâmetros passados por referência devem ser obrigatoriamente variáveis, uma vez que não faz sentido modificar o valor de uma constante ou de uma expressão.

Em linguagem C, para passar parâmetros por referência utilizaremos os ponteiros. Um ponteiro é uma variável que armazena endereço de memória de outras variáveis e é justamente isso que os parâmetros passados como referência devem receber. Portanto para indicar que um parâmetro deve receber uma referência devemos declará-lo da mesma forma que declaramos os ponteiros, isto é, utilizar o operador ` * ' antes do nome do parâmetro. O uso do operador '*' também será utilizado no corpo do módulo todas as vezes que desejarmos referenciar a um parâmetro é passado por referência.

Além do cuidado para especificar e manipular os parâmetros passados por referência na função, quando a função for chamada, teremos que lembrar de colocar o operador '&' na frente das variáveis que estivermos passando por referência para a função. Veja a implementação do algoritmo da troca.

Nome do Programa: troca
Objetivo: mostrar um exemplo de passagem por referência
Dados de Entrada: valores para duas variáveis
Saída: valores das variáveis invertidos

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

//prototipação das funções
void Troca ( int * a, int *b);

int main( void )
{
    int    num1, num2;


    printf("Entre com 2 números inteiros: ");
    scanf("%d %d", &num1, &num2);

    printf("\n num1=%d e num2=%d", num1, num2);
    Troca(&num1, &num2);
    printf("\n num1=%d e num2=%d", num1, num2);
     system ("pause");
     return 0;

}

void Troca (int *a, int *b)
{
    int aux;

    aux = *a;
    *a = *b;
    *b = aux;
}
```

Neste último exemplo, o que está acontecendo é que passamos para a função *Troca* o endereço das variáveis *num1* e *num2*. Estes endereços são copiados nos ponteiros *a* e *b*. Através do operador '*' estamos acessando o conteúdo apontado pelos ponteiros e modificando-o. Este conteúdo corresponde aos valores armazenados em *num1* e *num2*, que, portanto, estão sendo modificados.

7.6.3 Passagem por valor x por referência

Sempre que desejarmos passar parâmetros para uma função, devemos distinguir e saber quando utilizar passagem de parâmetros por valor ou por referência. Quando a função/procedimento precisa apenas de um valor inicial para o parâmetro utilizaremos passagem por valor. Quando desejarmos que as atualizações realizadas pelo módulo alterem as variáveis externas a ele, devemos utilizar passagem por referência. Para exemplificar vamos criar módulos com passagem por valor e por referência.

Exemplo: Suponha que desejamos fazer uma função para calcular a potência de um número positivo elevado a um determinado expoente. Nesse caso podemos realizar a passagem dos parâmetros base e expoente por valor, já que não queremos que o módulo altere-as, e retornar a potência calculada.

```
Nome do Algoritmo: Potencia 1
Objetivo: Calcular potência
Dados de Entrada: dois números inteiros positivos, indicando a base e o expoente
Saída: potência
<Especificação da interface dos módulos>
Módulos:
   Função inteiro Potencia ( inteiro base, inteiro expoente )
<Declaração das variáveis do módulo principal>
Variáveis:
   inteiro num1, num2, resultado
INÍCIO
   LEIA num1
   LEIA num2
   resultado ← Potencia(num1, num2)
   IMPRIMA resultado
FIM

<Especificação do corpo do módulo potência>

Função inteiro Potencia ( inteiro base, inteiro expoente )
   Variáveis
      inteiro cont, pot
   Início
      pot ← 1
      PARA cont DE 1 ATÉ  expoente PASSO 1 FAÇA
         pot ← pot * base
      FIM-PARA
      RETORNA pot
   Fim
Fim-Função
```

Exemplo: Outra forma de especificar a função potencia é passar também como parâmetro a referência da variável que receberá o resultado da potencia, além dos parâmetros base e expoente. Desta forma o parâmetro que receberá a potência calculada deverá ser passada por referência.

```
Nome do Algoritmo: Potencia 2
Objetivo: Calcular potência
Dados de Entrada: dois números inteiros positivos, indicando a base e o expoente
Saída: potência
<Especificação da interface dos módulos>
Módulos:
   Procedimento inteiro Potencia ( inteiro base, expoente, REF inteiro pot)
<Declaração das variáveis do módulo principal>
Variáveis:
   inteiro num1, num2, resultado
INÍCIO
 LEIA num1
   LEIA num2

   Potencia(num1, num2, resultado)

   IMPRIMA resultado
FIM

Procedimento Potencia ( inteiro base, expoente, REF inteiro pot)
   Variáveis
      inteiro cont
   Início
      pot ← 1

      PARA cont DE 1 ATÉ  expoente PASSO 1 FAÇA
         pot ← pot * base

      FIM-PARA

   Fim
Fim-Procedimento
```

Observe que nessa segunda implementação do módulo *Potencia,* o cálculo é atribuído diretamente ao parâmetro passado por referência.

Exemplo: Em nosso primeiro programa modularizado, soma dos quadrados, podemos perceber que a leitura das variáveis *x* e *y* ocorrem da mesma maneira. Por que não criar um módulo para realizar tal tarefa? Nesse caso, vamos criar um módulo para ler um número positivo que armazena o número lido e validado na variável passada como parâmetro. O que precisamos definir é se a passagem de parâmetro será por valor ou por referência. Como o módulo precisa alterar a variável passada como parâmetro, então utilizaremos passagem por referência.

```
Nome do Procedimento: Ler_número
Objetivo: Procedimento para ler um número positivo.
Dados de Entrada: parâmetro passado por referência para o procedimento.
Saída: número positivo armazenado no parâmetro passado por referência.

Procedimento Ler_numero( REF inteiro val )
    Início
        FAÇA
            < solicita a entrada do número >
            IMPRIMA "Entre com um número positivo:"
            LEIA val
        ENQUANTO val < 0              < lê enquanto entrarem números negativos >
    Fim
Fim-Procedimento
```

A implementação em linguagem C desse procedimento será da seguinte forma:

```
void Ler_numero (int *val)
{
    do {
        printf ("\nEntre com um numero positivo:");
        scanf ("%d", val);
    }while (*val < 0 );
}
```

Como vimos anteriormente, para referenciar um parâmetro passado por referência em linguagem C, devemos utilizar o operador * para obtermos o conteúdo do parâmetro. Então porque não utilizamos esse operador na chamada da função scanf( )? A função scanf( ) usa passagem por referência porque ela precisa alterar as variáveis que passamos para ela. Quando utilizamos o & passamos o endereço de uma variável. No procedimento *Ler_numero* a variável *val* já possui um endereço e por isso não precisamos utilizar o operador &.

Agora no nosso algoritmo, em vez de escrever as instruções para ler e validar uma variável, faremos a chamada do procedimento *Ler_numero* passando a referência para variável que desejamos ler. O módulo principal do nosso primeiro algoritmo será reescrito utilizando esse novo módulo.

```
//Alterações no módulo principal para utilizar o procedimento Ler_numero
<Especificação da interface dos módulos>
Módulos:
   Procedimento Apresentacao( cadeia programador, cadeia objetivo)
   Função real Quadrado( inteiro num )
   Procedimento Ler_numero ( REF inteiro val)

<Declaração das variáveis do módulo principal>
Variáveis:
   inteiro   x, y                  < números digitados pelo usuário >
             soma                  <quadrado>
   cadeia    nome,                 <nome do autor do algoritmo>
             descricao        <descrição objetivo do algoritmo>
< Conjunto de instruções do módulo principal >
INÍCIO
   nome ← "Fulano de Tal"
   descricao ← "Calcular a soma do quadrado de dois números"

   Apresentacao(nome, descricao)

   Ler_numero(x)
   Ler_numero(y)

   soma ← Quadrado( x ) + Quadrado(y)

   IMPRIMA soma
FIM
```

Já a nossa função principal será implementada da seguinte forma:

```
//Alterações no programa para utilizar a função ler_numero
//Declaração das Bibliotecas
#include <stdio.h>
//Definição de Tipos
typedef char cadeia[50];

//Definição das Constantes
#define NOME        "Fulano de Tal"
#define DESCRICAO "Calcular a soma do quadrado de dois números inteiros
positivos"

//prototipação das funções
void Apresentacao ( cadeia programador, cadeia objetivo) ;
int Quadrado (int num);
void Ler_numero(int *val);

// função principal
int main( void )
{
   int    x, y, soma;
   Apresentacao(NOME, DESCRICAO);

   Ler_numero(&x);
   Ler_numero(&y);
   soma =  Quadrado( x ) + Quadrado(y);
   printf( "\n Soma dos quadrados : %d", soma);
    return 0;
 }
```

7.7 Escopo dos Dados

O escopo de um dado ou variável está vinculado a sua visibilidade em relação aos módulos de um algoritmo.

Exemplos:

```
Função booleano xyz (inteiro z)
  Variáveis
     inteiro     x
  Início
       x ← z
       RETORNA verdadeiro
  Fim
Fim-Função
```

```
Função booleano abc (inteiro g)
  Variáveis
     inteiro     f
  Início
       f ← g
       RETORNA verdadeiro
  Fim
Fim-Função
```

Neste caso a variável x e o parâmetro z só podem ser utilizados no algoritmo xyz e não no algoritmo abc, da mesma forma a variável f e o parâmetro g só podem ser utilizados no algoritmo abc. ·O local onde uma variável pode ser utilizada define o seu **Escopo**.

Para que não existam ambiguidades na utilização dos identificadores, não poderá existir nenhum caso onde existam dois identificadores com o mesmo nome dentro do mesmo escopo, i.e., dentro de um escopo existe uma unicidade de nomes. O mesmo vale para os parâmetros, ou seja, não pode existir um parâmetro que tenha o mesmo nome de uma variável ou constante dentro do mesmo escopo. Em escopos distintos podem existir nomes iguais.

Exemplos:

```
Função booleano xyz (inteiro z)
  Variáveis
    inteiro     x
  Início
    x ← z
    RETORNA verdadeiro
  Fim
Fim-Função
```

```
Função booleano abc (inteiro g)
  Variáveis
    inteiro     x
  Início
    x ← g
    RETORNA verdadeiro
  Fim
Fim-Função
```

Nestes módulos não existem problemas com a utilização da variável x, pois existe um x para cada algoritmo e estes nomes (x) indicam posições de memórias diferentes.

7.7.1 Variáveis Locais e Globais

Podemos ter dois tipos de variáveis: as variáveis **globais** e as **locais**.

Uma variável é considerada **global** quando é declarada no início do algoritmo, antes da declaração das variáveis locais ao algoritmo, podendo ser utilizada por qualquer função/procedimento, assim a variável passa ser visível no algoritmo principal e nos outros módulos.

Uma variável é considerada **local** quando é declarada dentro de um módulo ou no algoritmo principal e é válida somente dentro da rotina à qual está declarada.

```
Variáveis:
      real     x
Função lógico abc (inteiro g)
   Variáveis
      inteiro      x
   Início
         x ← g
         retorna verdadeiro
   Fim
FimFunção
```

A variável x do tipo real definida fora do escopo de um algoritmo é dita global, enquanto a variável x declarada dentro do algoritmo tem escopo local a função. O que acontece quando existe uma declaração global e uma local para uma mesma variável ? As linguagens de programação normalmente tratam este tipo de ambigüidade da seguinte maneira :

1. Se existir uma declaração local para esta variável então a expressão usará esta variável

2. Caso contrário o compilador irá verificar se existe uma declaração global para a variável.

No exemplo acima a atribuição x ← g irá colocar na variável local x o valor da variável g, e não utilizará a variável global x.

7.8 Exercícios

1. Elabore um programa que efetue a leitura de três valores positivos (A, B e C) e apresente como resultado final a soma dos quadrados dos três valores lidos.

2. Escreva uma função para calcular o valor do fatorial de um número passado por parâmetro à função.

3. Faça uma função que, conta divisores que recebe um número inteiro $x > 0$ e devolve o número de divisores positivos que x tem.

Exemplo: O número 12 tem 6 divisores (1, 2, 3, 4, 6 e 12)

4. Faça uma função que recebe um número inteiro $p > 0$, e devolve 1 caso o número seja primo e 0 caso contrário.

5. Faça um programa que leia uma sequência de números inteiros e determine quais desses números são *divprimos*.

Dizemos que um número é *divprimo* se o número de divisores dele for primo.

Exemplo: O número 7 é *divprimo*, já que 7 tem 2 divisores, e 2 é primo. Já o número 12 não é *divprimo*, uma vez que o 12 tem 5 divisores.

Utilize necessariamente as funções feitas nos itens anteriores.

6. Escreva uma função que recebe dois números inteiros positivos e determina o produto dos mesmos, utilizando o seguinte método de multiplicação:

(a) dividir, sucessivamente, o primeiro número por 2, até que se obtenha 1 como quociente;

(b) paralelamente, dobrar, sucessivamente, o segundo número;

(c) somar os números da segunda coluna que tenham um número impar na primeira coluna.

O total obtido é o produto procurado.

Exemplo:

9	6	→6
4	12	
2	24	
1	48	→48
		54

7. Faça uma função que converte um número inteiro positivo em representação decimal para a representação binária.

8. Faça um programa permita converter um número em representação binária para decimal ou decimal para binária, dependendo da opção do usuário.

9. Faça uma função que recebe dois números inteiros, x e y, e retorna o quociente e o resto da divisão de x por y.

10. Sobre variáveis locais e globais, quais afirmações são verdadeiras?

a) Variáveis globais podem ser manipuladas e alteradas por qualquer módulo.

b) Variáveis locais podem ser manipuladas e alteradas apenas no módulo em que elas foram declaradas.

c) Um módulo pode ter variável com o mesmo nome de variáveis de outro módulo.

d) Variáveis com o mesmo nome indicam o mesmo endereço de memória.

11. Sobre parâmetros, quais afirmações são verdadeiras?

a) Parâmetros são dados entrada para os módulos.

b) Parâmetros são variáveis locais do módulo inicializadas na chamada do módulo.

c) Um parâmetro passado por referência é uma variável do módulo que recebe o endereço de memória da variável especificada na chamada.

d) Um parâmetro deve ser passado por referência para que o outro módulo altere o conteúdo desse parâmetro.

e) Quando um módulo finaliza sua execução, suas variáveis e seus respectivos valores continuam sendo válidos para os outros módulos.

12. Complete a tabela abaixo:

Considere a seguinte declaração:

```
int i, j;
char c;
float f;
```

Protótipo da Função	Chamada da Função	Certo ou Errado? Se errado, escreva a forma correta.
void x1(int a, float b);	x1(int i, float f);	
int x2 (int a, float b);	j = x2(f , i+10);	
char x3 (char letra);	c= x3(c);	
void x4 (int *a, int *b);	f = x4(i, j);	
void x5 (int *a, int *b);	j = x5(&i, &f);	
void x6 (void);	x6( );	
float x7 (float a);	f= x7(10.0) +x7(f +5.0);	

13. Mostre a saída para os trechos de programas abaixo.

```
int F1(int a, int b);

int main( void )
{
        int     x = 5; y = 15, z =30;
        z = F1( x,y);
        printf(" x= %d, y=%d, z=%d", x,y,z);
        return 0;
}
int F1(int a, int b)
{       int i;
        if (a>b)
            i=1;
        else
            i=0;
        return i;
}
```

Saída:_____

```
void F2(int a,int b, int *c);

int main( void )
{
        int     x = 10, y = 5, z = 30;
        F2( x, y, &z);
        printf(" x= %d, y=%d, z=%d", x,y,z);
        return 0;
}
```

```
void F2(int a,int b, int *c)
{       int i;

        *c = 0;
        for(i=0 ; i< b ; i++)
            *c = *c + a;
}
```

Saída:_____

```
void F3(int a, int b, int c);

int main( void )
{
        int     x= 45 , y =30, z= 25;
        F3( x,y, z);
        printf(" x= %d, y=%d, z=%d", x,y,z);
        return 0;
}

void F3(int a,int b, int c)
{
        a = b + c;
        b = a + c;
        c = a + b;
}
```

Saída:_____

```
void F4(int *a,int *b, int *c);

int main( void )
{
        int     x = 10 , y = 5, z = 6;
        F4( &x, &y, &z);
        printf(" x= %d, y=%d, z=%d", x,y,z);
        return 0;
}
void F4(int *a, int *b, int *c)
{
        *a = *a + *a;
        *b = *a + *b;
        *c = *c + *b;
}
```

Saída:_____