Material desenvolvido por
Prof. M.Sc. Adriano Brum Fontoura

Motivação

A automatização de tarefas é um aspecto marcante da sociedade moderna. O avanço tecnológico tem como elementos fundamentais a análise e a obtenção de descrições da execução de tarefas em termos de ações simples o suficiente, tal que pudessem ser automatizadas por uma máquina especialmente desenvolvida para este fim, O COMPUTADOR.
Na área de ciência da computação houve um processo de desenvolvimento simultâneo e interativo de máquinas (hardware) e dos elementos que gerenciam a execução automática (software) de uma dada tarefa. E essa descrição da execução de uma tarefa, como considerada acima, é chamada **algoritmo**.
O objetivo desta disciplina é a Lógica de Programação dando uma base teórica e prática, suficientemente, para que, o aluno domine a elaboração de algoritmos e esteja habilitado a aprender uma linguagem de programação.

Uma Visão Geral do Processo de Desenvolvimento de Software

Antes de começarmos a estudar lógica de programação, é importante que tenhamos uma visão geral do processo de desenvolvimento de software, uma vez que estudaremos lógica de programação com o objetivo de conseguir um bom embasamento para a prática da programação de computadores.

O processo de desenvolvimento de software consiste basicamente num conjunto de atividades divididas em etapas, onde o objetivo é, ao executar estas etapas, chegar à efetiva construção de um software[1].

Podemos encontrar na literatura em Informática, mais especificamente na área de engenharia de software, várias formas de representação das etapas em que consiste o processo de desenvolvimento de software. Estas formas de representação podem variar tanto na quantidade de etapas como nas atividades que devem ser realizadas por cada etapa.

A seguir apresentaremos uma forma de representação do processo de desenvolvimento de software que é bastante encontrada na literatura:

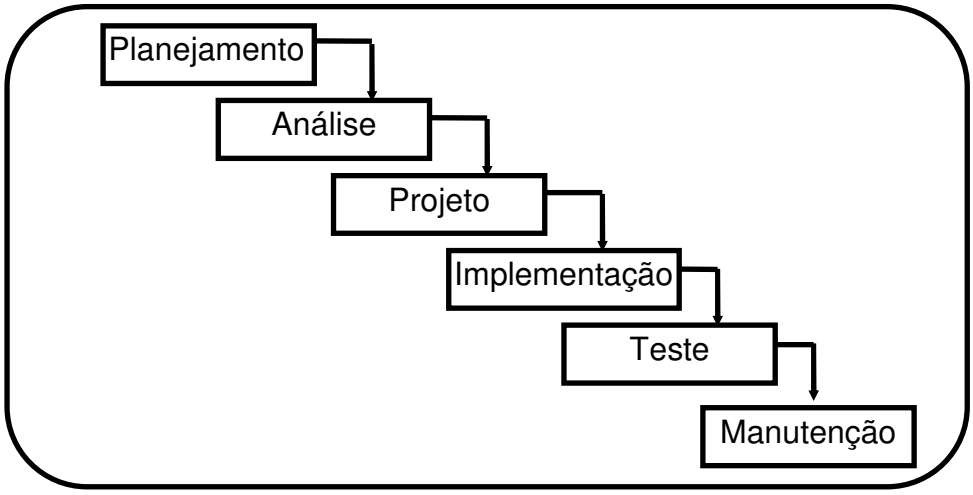


Figura 1 Etapas do processo de desenvolvimento de software.

Na Figura 1 podemos ver o processo de desenvolvimento de software dividido em 6 etapas. A seguir daremos uma rápida explicação das atividades realizadas por cada etapa.

[1] Conjunto de instruções (comandos) interpretáveis pelo computador.

• **Planejamento:** na etapa de planejamento é onde deve ser desenvolvido um plano inicial de desenvolvimento, levando em considerações questões como definição da abrangência do sistema, missão e objetivos do sistema, cronogramas de desenvolvimento, análise custo x benefício, levantamento inicial de informações etc.

• **Análise:** também chamada de análise de requisitos, é onde deve se obter um claro entendimento sobre o sistema. A análise proporciona a base para uma boa implementação do software. Nesta etapa são construídos os modelos do sistema.

• **Projeto:** também chamada de especificação do projeto, é onde propomos uma arquitetura de implementação para o software, que atenda aos requisitos do sistema identificados na análise. Aqui passamos a nos preocupar com os diversos aspectos computacionais necessários para uma boa implementação do software. Os algoritmos dos programas a serem implementados são construídos nesta fase.

• **Implementação:** a etapa de implementação é onde os programas são efetivamente construídos, a partir da arquitetura de implementação feita na etapa anterior. Nesta etapa é onde a atividade de codificação ocorre de forma massiva.

• **Teste:** nesta etapa todos os programas construídos serão testados de forma exaustiva. Existe uma grande variedade de testes que são realizados, indo desde o teste unitário dos módulos de programas até o teste de integração de todo o sistema de software.

• **Manutenção:** é onde ocorrem ajustes do software implementado, que podem ser ocasionados por vários motivos: erros de projeto identificados após a implementação e o teste do software, inovações tecnológicas, evolução do sistema etc.

Introdução à Lógica de Programação

Como esta é uma disciplina de lógica de programação, vamos iniciar nossos estudos procurando entender o que é **lógica** de uma forma geral. A seguir serão dadas algumas definições que procuram elucidar o termo lógica.

Lógica:

• [do grego logiké, que significa "arte de raciocinar"]. Na tradição clássica, aristotélico-tomista, conjunto de estudos que visam a determinar os processos intelectuais que são condição geral do conhecimento verdadeiro. (1ª. definição encontrada no Dicionário Aurélio da Língua Portuguesa)

• Coerência de raciocínio, de idéias. (6ª. definição encontrada no Dicionário Aurélio da Língua Portuguesa)

• Maneira de raciocinar particular a um indivíduo ou a um grupo: *a lógica da criança, a lógica primitiva, a lógica do louco*. (7a. definição encontrada no Dicionário Aurélio da Língua Portuguesa)

A lógica trata da correção do pensamento. Como filosofia ela procura saber por que pensamos de uma forma e não de outra. Poderíamos dizer também que a lógica é a arte de pensar corretamente e, visto que a forma mais complexa de pensamento é o raciocínio, a Lógica estuda ou tem em vista a "correção do pensamento". **A Lógica ensina a colocar Ordem no Pensamento.**

Exemplo:

Todo vulcano tem orelhas pontudas.
Spock é vulcano.
Logo, Spock tem orelhas pontudas.

- Se quisermos ter sucesso numa modalidade esportiva, devemos treinar, descansar e nos alimentarmos adequadamente.
- Se quisermos desenvolver bons programas de computador, devemos programá-lo logicamente, para que este possa resolver o problema desejado da forma mais otimizada possível, dado um conjunto de restrições. É neste ponto que entra o conceito de lógica de programação.

Lógica de Programação: raciocínio lógico empregado no desenvolvimento de programas de computador, fazendo uso ordenado dos elementos básicos suportados por um dado estilo de programação.

Uma boa lógica de programação é desenvolvida a partir de um conjunto de elementos, entre eles:
- **Organização**
- **Criatividade**
- **Perseverança**
- **Padronização**
- **Otimização**

Algoritmos e lógica de programação

O estudo de ambos é essencial no contexto do processo de criação de um *software* (programa de computador). O algoritmo está diretamente relacionado com a etapa de projeto de um software, mesmo antes de sabermos qual será a linguagem usada na codificação se especifica o software por meio de um algoritmo.
Nesta etapa e possível se verificar em um nível maior de abstração, se o software atenderá as necessidades da proposta original.

Algoritmizando a Lógica

Algoritmo

- É a descrição, de forma lógica, dos passos a serem executados no cumprimento de determinada tarefa.
- "O algoritmo pode ser usado como uma ferramenta genérica para representar a solução de tarefas independente do desejo de automatizá-las, mas em geral está associado ao processamento eletrônico de dados, onde representa o rascunho para programas (Software)".

Apesar do nome estranho, os algoritmos são muito comuns no nosso cotidiano, como por exemplo, em uma receita de bolo. Nela estão descritos os ingredientes necessários e a seqüência de passos ou ações a serem cumpridos para que se consiga fazer um determinado tipo de bolo.
Em um modo geral, um algoritmo segue um determinado PADRÃO DE COMPORTAMENTO, com objetivo de alcançar a solução do problema.
Padrão de Comportamento: imagine a seqüência de números: 1,6,11,16,21,26, ... Para determinar qual será o sétimo elemento dessa série, precisamos descobrir qual é a sua regra de formação, isto é, qual é o seu padrão de comportamento. Como a seqüência segue uma certa constância, facilmente determinada, somos capazes de determinar qual seria o sétimo termo ou outro qualquer.

Programa

- É uma seqüência finita de etapas que devem ser executadas pelo computador para resolver um determinado problema.
- Um programa de computador e a implementação de um determinado algoritmo utilizando um linguagem de programação

Formalização de um algoritmo

A tarefa de especificar os algoritmos para representar um programa consiste em detalhar os dados que serão processados por este programa e as instruções que vão operar sobre estes dados.
Esta especificação pode ser feita livremente, sem regras, mas é importante formalizarmos a descrição de um algoritmo segundo alguma convenção, para que todos os envolvidos na sua criação possam entendê-lo.

- Em primeiro lugar, é necessário definir um conjunto de regras que regulamentem a escrita do algoritmo, ou seja, regras de **sintaxe**.

P.exemplo: (algoritmo usando as sintaxe do VisuAlg)

(Algoritmo)	(Estrutura de um algoritmo/programa)
algoritmo "Primeiro"	Programa <nome_do_Programa>
var	Constantes
valor: inteiro	{Aqui declaramos as constantes}
Inicio	Variáveis
escreva("Informe o valor:")	{Aqui declaramos as variáveis}
leia(valor)	Inicio {início do bloco de execução}
escreval("Valor digitado = ", valor)	Escreva(<mensagem que sai no vídeo ou impr.>);
fimalgoritmo	Leia(<colocamos a variável a ser lida>);
	Escreva(<mensagem que sai no vídeo ou impr.>);
	Fim {fim do bloco de execução}

- Em segundo lugar é preciso estabelecer as regras que permitem interpretar um algoritmo, que são as regras de **semântica**.

Sintaxe e Semântica

Sintaxe

A **sintaxe** de um algoritmo resume-se nas regras para escrevê-lo corretamente. Em computação, essas regras indicam quais são os tipos de comandos que podem ser usados e também como neles escrever expressões
Por Exemplo:
Comando que envia uma mensagem para o monitor do computador

```
Escreva("Valor da Média");   { Program Design Language – PDL }   => Mensagem
printf("Valor da Media");    {Linguagem C}
Escreva(Média);              {Comando que exibe o resultado final, saída de dados} => Variável
printf(Media);               {Comando que exibe o resultado final, saída de dados}
```

Semântica

A **semântica** e a descrição de como as construções sintaticamente corretas são interpretadas ou executadas, ou seja, estabelece regras para sua interpretação.
Desta forma, a semântica de um algoritmo sempre acompanha a sua sintaxe, fornecendo um significado.
A importância da formalização de um algoritmo, sua sintaxe e semântica podem ser resumidas assim:

- Evitar ambigüidades, pois definem regras sintáticas e semânticas que sempre são interpretadas da mesma forma.
- Impedir a criação de símbolos ou comandos desnecessários na criação de um algoritmo: representam um conjunto mínimo de regras que pode ser utilizado em qualquer algoritmo.
- Permitir uma aproximação com as regras de uma linguagem de programação, fazendo, assim uma fácil tradução de um algoritmo para sua implementação.

Técnica de resolução por método cartesiano

A famosa frase de Descartes "***Dividir para conquistar***" é muito importante dentro da programação. È um método que ataca um grande problema, de difícil solução, dividindo-o em problemas menores, de solução mais fácil. Se necessário, pode-se dividir novamente as partes não compreendidas.
Esse método pode ser esquematizado em passos:

I. Dividir o problema em partes;
II. Analisar a divisão e garantir a coerência entre as partes;
III. Reaplicar o método, se necessário.

Por Exemplo: A solução de: $\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$

Planejamento reverso

Consiste em a partir do resultado final, determinar quais são os componentes básicos. Ou seja, a partir da saída desejada, devemos poder determinar, reversamente, quais são os componentes da entrada de dados necessários.
Por Exemplo: algoritmo que calcule o juro de um dado valor, onde partimos do valor final a nossa analise.

Método para construí um algoritmo

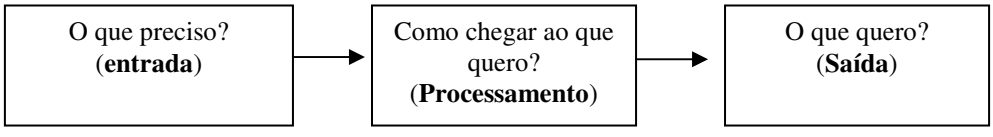
Utilizando os conceitos já desenvolvidos, esquematizaremos um método para construir um algoritmo logicamente correto:

A. **Identificar o problema**: Deve-se ler o enunciado de um exercício quantas vezes for necessário, até compreende-lo completamente. A maior parte da resolução de um exercício consiste na compreensão completa do enunciado.

B. **Identificar a Entrada de Dados:** Retirar a relação das entradas de dados do enunciado, descobrirmos quais são os dados que devem ser fornecidos ao programa, via teclado, a partir dos quais são desenvolvidos os processamentos necessários.

C. **Identificar a Saída de Dados:** Através do enunciado podemos descobrir quais são as informações que devem ser mostradas, ou seja, a relação de saídas de dados para compor o resultado final, objetivo do algoritmo.

D. **Identificar o Processamento:** Determinar o que deve ser feito para transformar as entradas nas saídas especificadas.
Nessa fase é que teremos a construção do algoritmo propriamente dito. Devemos determinar qual seqüência de passos ou ações é capaz de transformar um conjunto de dados nas informações de resultado. Para isso utilizarmos os fatores descritos anteriormente, tais como legibilidade, portabilidade, método cartesiano e planejamento reverso, e finalmente podemos construir o algoritmo.



Formas de Representação de Algoritmos

Algumas formas de representação de algoritmos tratam o problema apenas em um nível lógico, abstraindo-se de detalhes de implementação muitas vezes relacionados com uma linguagem de programação especifica. Por outro lado existem formas de representação de algoritmos que possuem

uma maior riqueza de detalhes e muitas vezes acabam por obscurecer a idéia principal, o algoritmo, dificultando seu entendimento
Dentre estas formas as mais conhecidas são:

- Descrição Narrativa
- Diagrama de Nassi-Shneidermam ou Diagrama de Chapin
- Fluxograma / Diagrama de blocos
- Pseudocódigo /Português Estruturado (Portugol)

Descrição Narrativa

Nesta forma de representação os algoritmos são expressos diretamente em linguagem natural. Como por exemplo, o algoritmo da troca de um pneu.

1. Afrouxar os parafusos;
2. Suspender o carro;
3. Retirar as porcas e o pneu;
4. Colocar pneu reserva;
5. Apertar as porcas;
6. Abaixar o carro;
7. Dar o aperto final nas porcas.

Esta representação é pouco usada na prática, devido ao fato que a utilização da linguagem natural muitas vezes dá oportunidade a más interpretações, ambigüidade e imprecisões.

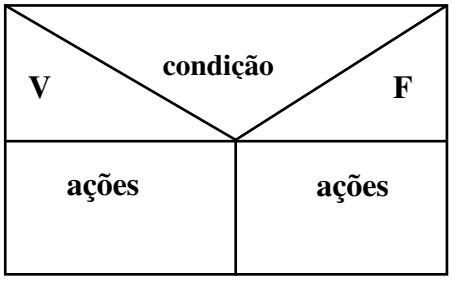
Diagrama de Nassi-Shneiderman

O diagrama foi desenvolvido por Nassi e Shneidermam e ampliado por Ned Chapin, os quais resolveram substituir o diagrama de blocos tradicional por um diagrama de quadros que permite apresentar uma visão hierárquica e estruturada da lógica do programa.
Esta ferramenta de representação oferece grande clareza para a representação de sequenciação, seleção e repetição num algoritmo, utilizando-se de uma simbologia própria. A idéia básica deste diagrama é representar as ações de um algoritmo dentro de um único retângulo, subdividido-o em retângulos menores, que representam os diferentes blocos de seqüência de ações do algoritmo. Seleção e repetição também são representadas de forma gráfica, dentro dos retângulos.

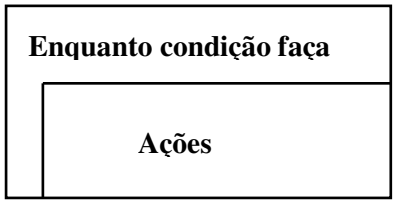
Seqüência

ação-1
ação-2
ação-n

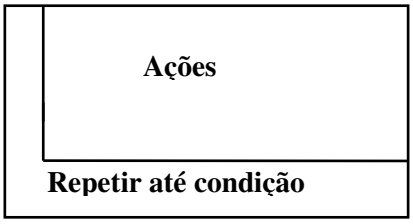
Seleção



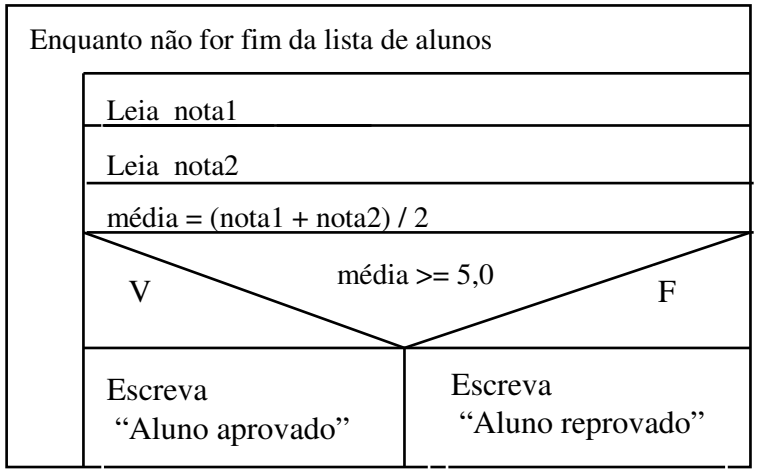
Repetição (teste inicial)



Repetição (teste final)



Exemplo:

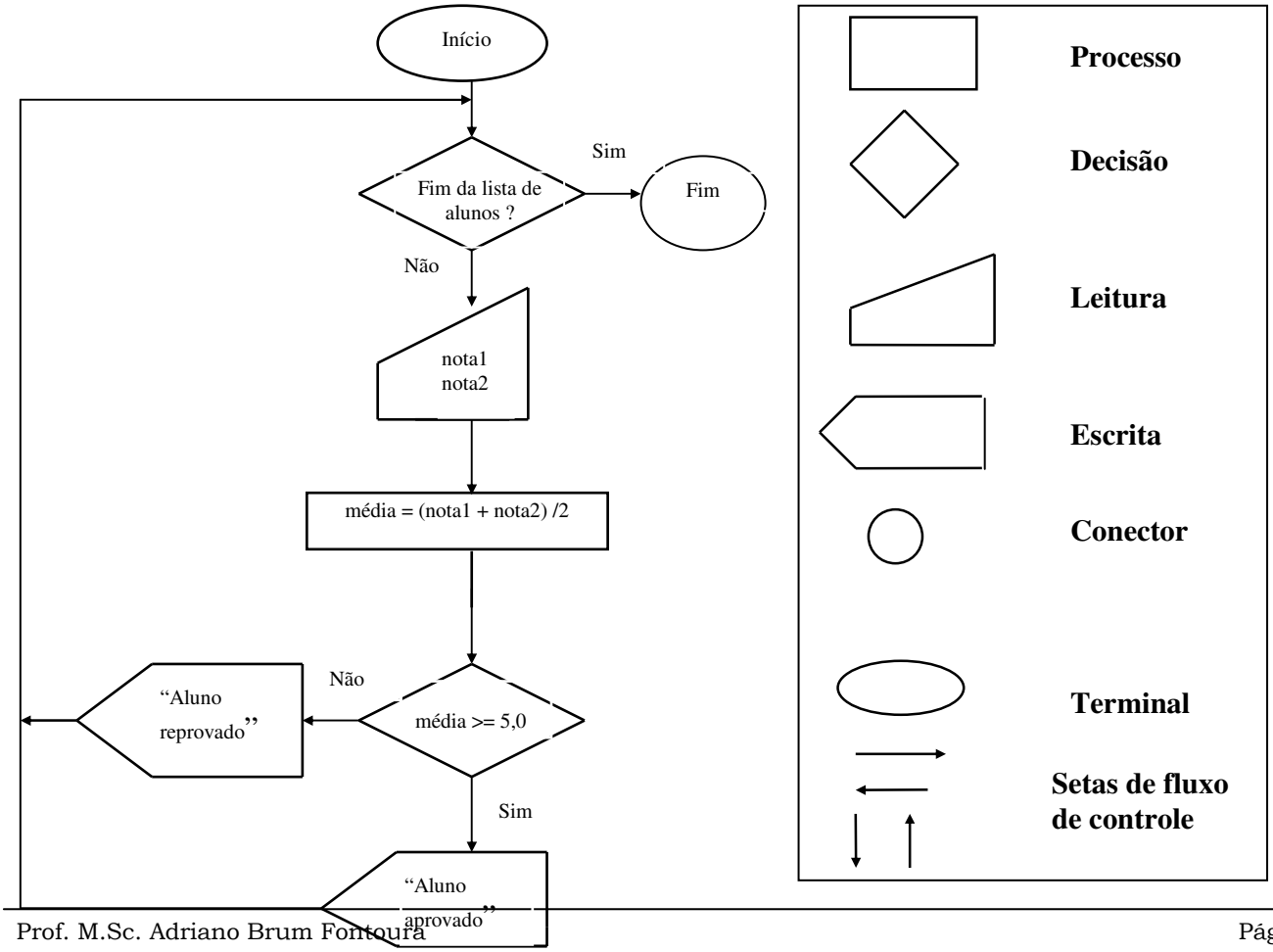


Embora os diagramas N-S ofereçam uma representação muito clara do algoritmo, à medida que os algoritmos vão se tornando mais complexos, fica difícil realizar os desenhos necessários numa única página, prejudicando a sua visualização.

Fluxograma

O fluxograma foi utilizado por muito tempo para a representação de algoritmos. No entanto, o seu grande problema é permitir o desenvolvimento de algoritmos não estruturados. Com o advento das linguagens de programação estruturada o fluxograma caiu em desuso. O fluxograma utiliza-se de símbolos específicos para a representação de algoritmos. Existe uma certa variação na simbologia empregada, apresentamos a seguir uma simbologia tradicionalmente usada:

Exemplo de Representação com Fluxograma:



Desenvolvimento top-down

Para resolver um diagrama correto, devemos considerar como procedimentos prioritários os itens seguintes:

I. Os diagramas devem ser feitos e quebrados em vários níveis. Os primeiros devem conter apenas as idéias gerais, deixando para as etapas posteriores os detalhamentos necessários;

II. Para o desenvolvimento correto de um diagrama de bloco, sempre que possível, deve ser desenvolvido de cima para baixo e da esquerda para direita;

III. È incorreto e "proibido" ocorrer cruzamento das linhas de fluxo de dados.

A seguir será abordada a segunda etapa que apresenta o detalhamento no que se refere à entrada e saída de dados do diagrama de bloco da Figura1.

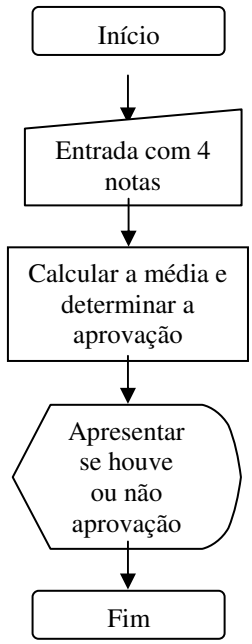


Figura 4 – Diagrama c/ entrada de notas e saída se aprovação

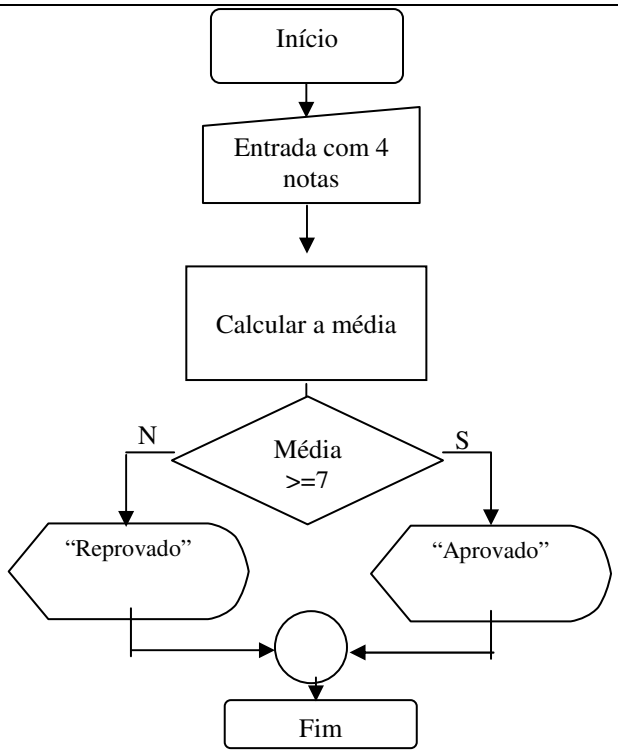


Figura 5 – Uso de uma condição em diagrama de bloco

Particularidade entre lógica

As representações gráficas de um diagrama de blocos podem ser feitas de várias maneiras e possuírem estruturas diferenciadas.

Lineares

A técnica lógica linear é conhecida como um modelo tradicional de desenvolvimento e resolução de um problema. Não está ligada a regra de hierarquia ou de estruturas de linguagens especificas de programação de computadores. Devemos entender que este tipo de procedimento está voltado à técnica matemática, a qual permite determinar a atribuição de recursos limitados, utilizando uma coleção de

elementos organizados ou ordenados por uma só propriedade, de tal forma que cada um deles seja executado passo a passo de cima para baixo, em que tenha um só “predecessor” e um só “sucessor”. Afigura 2.6 apresenta um exemplo deste tipo de lógica.

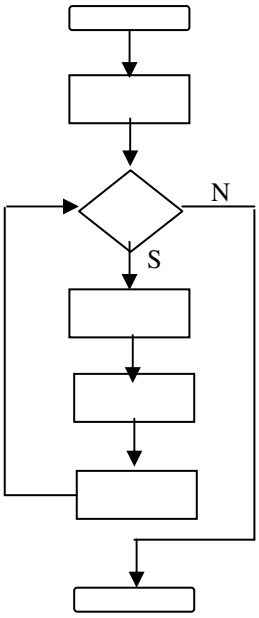


Figura 6: Exemplo de lógica linear

Estruturada

A técnica da lógica estruturada é a mais usada pelos profissionais de processamento eletrônico de dados. Possui características e padrões particulares, os quais diferem dos modelos das linguagens elaboradas por seus fabricantes. Tem como pontos fortes para elaboração futura de um programa, produzi-lo com alta qualidade e baixo custo.

A seqüência, a seleção e a interação são as três estruturas básicas para a construção do diagrama de blocos. A figura 2.7 seguinte apresenta um exemplo do tipo de lógica estruturada.

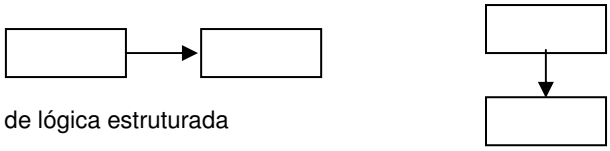


Figura 7: Exemplo de lógica estruturada

Modular

A técnica de a lógica modular deve ser elaborada como uma estrutura de partes independentes, denominada de módulos, cujo procedimento é controlado por um conjunto de regras. Segundo James Martin, suas metas são as seguintes:

- Decompor um diagrama em partes independentes;
- Dividir um problema complexo em problemas menores ou mais simples
- Verificar a correção de um modulo de blocos, independentemente de sua utilização como uma unidade de um processo maior.

A modularização deve ser desenvolvida, se possível, em diferentes níveis. Poderá ser utilizada para separar um problema em sistemas, um sistema em programas e um programa em módulos.

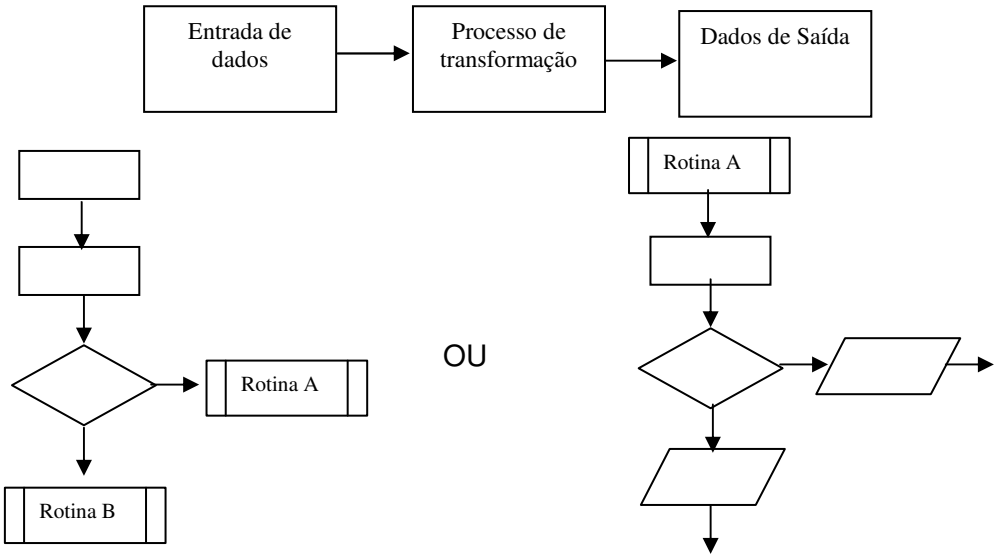


Figura 8: Exemplos de lógica modular

Português estruturado

Como foi visto até agora, o diagrama de blocos é a forma mais utilizada de notação gráfica, mas existe outra, que é uma técnica narrativa denominada pseudocódigo, também conhecida como português estruturado ou chamada por alguns de portugol.

Está técnica de algoritmização é baseada em uma PDL – *Program Design Language* (Linguagem de Projeto de Programação). A forma original de escrita é conhecida como inglês estruturado, muito parecida com a notação da linguagem PASCAL. A PDL (neste caso, o português estruturado) é usada como referência genética para uma linguagem de projeto de programação, tendo como finalidade mostrar uma notação para elaboração de algoritmos, os quais serão utilizados na definição, criação e desenvolvimento de uma linguagem computacional (Clipper, C, Pascal, Delphi, Visual-Objects) e sua documentação. Abaixo, é apresentado um exemplo deste tipo de algoritmo.

Estrutura Básica de um Algoritmo

PDL – *Program Design Language* ou Português Estruturado (Portugol)

Problema: Elabore um algoritmo que resolva o calculo da média, segundo a formula Media é igual a som nota1 e nota2 dividido por dois.

```
algoritmo "segundo"
var
  N1,N2, soma, media: real
Inicio
          escreva("Informe a Nota 1:") {comando que exibe uma mensagem da tela}
    leia(n1)        {Comando que realiza a leitura das variáveis, entrada de dados}
    escreva("Informe a Nota 2:")
          leia(n2)
        soma := n1 + n2
        media := soma/2
          Escreva(media)    {Comando que  mostrado a resultado final, saída de dados}
Fimalgoritmo
```

Linguagem C

```
#include <stdio.h>
int main()
{
  float n1,n2,soma,media;      // Declara as variáveis
  printf(" Informe Nota 1 ");  // Exibe uma mensagem na tela
  scanf("%f",&n1);             // Lê um valor para uma variável
  printf(" Informe Nota 2 ");
  scanf("%f",&n2);
```

```
  soma=n1+n2;                  //Formula da soma
  media=soma/2;                //Formula da média
  printf(" Media = %f \n",media);
  system("pause");             // Aguarda o usuário teclar algo para sair do programa
  return(0);
}
```

A diferença entre uma linguagem de programação de alto nível utilizada em computação e uma em PDL é que esta (seja escrita em português, inglês ou qualquer outro idioma) não pode ser compilada em um computador (por enquanto).Porém existem “Processadores de PDL“ que possibilitam traduzir essa linguagem numa representação gráfica de projeto.

Constantes

Entende-se por uma informação constante, aquela que não sofre nenhuma alteração no decorrer do tempo, ou seja, da execução do algoritmo. Por exemplo, na fórmula a seguir.

```
Algoritmo “ Teste"
Cont    {constante}
    Pi:= 3,14159
Inicio
<comandos>
Fim
```

Variáveis

Tem-se como definição de variáveis, tudo aquilo que é sujeito a variações. Variáveis são espaços de memória que são alocados para armazenar informações. Por que se precisa de variáveis?

R. Para armazenar valores que serão utilizados posteriormente. Ex: em um calculo, onde resultados intermediários podem ser armazenados e posteriormente processados para se obter o resultado final.

Imagine que a memória de um computador é um grande armário com diversas gavetas, onde cada gaveta pode conter somente um valor (numérico, caractere ou lógico). Deve-se concordar que é necessário identificar cada gaveta, para sua posterior utilização, através do nome.

$área = \pi . raio^2$ **Área da Circunferência**	$are = \frac{base.altura}{2}$ **Área de um triângulo**
PDL – *Program Design Language* ou Português Estruturado (Portugol)	

lgoritmo "terceiro"
// Função : circunferencia;
// Autor : Adriano
// Data : 12/2/2008
// Seção de Declarações
var
P, area, raio: real
inicio
P <- 3.14159 //Constante
Leia(raio)
area := pi * (raio * raio)
Escreval(area)
fimalgoritmo
	algoritmo "Quarto"
	// Função : triangulo;
	// Autor : Adriano
	// Data : 12/2/2008
	// Seção de Declarações
	Var
	base, alt, area :real
	Inicio
	escreva("Informe a base ")
	leia (base)
	escreva("Informe a altura ")
	leia (alt)
	area <-(base * alt) /2 //constante
	escreva("Area do triangulo = ",area)

	fimalgoritmo
Linguagem Pascal	**Lingagem C**

Program circunf;
Cont
Pi:= 3,14159;
Var
area, raio: real; **constante**
Begin
Write('Informe o raio');
Raed(raio);
area := pi * (raio * raio);
Write('Area = ',area);
End.
	#include <stdio.h>
	#define pi 3.14159
	int main() Constante
	{
	float area,raio;
	printf(" Informe o raio ");
	scanf("%f",&raio);
	area=pi*(raio*raio);
	printf(" Area = %f \n",area);
	system("pause");
	return(0);
	}

Tipos Primitivos de Dados

Podemos classificar os tipo de dados a serem processados, de forma resumida, como dados e instruções.

Inteiro

Toda e qualquer informação numérica que pertença ao conjunto dos números inteiros (negativa nula ou positiva). Exs: 39, 0, -56, entre outros.
a)Ele tem 15 irmãos. b)A temperatura desta noite será de –2 graus

Real

Toda e qualquer informação numérica que pertença ao conjunto dos números reais (negativa, nula ou positiva, inteiro ou fracionário). Exs:-4, 3, 0, 35, 1,23
a)Ela tem 1,73 metro de altura b) Meu saldo bancário é de - R$ 121,07.

Caractere

São caracterizadas como tipo caracteres, as seqüências contendo letras, números e símbolos especiais. Uma seqüência de caracteres deve ser indicada entre aspas(‘’ ’’). Este tipo de dado é também conhecido como alfanumérico, string, literal ou texto. Exs: ”Rua Alfa,52 Apto1”, “Fone: 574-9988”
a)Constava na prova: “Use somente caneta!”.

Lógico

São característicos tipos lógicos, os dados com valores verdadeiro e falso, sendo que este tipo de dado poderá representar apenas um dos dois valores. Ele é chamado por alguns de tipo boleano, devido a contribuição do filósofo e matemático inglês George Boole na área da lógica matemática. Os dados do tipo lógico, poderão ser apresentados e delineados pelo caractere ponto(.) ou não. Exs.**Falso.**, **FALSO** ou **.f.** (para o valor lógico :falso) e **.Verdadeiro. VERDADEIRO** ou.**v.** (para valor lógico : verdadeiro).
a) A porta pode estar *aberta* ou *fechada*

Tipo	Exemplo	Basic(VB)	**Pascal**/Delphi	Java/C++
Inteiro	2; 45; 100	Integer	Integer	Int
Real	2,456; 101,19	Single	Real	Float
Caractere/String	“A”; “4”; “Oi”	Byte / String	Char / String	Char / String
Lógico	Verdadeiro; Falso;	True / False	True / False	True / False

Nomenclatura e declaração de variáveis

O **nome** de uma variável faz referência ao endereço de memória onde essa variável se encontra. Existem algumas regras para utilização das variáveis:

- Nome de variável pode ter 1 o u mais caracteres
- O primeiro caractere de uma variável **sempre** deve ser uma letra
- Nenhuma variável pode ter espaços em branco em seu nome
- Nenhum nome de variável pode ser uma palavra reservada a uma instrução da linguagem usada
- Poderão ser utilizadas somente letras, letras e números
- Procure sempre utilizar variáveis que tenham sentido (Mnemônico). Ex: “Alt_homem” ao invés de “x”

Linguagens	**Variáveis X**	**Variáveis Y**
Visual Basic / Basic	Dim X as integer	Dim Y as single
Pascal / Delphi	X: integer;	Y:real;
Java/C# / C++	Int X;	Float Y;

Atribuição de vaiáveis

O comando de (=, :=, ou $\leftarrow$) permite fornecer um valor a uma certa variável, onde o tipo de informação deve ser compatível com o tipo de variável utilizada, ou seja, somente podemos atribuir “Pedro” a uma variável do tipo caractere. Exs: a:=”mesa”, b:=2+5-X, c:= -5,4-b

Quando uma variável é declarada (criada) qual o seu valor inicial?
R: não se pode saber o valor inicial da variável, pois na memória existem várias informações armazenadas, muitas podem estar sendo utilizadas pelo computador, mas podem existir espaços em que foram armazenadas informações anteriores, mas que não estão mais em uso. Esses espaços, mesmo com informações uma nova variável pode ser criada, e seu valor será um “lixo” qualquer da memória. Mas uma pode ser criada em um espaço vazio da memória, nesse caso seu valor será nulo. Para resolver esse problema, o valor inicial da variável, algumas linguagens inicializam suas variáveis com 0 (zero) ou nulo.
Suponha que fossem atribuídos os seguintes valores às seguintes variáveis:
A:= “mesa”, B:=0, C:=2, D:=-5.4, E:=”João”, E:=5.656

Veja abaixo como poderia ficar a memória do computador:

1	2	mesa	0
C:\		125	
BA		TXT	
-5.4	XYz		2
João	30		5.656

Utilizado	Não vazio, não em uso (lixo)	Vazio, não utilizado

Expressões matemáticas ou fórmulas matemáticas

Uma expressão matemática apresentada como: X = {43 . [55 : (30+2)]} nos algoritmos deve ser apresentada como: x := (43 * (55 / (30+2)))

Uma fórmula que matematicamente é representada como: $are = \frac{base.altura}{2}$ nos algoritmos deve ser apresentada como: área := (base * alt) / 2

Instruções Básicas ou Comandos Básicos

A partir deste momento, vamos aprender os conceitos e comandos básicos da linguagem de programação C, que será utilizada nesta disciplina com a finalidade de desenvolver a lógica de programação.
Dois pontos importantes são, sabermos a sintaxe dos comandos e entender a semântica dos mesmos, pois isto tornará o processo de aprendizagem mais fácil e rápido

Formato do Programa

Vejamos um primeiro programa em C:

```
main ()
{
  printf ("Ola! Eu estou vivo!\n");
}
```

O programa inicia com a função main, e o corpo do programa fica entre chaves { }. Compilando e execu este programa você verá que ele coloca a mensagem *Ola! Eu estou vivo!* na tela.

Comandos de entrada e saída

Todo o programa de computador salvo algumas exceções, consistem de três etapas ou pontos de trabalho: **Entrada** de dados, **Processamento** e **Saída** dos dados.

A entrada de dados e representada pela instrução: **scanf()**

scanf (string-de-controle,lista-de-argumentos);
scanf("%d",idade)
scanf(<parametro>,<parametro>)

A saída de dados e representada pela instrução: **printf()**

printf (string_de_controle,lista_de_argumentos);
printf("%d",A);
printf**(**<'mensagem'>**,<**variável>**)**;
printf ("O valor resultante é: %f",B);

Exemplo

```
1.  #include  <stdio.h>
2.  void main()
3.  {
4.    int x;
5.    scanf("%d",&x);
6.    printf("%d",x);
7.  }
```

Linha 1: diz ao compilador que ele deve incluir o arquivo-cabeçalho **stdio.h**, pois neste arquivo existem declarações de funções úteis para entrada e saída de dados.

Linha 4: declara uma variavel para armazenar a informação digitado pelo usuário.

Linha 5: Ler do teclado o valor informado pelo usuário.

Linha 6: Coloca o valor da variável x na tela do computador.

Linhas 3 e 7: Abrem e fecham o corpo do programa.

A entrada de dados e representada pela instrução (lendo string): **gets()**

gets(<variável>);
gets(nome);

Pede ao usuário que entre uma string, que será armazenada na string nome.

Exemplo

```
1.  #include <stdio.h>
2.  void main()
3.  {
4.      char buffer[10];
5.     printf("Entre com o seu nome");
6.     gets(buffer);
7.     printf("O nome é: %s", buffer);
8.     system("pause");
9.  }
```

Linha 4: Declara uma variável do tipo string

Linha 6: Lê uma variável do tipo string

Linha 7: Exibe a variável do tipo string, utilizando a formatação %s para exibir strings

Linha 8: O comendo system("pause") aguarda até que se tecle algo para continuar/finalizar o programa

Operadores aritméticos

São classificados em duas categorias:

- Unários: é quando atua na inversão de um valor, atribuindo o sinal positivo ou negativo.
- Binário: Quando atua em operações de expomemciação, multiplicação, adição, subtração.

Na resolução das expressões aritméticas, as operações têm uma hierarquia entre si. Para operações de mesma prioridade, seguimos a ordem especificada, isto é, primeiro resolvemos os operadores mais à esquerda e, depois, os mais à direita da expressão. Para alterar a prioridade da tabela, utilizamos parênteses mais internos.

Operador	Operação	Tipo	Prioridade de operação
+	Manutenção do sinal	Unário	1º
-	Inversão de sinal	Unário	1º
**	Exponenciação	Binário	2º
/	Divisão	Binário	3º
*	Multiplicação	Binário	3º
+	Adição	Binário	4º
-	Subtração	Binário	4º

Expressões lógicas

Operadores relacionais

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

= - igual	<> - diferente
< - menor	> - maior
<=- menor ou igual	>= - maior ou igual

EXS: AS VARIÁVEIS A E B TEM OS SEGUINTES VALORES: A:=5, B := 8

Teste	**Resposta**	**Teste**	**Resposta**
A > B	(falso)	A < B	(verdadeiro)
A <= B	(verdadeiro)	A = B	(falso)
A <> 15	(verdadeiro)	B = 2	(falso)

Operadores lógicos

Os operadores lógicos permitem trabalhar com o relacionamento de duas ou mais condições, ao mesmo tempo, ou seja, efetuando testes múltiplos. Neste caso se torna necessário trabalhar com operadores lógicos ou boleanos. Os lógicos são três: **e, ou, não**

Operado e (and): Também chamado de conjunção. Neste caso, todas as expressões condicionais componentes de uma conjunção, ou seja, de um conjunto de testes, devem ser verdadeiras para que a expressão resultante tenha valor verdadeiro.

Condição I	**Condição II**	**Resultado**
Verdadeira	Verdadeira	V
Verdadeira	Falsa	F
Falsa	Verdadeira	F
Falsa	Falsa	F

OPERADOR OU (OR): TAMBÉM CHAMADO DE DISJUNÇÃO. NESTE CASO QUANDO QUALQUER UMA DAS EXPRESSÕES COMPONENTES DA DISJUNÇÃO FOR VERDADEIRA, A EXPRESSÃO RESULTANTE TERÁ VALOR VERDADEIRO.

Condição I	**Condição II**	**Resultado**
Verdadeira	Verdadeira	V
Verdadeira	Falsa	V
Falsa	Verdadeira	V
Falsa	Falsa	F

Operador não (not): Também chamada de negação. Neste caso, a negação apenas inverte o valor verdadeiro da expressão. Se X for falso, não X é verdadeiro.

Condição I	**Resultado**
Verdadeira	F
Falsa	V

Exemplos:

a) 3 > 6 ou 4 < 5 F ou V Resultado: V	b) 4 < 7 e 5 > 9 V e F Resultado: F

Estruturas de Controle

Uma estrutura de controle permite a escolha de um ou um grupo de ações a ser executado quando determinada condição for ou não satisfeita. Essas condições são representadas por expressões lógicas ou relacionais.

Desvio condicional

Simples (if...)

Quando necessitamos testar uma condição antes de executar uma ação, utilizamos um desvio condicional simples. Veja o modelo:

se <expressão-lógica> entao <seqüência-de-comandos> fimse	if (condição) declaração;

```
#include <stdio.h>
int main ()
{
    int num;
    printf ("Digite um numero: ");
    scanf ("%d",&num);
    if (num>10)
            printf ("\n\nO numero e maior que 10");
            if (num==10)
           {
              printf ("\n\nVoce acertou!\n");
              printf ("O numero e igual a 10.");
           }
            if (num<10)
              printf ("\n\nO numero e menor que 10");
     system("pause");
     return(0);
}
```

Composto (if ... else...)

O desvio condicional Se quando o seu resultado do teste for VERDADEIRO, todos os comandos da *<seqüência-de-comandos-1>* são executados. Se o resultado for FALSO, os comando da *<seqüência-de-comandos-2> é que serão executados.*. Veja o modelo:

se <expressão-lógica> entao <seqüência-de-comandos-1> senao <seqüência-de-comandos-2> fimse	*if (condição)* * declaração_1;* *else* * declaração_2;*

Exercício – Faça o algoritmo	#include <stdio.h>
	int main ()
	{
	int num;
	printf ("Digite um numero: ");
	scanf ("%d",&num);
	if (num==10)
	{
	printf ("\n\nVoce acertou!\n");
	printf ("O numero e igual a 10.\n");
	}
	else
	{
	printf ("\n\nVoce errou!\n");
	printf ("O numero e diferente de 10.\n");
	}
	return(0);
	}

Composto encadeados (if ...else...)

Existem situações em que é necessário estabelecer verificações sucessivas. Quando uma ação é executada, ela poderá ainda estabelecer novas condições, isso significa condições dentro de condições. Esse tipo de estrutura poderá ter diversos níveis de condição, sendo chamados de aninhamento ou encadeamento. O encadeamento pode ser tanto para uma condição verdadeira como falsa.

Exemplo:

```
#include <stdio.h>
void main ()
{
      int num;
      printf ("Digite um numero: ");
      scanf ("%d",&num);
     if (num>10)
            printf ("\n\n O numero e maior que 10");
     else
       if (num==10)
      {
            printf ("\n\n Voce acertou!\n");
            printf ("O numero e igual a 10.");
      }
      else
        if (num<10)
          printf ("\n\n O numero e menor que 10");
     system("pause");
}
```

Múltiplas opções

Outro comumente utilizado, além do se... então senão fim_se, quando há mais de duas situações a serem testadas, é o comando **switch**.

escolha <expressão-de-seleção> caso <exp11>, <exp12>, ..., <exp1n> <seqüência-de-comandos-1> caso <exp21>, <exp22>, ..., <exp2n> <seqüência-de-comandos-2> ... outrocaso <seqüência-de-comandos-extra> fimescolha	*switch (variável)* *{* *case constante_1:* *declaração_1;* *break;* *case constante_2:* *declaração_2;* *break;* *.* *.* *.* *case constante_n:*

	declaração_n; *break;* *default* *declaração_default;* *}*

```
#include <stdio.h>
Void main ()
{
      int num;
      printf ("Digite um numero: ");
      scanf ("%d",&num);
      switch (num)
       {
              case 9:
                      printf ("\n\nO numero e igual a 9.\n");
                       break;
              case 10:
                      printf ("\n\nO numero e igual a 10.\n");
                       break;
              case 11:
                      printf ("\n\nO numero e igual a 11.\n");
                       break;
              default:
                      printf ("\n\nO numero nao e nem 9 nem 10 nem 11.\n");
      }
  }
```

Laços ou malhas de repetição

Quando uma seqüência de comandos deve ser executada repetidas vezes, tem-se uma estrutura de repetição. Esta estrutura, assim como a de decisão, envolve sempre a avaliação de uma condição.

Repetição determinada (for...)

Na repetição determinada o algoritmo apresenta previamente a quantidade de repetições.

for (inicialização; condição; incremento)
declaração;

```
1.  #include <stdio.h>
2.  void main ()
3.  {
4.      int count;
5.      for (count=1; count<=100; count++)
6.          printf ("%d ",count);
7.      system("cause");
8.  }
```

Linha 5: O comando for é composto por três argumentos sendo eles: *for (inicialização; condição; incremento*

Inicialização: Inicializa a variável, ou seja, é o valor ao qual iniciará o incrementou ou decremento.

Condição: é onde se realiza o teste para finalizar o looping

Incremento/Decremento: é onde soma-se um valor ao contador ou subtrai-se no caso do decremento

OBS: A repetição por padrão determina o passo do valor inicial até o valor final como sendo 1. Determinadas linguagens possuem passo –1 ou permitem que o programador defina o passo.

Repetição Indeterminada c/ validação inicial (while...)

É usada para repetir N vezes uma ou mais instruções. Tendo como vantagem o fato de não ser necessário o conhecimento prévio do número de repetições.

enquanto *<expressão-lógica>* faca *<seqüência-de-comandos>* fimenquanto	*while (condição) declaração;*

```
#include <stdio.h>
void main ()
{
   int i = 0;
   while ( i < 100)      // Inicia o laço com um teste inicial I tem que ser menor que 100
   {
        printf(" %d", i);
        i++;     // Incrementa (soma) 1 a variável i
   }
   system(“pause”);
}
```

Repetição Indeterminada c/ validação final (do ...while)

Assim como a estrutura ENQUANTO É usada para repetir N vezes uma ou mais instruções. Sua validação é final fazendo com que a repetição seja executada pelo menos uma vez.

repita *<seqüência-de-comandos>* ate *<expressão-lógica>*	*do* *{* *declaração;* *} while (condição);*

```
#include <stdio.h>
void main(){
  int i=0;
  do{
     printf("\n\n Escolha sua fruta pelo numero\n\n "
            "\t (1)...Mamao\n"
            "\t (2)...Abacaxi\n"
            "\t (4)...Sair\n");
     scanf("%d",&i);
     switch(i){
             case 1:
              printf("\t\t Voce escolheu Mamao \n ");
              break;
             case 2:
               printf("\t\t Voce escolheu Abacaxi \n ");
               break;
             case 4:
               printf("\t\t Saida! \n ");
               return(0);
             default:
               printf("\t\t Opcao invalida \n ");
               break;
     }
     getch();
     system("cls");
   }while((i>=1)||(i<=4));
    printf(" \n F I M  \n");
 getch();
}
```

O Comando break

Nós já vimos dois usos para o comando break: interrompendo os comandos **switch** e **for**. Na verdade, estes são os dois usos do comando break: ele pode quebrar a execução de um comando (como no caso do switch) ou interromper a execução de *qualquer* loop (como no caso do for, do while ou do do while). O break faz com que a execução do programa continue na primeira linha seguinte ao loop ou bloco que está sendo interrompido.

Contadores e Acumuladores

Contadores

È uma variável de controle, inteira, que serve para controlar quantas vezes um determinado trecho do algoritmo (programa) foi executado. Por exemplo: um programa que leia 100 valores, podendo eles serem somente negativos ou positivos (desconsiderando os valores nulos). A seguir, considere que o programa deva mostrar a quantidade de valores positivos digitados. Nesse caso, devemos fazer um teste a cada leitura, e no caso do valor ser positivo, adicionar +1 para a variável contador (contp := contp + 1).

Acumuladores

È uma variável de controle, inteira, que serve para acumular valores. Considere que um programa, além de ler 100 valores e mostrar a quantidade de números positivos, deva mostra o somatório destes valores.

Estruturas de Dados

Temos os seguintes tipos de dados básicos: numérico (inteiros, reais), lógico e caractere. Baseando-se nestes tipos básicos podemos construir tipos compostos. Os tipos compostos dividem-se em: **Homogêneo** (vetores e matrizes) e **Heterogêneos** (registros)

Tipos de dados homogêneos

A utilização deste tipo de estrutura de dados recebe diversos nomes, como: variáveis indexadas, variáveis compostas, variáveis subscritas, arranjos, vetores, matrizes, tabelas de memória ou array (inglês). Para simplificar chamaremos de vetores e matrizes. Vetores ou Matrizes só permitem trabalhar com um único tipo de dado, a única forma de trabalhamos com dados de tipos diferentes é trabalharmos com duas matrizes ou dois vetores.

- Vetores: são as variáveis compostas unidimensionais.
- Matrizes: São as variáveis compostas multidimensionais.

Vetores

Para definirmos o tipo do vetor seguimos a seguinte regra sintática:

tipo_da_variável nome_da_variável [tamanho];

Exs: Um vetor de 8 posições reais poderia ser declarado da seguinte forma:
Int notas: vetor[8] ;

Com o conhecimento adquirido até o presente momento, poderemos elaborar um programa que:

- Declare um vetor
- Exiba o vetor.
- Leia os dados e armazena nas respectivas posições do vetor
- Exiba os dados armazenados no vetor.

Como seria este algoritmo utilizando vetor em sua construção:

```
#INCLUDE <STDIO.H>

VOID MAIN(VOID){
  INT VETOR[3] ={0,0,0}; // DECLARA UM VETOR DE TAMANHO 3
  INT CONTADOR;
 // EXIBE O VETOR SEM DADOS
 //COMO O COMANDO FOR (INCIA DA POSIÇÃO ZERO, TESTA SE O CONTADOR TEM VALOR MENOR QUE 3,
INCREMENTA O CONTADO)
 FOR(CONTADOR=0; CONTADOR<3; CONTADOR++)
   PRINTF("\N EXIBE O POSICAO VETOR[%D] DADO INSERIDO: %D ",CONTADOR,VETOR[CONTADOR]);

 // INSER DADOS EM UM VETOR
 FOR(CONTADOR=0; CONTADOR<3; CONTADOR++){
    PRINTF("\NINFORME UM VALOR INTEIRO NA POSICAO %D: ",CONTADOR);
    SCANF("%D",&VETOR[CONTADOR]);
 }
 // EXIBE O VETOR COM DADOS INSERIDOS
 FOR(CONTADOR=0; CONTADOR<3; CONTADOR++)
    PRINTF("\N EXIBE O POSICAO VETOR[%D] DADO INSERIDO: %D ",CONTADOR,VETOR[CONTADOR]);

 GETCH();
}
```

Matrizes

Para definirmos o tipo da matriz seguimos a regra sintática do vetor, com um alteração que a de definir as dimensões da matriz. A matriz mais comum é a de duas dimensões, composta por linhas e colunas, mas vale ressaltar que em algum momento pode ser necessário trabalhar com matrizes de 3 ou mais dimensões. Na definição, também é necessário informar o número de linhas e o número de colunas que a matriz terá.
tipo_da_variável nome_da_variável Linhas][columas];

Vejamos como acessar (preencher) os valores na matriz, ou seja, o processo de armazenamento das informações.

```
#include <stdio.h>
#define NLIN  2    //número de linhas
#define NCOL 4   // número de colunas
void main(){
 int Matriz[NLIN][NCOL];  // declara a matriz de inteiro
 int i,j;

 // Leitura dos dados
 for(i=0; i < NLIN; i++)
   for(j=0; j < NCOL; j++)
        Matriz[i][j] = i;

// mostra os dados da matriz
for(i=0; i < NLIN; i++){
   printf("\n");
   for(j=0; j < NCOL; j++) {
       if (i==j)
         printf(" %d, ",Matriz[i][j]);
    }
}
getch();
}
```

Tipos de dados heterogêneos

Registros

Esta estrutura de dados, o registro, a qual consiste em trabalhar vários tipos de dados (campos) em uma mesma estrutura. Por esta razão, este tipo de dado é considerado heterogêneo. A seguir vejamos um exemplo do layout de um registro com suas informações, as quais recebem o nome de campos.

Cadastro Escolares
Nome.....:
Primeira Nota:
Segunda Nota:
Terceira Nota:

O registro está formado pelos campos: Nome, Pnota, Snota, Tnota e pode ser chamado de aluno.

Declaração de um registro em C

struct aluno { Char nome[20]; float nota1; float nota2; float nota2; }cadastro;	*struct nome_do_tipo_da_estrutura* *{* *tipo_1 nome_1;* *tipo_2 nome_2;* *...* *tipo_n nome_n;* *} variáveis_estrutura;*

```
#include <stdio.h>

#include <string.h>

// Cria a estrutura

struct ficha_pessoal {

      char nome [50];

      long int telefone;

      char rua [50];

      int numero;

      };

main (void){

struct ficha_pessoal ficha; // declara a estrutura

system("cls");
```

```
printf("\n Informe o Nome:" );

gets(ficha.nome);       fflush(stdin); // acessa para entrada de dados

printf("\n Infome o Telefone:");

scanf("%d",&ficha.telefone);

printf("\n Infome a Rua:");

fflush(stdin);   gets(ficha.rua);          fflush(stdin);

printf("\n\n EXIBE DADOS DIGITADOS \n\n");

printf("\n Nome....: %s \n",ficha.nome);  // exibe os dados

printf("\n Telefone: %d \n",ficha.telefone);

system("pause");
return 0;
}
```

Sub-Rotinas (modularização)

No geral, problemas complexos exigem algoritmos complexos. Mas sempre que é possível dividir um problema grande em pequenos. Desta forma, a cada parte teremos um algoritmo mais simples e este pequeno algoritmo chamamos de sub-rotina.

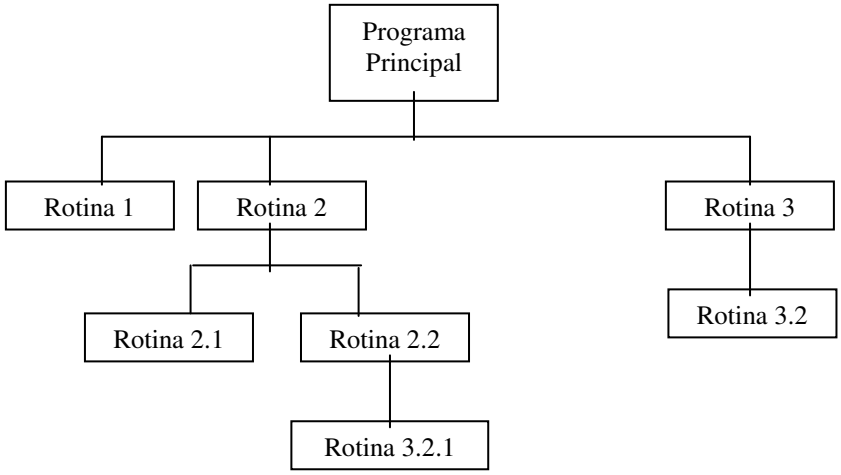
A sub-rotinas, além de facilitar a compreensão do algoritmo, também possibilita que estes trechos possam ser utilizados em outros algoritmos. Isso facilita a manutenção, pois se uma sub-rotina para verificar se um número é primo ou não esta incorreta, sabemos onde verificar este erro.

Quando as sub-rotinas ficam demasiadamente complexas, podemos dividi-las novamente. O processo de divisão de sub-rotinas em outras, é chamado de refinamento sucessivo.

O processo de programação torna-se bastante simples quando aplicamos algum método para a utilização de sub-rotina. Um dos mais usados na programação estruturada é Top-Down, o qual caracteriza-se basicamente por:

- Antes de iniciar o programa é desejável se ter em mente as tarefas principais que este deverá executar. Não é necessário saber como funcionarão, somente saber quais são.
- Deve-se ter em mente como será o funcionamento do programa principal, o que controlará toda as tarefas das sub-rotinas.
- Tendo o programa principal definido, detalha-se as sub-rotinas. São definidos vários algoritmos, um para cada rotina, para que se tenha visão do que será executado em cada módulo do programa. Isso também facilita a manutenção.

O método Top-Down pode ser graficamente visualizado abixo:



Procedimentos

Procedimentos, na realidade são trechos de programas independentes que, para serem processados devem ser referenciado (chamados) pelo programa principal. Quando uma rotina é chamada, ela é executada e ao seu término o controle do processamento retorna automaticamente a primeira linha de instrução após a linha que efetuou o chamado.

Exemplo:

Introdução às Funções/procedimentos

Uma função é um bloco de código de programa que pode ser usado diversas vezes em sua execução. O funções permite que o programa fique mais legível, mais bem estruturado. Um programa em C consiste, no fur várias funções colocadas juntas.

Abaixo o tipo mais simples de função:

```
#include <stdio.h>
void mensagem ()  /* Funcao simples: so imprime Ola! */
{
        printf ("Ola! ");
}
void main ()
{
        mensagem();
        printf ("Eu estou vivo!\n");
}
```

Variáveis globais e locais

- Uma variável é considerada **Global**, quando é declarada no início do algoritmo principal, podendo ser utilizada por qualquer sub-rotina subordinada a este algoritmo principal.
- A variável é considerada **Local**, quando é declarada no início dentro de uma sub-rotina é somente válida dentro da rotina à qual está declarada. Desta forma, as demais sub-rotinas ou até mesmo o algoritmo principal não poderão fazer use desta variável.

Parâmetros formais e reais

Os parâmetros têm por finalidade de servir como um ponto de comunicação bidirecional entre uma sub-rotina e o algoritmo principal ou uma outra sub-rotina hierarquicamente de nível mais alto. Estes parâmetros podem ser formais e reais.

Serão considerados **Parâmetros Formais** quando forem declarados por meio de variáveis juntamente com a identificação do nome da sub-rotina, os quais serão tratados da mesma forma que as variáveis globais ou locais.

```
int square (int x) /* Calcula o quadrado de x */
{
  printf ("O quadrado e %d",(x*x));
        return(0);
}
```

Serão considerados **Parâmetros Reais** quando substituírem os parâmetros formais, quando na sua utilização de uma sub-rotina. Vejamos o exemplo abaixo.

```
...
int main ()
{
   int num;
   printf ("Entre com um numero: ");
   scanf ("%d",&num);
   printf ("\n\n");
   square(num);
          return(0);
}
```

Passagem de parâmetros

A passagem dos parâmetros e feita quando ocorre a troca de parâmetros formais para reais. Eles podem ser passados de duas formas: por valor ou por referência.

Por Valor, caracteriza-se pela ação de dar apenas entrada de um tipo de dado em uma determinada sub-rotina. Desta forma, qualquer modificação que ocorra na variável local existente dentro da sub-rotina não afetará o valor do parâmetro passado.

Exemplo:

```
#include <stdio.h>
int prod (int x,int y)
 {
     return (x*y);
  }

 void main ()
  {
          int saida;
       saida=prod (12,7);
       printf ("A saida e: %d\n",saida);
     }
```

Por Referência, caracteriza-se pela ação de dar entrada obter saída de um tipo de dado em uma sub-rotina. Desta forma, qualquer modificação que ocorra na variável local afetará o valor do parâmetro passado por referência, pois a alteração é devolvida para a rotina chamadora.

Exemplo:

```
#include <stdio.h>
int prod (int *x,int *y)
{
      return (x*y);
}

   int main ()
{
      int saída,A=12,B=7,*px,*py;
      px=&A;
      py=&B;
      saida=prod (&px,&py);
      printf ("A saida e: %d\n",saida);
                  return(0);
}
```

Usa ponteiros.