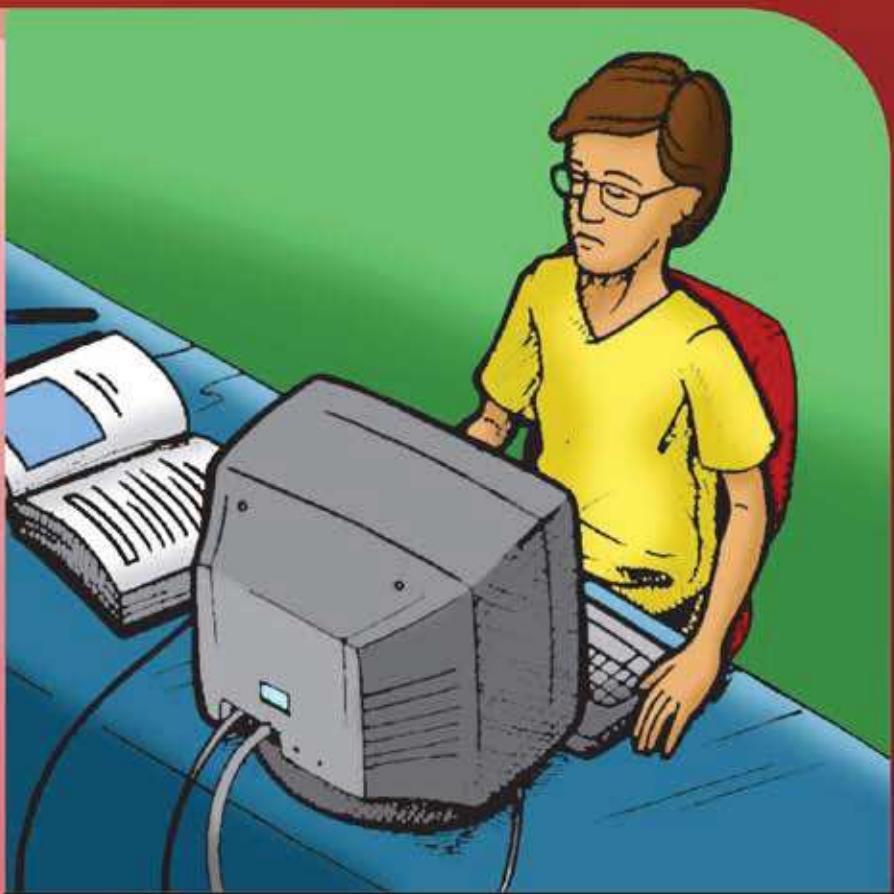


Introdução à Programação

Sandra de Albuquerque Siebra



FASCÍCULO 2



Universidade Federal Rural de Pernambuco

Reitor: Prof. Valmar Corrêa de Andrade

Vice-Reitor: Prof. Reginaldo Barros

Pró-Reitor de Administração: Prof. Francisco Fernando Ramos Carvalho

Pró-Reitor de Extensão: Prof. Paulo Donizeti Siepierski

Pró-Reitor de Pesquisa e Pós-Graduação: Prof. Fernando José Freire

Pró-Reitor de Planejamento: Prof. Rinaldo Luiz Caraciolo Ferreira

Pró-Reitora de Ensino de Graduação: Profª. Maria José de Sena

Coordenação de Ensino a Distância: Profª Marizete Silva Santos

Produção Gráfica e Editorial

Capa e Editoração: Allyson Vila Nova e Rafael Lira

Revisão Ortográfica: Ivanda Martins

Ilustrações: Allyson Vila Nova

Coordenação de Produção: Marizete Silva Santos



Sumário

Capítulo 1 - Conceitos Básicos sobre Lógica de Programação e Algoritmos..	6
Objetivos do Capítulo 1	6
Conteúdo Programático do Capítulo 1	6
Unidade 2 – Linguagem Algorítmica	8
E aí? Por onde começo?	11
Identificadores	14
Constantes, Variáveis e Tipos de Dados	15
CONSTANTES	15
VARIÁVEIS.....	16
TIPOS DE DADOS.....	17
DECLARAÇÃO DE VARIÁVEIS	18
Operadores e Expressões	20
Operador de atribuição.....	21
Operadores aritméticos e expressões aritméticas	22
Operadores relacionais	24
Instruções de Entrada e Saída	27
Sintaxe e Semântica.....	30
Estrutura Seqüencial	33

Estrutura Condicional	34
Estrutura de Repetição	40
Dicas para Escrever Bons Algoritmos	50

Capítulo 1 - Conceitos Básicos sobre Lógica de Programação e Algoritmos

Carga horária do capítulo: 30 horas

Objetivos do Capítulo 1

- Apresentar os conceitos básicos sobre algoritmos e lógica de programação.
- Apresentar formas distintas para representação do raciocínio da lógica algorítmica
- Apresentar os tipos de problema algorítmicos
- Apresentar componentes básicos de um programa em linguagem algorítmica (pseudocódigo).
- Desenvolver a lógica de programação fazendo uso, de forma ainda básica, de elementos, variáveis, operadores, expressões, estruturas seqüenciais, estruturas condicionais e estruturas de repetição.

Conteúdo Programático do Capítulo 1

Unidade 1 – Introdução a Algoritmos e Linguagens (10 horas)

- Noções de Lógica de Programação e Algoritmos
- Fases Básicas da Construção de Algoritmos.
- Representação de Algoritmos: fluxogramas, português estruturado e linguagem algorítmica (pseudocódigo)
- Conceitos Básicos de Lógica de Programação e Linguagens de Programação

Unidade 2 – Linguagem Algorítmica (20 horas)

- Identificadores
- Constantes, Variáveis e Tipos de Dados
- Operadores e Expressões (aritméticos, relacionais e lógicos)
- Instruções Primitivas (entrada e saída)
- Sintaxe e Semântica
- Estrutura seqüencial (início/fim)
- Estruturas de Decisão (simples, composta e múltipla)
- Estruturas de Repetição (condicional e com contador)
- Dicas para escrever bons algoritmos

Cada unidade conterá exercícios para fixação do assunto apresentado, assim como indicações de leituras adicionais. A Unidade 1 exigirá 10 horas de sua dedicação e a Unidade 2, o dobro de dedicação (20 horas).

Unidade 2 – Linguagem Algorítmica

A linguagem algorítmica consiste na definição de uma pseudolinguagem (também chamada de pseudocódigo) de programação para representar algoritmos. Esta pseudolinguagem, apesar de fazer uso de um português estruturado (o português que conhecemos, porém com regras para formação do algoritmo), se assemelha muito ao modo como os programas são escritos. Por isso, essa representação permite que os algoritmos nela representados possam ser traduzidos, quase que diretamente, para uma linguagem de programação (bom, não?).

A linguagem algorítmica não tem uma padronização. Você vai ver que em cada livro que você consultar, você poderá encontrar a linguagem com alguma modificação no formato. Porém, todos os formatos têm os mesmos tipos de estruturas, apenas escritas de forma um pouco diferente (por exemplo, em um tipo você pode encontrar a palavra fim para marcar o final do algoritmo e em outro tipo, como será o nosso caso, a palavra usada será finalgoritmo). Para o nosso estudo, vamos adotar a linguagem algorítmica que é usada dentro da ferramenta VisuAlg (<http://www.apoioinformatica.inf.br/visualg/objetivos.htm>), uma vez que, posteriormente, vamos utilizar essa ferramenta para a compilação dos nossos algoritmos. E qual o formato dessa linguagem, afinal? O formato geral de um algoritmo em linguagem algorítmica está representado no Quadro 4.



Quadro 4 - Formato geral de um algoritmo em linguagem algorítmica

Neste quadro, os trechos que estão entre os sinais < > são obrigatórios e os que estão entre colchetes [] são opcionais. Agora,

vamos explicar, resumidamente, o código do Quadro 1:

- Algoritmo é uma palavra que indica o início da definição de um algoritmo em forma de pseudocódigo.
- <nome_do_algoritmo> é um nome simbólico dado ao algoritmo com a finalidade de distinguí-los de outros algoritmos que você possa vir a criar. Esse nome deverá vir entre aspas.
- <declaração_de_constants> - é uma parte opcional do algoritmo e indica a área onde será feita a declaração dos valores constantes (que não vão mudar durante a execução do algoritmo). Mais detalhes, daqui a pouco.
- var <declaração_de_variáveis> – é uma parte opcional do algoritmo e indica a área onde será feita a declaração de variáveis (que vamos saber o que é daqui a pouco) do algoritmo. Uma vez que essa parte exista no algoritmo, deve-se indicar a área de declaração com a palavra var e, depois dela, obrigatoriamente, especificar que variáveis serão usadas no algoritmo. Paciência que explicaremos isso em detalhes, mais a frente.
- <definição de subalgoritmos> consiste de uma área opcional do algoritmo onde são definidos os subalgoritmos, ou seja, algoritmos que podem ser criados dentro de outros. Isso será assunto do módulo 2. Logo, por agora, apenas saiba que esse seria o lugar onde os sub-algoritmos, também chamados de sub-rotinas entrariam.
- início e finalgoritmo são respectivamente as palavras que delimitam o início e o fim do conjunto de instruções do corpo do algoritmo, que é o lugar onde vai ocorrer todo o processamento (realização de cálculos, leituras de dados, saída de dados, operações, etc).

No Quadro 1, as palavras indicadas em negrito são palavras reservadas da linguagem algorítmica. Você poderia então perguntar: e o que são palavras reservadas?

Palavras Reservadas são palavras que fazem parte da estrutura de uma linguagem (como, por exemplo, a linguagem algorítmica). Ou seja, elas têm significados pré-determinados dentro da linguagem e, por causa disso, não podem ser usadas para outra finalidade diferente daquela que foi especificada na linguagem. Por exemplo, na

linguagem algorítmica a palavra Algoritmo é uma palavra reservada que indica o início da definição do algoritmo. Dessa forma, você não pode usar essa palavra em nenhuma outra parte do algoritmo: nem na declaração de variáveis, nem como parte das instruções, comandos, cálculos ou expressões do corpo do algoritmo. A utilização indevida de palavras reservadas causam erros no processo de compilação. Apenas para ilustrar, as palavras reservadas da linguagem algorítmica que vamos utilizar são:

aleatorio	enquanto	funcao	para
algoritmo	entao	inicio	passo
arquivo	escolha	int	pausa
asc	escreva	inteiro	pos
ate	escreval	interrompa	real
carac	faca	leia	procedimento
caracpnum	falso	limpatela	repita
caractere	fimalgoritmo	logico	retorne
caso	fimenquanto	maiusc	se
compr	fimescolha	minusc	senao
copia	fimfuncao	mod	timer
cronometro	fimpara	nao	var
debug	fimprocedimento	numpcarac	vetor
e	fimrepita	ou	verdadeiro
eco	fimse	outrocaso	xou

Você não precisa se preocupar em decorar essas palavras! Nem nós vamos explicar todas elas agora. Essas palavras irão aparecendo no decorrer das explicações dessa unidade e, também, do próximo módulo. É só ter paciência, ok?

Ah, uma coisa que ia esquecendo, a linguagem algorítmica não faz distinção entre letras maiúsculas e minúsculas. Ou seja, a palavra Algoritmo seria reconhecida dentro da linguagem da mesma forma que a palavra algoritmo. Porém, há linguagens (como por exemplo, C e Java) onde essas palavras seriam consideradas diferentes.

Agora, também, apenas para ilustrar, ou seja, você não precisa se preocupar em entender esse código agora, vamos dar um exemplo do uso da linguagem algorítmica para aquele caso que já apresentamos em descrição narrativa e fluxograma de “calcular a média de dois alunos a partir de duas notas dadas”. Esse exemplo pode ser visto no Quadro 5.



Quadro 5 - Exemplo de uso da linguagem algorítmica para calcular a média de um aluno

Como afirmei antes: não se assuste! Mais à frente, vamos explicar melhor cada pedacinho desse código. E, logo, logo, ele vai ficar mais claro para você.

E aí? Por onde começo?

Antes de destrinchar melhor a linguagem algorítmica, há uma questão que muitos alunos costumam me perguntar: quando eu vou construir um algoritmo, por onde eu começo? Bem, vou dar umas dicas que podem ser úteis para você nesse sentido.

1. Primeiro: antes de começar, tente entender o que o problema está pedindo. Como fazer isso? Leia o enunciado do problema com atenção, para tentar entender, em linhas gerais, o que está sendo pedido. Se você não entender o que o problema quer dizer, fatalmente vai construir o algoritmo errado que, provavelmente, não vai resolver o problema, não vai satisfazer o objetivo. Fazendo um paralelo... Imagine que você está dirigindo com uma pessoa lhe ensinando o caminho para

chegar ao lugar desejado (o objetivo). Se a pessoa dissesse para você dobrar na próxima rua à direita e você entendesse errado e dobrasse na próxima rua à esquerda, com certeza você não chegaria ao destino desejado, não é?

2. Segundo: leia novamente o problema para identificar as três fases de resolução do problema que são:

ENTRADA – o que você identificou que vai precisar que o usuário ou alguém lhe informe? (via teclado, leitor de código de barras ou mouse). Ou seja, os valores de trabalho do algoritmo, o que precisa ser perguntado ao usuário. Normalmente, na descrição dos problemas elas aparecem, mais ou menos, assim:

- “Escreva um programa que recebe 2 notas de aluno e calcula a média aritmética delas.”
- “Escreva um algoritmo para ler o preço e a quantidade de um produto e calcula o preço a pagar pelo mesmo.”
- “Escreva um algoritmo para calcular o salário líquido de um funcionário, a partir do número de horas trabalhadas e da quantidade de dependentes. Sabe-se que o valor da hora é de 12 reais, que se recebe 40 reais por cada dependente e que é descontado um total de imposto de 12% do valor do salário bruto (salário sem descontos)”.

Você consegue identificar nos enunciados (descrições do problema) acima os valores que deverão ser fornecidos na entrada? Leia atentamente. Os valores estão indicados em itálico. Veja que é justamente o que você precisa ter para poder calcular o que o problema está pedindo. Por exemplo, no primeiro exemplo, você precisa das duas notas (entrada) para poder calcular a média solicitada. No segundo exemplo, você precisa do preço e da quantidade para calcular o valor a pagar. E, no último exemplo, você precisa do número de horas trabalhadas e da quantidade de dependentes para poder calcular o salário do funcionário. Qualquer dado que precise ser informado pelo usuário (por exemplo, através de digitação de valores) é considerado um dado de entrada.

PROCESSAMENTO - que tipo de cálculos serão necessários para resolver o problema? Há alguma checagem a ser feita com os dados? Existe alguma decisão a ser tomada dentro do algoritmo? Existem

operações de processamento que precisarão ser realizadas mais de uma vez? Ou seja, o que é preciso fazer para que o algoritmo gere as saídas esperadas? Vejamos os mesmos exemplos anteriores.

- No primeiro exemplo, para calcular a média aritmética, sabemos que o processamento é somar as duas notas e dividir por dois: $(\text{nota1} + \text{nota2})/2$
- No segundo exemplo, para calcular o valor a pagar, seria multiplicar o preço do produto, pela quantidade comprada: preço unitário x quantidade comprada
- O terceiro exemplo é o mais elaborado, pois precisaríamos fazer em mais de um passo o cálculo. Vamos lá! Se você reler o problema, teríamos que:

Para calcular o valor do salário bruto (só os ganhos da pessoa, sem os descontos), teríamos de multiplicar as horas trabalhadas pelo valor da hora fornecido (12 reais) e somar com a quantidade de dependentes, multiplicada pelo valor pago por dependente (fornecido como 40 reais). Dessa forma:

$$\text{Salário bruto} = (\text{horas trabalhadas} \times 12) + (\text{no. Dependentes} * 4)$$

Agora, para calcular o salário líquido, teríamos de descontar os impostos do salário bruto. O problema diz que é descontado do salário bruto, 12% de impostos . Dessa forma, ficaríamos com:

$$\text{Salário líquido} = \text{salário bruto} - ((12 * \text{salário bruto})/100)$$

- Um exemplo adicional de processamento seria “Leia um número qualquer e diga se ele é maior ou menor que zero, imprimindo uma mensagem apropriada”. Nesse caso, o processamento não seria calcular alguma coisa, mas sim fazer uma checagem (no caso se o número é maior ou menos que zero).
- **SAÍDA** – quais os resultados que necessitarão ser apresentados aos usuários? O que deve ser impresso pelo algoritmo na tela/monitor do computador? Geralmente, essa parte vem expressa no enunciado do problema, é o que o problema ta pedindo. Por exemplo, a saída do primeiro exemplo, é a média aritmética das notas. A saída do segundo exemplo, o preço a pagar pelo produto e no terceiro exemplo, o salário líquido do funcionário. Às vezes, na saída, você

pode precisar, também, dar alguma mensagem. Por exemplo, naquele caso de checar se o número é maior ou menor que zero, você, provavelmente, precisaria escrever na tela “O número é maior do que zero” ou “o número é menor do que zero”. Isso também é saída

Identificadores

Identificadores são nomes únicos definidos pelos programadores para identificar/distinguir os elementos de um algoritmo, tais como: o nome do algoritmo, as constantes e as variáveis (essas duas últimas, veremos daqui a pouco o que são). Para definir esses nomes únicos, algumas regras devem ser seguidas:

- O nome do identificador deve sempre começar por uma letra (caractere alfabético). Ou seja, o primeiro caractere do nome deve ser uma letra.
- Depois, os outros caracteres depois do primeiro podem ser letras ou números.
- Não é permitido o uso de espaço em branco ou de caracteres especiais, tais como: @, #, &, *, -, +, ?, \$, etc. O único caractere especial permitido é o _ (subscrito). Por exemplo, nome_completo.
- Maiúsculas e minúsculas são consideradas letras iguais, logo a palavra Nome é igual a palavra nome.
- O nome dos identificadores não deve ter acentuação, nem fazer uso do cedilha (ç).
- O nome do identificador não pode ser igual a nenhuma palavra reservada da linguagem algorítmica.
- O nome deve ser significativo. Ou seja, deve expressar o valor que ele pretende representar. Por exemplo, um identificador chamado X, não diz nada. Você não terá a menor idéia do que valor está guardado nele. Porém um identificador chamado nota1, você sabe que, provavelmente, o valor guardado nesse identificador é o da primeira nota.
- Você não pode ter dois identificadores de mesmo nome dentro do algoritmo.

Baseados nessas regras, alguns exemplos de identificadores válidos, corretos, são: salario_minimo, nome, cpf, numeroChassi. Agora, alguns exemplos de identificadores inválidos, incorretos, com o problema apresentado entre parênteses:

- vm (o nome não é significativo)
- 13salario (identificador tem de começar por uma letra)
- salárioR\$, %desconto, km/h, (o nome do identificador não poder ter símbolos especiais)
- horário, preço (não se deve usar acentuação ou cedilha)
- valor total (o identificador não pode ter espaço em branco)
- logico, passo, algoritmo (o nome do identificador não pode ser uma palavra reservada)

Constantes, Variáveis e Tipos de Dados

Como dissemos na seção anterior, constantes e variáveis são tipos de identificadores. Na verdade, elas são os elementos básicos que um programa manipula, ou seja, são os dados manipulados pelo programa. Vamos explicar isso melhor agora.

CONSTANTES

Constante é um determinado valor fixo que não se modifica ao longo do tempo, durante a execução de um programa. Você deve lembrar da palavra constante da matemática. Uma das constantes mais famosas daquela área é a chamada π (pi). Lembra dela? É aquela que vale 3,1415...? A idéia é ter um identificador que guarda esse valor constante que não vai ser modificado no decorrer do algoritmo. Seu uso poupa tempo quando se quer alterar um valor a cada execução do algoritmo (não durante a mesma execução), porque ao trocar o valor da constante, todas as instruções que manipulam aquele identificador da constante, irão, automaticamente, assumir o novo valor (mostraremos isso a você mais para frente, pode esperar...).

A declaração (definição) das constantes do algoritmo deve ser feita na área de declaração de constantes (vide Quadro 1), logo depois da definição do nome do algoritmo. O formato da declaração é o

seguinte:

<nome da constante> = <valor>

Por exemplo:

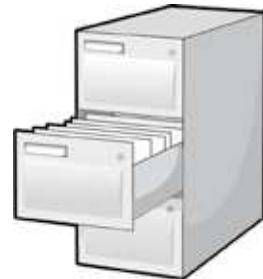
PI <- 3.1416

MAX <- 10

PAIS <- "Brasil"

Onde <- é o operador de atribuição (que falaremos mais a frente), mas para adiantar, ele pega o valor do lado direito e armazena no identificador da constante do lado esquerdo. Por exemplo, PI <- 3.1416, quer dizer que a constante PI vai receber (para armazenar durante a execução do algoritmo) valor 3.1416. Falaremos desse armazenamento daqui a pouco.

VARIÁVEIS



Na Matemática, uma variável é a representação simbólica dos elementos de um conjunto. Já nos algoritmos a variável vai armazenar um determinado valor de dados. E isso sempre vai ser necessário, porque todo dado manipulado pelo algoritmo vai precisar estar em uma variável. De fato, uma variável corresponde a uma posição de memória (a memória de trabalho do computador, que conhecemos como memória RAM), identificada por um nome (o identificador) e cujo conteúdo pode variar ao longo da execução do algoritmo. Fazendo um paralelo é como se a memória principal do seu computador fosse um armário, onde cada gaveta é uma variável onde eu posso armazenar alguma coisa. Mas claro que o que vai ser armazenado depende do tamanho da gaveta, né? E esse tamanho, vai ser especificado pelo tipo de dado da variável. Além disso, embora uma variável possa assumir

diferentes valores no decorrer do algoritmo, ela só pode armazenar um valor a cada instante. Antes de mostrar a declaração de variáveis, precisamos definir os tipos de dados. Vamos lá!

TIPOS DE DADOS

Toda variável possui, além do nome (o identificador), um tipo, responsável por definir o espaço que essa variável vai ocupar na memória. As variáveis podem ser basicamente de três tipos: Numéricas, literais ou lógicas (vide Figura 15).

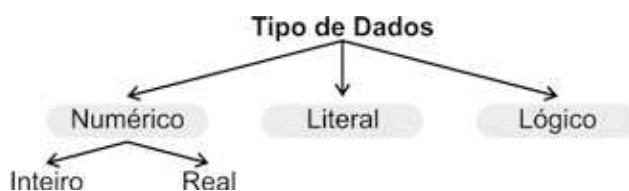


Figura 15 - Tipos de Variáveis

Tipo Numérico é específico para armazenamento de números (positivos ou negativos), que posteriormente poderão ser utilizados para cálculos. O tipo numérico podem ser ainda classificado como **Inteiro** ou **Real**. O tipo inteiro serve para armazenamento de números inteiros, ou seja, aqueles que não possuem ponto flutuante. Por exemplo, 234, 1, -89 e 27 e o tipo real serve para o armazenamento de números que possuam casas decimais. Por exemplo, 2.14, -9.0, 1200.09 e 0.23.

O **Tipo Literal ou Caractere** é específico para armazenamento de uma ou mais letras, números ou símbolos especiais. Porém, os números do tipo Literal não poderão ser utilizados para operações matemáticas. Geralmente, os dados literais são representados nos algoritmos pela coleção de caracteres, delimitada em seu início e término com o caractere aspas (""). Por exemplo: "QUAL ?", " ", "AbCdefGHi", "1-2+3=", "Ana Maria" e "0". Note que, por exemplo, "2" representa um dado do tipo literal, diferindo de 2 que é um dado do tipo inteiro. Com o segundo você poderá fazer cálculos matemáticos. Mas com o caractere não. Como saber a diferença? O dado do tipo caractere SEMPRE deverá aparecer entre " ".

O **tipo lógico** armazena somente dados lógicos que só possuem dois valores possíveis: Verdadeiro ou Falso. Esse tipo é usado em expressões lógicas. É comum encontrar em outras referências

outros tipos de pares de valores lógicos possíveis, tais como sim/não, 1/0, true/false. Porém, aqui adotaremos apenas verdadeiro (ou simplesmente V) ou falso (ou simplesmente F)

DECLARAÇÃO DE VARIÁVEIS

As fases de Entrada, Processamento e Saída podem manipular vários tipos de variáveis. Para definir que variáveis são essas é preciso fazer a declaração das mesmas. Em linguagem algorítmica as variáveis são declaradas na seção de declaração de variáveis (identificada pela palavra var), antes do início do algoritmo propriamente dito. A declaração dever ter o seguinte formato:

```
var <nomeVar1, ..., nomeVarN>: <tipo_dado_da(s)_variavel(eis)>
```

Alguns exemplos de declarações são:

```
var
    media, salário: real (veja que você pode declarar mais de uma variável por linha)
    contador: inteiro
    achou: logico
    nome, sexo: caractere
```

E o que o computador faz a partir dessa declaração? Bem, imagine que a Figura 16 é a memória do computador. Cada pedaço da memória tem um endereço, para poder ser acessada. Esse endereço é conhecido pelo sistema operacional (ex: windows, linux) do computador. Na figura, estamos “inventando” endereços, para tentar lhe explicar como as coisas funcionam. Por exemplo @#1, @#2 são endereços de memória na nossa figura, porém os endereços de verdade são bem mais complexos que apenas isso, são complicados de serem expressos. Algo como @#\$1208DFE ou pior.



Figura 16 - Exemplo de Memória do Computador

Quando declaramos variáveis, estamos reservando na memória uma área para aquela variável (para guardar o seu valor) e identificando essa área pelo nome da variável. O tamanho da área vai depender do tipo da variável (se é caractere, se é inteiro, se é real ou lógico). Por exemplo, suponhamos as declarações:

```
var nome: caractere
idade: inteiro
ativo: logico
```



Figura 17

Com essas declarações o compilador ou interpretador da linguagem, seguindo a ordem que as declarações foram feitas, pegaria uma área de memória qualquer (aleatoriamente), onde coubesse algo do tipo caractere, marcaria como ocupada e chamaria essa área de nome (vide Figura 17).

Depois o compilador/interpretador pegaria uma área (de novo, aleatoriamente, ou seja, não é nada seqüencial), que coubesse um número inteiro, marcaria a área como ocupada e a chamaria de idade. Da mesma forma, seria reservada uma área para um valor lógico, marcada como ocupada e essa área seria chamada de ativo, que poderia servir, por exemplo, para indicar se a pessoa está ativa ou aposentada, em termos de trabalho (vide Figura 18).



Figura 18 - Memória do Computador após a declaração de todas as variáveis.

Agora, se quiséssemos colocar um valor dentro de cada variável, poderíamos chamá-las pelo nome, que o compilador/interpretador, automaticamente, levaria para o endereço de memória correto. Por exemplo:

```

nome <- "Ana Maria"
idade <- 27
ativo <- V

```

Resultaria na modificação na memória do computador, apresentada na Figura 19.



Figura 19 - Memória do Computador após a atribuição de valores às variáveis

Se não tivéssemos a noção de variável, nós teríamos de conhecer cada endereço de memória do computador, para poder implementar algum programa. Já pensou a complicação? Para exemplificar, a nossa atribuição de valores passaria a algo do tipo:

```

//atribuindo o valor "Ana Maria" a um endereço de memória
@#$1208DFE <- "Ana Maria";
//atribuindo o valor 27 a um endereço de memória
@#$1118DCC = 27;

```

E com o detalhe que os endereços poderiam mudar de computador para computador, dependendo da marca e da capacidade de armazenamento da memória, do sistema operacional instalado, entre outras coisas. Por isso, usar variáveis é uma forma de facilitar o acesso para armazenamento e recuperação de valores na memória do computador. Pois agora, nós podemos atribuir valores aos nomes das variáveis e a linguagem se encarrega de gerenciar a memória para nós. Lembre, para fazer qualquer algoritmo que precise processar qualquer coisa que seja, você vai precisar declarar variáveis, porque são elas que armazenam os valores para você.

Operadores e Expressões

Agora que as variáveis já estão declaradas, para fazer a parte do PROCESSAMENTO da maior parte dos algoritmos, vamos precisar

fazer uso de operadores. Os operadores são os meios usados para incrementar, decrementar, comparar e avaliar dados dentro do computador. Já uma expressão faz uso dos operadores e de operandos para construir fórmulas que serão usadas na parte do processamento do algoritmo. Operandos são as variáveis ou valores usados na expressão. Por exemplo, na expressão $4 + 5$, os operandos são os números 4 e 5 e o operador é o + (que representa a soma). Já os operadores podem ser dos seguintes tipos: operador de atribuição (que a gente até já veio utilizando, antes de explicar pra valer, ele é aquela setinha <-), operadores aritméticos, operadores relacionais e operadores lógicos.

As expressões podem ser classificadas como expressões aritméticas (quando o cálculo retorna um valor numérico, inteiro ou real) ou expressões lógicas (quando o cálculo retorna um valor lógico, verdadeiro ou falso). Vamos explicar melhor isso tudo no texto a seguir.

Operador de atribuição

Serve para atribuir um valor a uma variável ou para copiar o conteúdo de uma variável para outra do mesmo tipo. O operador de atribuição é representado por \leftarrow (uma seta apontando para a esquerda). Em uma atribuição temos dois elementos fundamentais: o primeiro é o elemento para o qual estaremos realizando a atribuição e o segundo elemento é o que nos fornece o valor a ser atribuído. O primeiro elemento é sempre uma variável e o segundo é uma expressão, outra variável ou um valor. Alguns exemplos de uso são:

- **Nome** \leftarrow “**Sandra**” (aqui temos que a variável Nome recebe o valor Sandra)
- **Idade** \leftarrow **18** (a variável Idade recebe o valor 18)
- **Casado** \leftarrow **F** (a variável Casado recebe o valor F (falso))
- **Salário** \leftarrow **500.55** (a variável Salario recebe o valor 500.55)
- **Soma** \leftarrow **45 + 10** (a variável Soma recebe o valor da soma de 45 com 10, ou seja, 55)
- **Nome2** \leftarrow **Nome** (a variável Nome2 recebe o mesmo valor que Nome, ou seja, aqui você está copiando para Nome2 o que estava armazenado em Nome, ou seja, o valor “Sandra”).

Esse tipo de atribuição envolvendo duas variáveis só pode ser feito se as duas variáveis forem do mesmo tipo (para poderem ter a mesma quantidade de posições reservadas na memória (vide Figura 20).



Figura 20 - Variável Nome copiada para a Variável Nome2

O tipo da variável deve ser respeitado durante uma atribuição, ou seja, a variável que vai receber o valor da atribuição deve ser compatível com o valor a ser atribuído.

Operadores aritméticos e expressões aritméticas

Os operadores aritméticos são símbolos que representam operações matemáticas básicas. Eles estão implementados em, praticamente, todas as linguagens de programação, ainda que possam existir algumas ligeiras variações de uma linguagem para outra. Esses operadores são classificados em:

- Binários: atuam sobre dois operandos, ou seja, precisam de dois valores para poder agir. A multiplicação é um operador binário. Por exemplo, $5 * 4 = 20$.
- Unários: atuam sobre um único operando. São os sinais de + e -, quando indicam se o valor é positivo (maior que zero) ou negativo (menor que zero). Ex: -10 e +6

Na tabela 1, apresentamos todos os operadores binários aceitos pela linguagem algorítmica. Quanto mais na parte de cima da tabela estiver o operador, maior a sua prioridade quando usado em uma expressão. Porém, sabemos que a prioridade dos operadores pode ser alterada usando parênteses. Por exemplo, as expressões $(4 + 5) * 3$ e $4 + 5 * 3$ dariam o mesmo valor?

Não, não dariam. Na primeira expressão, a prioridade é de quem está entre parênteses, dessa forma, a soma seria resolvida primeiro e depois a multiplicação, ficando com $20 * 3 = 60$. Já na segunda expressão, como não há parênteses, seria considerada a prioridade

dos operadores. Pela tabela 1, a multiplicação tem prioridade sobre a soma. Dessa forma, teríamos $4 + 15 = 19$.

Operador	Tipo	Operação	Exemplo	Prioridade	
-	Unitário	Inversão do Sinal	- 10	1	
+	Unitário	Manutenção do Sinal	+ 8	1	
[^]	Binário	Potenciação	$2 ^ 3 = 8$	2	
\	Binário	Divisão Inteira	$6 \backslash 4 = 1$	3	
MOD	Binário	Resto da Divisão Inteira	$6 \text{ MOD } 2 = 0$	3	
/	Binário	Divisão	$6 / 4 = 1.5$	3	
*	Binário	Multiplicação	$6 * 4 = 1.5$	3	
-	Binário	Subtração	$6 - 4 = 2$	4	
+	Binário	Adição	$6 + 4 = 10$	4	

Tabela 1 - Operadores Aritméticos e suas Prioridades

Uma expressão aritmética é justamente o que usamos acima. É fazer uso dos operadores aritméticos para criar uma fórmula de cálculo. Expressões aritméticas retornam, necessariamente, um valor inteiro ou real. Vejamos outro exemplo: $10 + (3 + 1) / 2$. Quanto daria essa expressão? Avalie as prioridades... daria 12. Como? Primeiro deveria ser feito o cálculo entre parênteses $10 + 4 / 2$. Agora, a prioridade é da divisão (consulte a tabela 1), daí teríamos $10 + 2 = 12$.

Voltando para a Tabela 1, a maior parte dos operadores você já conhece. Porém, gostaria de destacar três deles que você pode não conhecer. A barra invertida (\backslash) equivale a divisão inteira de um número. E o que é isso? Quer dizer você realizar a divisão de um número por outro e não considerar (desprezar) a parte decimal obtida. É o caso do exemplo dado na Tabela 1: $6 \backslash 4$ é igual a 1, porque a divisão normal de 6 por 4 daria 1.5. Porém, na divisão inteira, a parte .5 (parte decimal) seria ignorada, dando como resultado apenas o valor inteiro 1. Outro operador é a potenciação ($^$). Só relembrando, se temos $X ^ Y$, quer dizer que X será multiplicado por ele mesmo Y vezes. Ex: $2 ^ 3 = 2 \times 2 \times 2 = 8$. Por último o operador MOD, que também pode ser representado por %. Ele representa o resto da divisão inteira. E o que é isso? Bem, vamos explicar. Quando você faz uma conta de divisão, você tem os elementos especificados na Figura 21. O MOD fornece justamente o valor do resto e não o do quociente, como é o caso da divisão normal. Logo, como ilustrado na figura: 10 MOD 2 seria igual a zero. Porque o resto da divisão de um número par por dois, sempre é ZERO (ou seja, a divisão é exata, não possui resto).



Figura 21 - Elementos da Divisão

Para ver se você entendeu, quanto seria 7 MOD 3? Seria um. Veja na Figura 22.



Figura 22 - Demonstração do cálculo de 7 MOD 3

E quanto seria 4 MOD 6? Mais difícil? Realmente, aqui temos uma exceção (vide Figura 23). Toda vez que se vai calcular o MOD, onde o valor do divisor é maior que o número a ser dividido (dividendo), o resto é o próprio dividendo. Isso porque como é resto da divisão inteira, não nos interessa esperar pelos outros valores da parte decimal, uma vez que divisão final daria: 0.666666666.



Figura 23 - Demonstração do cálculo de 4 MOD 6

Operadores relacionais

Operadores relacionais são sinais utilizados para realizar comparações entre valores, variáveis, expressões e constantes. Estes operadores sempre retornam valores lógicos (verdadeiro ou falso). Todos os operadores relacionais têm a mesma prioridade, logo, para estabelecer prioridades no que diz respeito a qual operação executar primeiro, você deve utilizar parênteses. Um detalhe é que as comparações realizadas só podem ser feitas entre dados do mesmo tipo. Por exemplo, você não pode comparar um inteiro e um caractere. Mas pode comparar dois números inteiros. Os operadores relacionais podem ser vistos na Tabela 2. Na tabela é dado um exemplo de uso de cada operador e que valor ele retornaria com a comparação (ou seja, se o valor é V – verdadeiro ou F – falso).

Operador	Comparação	Exemplo	Resultado
>	Maior que	$4 > 4$	V
<	Menor que	$4 < 4$	F
\geq	Maior ou igual	$4 \geq 4$	V
\leq	Menor ou igual	$4 \leq 4$	F
=	Igual	$4 = 4$	F
\neq	Diferente	$4 \neq 4$	V

Tabela 2 - Operadores Relacionais

Para ilustrar melhor, vamos também dar exemplos das outras comparações que podem ser realizadas. Considere que x tem valor 2, y tem valor 5, w tem valor 4 e j tem valor 3.

- variável e constante. Ex: $x = 3$ (o que retornaria um Falso, porque $x = 2$)
- variável e variável. Ex: $x \neq y$ (o que retornaria o valor Verdadeiro)
- variável e expressão. Ex: $j = (w - 1)$ (o que retornaria o valor Verdadeiro)
- expressão e expressão. Ex: $(x + 1) > (y + 4)$ (o que retornaria o valor Falso)

Os operadores relacionais são usados para criar expressões lógicas, que serão explicadas na subseção a seguir.

Operadores lógicos e expressões lógicas

Os operadores lógicos são também conhecidos como operadores booleanos, porque derivam da álgebra de lógica matemática desenvolvida por George Boole. A principal função destes operadores é construir expressões, chamadas de expressões lógicas, que são aquelas que resultam em valores lógicos verdadeiro/falso. Em termos práticos, eles vão conectar condições a serem testadas durante a execução do algoritmo. Em outras palavras, esses operadores servem para realizar operações sobre valores lógicos (verdadeiro/falso). Os operadores lógicos podem ser vistos na Tabela 3 e serão explicados a seguir.

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

Tabela 3 - Operadores Lógicos

NÃO (negação) – é o operador mais simples e de maior prioridade. Ele é um operador unário (opera sobre apenas uma variável) e serve para inverter o valor da expressão ou condição. Por exemplo, se a expressão for verdadeira, ele inverte para falsa e vice-versa.

Exemplo: **Não (4 < 3)** é avaliada como **Verdadeira**, porque a expressão ($4 < 3$) é falsa e Não falso é igual a verdadeiro.

E (conjunção) - é o próximo operador na ordem de precedência. Ele retorna Verdadeiro como resultado se (e somente se) ambos os operandos forem avaliados como Verdadeiros. Se qualquer um dos operandos for avaliado como Falso, ele retorna Falso. Esse operador corresponde ao **E** usado em português, como na frase “Só poderá se inscrever no concurso quem for maior de 18 anos E tiver conhecimentos básicos de informática”. Ou seja, se uma das condições não for verdadeira, o candidato não poderá se inscrever. Esse é o mesmo caso do operador lógico E. Se uma das condições for falsa, o resultado é falso. Vamos dar uma olhada nas combinações possíveis na Tabela 4.

Exemplo: a expressão **(3<4) E ((8+2) = 11)** resultaria em Falso. Porque a primeira expressão é verdadeira, mas a segunda é falsa. E, pela Tabela 4, você pode ver que verdadeiro E Falso = Falso.

Valor 1	Operador	Valor 2	Resultado
Falso	E	Falso	Falso
Falso	E	Verdadeiro	Falso
Verdadeiro	E	Falso	Falso
Verdadeiro	E	Verdadeiro	Verdadeiro

Tabela 4 – Uso do Operador E

OR (disjunção) - o operador **OU** retorna o resultado Verdadeiro se qualquer um dos dois operandos for avaliado como Verdadeiro. Novamente, o significado é similar ao uso da palavra **OU** em português, como, por exemplo, “Eu irei ao mercado se acabar o arroz ou o feijão”. Ou seja, se uma das condições for verdadeira, o resultado

já será verdadeiro. As combinações possíveis do operador OU podem ser vistas na Tabela 5.

Valor 1	Operador	Valor 2	Resultado
Falso	OU	Falso	Falso
Falso	OU	Verdadeiro	Verdadeiro
Verdadeiro	OU	Falso	Verdadeiro
Verdadeiro	OU	Verdadeiro	Verdadeiro

Tabela 5 – Uso do Operador OU

Exemplo: peguemos a mesma expressão usada antes, trocando o E por OU e vejamos: **(3<4) OU (8=11)** resultaria em Verdadeiro. Porque a primeira expressão é verdadeira e a segunda é falsa. E, pela Tabela 5, você pode ver que verdadeiro OU Falso = Verdadeiro.

Observações finais sobre operadores

Os operadores podem ser combinados para formar expressões. A prioridade entre eles pode ser vista na Tabela 6. Operadores com a mesma prioridade devem ser executados da esquerda para direita. Para alterar a ordem de prioridade, você deve utilizar parênteses.

Operadores	Prioridade
Aritméticos	1
Relacionais	2
Lógicos	3

Tabela 6 – Prioridade entre os operadores

Exemplos: Qual seria o resultado de **Não (3+4 > 10) E (10<> 8-2)**? Seria verdadeiro. Porque o primeiro termo da expressão, entre parênteses resulta em falso, mas como existe a negação na frente dele, ele se tornaria verdadeiro. O segundo termo da expressão é verdadeiro. Então, Verdadeiro E Verdadeiro resultam em Verdadeiro.

Instruções de Entrada e Saída

Para poder implementar no algoritmo as fases de **ENTRADA** e **SAÍDA**, fatalmente você vai precisar ter instruções específicas para essas partes. Isso porque na fase de ENTRADA um algoritmo pode receber dados através de dispositivos como teclado, mouse, um leitor de código de barras, etc. E na fase de SAÍDA, o algoritmo pode enviar dados para o monitor de vídeo ou para a impressora, entre

outros. Estes tipos de operações em que dados são recebidos por um algoritmo ou são enviados por um algoritmo para um dispositivo são chamados de operações de entrada e saída. Elas são instruções básicas que efetuam tarefas essenciais para o recebimento e apresentação de dados. Vamos explicar melhor cada uma.

A instrução de entrada de dados permite que informações que os usuários fornecem através dos dispositivos de entrada sejam transferidas para a memória do computador e armazenadas em variáveis (tal como uma atribuição, pegando o que é lido pelo dispositivo de entrada e guardando na variável). O formato do comando de entrada é o seguinte:

Leia (nomeVariavel) – onde nomeVariavel é o nome da Variável onde o valor lido será armazenado na memória do computador. Ex: **Leia (nome)**. Você também pode ler mais de uma variável por comando: **Leia (nomeVar1, ..., nomeVarN)**, porém, não aconselhamos essa prática, porque o algoritmo fica mais difícil de entender (menos legível). Ex: **Leia (nome, sexo)**.



Quando uma variável é declarada, esta apenas reserva uma posição na memória RAM (memória de trabalho) do computador. Ou seja, o conteúdo dessa posição é vazio. Assim, a cada variável criada deve-se ler ou atribuir um valor a ela. Diferente do operador “←”, que só atribui valores pré-definidos a uma variável, a instrução LEIA permite qualquer entrada de dados válida, fornecida pelo usuário, através de algum dispositivo de entrada.

A instrução de saída de dados é o meio pelo qual o resultado da fase de PROCESSAMENTO do algoritmo será exibido pelos dispositivos de saída de um computador (ex: vídeo e impressora), para que o usuário possa ver. Para simplificar, chamaremos a ativação do comando de saída de “impressão” (mesmo, nem sempre, ele enviando o resultado para a impressora). Há vários formatos possíveis para o comando de saída:

1. Apenas impressão de variável ou variáveis: **Escreva (nome**

Var1) – onde nomeVar1 é o nome da variável onde está o valor que você quer imprimir. Ex: **Escreva (salario)** iria imprimir o valor armazenado na variável salário, como por exemplo 2000. Você também pode imprimir mais de uma variável no mesmo comando de saída: **Escreva (nomeVar1, ..., nomevarN)**. Por exemplo: **Escreva (nome, endereço, cidade)**.

2. **Apenas impressão de texto: Escreva (“texto a ser mostrado ao usuário”)**. Por exemplo, Escreva (“Aula de Introdução a Programação”). Daí, apareceria na tela o texto especificado entre aspas.
3. **Impressão de texto e variáveis: Escreva (“textoA”, nomeVar, “textoB”, nomeVar1, ..., nomeVarN)**, ou seja, é intercalado no comando a escrita de texto e a impressão de valores de uma ou mais variáveis. Por exemplo: Escreva (“Nome= ”, nome, “ Telefone = ”, telefone). Na tela do computador, o que apareceria seria algo do tipo: **Nome= Ana Maria Telefone=32221090**



Usa-se “,” (vírgula) para separar o valor de uma variável com um texto explicativo. Veja que o nome da variável fica sem aspas, porque você quer pegar o valor que está gravado dentro da posição de memória com aquele nome. Ou seja, no caso de uma lista de variáveis, o conteúdo de cada uma delas é pesquisado na memória e enviado para o dispositivo de saída. E no caso do texto que está entre aspas, o que vai ser impresso é exatamente o que estiver escrito entre as aspas. Por exemplo:

media ← 15/2

Escreva (“A Média do Aluno é: ”, valorMedia)

O valor impresso na tela seria: A Média do aluno é: 7.5

Vamos ver agora como esses comandos entrariam em um algoritmo? Suponha que você quer construir um algoritmo para ler o preço unitário e a quantidade de um produto e quer imprimir o preço final a ser pago pelo cliente. Como ficaria?

Algoritmo Exemplo

```

var precoUnit, precoTot: real
    qtd: inteiro

inicio
    Leia(precoUnit, qtd)
    precoTot <- precoUnit * qtd
    Escreva(precoTot)
fimalgoritmo.
```



Há duas regras básicas para melhorar nossos algoritmos:

- 1) Antes de toda operação de leitura deve ser impressa uma mensagem informando ao usuário que dados ele deve fornecer para o programa.
- 2) Todo resultado enviado ao usuário, deve ser precedido de uma mensagem explicativa sobre que valor está sendo mostrado.

Assim sendo, vamos aplicar essas regras no algoritmo que apresentamos anteriormente.

Algoritmo Exemplo

```

var precoUnit, precoTot: real
    qtd: inteiro

inicio
    Escreva("Informe o preço unitário e a quantidade")
    Leia(precoUnit, qtd)
    precoTot <- precoUnit * qtd
    Escreva("O valor a ser pago será: ", precoTot)
fimalgoritmo.
```

Veja que agora o algoritmo está mais claro e as informações serão melhor apresentadas ao usuário.

Sintaxe e Semântica

A definição de qualquer linguagem comporta dois aspectos: a

forma da linguagem e o significado associado a essa forma. A estes aspectos damos respectivamente o nome de sintaxe e semântica da linguagem. Vamos detalhá-los.

A **sintaxe** dita as regras de como os comandos de uma linguagem e cada um dos seus termos deve ser construído corretamente. Ou seja, se o comando está correto ou não. Formalizando, sintaxe é um conjunto de regras formais que especificam a composição dos algoritmos a partir de letras, dígitos e outros símbolos. Por exemplo, o comando de entrada de dados, a sintaxe dele, ou seja, a sua forma correta é: **leia (nomeVariavel)**. Se você escrever `ler(nomeVariável)` a sintaxe vai estar incorreta, ou seja, a escrita do comando vai estar errada. A violação da sintaxe de uma instrução impede que ela seja executada. Ou seja, vai causar erro no algoritmo. Por exemplo, os comandos abaixo apresentariam problema de sintaxe:

- **mediaNotas ← “Ana Maria”** (toda média, geralmente, é um valor do tipo real e não poderia receber um valor do tipo caractere)
- **“nota” >= 7** (todo valor entre aspas é do tipo caractere. E um caractere não poderia ser comparado com um valor do tipo inteiro (como é o caso do sete), uma vez que só se podem comparar valores do mesmo tipo).

A semântica diz respeito ao significado lógico das instruções que serão executadas pelo computador. Ou seja, se os comandos fazem sentido. Porque um comando (ou seqüência deles) pode estar escrito corretamente, mas não fazer nenhum sentido na prática. Formalmente, semântica é o conjunto de regras que especifica o “significado” de qualquer programa ou algoritmo, sintaticamente válido. A checagem da semântica de um algoritmo é mais difícil que a checagem da sintaxe, porque ela depende do conhecimento do problema que está sendo resolvido e do contexto do mesmo. E, também, porque a violação da semântica não impede que o algoritmo seja executado. Todavia, ele processará e mostrará ao usuário um resultado errado.

Em resumo, temos que a sintaxe dita as regras de como as frases de um a linguagem e cada um dos seus termos deve ser construídos corretamente. Já a semântica se preocupa com o significado das frases construídas. Podemos construir frases sintaticamente corretas, mas com problema de semântica. Vamos dar um exemplo fora da área de computação para facilitar o seu entendimento. A frase “O gato

“comeu o rato” está, em termos da língua portuguesa, sintaticamente e semanticamente correta. Porque em termos de sintaxe, a frase tem sujeito, verbo e predicado e o verbo está corretamente conjugado. Semanticamente a frase também está correta, porque na vida real um gato pode mesmo comer um rato. Porém, se a frase fosse “O rato comeu o gato”, teríamos um problema. A frase continua sintaticamente correta. Porém a semântica apresenta problemas. Porque, na vida real, geralmente, um rato não come um gato. Agora, vamos dar um exemplo relacionado com programação, fazendo uso de uma estrutura que veremos na subseção 1.2.8, a estrutura SE. Suponha o trecho de código:

```
Se (media >= 7.0) então
    Escreva ("REPROVADO")
Senão
    Escreva ("APROVADO")
fimse
```

Você vai ver na subseção 1.2.8 que, sintaticamente, a estrutura “Se-Então-Senão” está correta, mas o bloco de instruções está semanticamente errado. Na vida real sabemos que o aluno está APROVADO se obtém media maior ou igual a sete. E não reprovado como mostra o algoritmo.

Os erros sintáticos são identificados pelos tradutores (compiladores e interpretadores), enquanto que os erros semânticos não o são. Por isso, os erros semânticos exigem mais atenção para corrigi-los. Agora que já sabemos o que é sintaxe e semântica e já vimos todos os conceitos mais básicos relacionados a algoritmos (variáveis, constantes, tipos de dados, operadores e expressões e comandos de entrada e saída), podemos partir para construir as estruturas de controle de um algoritmo. Existem 3 estruturas básicas de controle nas quais se baseiam os algoritmos:

- **Seqüencial:** conjunto de comandos separados que são executados em uma seqüência linear de cima para baixo.
- **Seleção, Condicional ou de Decisão:** a partir de um teste condicional, uma instrução, ou um conjunto de instruções, podem ser executados ou não, dependendo exatamente do resultado do teste efetuado.

- **Repetição:** uma instrução ou o conjunto de instruções que será executado repetidamente, de acordo com o resultado de um teste condicional ou de uma contagem.

Detalharemos cada uma delas na subseções a seguir.

Estrutura Seqüencial

Na estrutura seqüencial os comandos de um algoritmo são executados numa seqüência pré-estabelecida. Cada comando é executado somente após o término do comando anterior. Os comandos deverão ser executados numa seqüência linear, seguindo-se o texto em que estão escritos, de cima para baixo, se não houver indicação em contrário (vide Figura 24).

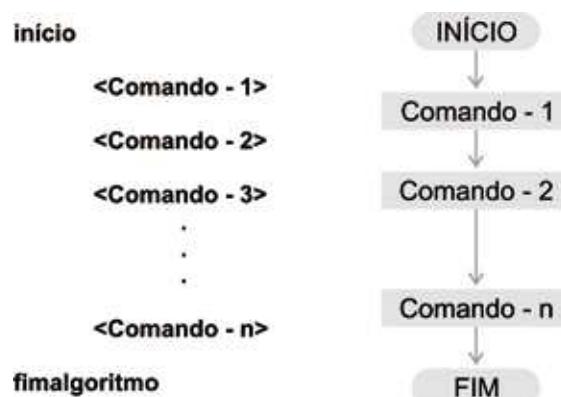


Figura 24 - Estrutura Sequencial

Nesta estrutura da Figura 24, o comando-3 somente será executado após o comando-2 e o comando-2 após o comando-1 e assim por diante. Uma estrutura de seqüencial é delimitada pelas palavras-reservadas **inicio** e **fimalgoritmo** e contém basicamente comandos de atribuição, de entrada e de saída. Em termos de fluxogramas, a estrutura seqüencial é caracterizada por um único fluxo de execução (um único caminho orientado) no diagrama. Vamos ver um exemplo de algoritmo seqüencial. Suponha um algoritmo para somar dois números e mostrar o resultado ao usuário.

Algoritmo Sequencial

```

var num1, num2, soma: inteiro
inicio
    Escreva("Informe o primeiro número")
    Leia(num1)

```

```

Escreva("Informe o segundo número")
Leia(num2)
soma <- num1 + num2
Escreva("A soma dos valores é: ", soma)
fimalgoritmo.

```

O algoritmo proposto respeita as regras de organização e sintaxe da linguagem algorítmica. Porém, qual é a primeira instrução a ser executada? Ele será executado de maneira sequencial, onde, a partir da palavra reservada **var**, as instruções vão sendo executadas de cima para baixo e a próxima instrução só é executada após a anterior ser encerrada. É importante observar que sempre será executado primeiro o bloco de declarações (para poder criar as variáveis na memória do computador, como explicado antes) e depois, na seqüência, o bloco de instruções. A execução da palavra reservada **fimalgoritmo** encerra a execução do algoritmo.

Estrutura Condisional

Estrutura também conhecida como de decisão ou de seleção, caracteriza-se pela execução de um bloco de instruções dependendo da veracidade de uma condição (representada por expressões com operadores relacionais e lógicos). Ou seja, a execução de um ou mais comandos dependerá de se uma condição é satisfeita (é verdadeira) ou não. A estrutura de decisão deve ser sempre usada quando há a necessidade de testar alguma condição e em função da mesma tomar uma atitude. Essa estrutura é muito utilizada, até porque no nosso dia-a-dia, estamos sempre tomando decisões, vejamos um exemplo:

Se estiver chovendo **então** tenho de fechar as janelas

Caso contrário (senão) as janelas podem ficar abertas

A estrutura se subdivide em outras três formas: Condicional Simples, Condicional Composta e Condicional Múltipla. Vamos analisar cada uma delas.

Estrutura condicional simples

Na estrutura condicional simples, uma única expressão lógica é avaliada, para decidir se o bloco de instruções será executado ou não. Em descrição narrativa, seria algo do gênero: “**Se** fizer sol **então**

eu irei à praia". A representação dessa condição em um fluxograma, seria algo como na Figura 25. Nesta mesma figura, o exemplo de uso do fluxograma mostra que se o Tempo for igual a sol, a condição é verdadeira (V) e o comando "vou à praia" será executado. Se a condição não for verdadeira, o comando não será executado.

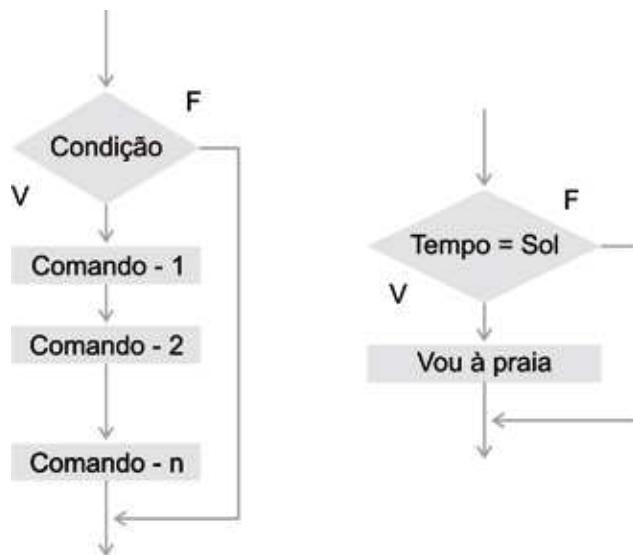


Figura 25 – Modelo Fluxograma e Exemplo para Estrutura Condicional Simples

Em linguagem algorítmica, a sintaxe para esse tipo de estrutura é:

Se <condição> então

```

<comando-1>
<comando-2>
...
<comando-N>
  
```

Fimse

Exemplo:

Se (idade < 18) então

escreva ("Esta pessoa é menor de idade")

fimse

Onde, se a <condição> for verdadeira será executado os comandos dentro do bloco, que pode ser apenas um ou vários comandos (<comando-1>...<comando-2>). Como mencionado anteriormente a <condição> é uma expressão lógica. No caso do exemplo, a frase só será apresentada ao usuário, se a idade for menor que 18.

Estrutura condicional composta

Na estrutura condicional composta, a condição irá determinar qual comando ou bloco de comandos será executado dentre dois listados. Na estrutura simples temos “Se...Então”, nesta caso teremos “Se.... Então.....Senão”. Dessa forma, além de executar se a condição for Verdadeira (V), a decisão composta também vai executar algo se a condição for Falsa (F). Ou seja, a seleção composta sempre executará um bloco condicional, pois se o resultado do teste condicional for verdadeiro um conjunto de instruções ou somente uma instrução será executada e se o resultado não for verdadeiro (for falso) uma outra instrução ou conjunto delas será executada também. Agora, será um caminho ou outro, nunca os dois blocos de instruções serão executados. Em descrição narrativa seria algo do gênero: “**Se** fizer sol **então** eu irei à praia **senão** eu irei ao shopping”. A representação dessa condição em um fluxograma, seria algo como na Figura 26.

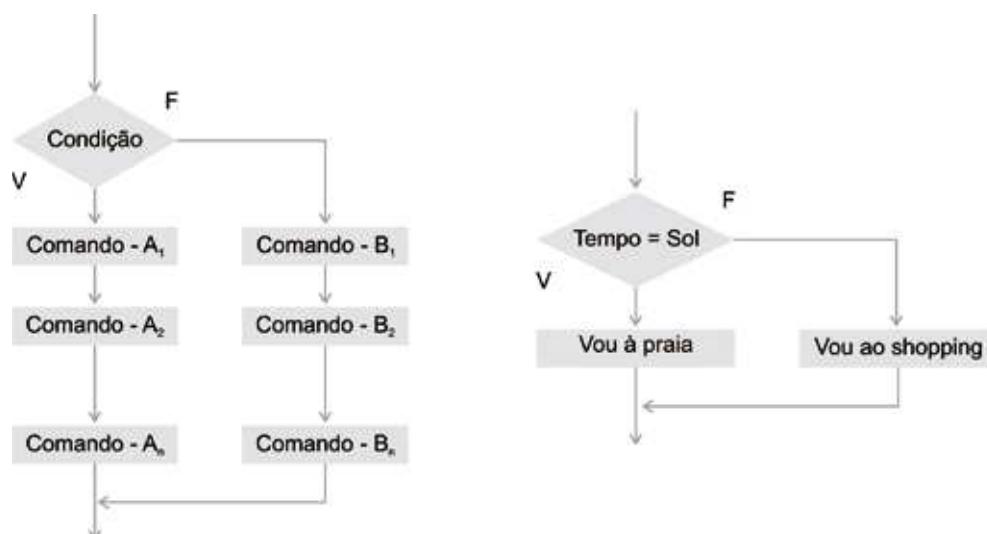


Figura 26 – Modelo Fluxograma e Exemplo para Estrutura Condicional Composta

Nesta mesma figura 26, o exemplo de uso do fluxograma mostra que se o Tempo for igual a sol, a condição é verdadeira (V) e o comando “vou à praia” será executado. Senão, se a condição não for verdadeira (for falsa), o comando “vou ao shopping” será executado. Em linguagem algorítmica, a sintaxe para esse tipo de estrutura é:

```

Se <condição> então
    <comando-A1>
    <comando-A2>
    ...
    <comando-AN>
  
```

senão

```
<comando-B1>
<comando-B2>
...
<comando-BN>
```

Fimse

Exemplo:

```
Se (media >= 7.0) então
    escreva ("Aluno Aprovado!")
senão
    escreva ("Aluno Reprovado!")
fimse
```

Onde, se a <condição> for verdadeira será executado os comandos dentro do bloco SE, que pode ser apenas um ou vários comandos (<comando-1>...<comando-2>. Já se a condição for falsa, será executado o comando ou bloco de comandos do SENAO. No caso do exemplo, a frase “Aluno Aprovado!” só apareceria se a média fosse maior do que 7. Senão apareceria fase de “Aluno Reprovado!”.

Estrutura condicional composta aninhadas

A instrução condicional (SE) pode ser encadeada uma dentro da outra, seja no bloco verdadeiro (então) ou falso (senão). O encadeamento destas instruções também é chamado de aninhamento de instruções de seleção ou de decisão. Não existe um limite de quantos testes condicionais podem estar dentro um do outro, tudo vai depender do problema a ser resolvido. Só para dar um exemplo de aninhamento, temos:

```
se <condição_1> então
    <bloco de comandos 1>
senão
    se (<condição_2>) então
        <bloco de comandos 2>
    senão
        se (<condição_3>) então
            :
        fimse
    fimse
```

fimse

Um exemplo de uso desse tipo de estrutura seria:

```

Algoritmo exemplo
var idade: inteiro
inicio
    escreva("Informe sua idade: ");
    leia(idade);
    se (idade < 18) então
        escreva ("Você é de menor")
    senão se (idade >=18 e idade < 30) então
        escreva ("Você é jovem")
    senão se (idade >= 30 e idade < 60) então
        escreva ("Você é adulto")
    senão se (idade >= 60) então
        escreva("Você é velho")
    fimse
    fimse
    fimse
fimalgoritmo

```

Veja que apenas uma das condições será executada. Porque a pessoa não poderá se encaixar em dois intervalos de idades diferentes. Ou seja, as opções de IF quando encadeadas ou aninhadas são mutuamente exclusivas.

Estrutura condicional múltipla

A estrutura condicional ou de decisão múltipla é uma especialização da estrutura de decisão composta aninhada, que permite a execução de opções mutuamente exclusivas. Dessa forma, ela é equivalente a uma estrutura aninhada.

Esse tipo de instrução checa a igualdade do valor avaliado com algumas opções e executa a instrução ou bloco de instrução correspondente ao valor selecionado. A sintaxe desse comando pode ser vista a seguir.

escolha <seletor>

```

caso <exp1>  <seqüência-de-comandos-1>
caso <exp2>      <seqüência-de-comandos-2>
...
outrocaso
    <seqüência-de-comandos-extra>
fimescolha

```

Neste comando o seletor é avaliado para ver em qual dos casos ele se encaixa. É checada a igualdade do seletor com cada uma das expressões presentes em cada caso. Quando uma igualdade ocorrer, a seqüência de comandos correspondente é executada e apenas ela. Se nenhum caso atender a seleção, o trecho denominado “outro caso” é executado (ou seja, a seqüência de comandos extra). Vejamos um exemplo. Suponha um trecho de programa para selecionar o mês do ano. Sendo que 1 é janeiro, 2 é fevereiro, 3 é março e assim por diante.

escolha (opcao)

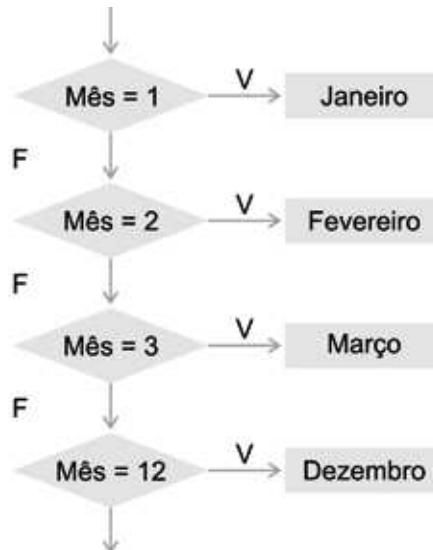
```

caso 1 mes < - "Janeiro"
caso 2 mes < - "Fevereiro"
caso 3 mes < - "Março"
caso 4 mes < - "Abril"
caso 5 mes < - "Maio"
caso 6 mes < - "Junho"
caso 7 mes < - "Julho"
caso 8 mes < - "Agosto"
caso 9 mes < - "Setembro"
caso 10 mes < - "Outubro"
caso 11 mes < - "Novembro"
caso 12 mes < - "Dezembro"

outro caso
    escreva ("Valor inválido!")

```

fimescolha



No exemplo acima, vai ser realizada uma comparação de igualdade para identificar qual opção tem o mesmo valor do seletor. Caso todas as condições sejam F, executa-se o “outro caso” (se este existir, uma vez que ele é uma parte opcional do comando) ou sai-se da estrutura sem executar nada. Se a igualdade for encontrada, vai ser executada a operação pertinente. O fluxograma ao lado do código ilustra, de forma resumida, o que acontece durante a execução. O teste dos casos é feito seqüencialmente .

Estrutura de Repetição

Adivinha pra que serve a repetição!... Isso mesmo! Para repetir uma instrução ou um bloco de instruções. Vamos tomar como base a preparação de uma xícara de café. Quais seriam os passos desse algoritmo?

1. Colocar o açúcar na xícara
2. Colocar o café na xícara (neste caso, teria de ser um café solúvel)
3. Colocar a água quente
4. Mexer
5. Mexer
6. Mexer
7. Mexer
8. Mexer
9. Mexer
10. Tomar o café

Veja que como há uma parte do algoritmo que se repete, ele poderia ser reescrito, de forma otimizada, como segue:

1. Colocar o açúcar na xícara
2. Colocar o café na xícara (neste caso, teria de ser um café solúvel)
3. Colocar a água quente

Repete 6 vezes

4. Mexer

FimRepete

5. Tomar o café

Note que a segunda versão faz a mesma coisa, porém, de maneira muito mais elegante. Nos algoritmos e programas existem situações que o processamento de um valor uma única vez não é suficiente para resolver o problema (por exemplo, a folha de pagamento de uma empresa não roda para apenas um funcionário, mas para todos os funcionários da empresa os mesmos cálculos). Quando isto ocorre, deve-se utilizar uma estrutura de repetição. Estruturas de repetição permitem que um ou mais instruções sejam executadas um número

definido de vezes, ou enquanto uma determinada condição não é alcançada. As estruturas de repetição também são conhecidas como Laços ou Loops e podem ser classificadas em dois tipos:

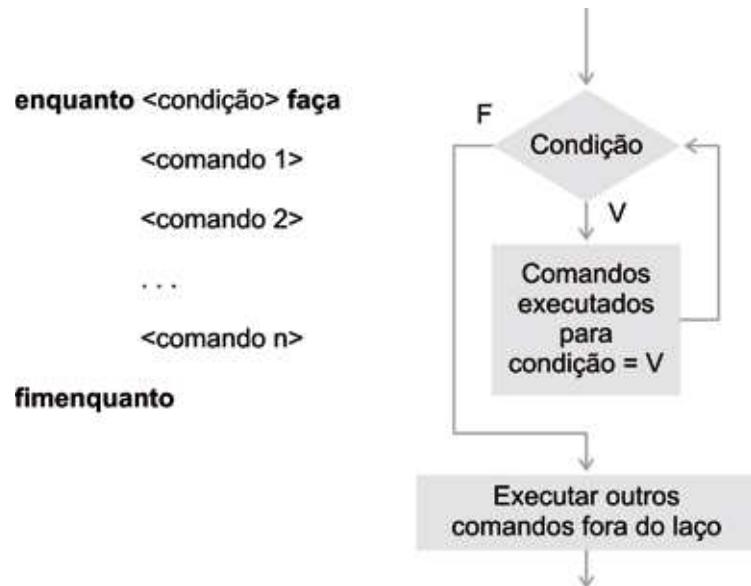
- **Condisional** – elas repetem um ou mais comandos até satisfazer a condição de repetição. São usadas quando não se sabe previamente quantas vezes deve-se executar as instruções do bloco de repetição. Esse tipo de estrutura de repetição condicional podem executar o teste da condição no início (ENQUANTO/FAÇA) ou no final (REPITA/ATÉ).
- **Contada** - repetem um número contado (pré-definido) de vezes um ou mais comandos. São usadas quando se sabe previamente quantas vezes deve-se executar as instruções do bloco de repetição. Para saber quando parar, esse tipo de estrutura de repetição utiliza uma variável contadora para controlar a quantidade de repetições. É representada pela estrutura PARA/FAÇA.

Vamos, a seguir, descrever e exemplificar cada uma dessas estruturas.

Estrutura de repetição ENQUANTO/FAÇA

Laço que verifica antes de cada execução, se é “permitido” executar o trecho do algoritmo. Ele se mantém repetindo o bloco de instruções enquanto uma dada condição permanecer verdadeira. Ou seja, faz o teste no início do laço. Se o resultado for V as instruções do laço são executadas. Depois ele volta para o início do laço e testa se a condição continua verdadeira. Se for, executa as instruções do laço denovo. E assim continua repetindo enquanto a condição testada for V. Quando a condição for falsa, o laço não será mais executado.

Essa estrutura é utilizada quando não sabemos o número de repetições e quando possuímos uma expressão que deve ser avaliada para que os comandos da estrutura sejam executados. Não se sabe de antemão quantas vezes o bloco de repetição será executado. Isto é, ele pode ser executado uma vez, várias vezes ou nenhuma vez. O formato desta estrutura pode ser vista a seguir, bem como o fluxograma que representa a sua execução.



Vamos dar um exemplo de utilização do enquanto/faça. Suponha um algoritmo para ler vários números positivos ou negativos. Esse algoritmo deve ler uma quantidade não definida de número, parando apenas quando for digitado o valor zero. Para cada número lido o algoritmo deve avaliar e escrever na tela se o número é positivo ou negativo.

1. **algoritmo** ExemploEnquantoFaça
2. **var** x: inteiro
3. **inicio**
4. **escreva**("Digite um valor")
5. **leia** (x)
6. **enquanto** ($x <> 0$) **faça**
7. **se** ($x > 0$) **então**
8. **escreva** ("O número é positivo")
9. **senão**
10. **escreva** ("O número é negativo")
11. **fimse**
12. **escreva**("Digite um valor")
13. **leia** (x)
14. **fimenquanto**
15. **fimalgoritmo**

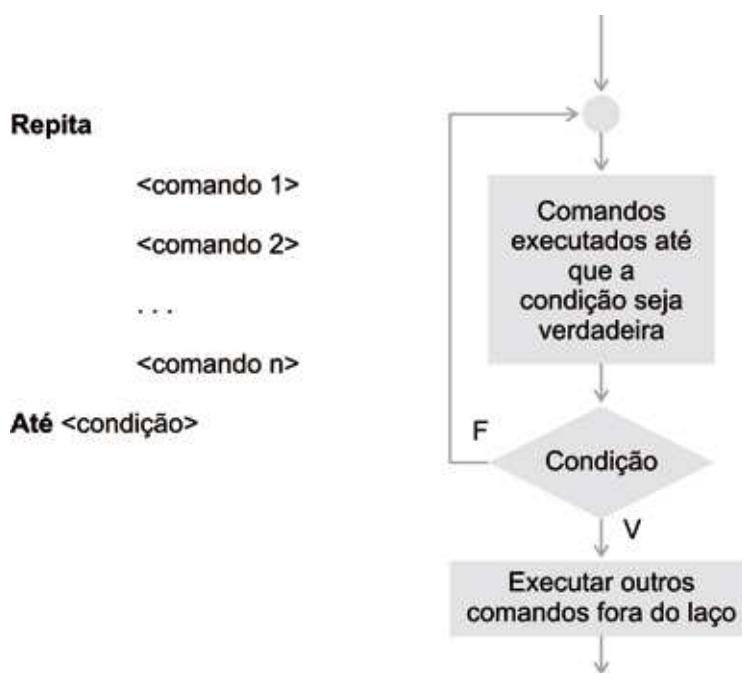
No algoritmo acima, a linha 1 define o algoritmo e o nome dele. Na linha 2 é feita a declaração da variável x que é do tipo inteiro. As linhas 3 e 15 delimitam o início e fim do algoritmo. Nas linhas 4 e 5 é feita a leitura do valor X antes de entrar no laço. Isso porque é necessário

que x tenha algum valor para poder ser testada a condição da linha 6. Se a condição for verdadeira, as linhas de 7 a 13 serão executadas. Senão, nenhuma delas o será. Caso entre no laço, as linhas de 7 a 13 serão executadas **enquanto a condição for verdadeira**. Ou seja, enquanto o valor de x for diferente de zero.

Estrutura de Repetição REPITA/ATÉ

O funcionamento desta estrutura é bastante parecido ao da estrutura enquanto. Porém os comandos contidos no interior do laço serão executados pelo menos uma vez, já que a condição de continuidade no laço só é testada no final do bloco de comandos. Ou seja, depois da execução do bloco a primeira vez, a condição é testada, caso ela seja falsa, os comandos serão executados novamente. A execução dos blocos é repetido **até que a condição (que é falsa) se torne verdadeira**.

O repita/até é utilizado quando não sabemos o número de repetições e quando os comandos devem ser executados pelo menos uma vez, antes da expressão ser avaliada. Ou seja, não se sabe de antemão quantas vezes o bloco de repetição será executado. Todavia é garantido que ele será executado pelo menos uma vez. O bloco de comandos é repetido toda vez que a condição for F. A execução do bloco é finalizada quando a condição for V. O formato desse comando está especificado a seguir.



Vamos supor o mesmo problema do exemplo da estrutura enquanto/faça. De checar se o número é positivo ou negativo. Como ficaria aquele problema usando a estrutura repita/até?

1. **algoritmo** ExemploRepitaAte
2. **var** x: inteiro
3. **inicio**
4. **repita**
5. **escreva**("Digite um valor")
6. **leia** (x)
7. **se** (x > 0) **então**
8. **escreva** ("O número é positivo")
9. **senão**
10. **escreva** ("O número é negativo")
11. **fimse**
12. **até** (x=0)
13. **fimalgoritmo**

Veja que agora, o teste da condição que define se o algoritmo continuará executando é feito no final do bloco de comandos (linha 12). Dessa forma, as linhas de 5 a 11 serão executadas ao menos uma vez, antes de chegar até a condição e, depois, só continuarão executando se a condição for falsa ($x \neq 0$). Porque se ela for verdadeira, ($x=0$) será encerrada a execução do laço, visto que é para ele executar até a condição ser verdadeira.

Toda repetição condicional pode ser representada por uma estrutura do tipo Enquanto/Faça ou Repita/Até. Todavia, perceba que a primeira repete somente quando a condição é V e a segunda repete somente quando a condição é F. Por isso, a condição de uma é a negação da outra. Nos laços condicionais a variável que é testada (no caso dos algoritmos de exemplo, a variável X), deve estar sempre associada a uma instrução que a atualize no interior do laço (no caso do algoritmo de teste, a atualização é a leitura de um novo valor para x). Pois, se isto não ocorrer, o algoritmo ficará repetindo indefinidamente este laço, gerando uma situação conhecida como “laço/loop” infinito”

1. **algoritmo** ExemploEnquantoFaça
2. **var** x: inteiro
3. **inicio**

```

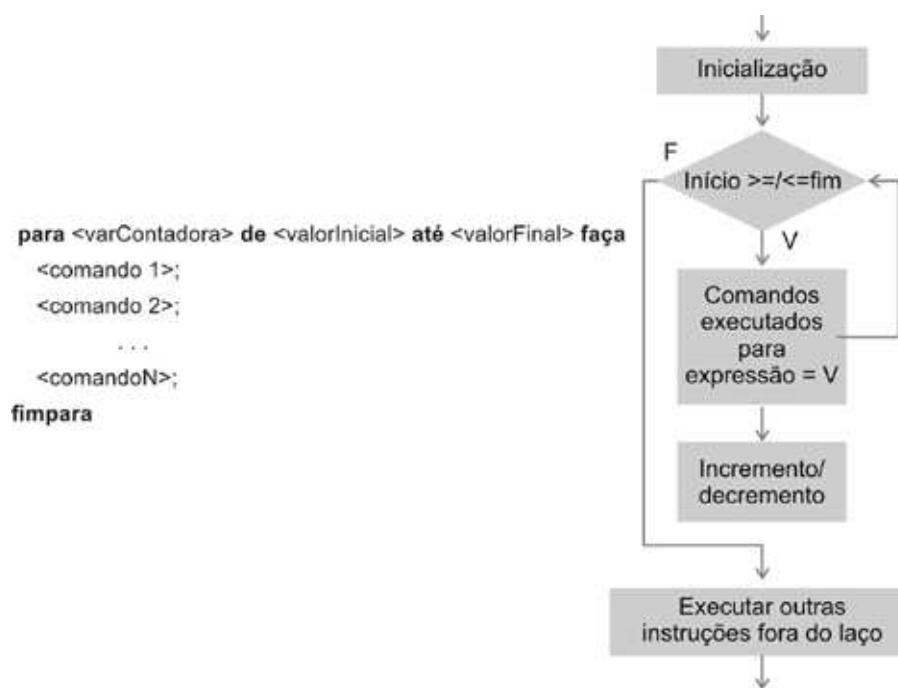
4.   escreva("Digite um valor")
5.   leia (x)
6.   enquanto (x<>0) faça
7.       se (x > 0) então
8.           escreva ("O número é positivo")
9.       senão
10.      escreva ("O número é negativo")
11.      fimse
12.      escreva("Digite um valor")
13.      leia (x)
14.  fimenquanto
15. finalgoritmo

```

O algoritmo acima entraria em loop infinito se os comandos das linhas 12 e 13 não fossem executados. Isso porque, se não fosse lido um novo valor para x, x poderia nunca chegar a ter o valor zero (condição de parada do laço), uma vez que o laço executa enquanto o x for diferente de zero.

Estrutura de Repetição PARA/FAÇA

Este tipo de estrutura é útil quando se conhece previamente o número de vezes que se deseja executar um determinado conjunto de comandos. Esta estrutura repete a execução dos comandos enquanto a variável contadora não atingir o valor final. Ressalta-se que a variável contadora é previamente inicializada (recebe um valor inicial) e é incrementada em uma unidade, automaticamente, a cada final de execução do bloco de comandos dentro do laço. Ou seja, sabe-se de antemão quantas vezes o bloco de repetição será executado, uma vez que ele irá de um valor inicial, especificado, a um valor final, também especificado e controlado pela variável de contadora. A variável contadora sempre será um número inteiro e ela não pode ser modificada dentro do bloco de comandos. O formato dessa estrutura está especificado a seguir.



Para exemplificar, suponha que gostaríamos de construir um algoritmo para ler 20 números e avaliar se esses números são positivos ou negativos, escrevendo na tela a mensagem apropriada. Considere que zero entrará na parte dos números positivos, para efeito de simplificar o algoritmo. Veja, neste algoritmo já é dito que se quer ler e avaliar 20 números. Então você já sabe o trecho do algoritmo que precisa ser executado 20 vezes. Então, este trecho ficará dentro do laço para.

- Atenção**
- O problema pediu para ler e avaliar 20 números. Logo o laço para vai de 1 até 20
1. **algoritmo** exemploParaFaca
 2. **var** num, contador: inteiro
 3. **inicio**
 4. **para** contador **de** 1 **até** 20 **faca**
 5. **escreva**("Digite um valor")
 6. **leia** (num)
 7. **se** (num \geq 0) **então**
 8. **escreva** ("O número é positivo")
 9. **senão**
 10. **escreva** ("O número é negativo")
 11. **fimse**
 12. **fimpara**
 13. **fimalgoritmo**

A variável contador, neste algoritmo chamada de contador, começa com o valor inicial 1 e vai até o valor final 20 (especificados

na linha 4) e a cada execução do bloco de comandos dentro do laço (da linha 5 a linha 11) a variável contadora é incrementada e o laço checa se ela ainda é menor ou igual ao valor final. Se for, a execução dos comandos dentro do laço continua, se não for, a execução do laço para.

É possível implementar um laço com contador, usando uma das estruturas de repetição condicionais (enquanto/faça ou repita/até). Porém, o controle do incremento da variável contadora passa a ser seu, não será mais feito automaticamente (tal qual é feito no laço para). Vejamos o exemplo a seguir, trocando a estrutura para/faça por enquanto/faça.

1. **algoritmo** exemploContadorEnquanto
2. **var** num, contador: inteiro
3. **inicio**
4. **contador <-1** ←
5. **enquanto** (contador <= 20) **faça**
6. **escreva**("Digite um valor")
7. **leia** (num)
8. **se** (num >= 0) **então**
9. **escreva** ("O número é positivo")
10. **senão**
11. **escreva** ("O número é negativo")
12. **fimse**
13. **contador <- contador + 1** ←
14. **fimpara**
15. **fimalgoritmo**

Atenção

Agora é preciso dar um valor inicial ao contador

Atenção

A variável contadora passa a necessitar ser incrementada por você. Antes o laço para executava esse passo, automaticamente.

Uma comparação que resumo tudo que foi apresentado até agora sobre as estruturas de repetição pode ser vista na Tabela 7.

	Enquanto/Faça	Repita/Até	Para/Faça
Tipo da Estrutura	Condisional	Condisional	Contada
Quantidade de Repetições	Zero ou mais vezes (pode não ser executada nenhuma vez)	Uma ou mais (vai ser executada, no mínimo, uma vez)	Vai ser executada do valor inicial ao valor final, ou seja, terá um número de execuções pré-definido

Tabela 7 - Comparação entre as estruturas de repetição

Estruturas de Repetição Aninhadas ou Encadeadas

Semelhante às estruturas de decisão composta, as estruturas de repetição também podem ser encadeadas/aninhadas. Esta abordagem é usada quando há a necessidade de se usar laços dentro de laços. Isso será bastante utilizado no Módulo II, quando formos falar de tipos estruturados de dados. Por isso, deixaremos para detalhar esse ponto, mais a frente. Porém, para que você tenha uma idéia do que estamos falando, vamos dar um exemplo.

Suponha que você precise fazer um algoritmo para gerar a tabuada da multiplicação de 1 a 10. Lembra dela? Acho que todos nós já passamos por ela (vide Figura 27).

Se você prestar atenção à tabuada a primeira coluna de cada bloco de números é sempre o mesmo número. Exemplo, a primeira coluna da tabuada do 1 é toda de número 1, a primeira coluna do bloco da tabuada do 2 é toda de número 2 e assim por diante. Já a segunda coluna de cada bloco, sempre varia de 1 a 10. Dê uma olhada na Figura 27 e veja isso. Viu?

Dessa forma, para gerar essa tabuada, vamos precisar de um laço que tem de ficar no mesmo número, até que se gerem os números

de um a dez. Vamos ver como ficaria esse algoritmo e vamos depois explicar mais detalhes.

$1 \times 1 = 1$	$3 \times 1 = 3$	$5 \times 1 = 5$
$1 \times 2 = 2$	$3 \times 2 = 6$	$5 \times 2 = 10$
$1 \times 3 = 3$	$3 \times 3 = 9$	$5 \times 3 = 15$
$1 \times 4 = 4$	$3 \times 4 = 12$	$5 \times 4 = 20$
$1 \times 5 = 5$	$3 \times 5 = 15$	$5 \times 5 = 25$
$1 \times 6 = 6$	$3 \times 6 = 18$	$5 \times 6 = 30$
$1 \times 7 = 7$	$3 \times 7 = 21$	$5 \times 7 = 35$
$1 \times 8 = 8$	$3 \times 8 = 24$	$5 \times 8 = 40$
$1 \times 9 = 9$	$3 \times 9 = 27$	$5 \times 9 = 45$
$1 \times 10 = 10$	$3 \times 10 = 30$	$5 \times 10 = 50$
	$7 \times 1 = 7$	$9 \times 1 = 9$
	$7 \times 2 = 14$	$9 \times 2 = 18$
	$7 \times 3 = 21$	$9 \times 3 = 27$
	$7 \times 4 = 28$	$9 \times 4 = 36$
	$7 \times 5 = 35$	$9 \times 5 = 45$
	$7 \times 6 = 42$	$9 \times 6 = 54$
	$7 \times 7 = 49$	$9 \times 7 = 63$
	$7 \times 8 = 56$	$9 \times 8 = 72$
	$7 \times 9 = 63$	$9 \times 9 = 81$
	$7 \times 10 = 70$	$9 \times 10 = 90$

Figura 27- A nossa tabuada de cada dia

1. **algoritmo** tabuada
2. **var** coluna1, coluna2, resultado: inteiro
3. **inicio**
4. **para** coluna1 **de** 1 **até** 10 **faça**
5. **para** coluna2 **de** 1 **até** 10 **faça**
6. resultado <- coluna1 * coluna2
7. **escreval**(coluna1, " x ", coluna2, " = ", resultado)
8. **fimpara**
9. **fimpara**
10. **fimalgoritmo**

Atenção

Controla o valor da primeira coluna da tabuada

Atenção

Controla o valor da segunda coluna da tabuada. Varia o valor da variável coluna2, enquanto o valor de coluna1 continua o mesmo

Neste algoritmo, quando o primeiro laço começa (linha 4), o valor da coluna1 fica fixo, até que se volte para executar esse laço denovo. Porém, isso só vai ocorrer, quando laço mais interno (da linha 5 a linha 8) terminar. Esse laço mais interno é o encarregado de gerar os valores de um a dez da segunda coluna de cada bloco da tabuada. Dessa forma, por exemplo, no comando da linha 6, o valor de coluna1 continua o mesmo dentro desse laço mais interno e o valor da coluna 2 vai mudando. Seria algo como mostrado na Tabela 8.

Coluna 1	Coluna 2	Coluna 1	Coluna 2

Tabela 8 - Exemplo de valores gerados pelos laços aninhados

Dicas para Escrever Bons Algoritmos

Bom, agora, a essas alturas, você já tem todas as instruções necessárias para construir vários tipos de algoritmos. Porém, vamos dar ainda algumas dicas para lhe ajudar a escrever realmente bons algoritmos, que sejam claros, fáceis de entender e de dar manutenção.

1. Procure não escrever mais do que um comando por linha
2. Depois de ler o enunciado do problema, tente identificar sempre as três fases: ENRADA, PROCESSAMENTO e SAÍDA. Isso lhe ajuda a esquematizar o algoritmo.
3. Para lembrar, posteriormente, de trechos de programas mais complicados que você pode esquecer como implementou, escreva comentários claros e objetivos no momento em que estiver escrevendo o algoritmo. Isto também facilita o entendimento do algoritmo. Porém, cuidado, comentários deverão acrescentar alguma coisa; não apenas descrever instruções. Instruções – dizem o que está sendo feito e os Comentários – dizem o porquê de está sendo feito. Por exemplo, para o comando lucro ← venda – custo, um comentário ruim seria “Este comando atribui a lucro o valor de venda menos o valor de custo” (isso está escrito, não precisa comentar). Um bom comentário seria: “Aqui é calculado o lucro total obtido com a venda dos produtos”. Uma linha de comentário deve ser iniciada com a marca // (duas barras). A linha iniciada com essa marcação não será traduzida pelo tradutor, serve apenas para informação ao programador e

documentação do programa/algoritmo.

4. Use comentários no início do algoritmo. Estes podem ter, entre outras informações, a descrição do que o algoritmo faz, o nome do autor do algoritmo, a data de implementação, etc. Por exemplo:

Algoritmo Tabuada

```
// Esse algoritmo gera a tabuada de multiplicação de 1 a 10.  
// Autora: Sandra de Albuquerque Siebra  
// Data: 20/08/2008  
var coluna1, coluna2, resultado: inteiro  
início ....
```

5. Escolha nomes de variáveis que sejam significativos. Evite variáveis com nomes do tipo: x, y, z, lm. Prefira nomes que expressem o valor que você vai guardar dentro da variável. Por exemplo: precoProduto, lucro, nome, idade, etc.
6. Procure alinhar os comandos de acordo com o nível a que pertençam. Isto é, destaque a estrutura na qual estão contidos. Este alinhamento é chamado de identação. Identação é a organização hierárquica das estruturas e suas instruções. Facilita visualizar o que está contido em que. Ela auxilia no entendimento do código e na busca de erros. Para ilustrar, qual dos dois trechos de algoritmo abaixo você acha mais fácil de entender? O da esquerda ou o da direita? (o da direita, está identado!)

```

algoritmo ExemploEnquantoFaça
var x: inteiro
inicio
escreva("Digite um valor")
leia (x)
enquanto (x<>0) faça
se (x > 0) então
escreva ("O número é positivo")
senão
escreva ("O número é negativo")
fimse
escreva("Digite um valor")
leia (x)
fimenquanto
fimalgoritmo

```

```

algoritmo ExemploEnquantoFaça
var x: inteiro
inicio
escreva("Digite um valor")
leia (x)
enquanto (x<>0) faça
se (x > 0) então
escreva ("O número é positivo")
senão
escreva ("O número é negativo")
fimse
escreva("Digite um valor")
leia (x)
fimenquanto
fimalgoritmo

```



Depois de ter visto vários tipos de instruções da linguagem algorítmica e ter inclusive lido algumas dicas para construção de algoritmos. E agora? Bem, vamos mostrar dois exemplos passo a passo de dois tipos de algoritmos para que, depois, você mesmo possa fazer os seus exercícios.

Problema 1: Construa um algoritmo para ler o nome, número de horas trabalhadas e número de dependentes de cada um dos 20 funcionários de uma empresa. Após a leitura, para cada funcionário, calcule e imprima na tela: o nome do funcionário, seu salário bruto, o total de descontos que ele sofrerá e seu salário líquido. Considere que:

- A empresa paga R\$10 por hora trabalhada e tem um adicional de R\$40 por dependente que o funcionário tenha.
- O salário bruto é o valor total dos ganhos do funcionário
- Sobre o salário bruto são descontados 8 % de INSS e 5% de IR (imposto de renda).
- O salário líquido é o valor do salário bruto, menos os descontos.

Agora, mãos à obra! Primeiro, faça a leitura do problema. Depois leia o problema tentando identificar qual a entrada, o processamento e a saída. Nisso temos:

ENTRADA: o nome, o número de horas trabalhadas e o número de dependentes de cada funcionário

PROCESSAMENTO: calcular o salário bruto, os impostos e o valor do salário líquido de 20 funcionários (veja, sabemos quantos funcionários são!)

SAÍDA: o nome do funcionário, seu salário bruto, os descontos e o valor do salário líquido.

O que mais podemos saber da leitura? Vamos precisar usar estrutura de repetição, porque precisamos fazer o mesmo processamento para 20 funcionários. Vamos precisar de uma variável para cada valor que queremos ler na entrada e para cada coisa que precisamos calcular no processamento. Vamos lá! Sabendo disso tudo, como ficaria o algoritmo? (lembre que as linhas iniciadas com // são comentários

```
=====

// você poderia dar qualquer nome ao algoritmo, eu resolvi chamar ele de calcula
Algoritmo calcula
    // aqui são feitas as declarações das variáveis. Lembre que vamos precisar de uma variável
    // contadora para o laço PARA/FAÇA (essa eu chamei de contador). Fora isso, uma variável
    // para cada valor que precisa ser lido (nome, horasTrab, dependentes) e uma variável para
    // cada valor a ser calculado (salarioBruto, salarioLiquido, inss e ir)

    Var nome: caractere
        horasTrab, salarioBruto, salarioLiquido, inss, ir: real
        dependentes, contador: inteiro

    inicio
        // como são 20 funcionários, precisaremos ler e calcular tudo 20 vezes.
        // Por isso, usamos um laço para/faça.
        para contador de 1 até 20 faça
            // a primeira coisa a fazer é pedir ao usuário para digitar os valores de entrada
            escreva ("*****")
            escreva ("Digite o nome do funcionário: ")
            leia(nome)

            escreva ("Digite a quantidade de horas trabalhadas: ")
            leia(horasTrab)
            escreva("Digite o número de dependentes do funcionário: ")
```

leia(dependentes)

```
// com os valores lidos, podemos calcular agora o salário bruto do funcionário
// multiplicando o valor em reais de cada coisa, pela sua quantidade
salarioBruto <- (10 * horasTrab) + (40 * dependentes)
// o INSS é 8% do salário bruto
inss <- (8*salarioBruto)/100
// o IR é 5% do salário bruto
Ir <- (5*salarioBruto)/100
// o salário líquido é o salário bruto subtraído dos descontos dos impostos
salarioLiquido <- (salarioBruto - inss) - ir
// depois de tudo calculado, imprimimos os resultados (poderia fazer tudo em um
// escreval só, fiz em mais de um só para organizar as impressões
// lembre escreval serve para imprimir na tela e depois pular uma linha)
escreval ("Nome: ", nome)
escreval ("Salário Bruto: R$ ", salarioBruto)
escreval ("Descontos: R$ ", (ir+inss))
escreval ("Salário Líquido: R$ ", salarioLiquido)
fimpara
```

fimalgoritmo

```
=====
```

Problema 2: Faça um algoritmo para ler uma quantidade indefinida de bois (código do boi e peso do boi), parando quando for digitado o código zero. Depois, calcule e imprima o código e o peso do boi mais gordo da fazenda. Considere, para simplificar, que não há dois pesos iguais. Neste problema temos: ENTRADA: o código e o peso do boi

PROCESSAMENTO: calcular qual o boi mais gordo

SAÍDA: o código e o peso do boi mais gordo

O que mais podemos saber da leitura? Vamos precisar usar estrutura de repetição. Porém, não pode ser um laço para/faça, porque vamos ler um número indefinido de bois. Mas, nós temos uma condição de parada (código do boi igual a zero). Dessa forma, podemos usar ou um laço enquanto/faça ou um laço repita/até. Vamos escolher esse último, porque o algoritmo teria de executar ao menos uma vez.

```
=====
```

Algoritmo pesado

```
// Vamos precisar de uma variável para cada valor que precisa ser lido (codBoi, peso)
```

```

// e uma variável para guardar o que vai ser calculado codGordo, pesoGordo.

Var codBoi, codigoGordo: inteiro
    peso, pesoGordo: real

inicio
    // antes de entrar no laço, temos de dar um valor inicial (inicializar) as variáveis
    // que vamos usar para os cálculos. Como não temos ainda o código do boi mais gordo
    // inicializamos o código do mais gordo com zero. E, como queremos descobrir o peso
    // do boi mais gordo, começamos o peso com zero também. Para que qualquer que seja
    // o peso do primeiro boi lido, ele já possa substituir esse valor zero, como você verá
    // abaixo.

    codigoGordo <- 0
    pesoGordo <- 0

repita
    // assim que entramos no repita, vamos ler os valores de entrada
    escreval("*****")
    escreva ("Digite o código do boi: ")
    leia(codBoi)
        // só continuo o algoritmo se o código lido for diferente de zero. Isso evita que o
        // primeiro código digitado seja zero e você tenha que executar o algoritmo todo,
        // antes de sair do laço – porque o teste do laço só é no final.

    se (codBoi <> 0) entao
        escreva ("Digite o peso do boi: ")
        leia(peso)
            // agora vamos usar uma estrutura condicional para tomar a decisão se esse boi
            // que acabou de ser lido é mais gordo do que o que já temos armazenado na
            // variável pesoGordo

        se (peso > pesoGordo) entao
            // se o peso do boi que foi lido for maior do que o peso que estava
            // armazenado, você deve substituir o valor armazenado, pelos valores
            // que acabaram de ser lidos.

            pesoGordo <- peso
            codigoGordo <- codBoi

    fimse

fimse
    // esse algoritmo continua até que a condição de parada (codBoi =0) seja satisfeita
    até (codBoi = 0)
    // quando sai do laço, deve imprimir o resultado
    escreval("O boi mais gordo é o de código: ", codigoGordo, " que pesa ", pesoGordo)

fimalgoritmo

```



Desafio

O que mudaria se que eu quisesse saber qual era o boi mais magro???



Conheça Mais

O assunto dessa unidade é linguagem algorítmica e qualquer livro de algoritmos ou de lógica de programação terá esse assunto (na verdade um ou mais capítulos sobre o assunto). Sugiro, então, a leitura dos capítulos 2 e 3 do livro-texto da disciplina: FORBELLONE, André Luiz. Lógica de Programação. Makron Books, 3^a. Edição, 2005.

Outro livro que pode ser consultado é o: FARRER, Harry; BECKER, C. G.; FARIA, E.C.; MATOS, H.F.; SANTOS, M.A.; MAIA, M.L. Algoritmos Estruturados. Editora LTC, 3^a. Edição, 1999.

Outra fonte de pesquisa pode ser a internet. Você pode consultar por “Linguagem Algorítmica” que virão várias dicas de apostilas, tutoriais, apresentações e sites. Além da própria página do VisuAlg: <http://www.apoioinformatica.inf.br/visualg/linguagem.htm>



Atividade de Estudo

Construa algoritmos em linguagem algorítmica para resolver os problemas abaixo.

Usando só estrutura sequencial

- 1) Leia o nome e o salário de um funcionário e exiba com resultado o nome do mesmo com o salário reajustado em 12%.
- 2) Leia 2 notas, calcule e exiba a média ponderada dessas notas. Considere que a nota1 tem peso 6 e a nota2 tem peso 4.
- 3) Faça um algoritmo que receba um valor em dólar e transforme o mesmo em reais, sabendo que a taxa de conversão é: 1

dólar = R\$ 1.70.

- 4) Leia uma distância em km entre dois pontos e o preço da gasolina em reais. Depois, exiba quantos litros de gasolina o carro irá consumir e quanto será o gasto em reais, sabendo que o carro em questão consegue percorrer 12 km com um litro de gasolina.
- 5) Leia as variáveis inteiros n1 e n2 e troque o valor destas variáveis. Isto é, n1 deve ficar com o valor de n2 e n2 deve ficar com o valor de n1. Exiba os valores antes e depois da troca.

Usando só estrutura de decisão

- 6) Leia um número inteiro e mostre uma mensagem indicando se este número é par ou ímpar (pra calcular se é par ou ímpar, use a função MOD).
- 7) Leia 3 notas, calcule a média aritmética dessas e, de acordo com o resultado, escreva como saída o valor da média e a mensagem apropriada: Reprovado (média < 4), Em Prova Final (média ≥ 4 e < 7) e Aprovado (média ≥ 7).
- 8) Leia a idade de um nadador e exiba sua categoria segundo as regras: Junior (5 até 7 anos); Infantil (8 até 12 anos); Pré (13 até 18) e Avançada (Idade > 18)
- 9) Desenvolva um algoritmo que funcione como uma calculadora, com as quatro operações básicas: adição, subtração, multiplicação e divisão. Para isso, deve ser pedido ao usuário o primeiro número, o sinal da operação e o segundo número. Dê uma mensagem se o sinal digitado não for um dos quatro presentes na calculadora e lembre que não existe número divisível por zero.
- 10)Leia três números inteiros e mostre-os em ordem crescente.
- 11)Leia um número inteiro entre 20 e 39 e escreva seu valor por extenso. Exiba uma mensagem apropriada se o número estiver fora desse intervalo.
- 12)Leia um número inteiro e positivo, calcule e imprima o resultado do seu fatorial. Lembre que o fatorial de um número N qualquer é igual a $1 \times 2 \times 3 \times \dots \times N - 1 \times N$. Ex: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$. Lembre também que $0! = 1$.

Usando estrutura de repetição (e talvez, outras estruturas)

12)Leia o nome e a idade de várias pessoas até que uma idade igual a ZERO seja digitada. E, ao final, informe o nome da pessoa mais velha e o nome da pessoa mais nova, entre todos os nomes lidos. Para simplificar, considere que não existem idades iguais.

13)Escreva um algoritmo para calcular e imprimir a soma dos números inteiros e pares de 1 a 50.

14)Um órgão de pesquisa deseja efetuar um mapeamento das condições econômicas das pessoas de uma determinada região e efetuou uma pesquisa perguntando Nome, idade, sexo, se trabalha, e quantos salários mínimos recebem por mês. Elaborar um algoritmo para determinar:

- A quantidade de pesquisas realizadas por sexo (quantos homens e quantas mulheres foram entrevistados)
- A quantidade pessoas menores de 18 anos que trabalham
- A renda média dos homens
- A renda média das mulheres
- Nome e Renda da pessoa que trabalha e que tem a menor renda
- Quantidade de pessoas que não trabalham
- A média de idade dos entrevistados

Considere:

Nome = “FIM” indica que já foram lidos todos as pesquisas ou seja, é a condição de parada do algoritmo.

Sexo vai ser informado como “M” ou “F”;

Idade deverá estar entre 00 e 100 anos

Se trabalha deverá receber um valor lógico, ou seja, V ou F (verdadeiro ou falso)

15)Escreva um programa que apresente a série de Fibonacci até o décimo quinto termo. A série de Fibonacci é formada pela seqüência: 1, 1, 2, 3, 5, 8, 13, 21, 34, ... etc. Consegue ver a lógica de criação da mesma? Pense um pouquinho, não é tão

difícil assim!

16) Faça um algoritmo para calcular e escrever a soma da seguinte série:

$$\text{Soma} = (X^{25} / 1) - (X^{24} / 2) + (X^{23} / 3) - (X^{22} / 4) + \dots + (X / 25)$$

17) Uma empresa de compras através da Internet decidiu realizar uma pesquisa entre os seus clientes para determinar o perfil dos mesmos. Considerando uma amostra de 200 clientes, faça um algoritmo que leia o nome, a idade de seus clientes e se eles têm acesso à Internet e forneça como resultado:

- A média de idade dos clientes;
- A quantidade de mulheres;
- Quantas pessoas têm acesso a Internet.
- O nome e a idade do homem mais velho que tem acesso a Internet
- O nome e a idade da mulher mais nova.



Resumo



A linguagem algorítmica é uma das formas de representação de algoritmos mais utilizadas. Porém, ela não é padronizada. Logo, adotamos na nossa disciplina a linguagem algorítmica utilizada dentro da ferramenta VisuAlg, que usaremos depois para executar nossos algoritmos.

Para começar a construir o algoritmo em linguagem algorítmica precisamos sempre identificar as três fases de um programa: entrada, processamento e saída.

Na fase de entrada, as informações (entradas) são armazenadas

em variáveis. As variáveis podem ser de quatro tipos diferentes: inteiro (para valores numéricos sem parte decimal), real (para valores numéricos com parte decimal), caracter (para letras, números e símbolos) e lógico (que só pode assumir valor verdadeiro ou falso). Nós podemos colocar valores diretamente nas variáveis pelo comando de atribuição ou podemos fazer a leitura dos dados usando o comando Leia(nomevariável). Esse comando pega valores do teclado, leitores de código de barras, etc e armazena nos espaços da memória do computador, nomeados pelas variáveis.

É na fase de processamento que são realizados os cálculos dos algoritmos. Por isso, nessa parte você vai fazer uso de operadores e expressões. Os operadores podem ser do tipo aritmético (+, -, *, /, \, MOD, ^), relacional (<, >, <=, >=, <>, =), lógico (não, e, ou), de atribuição (<-) e de caracter (+ significando concatenação de valores).

A parte de processamento pode fazer uso de:

- estruturas seqüenciais - onde um comando vem atrás do outro e eles são executados, sequencialmente, seguindo a ordem de aparição.
- estruturas de decisão ou condicionais, que podem ser de quatro tipos: condicionais simples (se-então), condicionais compostas (se-então-senão), condicionais aninhadas (quando colocamos estruturas condicionais, dentro de estruturas condicionais) e de decisão múltipla (escolha)
- estruturas de repetição – que podem ser de dois tipos, condicionais ou contada. As condicionais podem realizar o teste no início (enquanto/faça) ou no final (repita/até). Elas costumam ser usadas quando não se sabe quantas vezes será preciso repetir um determinado bloco de comandos. A estrutura de repetição contada é usada quando se sabe o número de repetições e é representada pelo comando para/faça.

Qualquer uma das estruturas podem ser utilizadas juntas, colaborando para a resolução do problema.

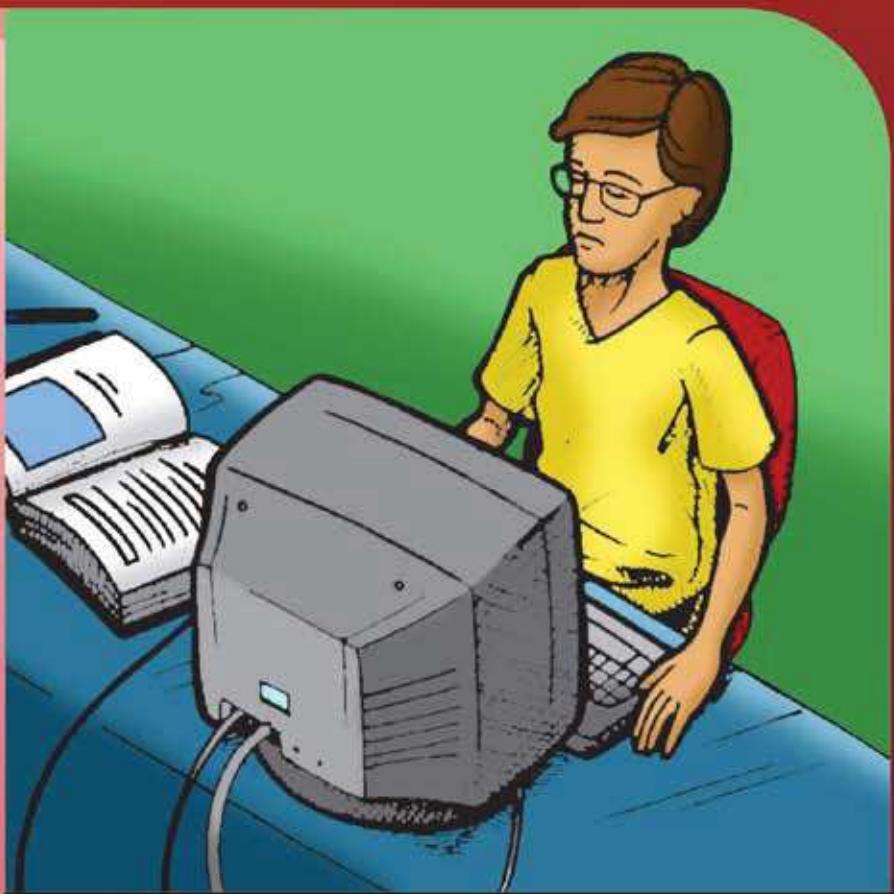
Já na fase de SAÍDA, geralmente, se imprimem valores ou mensagens na tela. Para imprimir variáveis, textos ou as duas coisas na tela, se usa o comando escreva("texto", variável).

No próximo módulo vamos estudar os tipos estruturados de

dados. Eles servem para agrupar dados do mesmo tipo ou de tipos diferentes em conjuntos. E isso vai lhe dar muito mais “poder de fogo” na construção dos algoritmos.

Introdução à Programação

Sandra de Albuquerque Siebra



FASCÍCULO 3



Universidade Federal Rural de Pernambuco

Reitor: Prof. Valmar Corrêa de Andrade

Vice-Reitor: Prof. Reginaldo Barros

Pró-Reitor de Administração: Prof. Francisco Fernando Ramos Carvalho

Pró-Reitor de Extensão: Prof. Paulo Donizeti Siepierski

Pró-Reitor de Pesquisa e Pós-Graduação: Prof. Fernando José Freire

Pró-Reitor de Planejamento: Prof. Rinaldo Luiz Caraciolo Ferreira

Pró-Reitora de Ensino de Graduação: Profª. Maria José de Sena

Coordenação de Ensino a Distância: Profª Marizete Silva Santos

Produção Gráfica e Editorial

Capa e Editoração: Allyson Vila Nova e Rafael Lira

Revisão Ortográfica: Ivanda Martins

Ilustrações: Allyson Vila Nova

Coordenação de Produção: Marizete Silva Santos



Sumário

Fascículo 3 - Conceitos Avançados sobre Algoritmos	5
Objetivos do Módulo 3	5
Conteúdo Programático do Módulo 3	6
Unidade 3 – Ambientes de Programação Algorítmica.....	7
O VisuAlg	8
Testes de Algoritmos (Teste de Mesa).....	9
Unidade 4 –Estruturas de Dados Homogêneas	13
Estruturas Homogêneas Unidimensionais (Vetores).....	15
Estruturas Homogêneas Bidimensionais (Matrizes).....	19
Unidade 5 – SubAlgoritmos	25
Procedimentos.....	28
Passagem de Parâmetros	31
Escopo de Variáveis	38
Funções	40
Resumo	53

Fascículo 3 - Conceitos Avançados sobre Algoritmos

Carga horária do Módulo: 15 horas

Objetivos do Módulo 3

- Apresentar uma ferramenta para compilação de algoritmos
- Apresentar conceitos sobre testes de algoritmos
- Apresentar conceitos relacionados a estrutura de dados homogênea unidimensional (vetor)
- Apresentar conceitos relacionados a estrutura de dados homogênea multidimensional (matrizes)
- Desenvolver o conceito de programação modular
- Implementar programas que fazem uso de subalgoritmos

Conteúdo Programático do Módulo 3

Unidade 3 – Ambiente de Programação Algoritmica

- O VisuAlg
- Testes de Algoritmos (Teste de Mesa)

Unidade 4 – Tipos Estruturados de Dados

- Estrutura de Dados Homogênea: array unidimensional (vetor) e array bidimensional (matriz)

Unidade 5 – Subalgoritmos

- Procedimentos
- Funções
- Passagem de parâmetros (por valor e por referência)

Cada unidade conterá exercícios para fixação do assunto apresentado, assim como indicações de leituras adicionais. A Unidade 3 exigirá 3 horas de sua dedicação, a Unidade 4 cerca de 5 horas e a Unidade 5 mais 7 horas.

Unidade 3 – Ambientes de Programação Algorítmica

Para amenizar a vontade de sair da “programação no papel” que a maior parte dos alunos de Introdução à Programação experimenta, sugerimos nesta etapa inicial da disciplina, uma vez que você já sabe criar um algoritmo básico, o uso de uma ferramenta de compilação algorítmica.

O uso de uma ferramenta como essas permite aos alunos iniciantes em programação, o exercício dos seus conhecimentos em um ambiente próximo da realidade da programação convencional. E isso pode ser bem interessante! Pois, nos anos que ensino a disciplina de Introdução à Programação (há mais de dez anos) notei que, para o aluno, fica muito abstrato programar “no papel”. Por outro lado, submetê-lo “de cara” aos rigores do uso de uma ferramenta para programação em uma linguagem como C, por exemplo, aumentaria em muito a complexidade do aprendizado. Dessa forma, acho recomendável, fazer uso de um ambiente integrado, onde o aluno tenha à disposição ferramentas como: editor para digitação do algoritmo; tradutor (compilador ou interpretador), a fim de converter o algoritmo para linguagem de máquina e executá-lo e, se necessário, depurar erros é de extrema valia para o aprendizado; podendo, assim, amenizar a “sede” de usar uma ferramenta de programação.

Existem várias ferramentas desse tipo que foram implementadas em universidades brasileiras. Algumas das que encontramos com maior destaque foram:

- Portugol/Plus - <http://www.c5.cl/ieinvestiga/actas/ribie98/118.html>
- G-Portugol - http://gpt.berlios.de/manual_nodes/manual.html
- ILA - Interpretador de Linguagem Algorítmica http://www.exatec.unisinos.br/_professores/gerador.php?professor=crespo&id_menu=434&id_submenu=189
- WebPortugol - <http://www.univali.br/webportugol>
- AlgoMais - http://www.brasilacademic.com/maxpt/links_goto.asp?id=808

- Hall - <http://www.paim.pro.br/hall/>
- VisuAlg - <http://www.apoioinformatica.inf.br/download.htm>

Infelizmente, não há padronização entre a linguagem algorítmica utilizada por qualquer uma delas, fazendo com que fosse preciso optar por adotar apenas uma. Dessa forma, optamos por indicar o uso da ferramenta VisuAlg. A limitação é que a mesma executa apenas em ambiente Windows. Porém, é uma ferramenta gratuita, estruturada, com ambiente integrado para edição do algoritmo, interpretação e execução, tem facilidades para depuração de erros, uma interface simples e o arquivo de instalação é pequeno, podendo ser baixado inclusive por quem tem internet discada. A seguir daremos mais detalhes sobre essa ferramenta.

O VisuAlg

O VisuAlg é uma ferramenta simples, que não depende de bibliotecas, máquinas virtuais ou outros componentes. Sua instalação não copia arquivos para nenhuma outra pasta a não ser aquela em que ela for instalada e exige cerca de 1 MB de espaço em disco. Pode ser executado sob Windows 95 ou posterior e tem melhor aparência com resolução de vídeo de 800x600 ou maior. Os detalhes sobre a ferramenta, seus objetivos, desenvolvimento, linguagem algorítmica utilizada e explicação das telas, podem ser encontrados em:

<http://www.apoioinformatica.inf.br/visualg/objetivos.htm> Reserve um tempo para dar uma olhada nesse link, pois ele contém informações detalhadas sobre a ferramenta para que você possa utilizá-la da melhor maneira. Além do material on-line do link acima, no ambiente está disponível o manual do VisuAlg, bem ilustrado e detalhado. E a linguagem algorítmica utilizada pelo VisuAlg? Bem, não precisa se preocupar, pois ela é exatamente igual à sintaxe dos comandos que foram apresentados até agora.

Para baixar a ferramenta, use o link <http://www.apoioinformatica.inf.br/download.htm>. Nele escolha a opção VisuAlg. Vai iniciar o download de um arquivo chamado setupv2.exe. Depois que baixar esse arquivo, basta clicar duas vezes sobre o nome dele e o VisuAlg será instalado. Um ícone para ativação do VisuAlg (Figura 1) será criado na área de trabalho, após a instalação. Clicando neste ícone, o ambiente é aberto (Figura 2) e você já pode começar a digitar o seu algoritmo,

aproveitando o esqueleto de código que já vem dentro da janela do editor.



Atalho para Ativação do VisuAlg

No mais, algumas observações importantes para usar a ferramenta são: ela permite apenas um comando por linha. Na versão atual do VisuAlg (2.0), com exceção das rotinas de entrada e saída, não há nenhum subalgoritmo embutido, calculo de raiz quadrada, incremento, etc. Todas as palavras-chave do VisuAlg foram implementadas sem acentos ou cedilha. Portanto, o tipo de dados lógico é definido como logico, o comando se..então..senão é definido como se..entao..senao, e assim por diante. O VisuAlg, também, não distingue maiúsculas e minúsculas no reconhecimento de palavras-chave e nomes de variáveis.

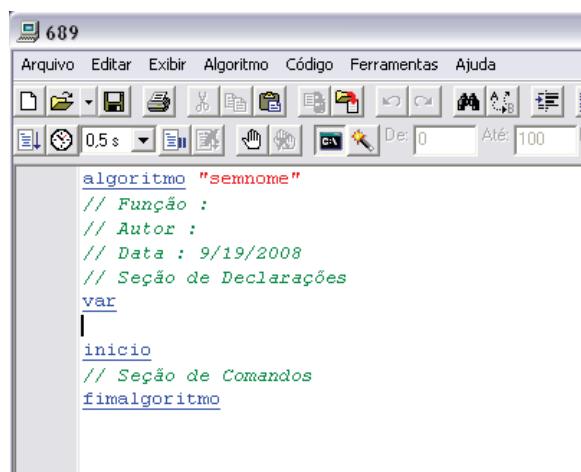


Figura 2- Tela do VisuAlg

Para praticar, você pode usar o VisuAlg para implementar alguns dos algoritmos resolvidos no Fascículo 1.

Testes de Algoritmos (Teste de Mesa)

Para testar se as funcionalidades implementadas em um algoritmo estão corretas é necessário testar o algoritmo, verificando o conteúdo das variáveis passo a passo. Para efetuar esta tarefa, costuma-se

utilizar o chamado Teste de Mesa ou Teste Exaustivo. Realizar este teste significa seguir passo a passo as instruções do algoritmo, de maneira precisa, para verificar se o que foi implementado está correto ou não, a partir dos resultados gerados e dos valores parciais de cada variável. Este teste permitirá que o programador visualize o comportamento de todo o processo, cheque se o algoritmo está correto e corrija eventuais erros, se existirem. Em Informática, dá-se o nome de “bugs” aos erros de um programa. O processo de identificação e correção dos erros denomina-se “debugging”. Os erros podem ser dos seguintes tipos:

- *Erros Sintáticos* - ocorrem quando as instruções do programa não são escritas de acordo com a sintaxe da linguagem sendo usada. Por exemplo: se eu esquecesse um **fimse** de um comando **se**, o comando estaria incompleto. Não estaria definido conforme a sintaxe da linguagem. Por isso, ocorreria o erro sintático. Este tipo de erro impede a execução do algoritmo ou programa.
- *Erros Semânticos (ou lógicos)* - ocorrem quando o código escrito pelo programador não gera o resultado desejado por este. Ou seja, o código está sintaticamente correto, mas o resultado gerado pelo algoritmo não está correto. Erros de lógica podem ser simplesmente uso incorreto de operadores (+ no lugar de -, usar o operador OU no lugar de usar o E), atribuições erradas (por exemplo, pedir para digitar o nome da pessoa e guardar o que for digitado na variável endereço), etc.

Os erros sintáticos são fáceis de encontrar e corrigir porque, geralmente, o compilador se encarrega de apontá-los e, normalmente, dá uma indicação do tipo de erro. O programa só é executado quando não existem mais erros sintáticos. Já os erros semânticos são de detecção mais difícil, uma vez que os compiladores não podem encontrar erros de lógica, já que não tem conhecimento sobre o que o programador deseja fazer. Dessa forma, erros de lógica só podem ser encontrados e remediados pelo programador. Esse tipo de erro pode fazer com que o programa exiba comportamentos inesperados.

E como se faz o teste de mesa?

1. Leia o algoritmo que foi escrito
2. Crie uma coluna para cada uma das variáveis declaradas no algoritmo e uma coluna para a saída de dados (o que vai ser

impresso na tela)

3. Em seguida, acompanhe linha a linha a execução do algoritmo, anotando nas colunas apropriadas cada mudança de valor das variáveis ou do que foi escrito na tela.
4. Preste atenção nas estruturas condicionais (porque pode haver instruções que não serão executadas) e nas estruturas de repetição (porque pode haver trechos de instruções que devem ser executados mais de uma vez).
5. Siga a execução até chegar ao final do algoritmo

Uma animação interessante sobre um exemplo de teste de mesa simples pode ser vista em <http://www.brasilacademic.com/ed/testemesa.htm>. Vamos dar mais um exemplo. Suponha um algoritmo para ler duas notas de um aluno e calcular a média das mesmas. Depois indicar se o aluno foi aprovado ($\text{média} \geq 7$) ou não. Suponha também que os valores digitados para as notas serão 8.0 e 9.0, respectivamente.

```

1. algoritmo "calcularMedia"
2. var N1, N2, Media : real
3. inicio
4.     escreva("Digite o valor da Primeira Nota ")
5.     leia(N1)
6.     escreva("Digite o valor da Segunda Nota ")
7.     leia(N2)
8.     Media <- (N1 + N2)/2
9.     se (Media  $\geq 7$ ) entao
10.        escreva("Aprovado. Média = ", Media)
11.    senao
12.        escreva("Reprovado. Média = ", Media)
13.    fimse
14. fimalgoritmo
```

Seguindo os passos que foram explicados do teste de mesa, vai ser criada uma coluna para cada variável do algoritmo e uma coluna para o que vai aparecer na tela. Em seguida, você vai seguindo, linha a linha, passo a passo a execução do algoritmo, com os valores de teste sugeridos e vai preenchendo a tabela criada (vide Tabela 1), até terminar o algoritmo.

Executando	Variáveis			Aparece na tela
	NºLinha	N1	N2	Media

Tabela 1 - Exemplo de teste de mesa

Como as variáveis foram preenchidas corretamente e o resultado impresso na tela está correto, o algoritmo criado está correto.

Unidade 4 –Estruturas de Dados Homogêneas

As estruturas de dados homogêneas correspondem a conjuntos de elementos de um mesmo tipo, representados por um único nome. E isso é muito útil! Quer ver? Imagine, por exemplo, como poderíamos construir um algoritmo, que lesse os nomes de 500 pessoas e imprimisse um relatório destes mesmos nomes ordenados alfabeticamente. Com o que sabemos até agora, isso não seria uma tarefa simples! Isso porque teríamos que definir 500 variáveis do tipo caracter e ler essas 500 variáveis, como é mostrado abaixo:

```

1. algoritmo "Inviável"
2. var
3.     nome1, nome2, nome3, nome4, nome5,..., nome499, nome500: caracter
4. inicio
5.     escreva("Digite o nome 1: ")
6.     leia(nome1)
7.     escreva("Digite o nome 2: ")
8.     leia(nome2)
9.     ...
10.    escreva("Digite o nome 500: ")
11.    leia(nome500)
12.    ...
13. fimalgoritmo
```

Não seria possível nem usar um laço para facilitar a leitura, visto que o nome das variáveis muda (nom1, nome2, etc). Dessa forma, seria enorme o tamanho do algoritmo e o trabalho braçal necessário para construí-lo.

Para resolver problemas como este e outros, existem as estruturas de dados. A declaração de uma estrutura de dados corresponde, na verdade, à declaração de várias posições de memória, cujo identificador difere apenas por um índice. Onde este índice corresponde a um valor numérico começando por 0. E cada posição do conjunto, representada por um índice pode receber valores no decorrer do algoritmo, como se fosse uma variável comum. Por exemplo, para o algoritmo anterior,

poderíamos ter uma estrutura de dados chamada Alunos e ela referenciar um conjunto de posições de memória, referenciadas por um índice e em cada um dos índices, poderia ser armazenado um nome.



As estruturas homogêneas podem variar quanto à quantidade de índices necessários para localizar um valor dentro do conjunto. A estrutura de dados homogênea unidimensional (Figura 3, estrutura V), também conhecida como vetor, necessita apenas de um índice para localização dos elementos do conjunto. Já a estrutura de dados homogênea bidimensional (Figura 3 – estrutura M), também conhecida como matriz (aquele mesma matriz que você estudou na matemática!), necessita de dois índices (linha e coluna) para localização de um elemento dentro do conjunto. Existem ainda estruturas multidimensionais, onde é utilizado um índice para localizar o elemento em CADA uma das dimensões da estrutura. Por exemplo, uma estrutura tridimensional (Figura 3 – estrutura T), como tem 3 dimensões, necessitaria de 3 índices para localizar um elemento dentro da estrutura.

Figura 3 - Exemplos de Estruturas Homogêneas

Cada elemento das estruturas (Figura 3) seria referenciado através de índices (que sempre iniciam no valor zero).

- Para localizar um elemento do vetor V, usa-se apenas um índice. Exemplos: $V[1] = 4$ e $V[2] = 10$.
- Para localizar um elemento dentro da matriz M usam-se dois índices, o da linha e o da coluna. Por exemplo, $M[1,1] = 55$, ou

seja, o elemento da linha 1, coluna 1 da matriz. Outro exemplo, $M[2,3] = 23$.

- Para localizar um elemento dentro da estrutura tridimensional T, seriam usados 3 índices. Um para localizar a “cartela” de elementos, depois, um índice para a linha e outro para a coluna. Por exemplo, $T[0,3,1] = 88$, seria pegar a “cartela” 0, a da frente e, nela, a linha 3 e a coluna 1. Outro exemplo, $T[2,0,1] = 6$

Exploraremos nas seções a seguir apenas as estruturas unidimensionais e bidimensionais, apesar de ainda dar um exemplo de estrutura multidimensional.

Estruturas Homogêneas Unidimensionais (Vetores)

Estruturas de dados homogêneas unidimensionais, também conhecidas como vetores, são referenciadas por um único índice.



Por exemplo, se definíssemos um conjunto de nomes de alunos de uma turma chamado Alunos, cada aluno gravado dentro da estrutura, seria referenciado por um índice. Essa estrutura está representada na Figura 4. Se desejássemos pegar o aluno da posição 4, usaríamos o nome do vetor e, entre colchetes, o índice desejado: Alunos[4], daí conseguiríamos pegar o valor “José”.

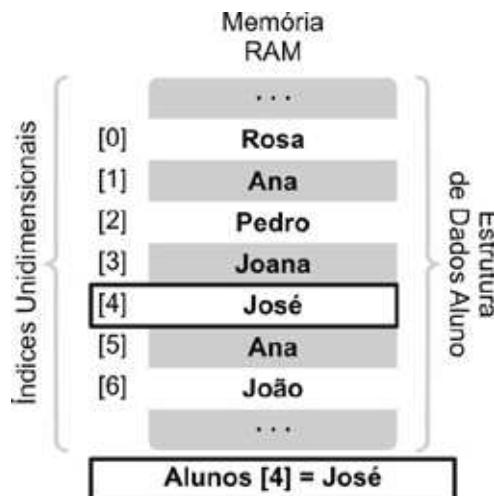


Figura 4 - Exemplo de Estrutura Unidimensional

O vetor é declarado na área de declaração de variáveis e a sintaxe para declaração é:

nomeVetor : vetor [VI..VF] de Tipo

onde:

nomeVetor é o nome que você quer dar a variável vetor na memória.

vetor é palavra reservada para indicar a criação da estrutura homogênea

[VI..VF] Vi é o valor inicial do índice do vetor e VF o valor final. Eles devem ser SEMPRE, números inteiros positivos.

Tipo = é o tipo de elemento que você quer gravar dentro do vetor.

Lembrando que todas as posições do vetor terão esse mesmo tipo (inteiro, real, caracter ou lógico).

Exemplo: o vetor alunos que apresentamos na Figura 4 poderia ser definido como:

var alunos: vetor [0..6] de caracter

Isso seria equivalente a ter 7 variáveis de mesmo nome, só distingüíveis pelos seus índices (alunos[0], alunos[1], alunos[2], alunos[3], alunos[4], alunos[5] e alunos[6]). Outros exemplos de declarações de vetores são:

// um vetor de 21 posições de números inteiros

idades: vetor [0..20] de inteiro

//dois vetores de 10 posições de números reais

receitas, despesas: vetor [0..9] de real

Para se atribuir um valor a um elemento do vetor devemos utilizar o seguinte padrão:

nomeVetor[índice] <- valor

onde o índice deve ser uma posição válida, existente dentro do vetor. Por exemplo:

alunos[1] <- "João da Silva"

idades[0] <- 35

idades[3] <- idades[0] //aqui as duas posições ficariam com o mesmo valor 35

i <- 5

idades[i] <- 45

Todos esses exemplos fizeram uso de posições válidas. Se eu tentasse acessar uma posição que não existe em um vetor, isso acarretaria um erro sintático e seria indicado na compilação do algoritmo. Ex: **alunos[10] <- "Maria Aparecida"**. O vetor aluno, anteriormente definido, só tem sete posições, acessar a posição 10 causaria um erro.

Para percorrer todas as posições de um vetor, deve-se usar alguma estrutura de repetição. Geralmente, utiliza-se a estrutura para..faça, porque sabemos o valor inicial e final dos índices do vetor. Por exemplo, para percorrer o vetor alunos, imprimindo o valor de cada posição, faríamos o seguinte:

```
para indice de 0 até 6 faça
```

```
    escreva (alunos[indice])
```

```
fimpara
```

Com isso, a cada vez que o laço fosse executado, uma posição diferente do vetor seria impressa, porque o valor do índice estaria mudando. Vamos agora juntar tudo? Que tal criar um algoritmo completo que faça uso de vetor? Isso é importante para você tentar entender como manipular um vetor.

Exemplo Completo: Faça um algoritmo para ler valores para preencher um vetor de 6 posições de números inteiros, imprimir os

valores lidos e, depois, imprimir quantos números entre os lidos são números positivos.

```

1. Algoritmo "vetores"
2. var
3.     numeros: vetor [0..5] de inteiro // declaro o vetor
4.     // declaro um índice para caminhar no vetor
5.     // e um contador para os números positivos
6.     indice, numerosPos: inteiro
7. inicio
8.     numerosPos <- 0 // para poder contar os números positivos
9.     // faz um laço para ler os valores que vão preencher o vetor
10. para indice de 0 ate 5 faca
11.         escreva("Digite o número da posição ", indice)
12.         leia (numeros[indice])
13.         // testa se número é positivo
14.         se (numeros[indice] > 0) entao
15.             //se for positivo, incrementa o contador numerosPos
16.             numerosPos <- numerosPos + 1
17. fimse
18. fimpara
19. // faz outro laço para imprimir os números armazenados no vetor
20. para indice de 0 ate 5 faca
21.         escreval (numeros[indice])
22.         fimpara
23.         // imprime o total de números positivos contados
24.         escreva ("Total de números positivos: ", numerosPos)
25. finalgoritmo
```



Reflexão

E se agora, eu quisesse que o algoritmo lesse 100 números e não apenas 6? O que mudaria no algoritmo acima?

```

1. Algoritmo "vetores"
2. var
3.     numeros: vetor [0..99] de inteiro // declaro o vetor
4.     // declaro um índice para caminhar no vetor
```

```

5.      // e um contador para os números positivos
6.      indice, numerosPos: inteiro
7. inicio
8.      numerosPos <- 0 // para poder contar os números positivos
9.      // faz um laço para ler os valores que vão preencher o vetor
10.     para indice de 0 ate 99 faca
11.         escreva("Digite o número da posição ", indice)
12.         leia (numeros[indice])
13.         // testa se número é positivo
14.         se (numeros[indice] > 0) entao
15.             //se for positivo, incrementa o contador numerosPos
16.             numerosPos <- numerosPos + 1
17.         fimse
18.     fimpara
19. // faz outro laço para imprimir os números armazenados no vetor
20.     para indice de 0 ate 99 faca
21.         escreval (numeros[indice])
22.     fimpara
23.     // imprime o total de números positivos contados
24.     escreva ("Total de números positivos: ", numerosPos)
25. finalgoritmo

```

Já imaginou se tivesse de declarar um nome de variável para cada um dos valores a serem lidos? Pois é... resolver o problema anterior sem vetores implicaria no árduo trabalho de ter que declarar e manipular 100 variáveis! Mas usando vetores, você só vai mudar a quantidade de posições do vetor e os limites de valores dentro dos laços e pronto! Todo o restante do algoritmo continuaria igual.

Estruturas Homogêneas Bidimensionais (Matrizes)

Estruturas de dados homogêneas bidimensionais, também conhecidas como matrizes, são referenciadas por um dois índices: um para indicar a linha da matriz e outro para indicar a coluna da matriz onde se encontra o dado.

Por exemplo, como definir um conjunto para armazenar as médias de 7 alunos obtidas em cada uma das 5 disciplinas que eles cursam? Poderíamos fazer cada linha de uma matriz representar um aluno e cada coluna representar uma disciplina (vide Figura 5). Dessa forma, a média do quinto aluno, na primeira disciplina seria representada por: `notas[4,0]` e seria igual ao valor 10.0, como pode ser visto na Figura 5. Isso porque a linha 0 representa o primeiro aluno, a linha 1 o segundo aluno e assim por diante. A coluna 0 representa a primeira disciplina, a coluna 1 a segunda disciplina e assim por diante.

Memória RAM

Estrutura de Dados Notas

		Índices das Colunas				
		[0]	[1]	[2]	[3]	[4]
Índices das Linhas	[0]	7,5	8,0	5,0	8,5	9,0
	[1]	8,0	7,0	6,5	8,5	10,0
	[2]	4,5	8,0	7,0	7,0	8,5
	[3]	9,0	6,5	8,5	9,0	9,0
	[4]	10,0	9,0	8,5	9,0	10,0
	[5]	5,0	7,5	6,5	8,0	9,0
	[6]	10,0	9,0	8,5	9,5	8,0

notas [4,0] = 10,0

Figura 5 - Exemplo de Estrutura Bidimensional

A matriz **notas** ocupa, ao todo, 35 posições de memória (notas de 7 alunos X 5 disciplinas), o que seria equivalente a criar 35 variáveis isoladas no algoritmo, para poder resolver o mesmo problema. Mais complicado, não é?

Ressalta-se que a matriz **notas** não armazena o nome dos alunos ou o nome das disciplinas. Ela armazena somente Notas! (lembre: estruturas de dados homogêneas! Toda posição deve ter o mesmo tipo de dados, que, no caso, são valores reais que são as médias). E como declarar isso? A matriz é declarada na área de declaração de variáveis e a sintaxe para declaração é:

nomeMatriz : vetor [linhaI..linhaF, colunaI..colunaF] de Tipo

onde:

nomeMatriz é o nome que você quer dar a variável matriz na memória.

vetor é palavra reservada para indicar a criação da estrutura homogênea

[linhal..linhaF, colunal..colunaF] a matriz precisa da definição de quantas linhas e quantas colunas a estrutura vai ter. Dessa forma, linhal é o valor inicial do índice de linhas e linhaF é o valor final do índice de linhas. Já colunal é o valor inicial do índice de colunas e colunaF o valor final do índice de colunas.

Tipo = é o tipo de elemento que você quer gravar dentro da matriz. Lembrando que todas as posições terão esse mesmo tipo (inteiro, real, caracter ou lógico).

Sendo assim, a matriz apresentada na Figura 5 poderia ser definida como:

```
var notas: vetor [0..6, 0..4] de real
```

Isso seria equivalente a ter as 35 variáveis de mesmo nome, anteriormente mencionadas, que seriam distinguidas uma das outras pelos seus índices (notas[0,0], notas[0,1], notas[0,2], notas[0,3], notas[0,4], notas[1,0] e assim por diante). Outros exemplos de declarações de matrizes são:

```
// uma matriz de 3 linhas por 4 colunas de números inteiros
valores: vetor [0..2, 0..3] de inteiro
//uma matriz de 2 linhas por 3 colunas de nomes alfanuméricos
nomes: vetor [0..1, 0..3] de caracter
```

Cada posição dentro da matriz é identificada pelo valor do índice da linha e pelo valor do índice da coluna. Por exemplo, se fosse criada uma matriz M de 3 linhas por 3 colunas, cada posição seria acessada como ilustrado na Figura 6. A posição da linha 0, coluna 0 é especificada por M_{00} , a posição da linha 2, coluna 1, é especificada por M_{21} e assim por diante.

M	0	1	2
0	m_{00}	m_{01}	m_{02}
1	m_{10}	m_{11}	m_{12}
2	m_{20}	m_{21}	m_{22}

```
// o primeiro para caminha nas linhas
para linha de 0 até 6 faça
    // o segundo para caminhar nas
    colunas
        // de uma mesma linha
        para coluna de 0 até 4 faça
            escreva (notas[linha,
            coluna])
        fimpara
```

26. **escreva**(notas[0,0])
27. **escreva**(notas[0,1])
28. **escreva**(notas[0,2])
29. **escreva**(notas[0,3])
30. **escreva**(notas[0,4])
31. **escreva**(notas[1,0])
32. **escreva**(notas[0,1])
33. **escreva**(notas[0,2])
34. **escreva**(notas[0,3])
35. **escreva**(notas[0,4])
36. ...
- 37. **escreva**(notas[6,4])

Veja que quando entra no primeiro laço, o contador linha recebe o valor 0 e para esse mesmo valor, o contador coluna varia de 0 até

4. Depois volta-se para o contador linha que passar a ter o valor 1 e, novamente, o contador coluna varia de 0 até 4.

Vamos agora juntar tudo? Para isso, faça um algoritmo para ler as médias de 7 alunos em cada uma das cinco 5 disciplinas que eles cursam (a matriz nota da Figura 5). E depois imprimir essas notas.

```

1. Algoritmo "matrizes"
2.     var
3.         notas: vetor [0..6, 0..4] de real // declaro a matriz
4.             // declaro um índice para linha e outro para coluna
5.             linha, coluna: inteiro
6.         inicio
7.             // Primeiro lê os valores das médias
8.             // Faz um laço para andar nas linhas
9.             para linha de 0 ate 6 faca
10.                // Depois faz um laço para andar nas colunas
11.                para coluna de 0 ate 4 faca
12.                    escreva("Digite a média do aluno ", linha, " da disciplina ",
13.                      coluna)
13.                    leia (notas[linha,coluna])
14.                fimpara
15.            fimpara
16.            // faz outro laço para imprimir os números armazenados na matriz
17.            para linha de 0 ate 6 faca
18.                para coluna de 0 ate 4 faca
19.                    escreva("Média", linha, coluna, " é igual a ", notas[linha,
20.                      coluna])
20.                fimpara
21.            fimpara
22.        fimalgoritmo
```

Da mesma forma que no vetor, seria fácil aumentar a quantidade de alunos ou de disciplinas, apenas trocando o tamanho da matriz e os limites dos laços **para..faça**.

Agora, só para ilustrar, como ficaria se a gente acrescentasse uma nova dimensão à matriz anteriormente definida? Por exemplo, faça um algoritmo para ler as médias de 7 alunos, em 5 disciplinas, em 10 cursos diferentes (estrutura tridimensional).

```

1. Algoritmo "tridimensional"
2. var
3.     notas: vetor [0..6, 0..4, 0..9] de real // declaro a estrutura
4.     // declaro um índice para cada dimensão
5.     aluno, disciplina, curso: inteiro
6. inicio
7.     // Primeiro lê os valores das médias
8.     // Faz um laço para andar nos alunos
9.     para aluno de 0 ate 6 faca
10.        // Depois faz um laço para andar nas disciplinas
11.        para disciplina de 0 ate 4 faca
12.            // E, por fim, um laço para andar nos cursos
13.            para curso de 0 ate 9 faca
14.                escreva("Digite a média: ")
15.                leia (notas[aluno, disciplina, curso])
16.            fimpara
17.        fimpara
18.    fimpara
19. fimalgoritmo

```

Cada dimensão da matriz é manipulada por um laço exclusivo. Por isso, ao se adicionar uma nova dimensão, deve-se criar uma nova variável e um novo laço para manipulá-la. Ou seja, é criado um laço **para..faça** para cada dimensão a ser percorrida e a referência ao elemento da estrutura é feita pela quantidade de índices correspondente a quantidade de dimensões.

Unidade 5 – SubAlgoritmos

A modularização é uma característica muito importante no desenvolvimento de programas. Ela é um método utilizado para facilitar a construção de grandes algoritmos, através de sua divisão em pequenas etapas, que são os subalgoritmos. Ao invés de escrever-se um algoritmo grande, escrevem-se vários algoritmos menores, os quais, não isoladamente, mas em conjunto, resolvem o problema proposto. É o “dividir para conquistar”.

Outra ocasião em que o uso de subalgoritmos é conveniente é quando uma determinada tarefa é efetuada em diversos lugares no mesmo algoritmo. Nisso, ao invés de escrever o mesmo trecho diversas vezes, pode-se escrever um subalgoritmo com aquele trecho de código e chamá-lo diversas vezes. Vamos ilustrar isso com um exemplo: Faça um algoritmo para ler três valores inteiros N1, N2 e N3 e coloca-los em ordem crescente, para ao final imprimir os valores. Use apenas estruturas simples, não precisa usar vetores ou algo do gênero.

```

1. Algoritmo "ordena"
2. var
3.     N1, N2, N3, aux: inteiro
4. inicio
5.     // Primeiro lê os valores
6.     escreva("Digite N1: ")
7.     leia(N1)
8.     escreva("Digite N2: ")
9.     leia(N2)
10.    escreva("Digite N3: ")
11.    leia(N3)
12.    //faz comparações para colocar o maior valor na
variável N3
13.    se (N1 > N2) e (N1>N3) então
14.        aux <- N1
15.        N1 <- N3
16.        N3 <- aux
}

```

```

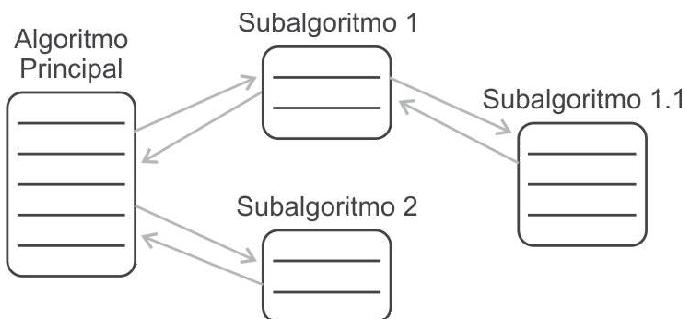
17.      senão
18.          se (N2 > N1) e (N2 > N3) então
19.              aux <- N2
20.              N2 <- N3
21.              N3 <- aux
22.      fimse
23.  fimse
24.      // agora que o maior já está em N3, basta ver quem
        é o maior entre os
25.      // outros dois elementos
26.      se (N1>N2) então
27.          aux <- N1
28.          N1 <- N2
29.          N2 <- aux
30.      fimse
31.      escreval(N1, " ", N2, " ", N3)
32.  finalgoritmo

```

Analisando o código, vemos que há trechos (que foram destacados com chaves) que realizam exatamente a mesma tarefa: trocar o valor de dois elementos. Justamente trechos como esses são bons candidatos a subalgoritmos. Mas vamos definir melhor, o que é mesmo um subalgoritmo? Bem, um subalgoritmo é um algoritmo menor que auxilia o algoritmo principal através da realização de uma determinada subtarefa bem definida (até agora, tudo que fizemos estava só no algoritmo principal). O subalgoritmo é também chamado de *subprograma, subrotina ou módulo*. Ele é declarado antes do início do algoritmo principal e pode ser chamado em qualquer ponto após sua declaração. Ou seja, dentro do algoritmo principal ou a partir de outro subalgoritmo definido abaixo dele. Por exemplo, na Figura 7, o subalgoritmo 1.1 é chamado a partir do subalgoritmo 1.

Os subalgoritmos são chamados dentro do corpo do algoritmo principal ou de outro subalgoritmo como se fossem comandos. Daí o fluxo de execução do que está sendo executado é desviado para executar o subalgoritmo e, após seu término, a execução continua a partir do ponto seguinte à chamada do subalgoritmo (Figura 7). Ou seja, a chamada de um subalgoritmo simplesmente gera um desvio

provisório no fluxo de execução do algoritmo principal.



<-- Algoritmo Principal

4. // ***** LOCAL PARA A DECLARAÇÃO DOS SUBALGORITMOS *****

5. **início**

Há dois tipos de subalgoritmos: PROCEDIMENTOS e FUNÇÕES.
Vamos detalhar cada um deles a seguir.

Procedimentos

Um procedimento é um subalgoritmo que não retorna, **explicitamente**, valores ao algoritmo principal ou a outro subalgoritmo que o tenha chamado. Ele pode retornar valores apenas por meio dos parâmetros, e nunca explicitamente como no caso das funções (que veremos depois) que usam a instrução Retorne.

Sua declaração, como descrito anteriormente, deve estar entre o final da declaração de variáveis do algoritmo principal e a linha **inicio** do mesmo e obedece à seguinte sintaxe:

```
procedimento nomeProcedimento ([declarações-de-parâmetros])
[var declaração de variáveis locais]
inicio
    //Seção de Comandos
fimprocedimento
```

onde:

nomeProcedimento obedece às mesmas regras de nomenclatura de um identificador (vistas no capítulo 1).

[declarações-de-parâmetros]

Um procedimento pode não receber nenhum parâmetro ou pode ter um ou mais parâmetros. Parâmetros são a forma de comunicação entre quem chama o procedimento e o procedimento. São valores de entrada, que precisam ser enviados por quem chamou o procedimento. Essas declarações vão ter o seguinte formato:

[var] seqüência-de-parâmetros : tipo-de-dado

Cada declaração entre um mesmo tipo de dado é separada por vírgula, mas a declaração entre tipos de dados diferentes é separada por ponto-e-vírgula. A presença (opcional) da palavra-chave **var** indica que a passagem de parâmetros é feita *por referência*; caso contrário, a passagem de parâmetros será feita *por valor* (explicaremos isso melhor na subseção a seguir).

[**var** declaração de variáveis locais]

De modo análogo ao algoritmo principal, a seção de declaração de variáveis locais começa com a palavra-chave **var**, e continua com a seguinte sintaxe: <lista-de-variáveis> : <tipo-de-dado>. As variáveis

locais só são vistas e só podem ser usadas dentro do procedimento

Na seção de comandos, você pode ter qualquer instrução que foi vista até agora: estrutura seqüencial, condicional ou de repetição, comandos de entrada e saída de dados, operações etc. Lembre que um subalgoritmo (procedimento ou função) é um algoritmo, só que menor, específico para realizar uma tarefa. Logo, pode ter tudo que um algoritmo pode ter.

E depois do procedimento declarado, como eu faço uso dele?
Como eu posso chamá-lo?



CHAMADA DE UM PROCEDIMENTO

A chamada de um procedimento é feita pelo nome dele, como um comando simples, dentro do algoritmo principal, de outro subalgoritmo ou dentro do próprio procedimento (quando um procedimento chama ele mesmo, está se fazendo uso de recursão – esse assunto só será apresentado na linguagem C, no módulo final da disciplina).

Sintaxe: **nomeProcedimento(valores para os parâmetros)**

Note que é parecido com a ativação do comando instruções Leia e Escreva, porque eles são também procedimentos. Um procedimento não pode ser chamado no meio de expressões ou em atribuições como no caso de funções.

Ao se chamar um procedimento, deve-se observar se ele tiver parâmetros definidos. Se ele tiver, quem o chamou vai ter que passar um valor para cada parâmetro. A quantidade dos parâmetros, sua seqüência e respectivos tipos não podem mudar: devem estar de acordo com o que foi especificado na sua correspondente declaração.

Que tal um exemplo completo agora, para ver se você está entendendo? Vamos pegar aquele algoritmo ordena do início da Unidade 4 e vamos agora, implementá-lo usando procedimento. O trecho que se repete, que é o da troca de valor de duas variáveis, é o que vai ser implementado como procedimento.

O procedimento

```

1. Algoritmo "ordena2"
2. var
3.     N1, N2, N3: inteiro
4. // depois da declaração de variáveis deve ser definido o procedimento
5.     procedimento troca (var num1, num2: inteiro)
6.         var aux: inteiro // declaração de variável local
7.         inicio
8.             aux <- num1
9.             num1 <- num2
10.            num2 <- aux
11.        fimprocedimento
12.    inicio
13.        // Primeiro lê os valores
14.        escreva("Digite N1: ")
15.        leia(N1)
16.        escreva("Digite N2: ")
17.        leia(N2)
18.        escreva("Digite N3: ")
19.        leia(N3)
20.        //faz comparações para colocar o maior valor na variável N3
21.        se (N1 > N2) e (N1>N3) então
22.            troca(N1, N3)
23.        senão
24.            se (N2 > N1) e (N2 > N3) então
25.                troca(N2, N3)
26.            fimse
27.        fimse
28. // agora que o maior já está em N3, basta ver quem é o maior entre os
29. // outros dois elementos
30.        se (N1>N2) então
31.            troca(N1,N2)
32.        fimse
33.        escreval(N1, " ", N2, " ", N3)
34.    finalgoritmo

```

Chamadas do Procedimento

- 25.
- 31.

Observe: o procedimento troca foi definido da linha 5 a linha 11. Ou seja, depois da declaração de variáveis do algoritmo principal (que é o ordena2) e antes do início do mesmo (linha12). O procedimento troca possui dois parâmetros (linha 5) chamados num1 e num2. Como na declaração dos parâmetros usou-se a palavra **var**, a passagem está sendo feita por referência (explicaremos isso melhor a seguir). Também foi definida uma variável local para o procedimento (linha 6), chamada aux. Você consegue notar a diferença entre variável local e parâmetro?

O valor de cada parâmetro (que é chamado de *argumento*) vai ser recebido de quem chamar o procedimento, ou seja, ele recebe valores externos ao procedimento. Já a variável local, ela serve para ser usada dentro do procedimento, ela não recebe valor externo algum.

O procedimento internamente tem seus comandos, que são os que realizam a troca de dois valores (linha 7 a 11). Após declarado, o procedimento vai poder ser chamado dentro do algoritmo principal. As chamadas são feitas nas linhas 22, 25 e 31. Na hora que o procedimento é chamado (para chamá-lo é só usar o nome dele) deve-se observar que valores ele está esperando como parâmetro. Da declaração do procedimento (linha 5) sabemos que ele está esperando duas variáveis inteiras. Por isso, na hora de chamar o procedimento, temos de enviar para ele o que ele está esperando. No caso, na primeira chamada, passamos como argumento N1 e N3 (linha 22), na segunda chamada passamos como argumento N2 e N3 (linha 25) e na terceira chamada passamos N1 e N2 (linha 31), que, por causa da declaração dos parâmetros do procedimento (linha 5), teriam de ser todas variáveis do tipo inteiro.



Reflexão

Mas como funciona essa tal passagem de parâmetros?

Passagem de Parâmetros

Passagem de parâmetros é um esquema que torna a comunicação entre o subalgoritmo e quem o ativa mais clara, definindo muito bem as informações que entram e que saem de um subalgoritmo e a maneira

como estas informações serão trocadas com a unidade que o ativou. Neste esquema, são definidos os PARÂMETROS na declaração do subalgoritmo (procedimento ou função) e os ARGUMENTOS na chamada do subalgoritmo. Ou seja, os argumentos são o que passamos para o subalgoritmo, no momento da chamada do mesmo. ARGUMENTOS e PARÂMETROS devem corresponder por tipo e posição. Se uma destas condições falhar, ocorrerá um erro de compilação. Por exemplo, se for definido um procedimento:

```
procedimento teste (valor: inteiro; nome: caractere) // definição dos parâmetros
```

Você não poderia ativar esse procedimento no programa chamador de nenhuma das formas abaixo:

```
teste ("Maria", 10) // a posição dos argumentos não equivale a declaração
teste (10, 2.5) // o tipo dos argumentos está errado, não corresponde aos parâmetros do procedimento
// Supondo x do tipo real e com valor 2.5 e y do tipo caractere com valor "Ana"
teste(x, y) // o tipo dos argumentos está errado
```

O correto seria passar um valor inteiro e um valor do tipo caractere, nesta ordem, que é o que o procedimento está esperando como entrada, para ser ativado. No último exemplo, quando o procedimento teste é chamado, o argumento X do tipo real é associado ao parâmetro chamado *valor* do procedimento que é do tipo inteiro, por isso estaria errado. Logo, tornamos a frisar, os argumentos e parâmetros devem corresponder em tipo, quantidade e posição.

A substituição dos parâmetros pelos argumentos no ato da invocação ou chamada de um subalgoritmo é denominada de *passagem de parâmetros* e ela realizar-se por dois mecanismos distintos: *passagem por valor (ou por cópia)* e *passagem por referência*.

Passagem de Parâmetros por Valor

Quando um parâmetro é passado por valor, o valor do argumento do momento da ativação é avaliado e COPIADO para o parâmetro correspondente do subalgoritmo. Ou seja, o argumento é apenas um valor de entrada para o parâmetro correspondente. Dessa forma, qualquer alteração feita, dentro do subalgoritmo, no parâmetro não vai afetar o argumento fora da subalgoritmo. Isto, porque, o que está sendo

manipulado dentro do subalgoritmo é uma cópia do argumento.

A passagem de parâmetros por valor é indicada pela ausência da palavra VAR na parte de declaração de parâmetros do subalgoritmo. Ela é usada quando não desejamos alterar o valor do argumento, apenas utilizá-lo. Vamos dar um exemplo. Suponha um algoritmo simples que faz uso de um procedimento para incrementar em uma unidade o valor de uma variável e imprime o valor da variável antes e depois da chamada do procedimento.

1. Algoritmo "exemploValor"

```

2. var numero: inteiro
3.     procedimento incrementa(valor: inteiro)
4.     inicio
5.         valor ← valor + 1
6.         escreva("Dentro do procedimento, Valor= : ", valor) // imprime 2
7.     fimprocedimento
8.     inicio
9.         numero ← 1
10.        escreva("Antes do procedimento, número = ", numero) // imprime 1
11.        incrementa(numero)
12.        escreva("Depois do procedimento, número = ", numero) // imprime 1
13. fimalgoritmo

```

Quando algoritmo inicia (linha 8) a variável número recebe o valor 1 (vide Figura 8). Por isso, a primeira impressão (linha 10) vai imprimir a frase “Antes do procedimento, número = 1”.

M E M Ó R I A D O C O M P U T A D O R

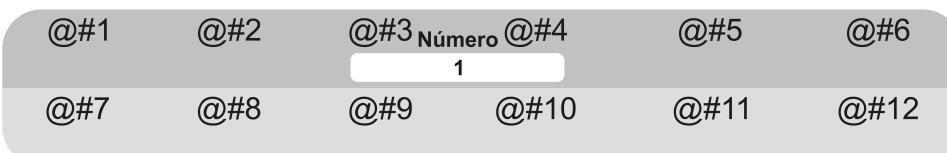


Figura 8 – Exemplo da memória do computador ANTES e DEPOIS da chamada do procedimento

Daí o procedimento é chamado (linha 11). Quando ele é chamado é feita uma *passagem por valor* (veja que na declaração do parâmetro do procedimento na linha 3, não foi usada a palavra **var**) do argumento *numero* para o parâmetro *valor*. Ou seja, é feita uma cópia do valor 1 que está dentro da variável *numero* do algoritmo principal, para a

variável chamada *valor* que pertence ao procedimento (Figura 9). As variáveis e parâmetros do procedimento só passam a existir na memória do computador, a partir do momento em que o procedimento é chamado.



Figura 9 - Exemplo da memória do computador assim que o procedimento é chamado

Dentro do procedimento o parâmetro *valor* é incrementado (linha 5 e Figura 10), por isso, dentro do procedimento o que é impresso (linha 6) é: “*Dentro do procedimento, Valor = 2*”.



Figura 10 - Exemplo da memória do computador DURANTE a execução do procedimento

Quando o procedimento é encerrado (linha 7), todas as variáveis e parâmetros que foram criados pelo mesmo na memória são retirados dela. Ou seja, tudo que é criado pelo procedimento deixa de existir, voltando para o que está representado na Figura 8. Dessa forma, vemos que o que foi manipulado durante a execução do procedimento foi uma cópia e, por isso, o valor passado como argumento (a variável *numero*) não é afetado por qualquer modificação feita dentro do procedimento. Tanto que a impressão realizada após a chamada do procedimento (linha 12) imprime “*Depois do procedimento, número = 1*”. Ou seja, imprime o mesmo valor de antes da chamada do procedimento. Essa é a filosofia da passagem de parâmetros por valor.



Reflexão

E agora, qual a diferença para a *passagem por referência*?

Passagem de Parâmetros por Referência

A passagem de parâmetros por referência (também chamada de por endereço) é utilizada quando desejamos alterar o valor do ARGUMENTO passado por quem ativa o subalgoritmo. Na verdade, na passagem de parâmetros por referência, toda alteração feita no parâmetro vai alterar o argumento associado a este parâmetro. Assim, quando a passagem é por referência, isto significa que o parâmetro é de *entrada e saída*.

Quando a passagem é por referência, o que é fornecido ao parâmetro do subalgoritmo não é uma cópia do valor do argumento, mas a sua *localização na memória do computador* (endereço de memória). Ou seja, quando é utilizada uma passagem por referência, o parâmetro e o argumento partilham o mesmo espaço de memória e, consequentemente, qualquer modificação feita no parâmetro reflete no argumento. A passagem por referência é indicada pela presença da palavra VAR antecedendo a declaração do parâmetro do subalgoritmo.

`procedimento incrementa(var valor: inteiro)`

Todo parâmetro é passado por valor, exceto quando indicado que a passagem de parâmetros é por referência (quando for utilizada a palavra reservada **var**). Vamos agora mostrar o uso da passagem por referência no mesmo exemplo dado anteriormente para a passagem por valor. E vamos observar o que muda.

1. **Algoritmo** "exemploReferencia"
2. **var** numero: inteiro
3. **procedimento** incrementa(**var** valor: inteiro)
4. **inicio**
5. valor ← valor + 1
6. escreva("Dentro do procedimento: ", valor) // **imprime** 2
7. **fimprocedimento**
8. **inicio**

```

9.           numero ← 1
10.          escreva("Antes do procedimento, número = ", numero) // imprime 1
11.          incrementa(numero)
12.          escreva("Depois do procedimento, número = ", numero) // imprime 2
13. finalgoritmo

```

Quando algoritmo inicia (linha 8) a variável número recebe o valor 1 (vide Figura 11). Por isso, a primeira impressão (linha 10) vai imprimir a frase “Antes do procedimento, número = 1”.



Figura 11 – Exemplo da memória do computador ANTES da chamada do procedimento

Daí o procedimento é chamado (linha 11). Quando ele é chamado é feita uma *passagem por referência* (veja que na declaração do parâmetro do procedimento na linha 3, foi usada a palavra **var**) do argumento *numero* para o parâmetro *valor*. Ou seja, o argumento *numero* e o parâmetro *valor* vão compartilhar o mesmo espaço de memória (Figura 12). Lembrando que as variáveis e parâmetros do procedimento só passam a existir na memória do computador, a partir do momento em que o procedimento é chamado.

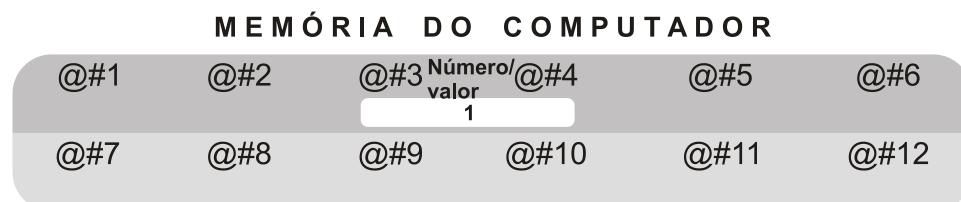


Figura 12 - Exemplo da memória do computador assim que o procedimento é chamado

Dentro do procedimento o parâmetro *valor* é incrementado (linha 5), automaticamente, o argumento *numero* também é incrementado, porque eles referenciam a mesma posição de memória (Figura 13). Dentro do procedimento o que é impresso (linha 6) é: “*Dentro do procedimento, Valor = 2*”.

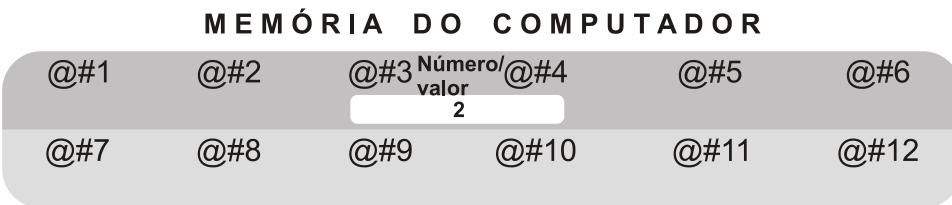


Figura 13 - Exemplo da memória do computador DURANTE a execução do procedimento

Quando o procedimento é encerrado (linha 7), todas as variáveis e parâmetros que foram criados pelo mesmo na memória são retirados dela. Dessa forma, a referência do parâmetro *valor* é desacoplada do argumento *numero*, porém as mudanças que foram feitas permanecem (Figura 14). Desse modo, vemos que o que foi manipulado durante a execução do procedimento afeta, diretamente, os argumentos passados por referência. Por isso, a impressão realizada após a chamada do procedimento (linha 12) imprime “*Depois do procedimento, número = 2*”. Ou seja, o valor do argumento do algoritmo principal foi realmente afetado pelo que foi executado dentro do procedimento.



Figura 14 - Exemplo da memória do computador DURANTE a execução do procedimento

Em um mesmo subalgoritmo, podemos fazer uma passagem por valor e outra por referência. Veja o exemplo abaixo.

1. **algoritmo** “exemploPassagemParametros”
2. **var** N1,N2 : inteiro
3. **procedimento** muda(X:inteiro; **var** Y:inteiro)
4. **inicio**
5. X <- 1
6. Y <- 1
7. **fimprocedimento**
8. **inicio**
9. N1 <- 0
10. N2 <- 0
11. muda(N1,N2)
12. escreva(N1)

13. escreva(N2)

14. fimalgoritmo

Que valores de N1 e N2 serão impressos (linhas 12 e 13), após a chamada do procedimento?

O argumento N1 será associado ao parâmetro X e será feita uma passagem de parâmetro por valor (não existe a palavra reservada **var** na definição do parâmetro). Dessa forma, como a passagem foi por valor, o conteúdo de N1 não é modificado. Assim, N1 continuaria com o valor zero.

Já o argumento N2 será associado ao parâmetro Y e será feita uma passagem por referência (veja que existe a palavra **var** antes da definição do parâmetro Y). Desse modo, como foi possível observar, a modificação feita em Y afetou diretamente N2. Sendo assim, o valor de N2 a ser impresso seria o valor UM. Ficou claro? Espero que sim!

Apesar de termos dado todos os exemplos de passagem de parâmetro usando procedimentos, esses tipos de passagem de parâmetro acontecem também nas funções. Usamos procedimentos para ilustrar apenas porque só os explicamos até agora.

Escopo de Variáveis

Cada subalgoritmo, além de ter suas próprias variáveis (chamadas de variáveis LOCAIS), que existem apenas durante a sua execução e só fazem sentido dentro do subalgoritmo, podem também ter acesso às variáveis de quem o chamou e às variáveis definidas no algoritmo principal.

Na verdade, as variáveis declaradas na parte inicial do algoritmo principal ficam disponíveis para uso em qualquer parte desse algoritmo ou por qualquer subalgoritmo. Por isso, elas são chamadas de variáveis GLOBAIS. Porém, para manter a modularidade dos programas, NÃO É RECOMENDADO que se faça uso de variáveis globais dentro dos subalgoritmos. E sim, ao invés disso, é importante que se use passagem de parâmetros.

As variáveis GLOBAIS existem na memória do computador durante toda a execução do algoritmo, ou seja, até que o algoritmo completo chegue ao seu final. Já as variáveis LOCAIS somente são

criadas quando o subalgoritmo que as contém é ativado e, ao seu término, elas são liberadas/desalocadas da memória, tornando-se inacessíveis.

```

1. algoritmo "exemploEscopo"
2. var N1,N2, resultado : inteiro ←
3.
4.     procedimento soma(X, Y:inteiro; var Z:inteiro) ←
5.         inicio
6.             Z <- X + Y
7.         fimprocedimento
8.
9.         procedimento troca(var A, B:inteiro) ←
10.
11.            var aux: inteiro ←
12.            inicio
13.                aux <- A
14.                A <- B
15.                B <- aux
16.            fimprocedimento
17.            inicio
18.                N1 <- 10
19.                N2 <- 5
20.                soma(N1, N2, resultado)
21.                troca(N1, N2)
22.                escreva(N1, N2, resultado)
23. fimalgoritmo

```

As variáveis N1, N2 e resultado são variáveis globais, porque foram definidas na declaração de variáveis do algoritmo principal (linha 2). Por isso, elas podem ser acessadas e manipuladas de qualquer lugar. E vão existir na memória do computador até o final do algoritmo (linha 23).

As variáveis X, Y e Z definidas dentro do procedimento soma (linha 4) só podem ser manipuladas e acessadas dentro deste procedimento.

As variáveis A e B (parâmetros) e aux (variável local) definidas dentro do procedimento troca (linhas 9 e 11) só podem ser acessadas e manipuladas dentro deste procedimento.

Funções

O conceito de funções é originário da idéia de função matemática, onde um valor é calculado a partir de outro(s) valor(es) fornecido(s) à função. No contexto de programação, uma função é um subalgoritmo que, além de executar uma determinada tarefa, retorna, explicitamente, um valor para quem a chamou (o algoritmo principal ou a outro subalgoritmo que a tenha chamado), que é o resultado de sua execução. Esse valor é retornado através da instrução Retorne. E, devido a esse fato, a chamada de uma função aparece como uma expressão, e não apenas como um comando, como foi o caso do procedimento. Porque tem de existir alguma variável para receber ou alguma expressão para usar o valor retornado pela função.

A declaração da função, de forma análoga ao procedimento, deve estar entre o final da declaração de variáveis do algoritmo principal e a linha **inicio** do mesmo e obedece à seguinte sintaxe:

```
funcao nomeFuncao ([declarações-de-parâmetros]) : Tipo de Retorno
[var declaração de variáveis locais]
inicio
// Seção de comandos
retorne variável ou expressão de retorno
fimfuncao
```

onde:

nomeFuncao obedece às mesmas regras de nomenclatura de um identificador (vistas no capítulo 1).

[declarações-de-parâmetros]

Uma função pode não receber nenhum parâmetro ou pode receber um ou mais parâmetros, por valor ou por referência. Essas declarações vão ter o seguinte formato:

[var] seqüência-de-parâmetros : tipo-de-dado

Cada declaração entre um mesmo tipo de dado é separada por vírgula, mas a declaração entre tipos de dados diferentes é separada por ponto-e-vírgula. Tal como no procedimento, a presença (opcional) da palavra-chave **var** indica que a passagem de parâmetros é feita *por referência*; caso contrário, a passagem de parâmetros será feita *por valor* (como explicado anteriormente).

: Tipo de Retorno – a função retorna explicitamente um valor para

quem a chamou. Esse tipo de retorno especifica justamente o que o próprio nome indica: qual o tipo de dados que a função vai retornar.

[`var` declaração de variáveis locais]

De modo análogo ao algoritmo principal, a seção de declaração de variáveis locais começa com a palavra-chave `var`, e continua com a seguinte sintaxe: `<lista-de-variáveis> : <tipo-de-dado>`. As variáveis locais só são vistas e só podem ser usadas dentro da função.

Na seção de comandos, você pode ter qualquer instrução, tal qual já foi explicado quando apresentamos os procedimentos.

retorne variável ou expressão de retorno - a instrução `retorne` é um comando simples usado apenas nas funções e tem o efeito de parar a execução da função e enviar um valor para o algoritmo principal ou para outro subalgoritmo que a tenha chamado. A variável ou expressão de retorno deve ser exatamente do mesmo tipo especificado na declaração da função. Por exemplo, se foi dito que a função retornava um valor inteiro. A variável ou expressão de retorno tem de ser um valor inteiro. Toda função deve ter em seu corpo de instruções, pelo menos, uma instrução `Retorne`.

E depois da função declarada, como eu faço uso dela? Como eu posso chamá-la?



CHAMADA DE UMA FUNÇÃO

A chamada de uma função, por parte do algoritmo principal ou por outro subalgoritmo, é feita pelo simples aparecimento de seu nome, com os respectivos parâmetros entre parênteses, dentro de uma expressão ou atribuição. Lembrando que os argumentos usados no ato da chamada da função devem manter uma correspondência (quantidade, ordem e tipo) com os parâmetros definidos na função. A função é executada e, ao seu término, o trecho do comando que a chamou é substituído pelo valor retornado pela mesma dentro da expressão ou atribuição em que se encontra e a avaliação da

expressão ou atribuição prossegue normalmente.

Vamos dar dois exemplos, para tentar deixar tudo mais claro: o primeiro chama a função de dentro de uma expressão e o segundo chama a função usando atribuição. Suponha que precisamos fazer um algoritmo, que faça uso de função, para checar se uma pessoa é maior ou menor de idade, dando a mensagem apropriada.

```

1. algoritmo "exemploFuncao"
2. var idade: inteiro
3.         função checaIdade(id:inteiro): logico
4.         inicio
5.             se (id >= 18) então
6.                 retorne verdadeiro
7.             senão
8.                 retorne falso
9.             fimse
10.            fimfuncao
11.            inicio
12.                escreva("Digite o valor da idade: ")
13.                leia(idade)
14.                se (checaIdade(idade)=verdadeiro) entao
15.                    escreval("A pessoa é de maior ")
16.                    senao
17.                    escreval("A pessoa é de menor ")
18.                fimse
19.            fimalgoritmo
```

A função é declarada na linha 3 e tem como tipo de retorno um valor lógico. Por causa disso, o que pode ser retornado no comando **retorne** são apenas valores lógicos (linhas 6 e 8).

Veja que a função foi chamada dentro de uma expressão (linha 14) que faz parte de uma estrutura condicional. O **SE** só será avaliado, depois que a função for chamada e executada, pois ele precisa do valor de retorno da mesma para poder realizar a comparação.

Outra forma de chamar uma função é usando uma **atribuição**.

```

1. algoritmo "exemploFuncao2"
2. var num1, num2, resultado: inteiro
```

```

3.    função somar(x, y:inteiro): inteiro
4.    var resposta: inteiro
5.    inicio
6.        resposta <- x + y
7.        retorne resposta
8.    fimfuncao
9.    inicio
10.       escreva("Digite Num1: ")
11.       leia(num1)
12.       escreva("Digite Num2: ")
13.       leia(num2)
14.       resultado <- somar(num1, num2)
15.       escreva("A soma dos valores é : ", resultado)
16. finalgoritmo

```

A função é declarada na linha 3 e tem como tipo de retorno um valor inteiro. Por causa disso, o que pode ser retornado no comando **retorne** só pode ser um valor inteiro (linha 7).

Veja que a função foi chamada dentro de uma atribuição (linha 14). Nisso a função é ativada/chamada e executada e o valor que ela retornar será atribuído a variável resultado do algoritmo principal.

OBSERVAÇÃO:

Vários programas, linguagens e ambientes possuem funções pré-definidas que podem ser utilizadas diretamente. Ou seja, já estão implementadas e prontas para usar. No VisuAlg também tem disso! Para saber as funções pré-definidas do VisuAlg, pressione (**CTRL + J**) dentro do ambiente do VisuAlg e aparecerá uma lista delas (Figura 15). Para utilizá-las, selecione a função desejada com o mouse ou as setas de direção do teclado e aperte o **Enter**. Depois é só fornecer os parâmetros que a função esteja precisando para executar.

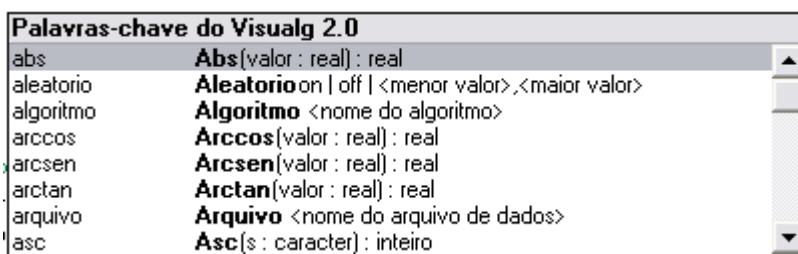


Figura 15 - Funções Pré-definidas do VisuAlg



Aprenda Praticando

Muitas novidades nesse módulo, não? Então vamos dar um exemplo relacionado com cada assunto, para que você possa compreender melhor a construção dos algoritmos.

Problema 1: SOBRE VETORES - Faça um algoritmo para ler a idade de 20 pessoas e, depois, imprimir apenas as idades que estiverem acima da média das idades lidas.

A partir da leitura do enunciado temos:

ENTRADA: as idades de 20 pessoas

PROCESSAMENTO: calcular a média de idades

SAÍDA: as idades que estiverem acima da média.

Logo, neste algoritmo há duas opções, ou declarar 20 variáveis idades ou usar vetores. Como apresentado neste capítulo, o uso de vetores seria mais vantajoso e mais simples de manipular. Mas não poderia ser usada uma simples variável um laço? Não, porque precisamos saber as idades lidas, visto que após calcular a média, temos de imprimir, entre as idades lidas, apenas aquelas que estiverem acima da média.

```

1. Algoritmo "ExemploVetor"
2. var idades: vetor[0..19] de inteiro // 20 elementos: de 0 a 19
3.     soma, indice: inteiro //índice é a variável contadora para o laço para..faça
4.     media: real // servirá para armazenar a média calculada
5.     inicio
6.         soma <-0 // para poder acumular a soma das idades, inicia a soma
com zero
7.         para indice de 0 até 19 faça
8.             escreval("informe a idade ", indice)
9.             leia(idades[indice])
10.            soma <- soma + idades[indice]
11.        fimpara
12.        // quando sai do laço que somou as idades, calcula a média delas
13.        media <- soma / 20
14.        // percorre denovo o vetor agora para imprimir as idades que estão
15.        // acima da média de idades calculada
16.        para indice de 0 até 19 faça
17.            se (idades[índice] > media) entao
18.                escreva(idades[indice])
19.            fimse
20.        fimpara
21. fimalgoritmo

```

Problema 2: SOBRE MATRIZES - Faça um algoritmo para ler uma matriz 4 x 4 contando números inteiros. Depois imprima a soma de cada linha da matriz e, ao final, a soma de todos os elementos da matriz. A partir da leitura do enunciado temos:

ENTRADA: matriz 4 x 4 de números inteiros

PROCESSAMENTO: calcular a de cada linha da matriz e a soma de todos os elementos da matriz

SAÍDA: o valor da soma de cada linha e o valor total da soma de todos os elementos da matriz.

Logo, vamos começar o algoritmo criando a matriz. Vamos precisar de duas variáveis, uma para acumular a soma das linhas e outra para acumular a soma total dos valores. Precisaremos também no algoritmo de dois laços, um para preencher a matriz e outro para calcular a soma das linhas e a soma total.

```

22. =====
23. Algoritmo "ExemploVetor"

```

```

24. var numeros: vetor[0..3, 0..3] de inteiro
25.      somaLinha, somaTotal: inteiro //as variáveis para acumular as somas
26.      linha, coluna: inteiro // índices para o laço para...faça para percorrer a matriz
27. inicio
28. // sempre para ler matrizes, você vai precisar de dois laços aninhados, um para andar
nas
29. // linhas e outro para andar nas colunas.
30.     para linha de 0 até 3 faça
31.         para coluna de 0 até 3 faça
32.             escreval("Digite um número ")
33.             leia(numeros[linha, coluna])
34.         fimpara
35.     fimpara
36.     somaTotal <-0 // zera, para poder acumular a soma total dos elementos da
matriz
37.     // Depois da matriz preenchida, vamos fazer um laço para realizar as devidas
somas
38.     para linha de 0 até 3 faça
39.         // a cada nova linha a variável que acumula a soma das linhas deve ser zerada
40.         somaLinha <-0
41.         para coluna de 0 até 3 faça
42.             somaLinha <- somaLinha + numeros[linha,coluna]
43.         fimpara
44.         // quando sai do laço que anda nas colunas, terminou a soma de uma
linha toda.
45.         // Por isso, imprimimos o resultado da soma dessa linha e acumulamos
na
46.         // variável somaTotal o valor da soma dessa linha
47.         escreval("A soma da linha ", linha, " é igual a ", somaLinha)
48.         somaTotal <- somaTotal + somaLinha
49.     fimpara
50.     // quando sair de todos os laços, imprime o resultado da somaTotal
51.     escreva("A soma de todos os valores da matriz é: ", somaTotal)
52. finalgoritmo

```

Problema 3: SUBALGORITMO - Faça um algoritmo para ler vários números inteiros e positivos (até que seja digitado o valor zero) e imprimir na tela se o número é par ou ímpar. Se o número lido não for positivo, dê uma mensagem apropriada e não julgue o número. Faça uso de subalgoritmo para resolver o problema.

A partir da leitura do enunciado temos:

ENTRADA: vários números inteiros e positivos (até que seja digitado o valor zero)

PROCESSAMENTO: testar se os números são pares ou ímpares

SAÍDA: dizer se o número é par ou ímpar ou dar uma mensagem apropriada se o número for negativo.

Há várias formas de resolver o problema. Pedi para resolverem usando subalgoritmo para podermos exercitar. A parte do problema que está bem definida e lembra um subalgoritmo, é justamente a parte de “julgar se o número é par ou ímpar”. Vamos implementar isso usando um procedimento. Para a estrutura de repetição, como não temos uma quantidade definida de números para ler, não devemos usar o laço para..faça. Podemos usar qualquer um dos outros.

1. **Algoritmo** “ExemploSubalgoritmo”
2. **var** numero: inteiro // o número a ser lido
3. // subalgoritmos devem ser declarados antes do início do algoritmo principal
4. // a passagem de parâmetros será por valor, porque não desejamos fazer nenhuma
5. // modificação no valor passado como parâmetro. Ele serve apenas como dado de entrada
6. **procedimento** julgar (valor: inteiro)
7. **inicio**
8. // para checar se o número é par, uso o MOD que pega o resto da divisão inteira
9. // Porque se o número for divisível por dois (resto zero), ele é par.
10. **se** (valor MOD 2 = 0) **entao**
11. **escreva**("O número é par!")
12. **senao**
13. **escreva**("O número é ímpar")
14. **fimse**
15. **fimprocedimento**
16. **inicio**
17. // vou usar o laço REPITA, para executar a leitura pelo menos uma vez.
18. **repita**
19. **escreval**("Digite um número ")
20. **leia**(numero)
21. // como o usuário pode digitar na primeira vez o zero (condição de parada)
22. // só devemos continuar se o valor lido for diferente de zero

```

23.           se (numero <> 0) entao
24.               // o enunciado do problema pede para só trabalharmos com
números positivos
25.               // logo, se ele for negativo, temos de dar a mensagem
apropriada ao usuário
26.           se (numero < 0) entao
27.               escreva("Digite apenas números positivos!")
28.           senao
29.               // se o número for positivo, chamamos o procedimento
julgar
30.               julgar(numero)
31.           fimse
32.       fimse
33.       ate (numero = 0) // condição de parada do laço
34.   finalgoritmo

```



Conheça Mais

Para saber mais sobre o VisuAlg e sobre dicas para uso da linguagem algorítmica dentro dele, você pode acessar a página da ferramenta:

<http://www.apoioinformatica.inf.br/visualg/linguagem.htm>

Ou dar uma lida nos seguintes materiais que estarão disponibilizados no ambiente: “Manual do VisuAlg” e “A Linguagem do VisuAlg 2.0” (este também pode ser encontrado em: <http://www.scribd.com/doc/2177331/Linguagem-Visualg2-0>). Esse material tem todo o assunto referente a linguagem algorítmica.

Quanto aos assuntos Estruturas de Dados Homogêneas e SubAlgoritmos, você pode ler os capítulos referentes a esses assuntos em qualquer livro sobre algoritmos ou lógica de programação. Porém, continuo indicando mais fortemente os livros:

FORBELLONE, André Luiz. Lógica de Programação. Makron Books, 3^a. Edição, 2005.

FARRER, Harry; BECKER, C. G.; FARIA, E.C.; MATOS, H.F.; SANTOS, M.A.; MAIA, M.L. Algoritmos Estruturados. Editora LTC,

3ª. Edição, 1999.

Outra fonte de pesquisa pode ser a internet. Você pode consultar pelo título de cada assunto (por exemplo, “Vetores” ou “Estrutura de Dados Unidimensional”) que conseguirá uma série de apostilas, apresentações, tutoriais, etc. Alguns exemplos de sites de outros professores são:

<http://www.manzano.pro.br/algoritmo.html>

http://200.136.79.4/informatica/alunos/tutoriais/C/LCPP_aulaIII_vetor.htm

[\(muito bom e com exercícios on-line para você responder!\)](http://www.ucb.br/programar/algoritmo/estruturadedadosV.htm)

<http://www.ic.unicamp.br/~ducatte/mc102/>

<http://mc102.unicamp.googlepages.com/agenda>

<http://david.coresoft.com.br/wiki/doku.php?id=ip>

http://www.gia.deinf.ufma.br/~geraldodisciplinas/intro_ee_2008_2.html

[http://almerindo.devin.com.br/index.php?option=content&task=vie w&id=138&Itemid=33](http://almerindo.devin.com.br/index.php?option=content&task=view&id=138&Itemid=33)

<http://www.airamez.com.br/Material.htm>

<http://www-usr.inf.ufsm.br/~pasin/algo/index.html>

<http://www.inf.ufrgs.br/~cora/Inf040/Aula06.ppt>

[\(apesar de mostrar o material em pascal, você vai reconhecer muitos dos nossos algoritmos nele\)](http://www.dcc.ufrj.br/~jonathan/docsPascal/apostila/capa.html)

Algumas vídeo-aulas no youtube (são uma seqüência de aulas, estou indicando apenas o link da primeira de cada tipo):

http://br.youtube.com/watch?v=3hv5_hWPle0

<http://br.youtube.com/watch?v=kaJ4nJHcYLE>

<http://br.youtube.com/watch?v=kGzFwgrZUXA>



Atividade de Estudo

Construa algoritmos em linguagem algorítmica para resolver os problemas abaixo. Lembre das boas práticas de programação!

ESTRUTURA UNIDIMENSIONAL

- 1) Leia um vetor chamado NUMEROS de 10 números inteiros e exiba qual o maior e o menor número desse vetor.
- 2) Estenda a questão 1 para, a partir do vetor NUMEROS completamente preenchido, escrever os vetores VP (que será um vetor de números pares) e VI (que será um vetor de números ímpares).
- 3) Estenda a questão 1 para, a partir do vetor NUMEROS, gerar o vetor OPOSTO, que corresponde ao vetor NUMEROS ao contrário. Por exemplo, se o vetor NUMEROS tiver os números 2 4 e 6, o vetor OPOSTOS terá os números 6 4 e 2.
- 5) Leia um vetor LESTRAS de 20 letras e, depois, escreva a quantidade de vogais que existem dentro do vetor e o percentual de consoantes.
- 6) Leia dois vetores A e B de tamanho 10. Depois gere, a partir deles, um vetor C, o qual corresponde à intercalação dos elementos dos vetores A e B. Ou seja, você deve colocar em C um elemento de A, depois um elemento de B, outro elemento de A, outro elemento de B e assim sucessivamente. Todos os elementos de A e B devem estar em C.
- 7) Estenda a questão 6 anterior para, a partir dos vetores A e B, gerar o vetor INTERSECAO que corresponde aos elementos que estão no vetor A e também estão no vetor B. Ou seja, o vetor INTERSECAO será um vetor com os elementos em comum entre os vetores A e B.

ESTRUTURA BIDIMENSIONAL

- 8) Leia duas matrizes A e B (5 X 5) de números inteiros e, a partir delas, gere a matriz SOMA (correspondente a soma das duas matrizes: A + B) e a matriz SUBTRACAO (correspondente à subtração entre as duas matrizes: A - B).

9) Leia uma matriz N (2X3) de números inteiros e positivos e gere uma matriz C, do mesmo tamanho, de caracteres, onde cada posição C[linha, coluna] terá o valor “P” ou “I” dependendo de se o número da posição correspondente em N[linha, coluna] for PAR ou IMPAR, respectivamente.

10)Leia uma matriz 4 x 4 de números inteiros e escreva os elementos da diagonal principal desta matriz. Para facilitar, observe o desenho abaixo. Em destaque está a diagonal principal. Como podem ser identificadas as posições desses elementos?

M	0	1	2
0	m_{00}	m_{01}	m_{02}
1	m_{10}	m_{11}	m_{12}
2	m_{20}	m_{21}	m_{22}

11)Leia as 4 notas de cada um dos 25 alunos de uma disciplina e escreva as notas que estão abaixo da média geral da turma e a porcentagem de notas acima da média da turma.

12)Leia uma matriz contendo o preço de 5 fornecedores para 10 produtos e calcule e imprima o preço médio de cada fornecedor para os 10 produtos.

13)Leia um vetor com os nomes dos 50 alunos de uma turma e uma matriz com as três notas desses alunos na mesma disciplina. Depois, calcule e escreva o nome dos alunos com média superior ou igual à média geral da turma.

DESAFIO: Leia um vetor com os nomes dos 500 candidatos de um concurso, outro vetor com o gabarito das respostas (“a”, “b”, “c”, “d” ou “e”) das 60 questões da prova e uma matriz com as respostas dos candidatos para a prova. Ao final, escrever os nomes dos candidatos que obtiveram a maior quantidade de acertos.

SUBLGORITMOS

14)Crie um procedimento para exibir a **tabuada da soma** de um número que deve ser passado como parâmetro. Por exemplo, se vier o número 1, você deve imprimir a tabuada da soma do número 1 (a soma de um com todos os outros números de 0

a 9). Faça também o algoritmo principal que vai chamar esse procedimento.

15) Escrever um algoritmo para determinar se um determinado número inteiro é positivo ou negativo. Utilizar um subalgoritmo de função que retorna um valor lógico para indicar se o valor recebido é positivo ou negativo.

16) Escreva um algoritmo que lê um número não determinado de valores, todos inteiros e positivos, um valor de cada vez, até que seja digitado o valor ZERO. A cada número, chamar uma função para calcular o fatorial do mesmo. Escrever cada número lido e seu fatorial. Se o número for negativo, dar uma mensagem apropriada.

17) Faça uma função (e o algoritmo principal que utiliza a mesma) que recebe a média final de um aluno por parâmetro e retorna o seu conceito, conforme a tabela abaixo:

Nota	Conceito

18) Faça um algoritmo que usa uma função que recebe, por parâmetro, a altura e o sexo de uma pessoa e retorna o seu peso ideal. Para homens, calcular o peso ideal usando a fórmula peso ideal = $72.7 \times \text{altura} - 58$ e, para mulheres, peso ideal = $62.1 \times \text{altura} - 44.7$.

Super Desafio

Escreva um algoritmo que apresente um menu com as seguintes opções:

- 0 Inicializar vetor
- 1 Inserir elemento no vetor
- 2 Remover elemento do vetor
- 3 Escrever na tela o Vetor

4 Sair

Quando for escolhida a opção número 0, um subalgoritmo apropriado deve ser ativado para colocar ZERO em todas as posições do vetor. O subalgoritmo deve receber como parâmetro o vetor a ser inicializado.

Quando for escolhida a opção número 1, um subalgoritmo apropriado é ativado para inserir um elemento (número inteiro positivo) em um vetor. O subalgoritmo deve receber como parâmetro o número a ser inserido e o vetor onde ele deve ser inserido. **Regras:** A inserção sempre será realizada no final do vetor (posições que possuam o valor ZERO são consideradas disponíveis). Se não for possível incluir no final (não há mais posições disponíveis no final do vetor), deve-se procurar se existe pelo meio do vetor alguma posição com valor ZERO. Se houver, inserir o elemento naquela posição. Se não houver, avisar ao usuário que o vetor está realmente cheio.

Quando for escolhida a opção número 2, um subalgoritmo apropriado é ativado para deletar um elemento do vetor. O subalgoritmo deve receber como parâmetro a posição (índice) do elemento a ser eliminado e o vetor. **Regras:** se a posição que se deseja deletar não existir ($\text{índice} > 30$) deve-se imprimir uma mensagem apropriada. Se a posição que se deseja deletar tiver valor zero (posição ainda sem elementos), também deve ser impressa uma mensagem apropriada. Cada elemento que for deletado deve ser substituído pelo valor ZERO.

Quando for escolhida a opção número 3, um subalgoritmo para imprimir todo o vetor na tela deverá ser ativado. O subalgoritmo deverá receber como parâmetro o vetor a ser impresso. **Regras:** só devem ser impressas as posições diferentes de ZERO. E ao final da impressão, dizer quantos números foram impressos (Total de números impressos = XX).

OBS: Considere que existem dois vetores de tamanho 30 de números inteiros e positivos.

Resumo

Neste fascículo, foram apresentados os conceitos avançados da linguagem algorítmica e um ambiente para programação usando essa linguagem (o que pode facilitar a prática dos exercícios). Além disso,

foi demonstrada uma forma de testar os algoritmos escritos, chamado teste de mesa.

Entre os conceitos avançados estudados estava o de estruturas homogêneas, que são conjuntos de dados de um único tipo agrupados sob um mesmo nome. Essas estruturas são classificadas conforme o número de dimensões do conjunto ou, em outras palavras, conforme a quantidade de índices necessários para localizar um elemento dentro do conjunto (geralmente, um índice para cada dimensão). Dessa forma, as estruturas podem ser: unidimensionais, bidimensionais, tridimensionais ou multidimensionais. O foco do nosso estudo nesta disciplina está apenas nas estruturas unidimensionais (vetores) e bidimensionais (matrizes).

Vetores são estruturas de dados homogêneas unidimensionais que permitem agrupar diversas informações dentro de uma variável única. Estas correspondem a um grupo de posições contínuas na memória que possuem o mesmo nome e o mesmo tipo de dado e são acessadas por um ÚNICO índice. Seu tamanho é definido por constantes inteiras e positivas e a definição do seu nome segue as mesmas regras aplicadas para identificadores

Matrizes são estruturas de dados homogêneas bidimensionais, ou seja, elas necessitam de dois índices para individualizar um elemento do conjunto: um índice para a linha e outro índice para a coluna. Por exemplo, declarar uma matriz M, de 4 linhas por 3 colunas (4×3), constituída de elementos numéricos inteiros fará com que passe a existir uma estrutura de dados agrupada na memória do computador, capaz de armazenar $4 \times 3 = 12$ elementos inteiros, endereçáveis por um par de índices, com o primeiro indicando a linha e o outro, a coluna.

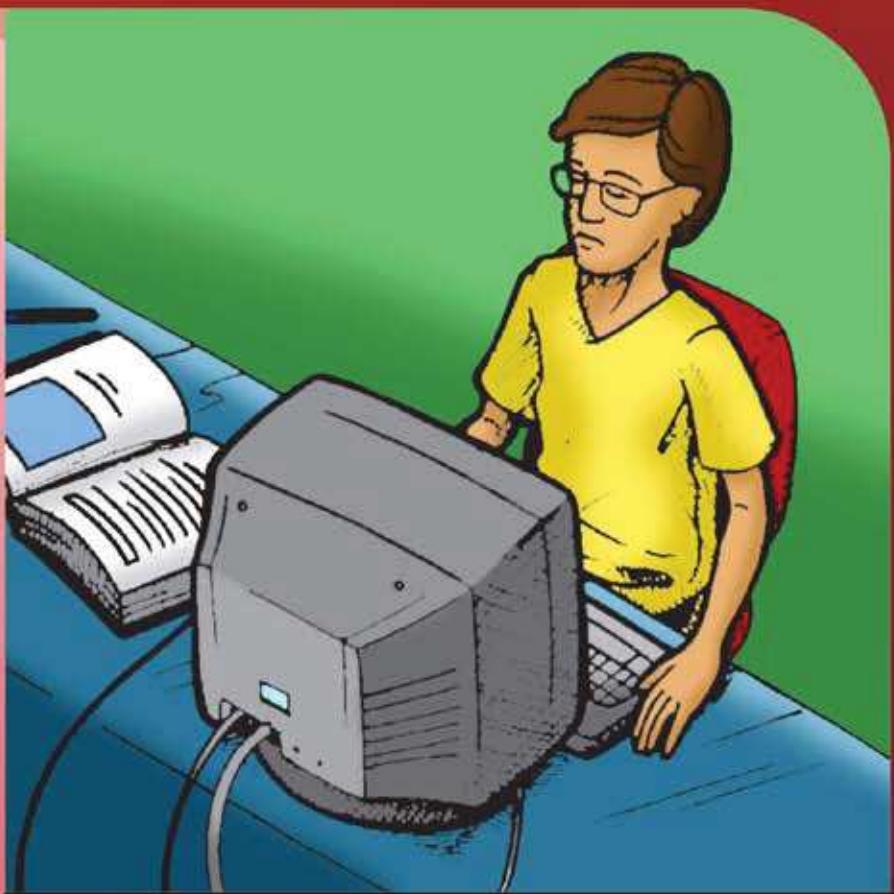
Outro conceito avançado estudado foi o de subalgoritmos. Subalgoritmo é um algoritmo que, geralmente, resolve um pequeno problema, e que está subordinado a um outro algoritmo. Esta subordinação deve-se ao fato de que o subalgoritmo só será acionado se solicitado pelo algoritmo principal (também é possível que um subalgoritmo chame outro subalgoritmo). Nesse caso, o algoritmo completo passa a ser dividido em um algoritmo principal e em diversos subalgoritmos (tantos quantos forem necessários ou convenientes para a resolução do problema). O algoritmo principal é aquele por onde começa a execução, e chama, eventualmente, os demais subalgoritmos.

Os subalgoritmos podem ser de dois tipos: procedimentos ou funções. A diferença entre eles é que as funções retornam a quem as chamar um valor, explicitamente. E o procedimento não. Ambos os tipos podem receber parâmetros. Os parâmetros podem ser recebidos por valor ou por referência. Quando a passagem é feita por valor, uma cópia do argumento do programa chamador é passada para o parâmetro. Assim o argumento original nunca é modificado. Quando a passagem é feita por referência, o argumento e o parâmetro passam a referenciar um mesmo endereço de memória. Dessa forma, qualquer modificação feita dentro do subalgoritmo no parâmetro, é refletida no argumento original, modificando-o. Indica-se que a passagem é por referência utilizando-se a palavra reservada VAR na declaração dos parâmetros do subalgoritmo.

Pronto! Agora finalizamos o primeiro módulo da disciplina! Finalizamos todo o estudo da linguagem algorítmica. Agora, já podemos partir para começar a implementar nossos programas em uma linguagem de programação. Essa linguagem será linguagem C e o estudo da mesma será o conteúdo de todo o segundo módulo. Até lá!

Introdução à Programação

Sandra de Albuquerque Siebra



FASCÍCULO 4



Universidade Federal Rural de Pernambuco

Reitor: Prof. Valmar Corrêa de Andrade

Vice-Reitor: Prof. Reginaldo Barros

Pró-Reitor de Administração: Prof. Francisco Fernando Ramos Carvalho

Pró-Reitor de Extensão: Prof. Paulo Donizeti Siepierski

Pró-Reitor de Pesquisa e Pós-Graduação: Prof. Fernando José Freire

Pró-Reitor de Planejamento: Prof. Rinaldo Luiz Caraciolo Ferreira

Pró-Reitora de Ensino de Graduação: Profª. Maria José de Sena

Coordenação de Ensino a Distância: Profª Marizete Silva Santos

Produção Gráfica e Editorial

Capa e Editoração: Allyson Vila Nova, Rafael Lira e Alesanco Andrade

Revisão Ortográfica: Ivanda Martins

Ilustrações: Allyson Vila Nova

Coordenação de Produção: Marizete Silva Santos



Sumário

Fascículo 4 - Linguagem de Programação C	6
Objetivos do Fascículo 4	6
Conteúdo Programático do Fascículo 4	6
4.1 Unidade 6 – Noções Básicas da Linguagem C	8
4.1.1 Breve Histórico da Linguagem C.....	9
4.1.2 A Linguagem C	10
4.1.3 O Dev C++.....	11
4.1.4 Nosso Primeiro Programa em C.....	14
4.1.5 Estrutura de um Programa em C.....	18
4.1.6 Comentários	19
4.1.7 Diretivas de Compilação.....	20
4.1.8 Palavras Reservadas	22
4.1.9 Identificadores	22
4.1.10 Constantes	23
4.1.11 Variáveis	29
4.1.12 Tipos de Dados Básicos (ou Tipos Primitivos)	30
4.1.13 O Tipo String.....	31
4.1.14 Modificadores de Tipos de Dados	32

4.1.15 Declaração de Variáveis.....	33
4.1.16 Inicialização de variáveis.....	37
4.1.17 Operadores e Expressões.....	38
4.1.18 Entrada e Saída de Dados	49
4.1.19 Impressão Formatada de Dados	50
4.1.20 Leitura Formatada de Dados	55
4.1.21 Outras Formas de Entrada e Saída e Dados	59
4.1.22 Aprenda Praticando	62

Fascículo 4 - Linguagem de Programação C

Carga horária do Módulo: 45 horas

Objetivos do Fascículo 4

- Apresentar os componentes básicos de um programa utilizando uma linguagem de programação estruturada
- Apresentar uma ferramenta para compilação de programas em uma linguagem estruturada
- Desenvolver a habilidade de programação fazendo uso de variáveis, operadores, expressões, estruturas seqüenciais, estruturas de repetição, tipos estruturados de dados (homogêneos e heterogêneos) e modularização de programas
- Incentivar a prática de desenvolvimento de programas de forma a aprimorar no aluno a sua lógica de resolução de problemas

Conteúdo Programático do Fascículo 4

Unidade 6 – Noções Básicas da Linguagem C (10 horas)

- Ambiente de Programação C
- Comentários, palavras reservadas e identificadores
- Constantes e Variáveis
- Tipos de dados
- Declaração de Variáveis
- Operadores e Expressões
- Entrada e Saída de Dados
- Estruturas Seqüenciais

Unidade 7 – Estruturas de Controle de Fluxo (7 horas e meia)

- Estruturas Condicionais
- Estruturas de Repetição
- Desenvolvimento e Depuração de Programas

Unidade 8 – Tipos Estruturados de Dados (10 horas)

- Tipos de Dados Homogêneos: vetores e matrizes
- Tipo de Dados Heterogêneo: registros

Unidade 9 – Subalgoritmos (10 horas)

- Modularização: funções, procedimentos, passagem de parâmetros e recursão

Unidade 10 – Desenvolvimento de Projeto (7 horas e meia)

- Bibliotecas e Funções Úteis
- Guia para o desenvolvimento de projetos
- Exemplos Práticos

Cada unidade conterá exercícios para fixação do assunto apresentado, assim como indicações de leituras adicionais. A Unidade 6 exigirá 10 horas de dedicação, uma vez que você estará começando com o ambiente de programação que usa a linguagem C. A unidade 7 exigirá 7 horas e meia. As unidades 8 e 9 exigirão um pouco mais: 10 horas de dedicação e lhe restarão 7 horas e meia para desenvolvimento do projeto final da disciplina, para o qual você encontrará diversas orientações na Unidade 10. Tenha em mente que a maior parte dos conceitos que serão vistos nessas unidades, você já utilizou em linguagem algorítmica (com exceção de registros e recursão), logo, a novidade será implementar os conceitos na linguagem C.

Mais do que nunca você vai precisar praticar. Logo, arregace as mangas e vamos lá!

4.1 Unidade 6 – Noções Básicas da Linguagem C

Pronto! Chegou a hora! Agora, vamos começar a programar de verdade! Não que o que você estava fazendo até o momento não fosse programação. Sim, era! Mas em uma pseudolínguagem. O que era criado era um algoritmo e não um programa. Pois, como vimos na unidade 1, um programa = algoritmo traduzido para uma linguagem de programação. E a primeira verdadeira linguagem de programação que você vai estudar neste curso é a linguagem C.

Mas por que C? Por diversos motivos que vamos explicar mais à frente nesta unidade, mas entre eles, destacamos dois: 1) C é a segunda linguagem mais usada no mundo, sendo superada apenas pela linguagem Java, (Fonte: <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>). Porém, se considerarmos apenas linguagens estruturadas, C é a primeira colocada. 2) C tem excelente desempenho e é extremamente versátil, podendo ser usada para desenvolver os mais diferentes tipos de softwares. Para ilustrar, C foi usada para desenvolver muitos dos softwares que conhecemos tais como:

- o sistema operacional UNIX ,
- a planilha eletrônica Microsoft Excel,
- o gerenciador de base de dados Microsoft Access,
- um montador que foi muito famoso nos anos 90: o clipper,
- utilitários como o editor de formulários FormTool,
- diversos programas de automação industrial,
- linguagens como o power builder e o visual basic

Além disso, C inspirou a criação de algumas das linguagens visuais mais populares dos dias de hoje: Java, C++ e C#. E foi usado para criar muitos dos efeitos especiais de filmes como Star Trek (Jornada nas Estrelas) e Star War (Guerra nas Estrelas).

Atenção

 Antes de começar,
um lembrete:
Programação
só se aprende
praticando!

E aí, você imaginava que C podia fazer tudo isso? Pois é! Vamos aprender C!

A melhor forma de estudar uma linguagem de programação é implementando, na prática, os algoritmos. Sugiro que você realmente implemente os exemplos e exercícios apresentados neste fascículo, usando o ambiente de programação indicado. Pois apenas estudar,

ler os exemplos de forma teórica não basta!

Deixe eu lhe dar um exemplo: eu tenho uma grande amiga que sabe toda a teoria sobre cozinhar e até, na casa dela, existem todos os equipamentos, panelas e apetrechos mais modernos para a cozinha. Ela sabe, inclusive, dezenas de receitas decoradas. Porém, na prática, quando vai para a cozinha, ela é um desastre! Isso porque ela quase nunca pratica, quase nunca cozinha, apenas gosta muito de ler livros de receitas e de assistir a programas de culinária. Você não vai querer ser uma pessoa que sabe programar só na teoria né?

Até porque só quando utilizamos o compilador aprendemos os “macetes” da programação. Aprendemos a lidar com as mensagens de aviso e mensagens de erro. Aprendemos a corrigir os erros encontrados e nos deparamos com a oportunidade única de ver o computador fazer o que mandamos!

Para aprender bem a usar a linguagem C é necessário, além do domínio da sintaxe da linguagem em si, uma familiaridade com o compilador e a experiência em achar erros (“bugs”) nos programas. Por isso mesmo, não esqueça: digite, compile e execute os exemplos apresentados. Você vai ver como isso vai acelerar o seu aprendizado!
Bons Estudos!

4.1.1 Breve Histórico da Linguagem C

Por que esse nome C? De onde veio a linguagem C? Como ela surgiu? Quem a criou? Antes de lhe apresentar a linguagem, vamos responder a essas perguntas.

A origem do nome da linguagem C é muito simples. É a linguagem que sucede a linguagem B. Por sua vez, a linguagem B teve seu nome retirado da inicial do local onde ela foi desenvolvida: Laboratórios Bell.

A primeira versão de C foi criada por Dennis Ritchie, em 1972, nos laboratórios Bell para ser incluído como um dos softwares a serem distribuídos juntamente com o sistema operacional Unix do computador PDP-11, na equipe certificada por Ken Thompson. Foi inicialmente publicada no livro “The C Programming Language” (tido como “a bíblia” da linguagem C), por B.W. Kernighan & D.M. Ritchie em 1978.

Sua primeira utilização importante foi a reescrita do Sistema

Operacional UNIX, que até então era escrito em assembly. Em meados de 1970 o UNIX saiu do laboratório para ser liberado para as universidades. Foi o suficiente para que o sucesso da linguagem atingisse proporções tais que, por volta de 1980, já existiam várias versões de compiladores C oferecidas por várias empresas, não sendo mais restritas apenas ao ambiente UNIX, porém compatíveis com vários outros sistemas operacionais (tais como o windows, DOS, etc).

Em 1983, a ANSI (American National Standards Institute) fundou uma comissão para definir uma versão padronizada para a linguagem C. Esta versão chamou-se ANSI C. Desta forma, simplesmente compilando o código fonte em qualquer sistema, um programa escrito em ANSI C funciona em praticamente qualquer computador. E todos os ambientes de programação em C entendem o ANSI C (sendo que alguns deles tem mais bibliotecas e funções além dessa parte padronizada, que é específica do ambiente/implementação).

É justamente a estrutura do ANSI C, o C padronizado pela ANSI, o foco principal do estudo deste fascículo.

4.1.2 A Linguagem C

A linguagem C é uma linguagem de programação genérica, ou seja, ela serve para o desenvolvimento de programas em áreas diversas, como comercial, científica e de engenharia. Foi desenvolvida por programadores para programadores tendo como meta características de flexibilidade e portabilidade (o ANSI C). Algumas características da linguagem C são:

- C é uma linguagem compilada: lê todo o código fonte e gera o código objeto (ling. de máquina).
- C é “case sensitive”, ou seja, ele faz distinção entre maiúsculas e minúsculas. Dessa forma as palavras ANA, ana, Ana e AnA são consideradas diferentes pelo C.
- O C é Portável (se for usado apenas o padrão ANSI)
- Gera código compacto e rápido, sendo indicado para construção de compiladores, editores de texto, bancos de dados, softwares de comunicação, drivers, sistemas operacionais, etc.

Outra característica interessante da linguagem C é que ela pode

ser considerada como uma linguagem de médio nível. Isso porque ela possui instruções que a tornam ora uma linguagem de alto nível e estruturada como o Pascal, se assim se fizer necessário, ora uma linguagem de baixo nível, pois ela possui instruções tão próximas da máquina, como as que só o Assembler possui. De fato com a linguagem C podemos construir programas organizados e concisos (como o Pascal), ocupando pouco espaço de memória e com alta velocidade de execução (como o Assembler). Infelizmente, dada toda a flexibilidade da linguagem, também poderemos escrever programas desorganizados e difíceis de serem compreendidos, por isso, precisamos ter cuidado na implementação dos nossos programas na linguagem C!

A linguagem C pode gerar programa executáveis. A Figura 1 ilustra o processo de geração do código executável final (código que pode ser executado pelo sistema operacional). Nesta figura, no primeiro passo o pré-processador realiza funções como: o processamento de macros e a inclusão de arquivos (referências a arquivos são substituídas pelo próprio arquivo), preparando o código fonte para ser compilado. Depois, o compilador analisa o código fonte e o converte para o código-objeto (linguagem de máquina). Por fim, na última etapa, o ligador (ou linkeditor) junta o código-objeto com as bibliotecas necessárias para execução do mesmo, a fim de gerar o programa executável.



Figura 1 - Geração de Executável

4.1.3 O Dev C++

Existem diversos compiladores para a linguagem C, como por exemplo: Turbo C (16 bits), MS-Visual C, Borland C-Builder, gcc

(padrão Linux), TcLite, KDevelop, Anjuta, Eclipse CDT, Code::Blocks, BabyDevelop, Dev C++, entre outros. Entre esses escolhemos o Dev C++ para utilizar nesta disciplina. Por que? Bem, vamos explicar.

Primeiro motivo, o Dev-C++ foi desenvolvido por Colin Laplace, Mike Berg e Hongli Lai e é um software livre sob a GNU General Public License. Isto significa, entre outras coisas, que ele pode ser distribuído e copiado à vontade, gratuitamente.

Segundo, para facilitar o desenvolvimento e compilação de um programa em C, o Dev-C++ (vide Figura 2) oferece um ambiente que integra um editor de textos a um compilador para linguagem C, além de ter ferramentas e facilidades para “debugar” um programa, ou seja, procurar pelos erros existentes no mesmo. Além disso, ele usa cores diferentes para destacar comandos, comentários, textos e palavras reservadas, tem a capacidade de editar múltiplos arquivos e, nele, é possível trabalhar com programas-fontes isolados ou usando a noção de arquivos de um projeto.

Terceiro, o Dev-C++ permite criar programas para console ou interface gráfica e, também, criar bibliotecas estáticas ou dinâmicas.

Quarto, o compilador usado pelo Dev-C++ é um dos mais antigos e mais utilizados compiladores C do mundo: o gcc (Gnu C Compiler).

E quinto e último motivo: o Dev-C++ é fácil de instalar, pequeno para baixar e tem uma versão em português.

O Dev-C++ possui versão para windows e linux. Porém, curiosamente, a versão para linux está sendo vendida e a para windows é gratuita.

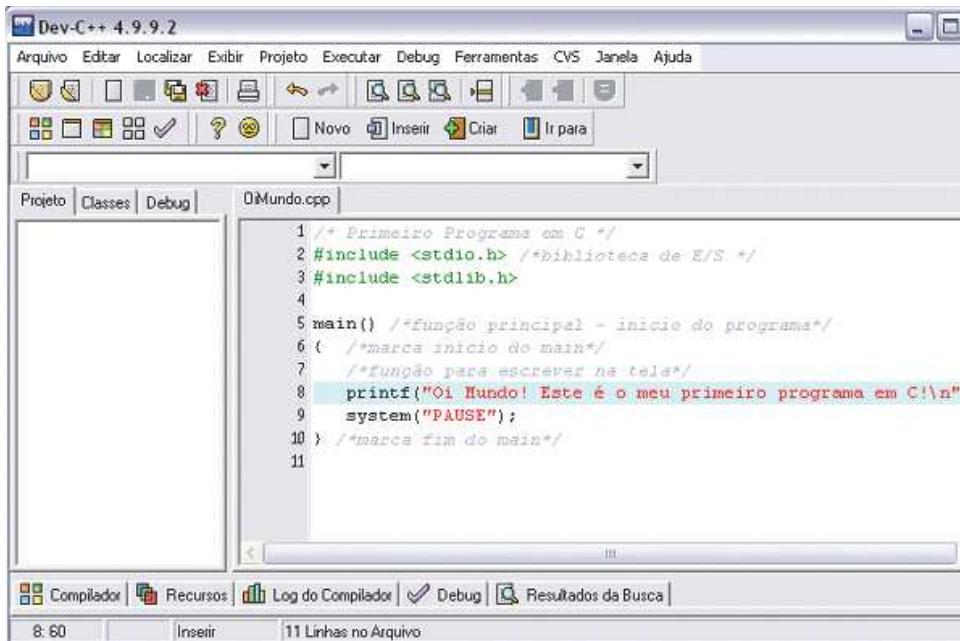


Figura 2- Interface do Dev-C++

VNVVExplicados os motivos da escolha, onde você pode baixar o Dev-C++? Além dos vários links que podem ser achados usando uma ferramenta de busca, os sites oficiais e mais confiáveis são: <http://bloodshed.net/devcpp.html> e <http://sourceforge.net/projects/dev-cpp/>

Para instruções sobre como instalar o Dev-C++ (que não é nada complicado), você pode consultar o “Introdução ao DevC++” que está disponível no ambiente Moodle da disciplina, como material complementar. Você também vai encontrar um guia de instalação no link: http://www.inf.pucrs.br/~flash/cbp/installa_devc/

Apesar da versão do Dev-C++ para linux ser vendida e ainda não estar tão estável, os usuários linux tem uma ampla variedade de ambientes que podem ser utilizados, tais como o KDevelop (que dá suporte ao desenvolvimento em diversas linguagens e não apenas o C e o C++) e está disponível em: <http://www.kdevelop.org>), o Anjuta (disponível em: <http://www.anjuta.org>) e o codeblock (<http://www.sci.brooklyn.cuny.edu/~goetz/codeblocks/>). Os dois primeiros são exclusivamente para linux e o último possui instaladores para diversas versões do windows, do linux e do macintosh.

Como serão os nossos programas? Bem, no Windows, há basicamente dois tipos de programas de usuário:

- Aplicações para o console (Console Applications), que são executadas numa janela de texto (também conhecida como

Prompt do MS-DOS), fornecida, muitas vezes, pelo próprio ambiente de desenvolvimento e

- Aplicações Gráficas (GUI Applications), que usam janelas, menus e outros elementos visuais como partes de sua interface com o programador.

Nesta disciplina, todos os programas em C que faremos serão aplicações para o console (vide exemplo na Figura 3).

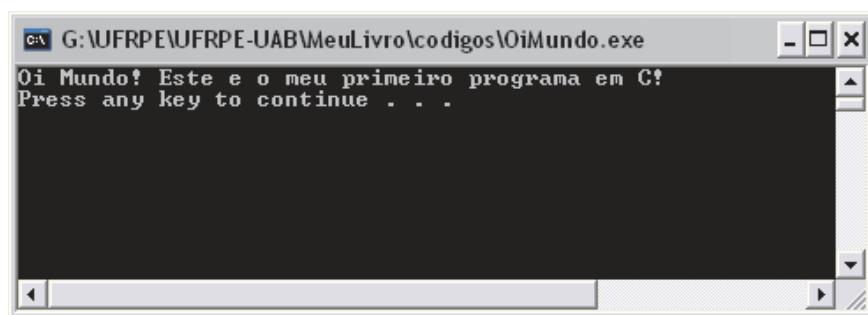


Figura 3 - Exemplo de Execução de Aplicação para Console

4.1.4 Nosso Primeiro Programa em C

Para que você possa usar o ambiente Dev-C++ e praticar como digitar, compilar e executar um problema, nada melhor do que o famoso programa “Oi Mundo!” (ou também chamado “HelloWorld”). Ele é bem simples, apenas imprime na tela uma mensagem, mas, no aprendizado de qualquer linguagem, geralmente, se passa por esse programa. Como mencionado, a idéia é ter o primeiro contato com o ambiente Dev-C++ e com a linguagem C. Logo, não se preocupe em entender detalhes da sintaxe do programa agora. Vamos lá?

Passo 1: Ative o Dev-C++ (deve haver um ícone na área de trabalho – Figura 4) ou você pode ativá-lo acessando o menor Iniciar >> Programas >> Bloodshed Dev-C++ >> Dev-C++. Com o ambiente aberto, vamos criar o nosso primeiro programa. Acesse o menu Arquivo e escolha as opções Novo >> Arquivo Fonte (ao invés disso, você também poderia ter clicado no terceiro ícone da esquerda para a direita, da primeira fileira, representado por uma folha em branco). Dessa forma, o ambiente estará pronto para você digitar o seu primeiro programa (tal qual pode ser visto na Figura 2, anteriormente apresentada). Sendo assim, digite o código abaixo apresentado. Todos os textos delimitados por /* e */ são comentários. Ou seja, não serão traduzidos pelo compilador, servem apenas para a documentação do

código.



Figura 4 - Ícone do Dev-C++

```

1. /* Primeiro Programa em C */
2. #include <stdio.h> /* uso de biblioteca */
3. #include <stdlib.h> /* uso de biblioteca */
4. main() /* função principal – inicio do programa */
5. { /*marca inicio do main */
6.     printf("Meu primeiro programa em C \n"); /*função para escrever na tela*/
7.     system("PAUSE");
8. } /* marca fim do main */
  
```

Passo 2: Após digitar o código acima, salve o programa usando o quarto ícone da esquerda para a direita, da primeira fileira (representado por um disquete) ou use as opções de menu Arquivo >> Salvar (ou ainda a tecla de atalho Ctrl + S). Grave o programa com o nome **oiMundo.c** Todo programa na linguagem C, tem a extensão (letras depois do ponto no nome do arquivo) **.c**

Lembre sempre de salvar os seus programas! Se for um programa grande, vá salvando por partes, para evitar perdê-lo se houver algum pique de energia, ou se o computador travar por qualquer motivo que seja.

Passo 3: Uma vez com o programa pronto e salvo, vamos compilá-lo. Para compilar um programa no Dev-C++ você deve escolher pelo menu Executar >> Compilar ou usar o primeiro botão da esquerda para a direita da segunda fileira de ícones (indicado na Figura 5). Se não houver erro, você deve visualizar a sub-janela “Compile Progress” vista na Figura 5, na qual você deve clicar no botão Fechar. Observe que a janela indica que não houve nenhum erro.

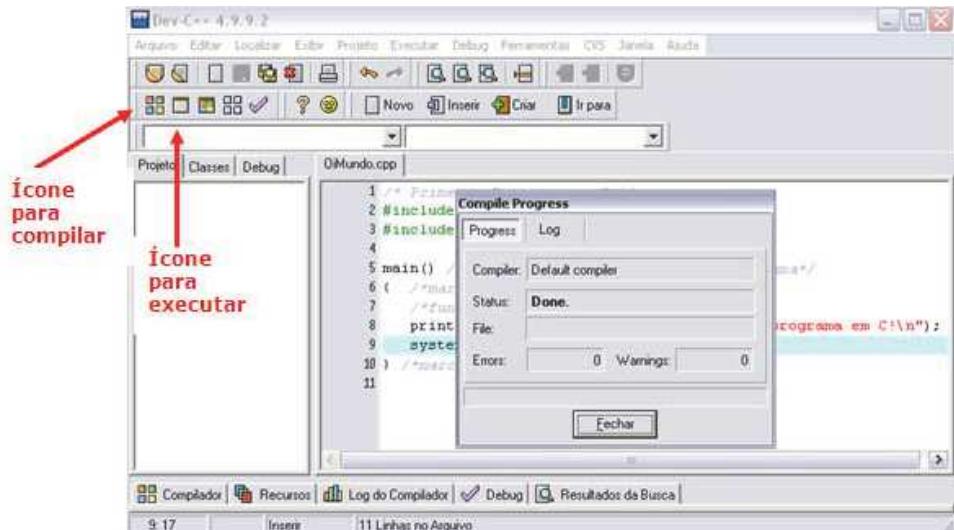


Figura 5 - Janela que indica o estado da compilação

Se houver erros na compilação, será apresentada uma lista de erros na parte inferior da janela principal (Figura 6) e a linha do primeiro erro aparecerá selecionada. Normalmente, os erros mais comuns são: erros de digitação (escrita equivocada das palavras), esquecimento de ponto-e-vírgula no final dos comandos (que é o caso do erro apresentado na Figura 6, onde falta um ponto-e-vírgula no final do comando *printf*, antes do comando *system*), falta de aspas, falta de parênteses ou chaves. Tenha muita atenção com letras minúsculas e maiúsculas, pois, elas representam coisas diferentes para a linguagem C, logo, escrever *Printf* é diferente de escrever *printf* (o correto!). Se o programa apresentar erros, eles devem ser corrigidos e, depois, o programa deve ser re-compilado.

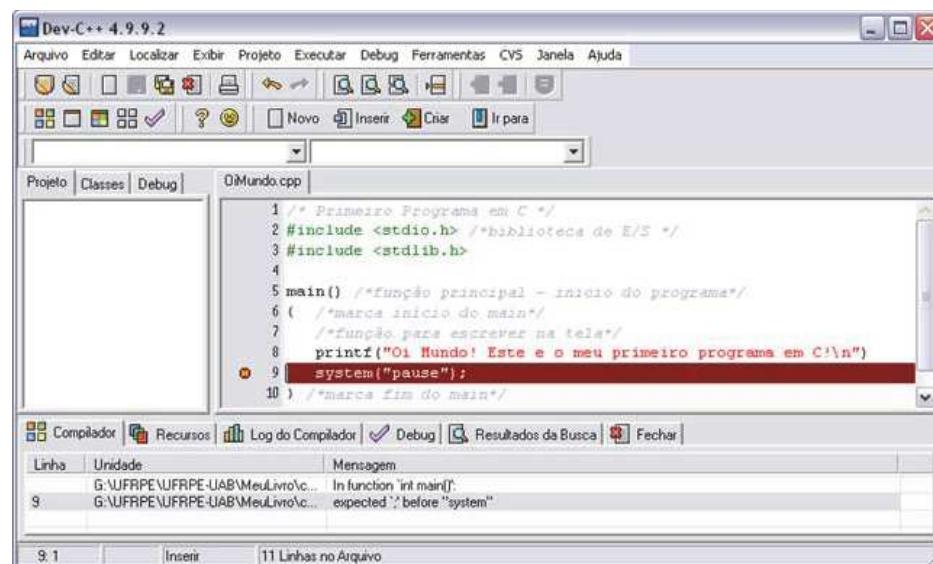


Figura 6 - Código apresentando erros de compilação no Dev-C++

Se não houver erros no programa e a compilação tiver sido executada a contento, você pode, então, executar o programa de uma das seguintes formas: pelo menu Executar >> Executar ou usando o segundo botão da esquerda para a direita da segunda fileira de ícones (indicado na Figura 5). Ou usando as teclas de atalho Ctrl + F10. O resultado da execução será a janela já apresentada na Figura 3, onde a frase “Oi Mundo! Este é o meu primeiro programa em C!” está impressa.

E aí, conseguiu digitar, compilar e executar o programa? Se sim, ótimo! Se não, tente novamente, até conseguir, ok? Só depois disso, continue a ler este texto.

Em todas as linhas do programa existem comentários. Como anteriormente mencionado, os comentários não são traduzidos para linguagem de máquina pelo compilador, são simplesmente ignorados pelo mesmo. Ex: /* Primeiro Programa em C */

Vejamos agora a segunda e terceira linhas do programa:

2. `#include <stdio.h> /* uso de biblioteca */`
3. `#include <stdlib.h> /* uso de biblioteca */`

Estas linhas usam a diretiva **#include** para inserir os arquivos de cabeçalho stdio.h e stdlib.h no seu programa. Os arquivos de cabeçalho são arquivos que contém definições, macros e funções que são muito utilizadas em todos os programas. Eles também podem ser considerados bibliotecas da linguagem. Normalmente, todo compilador C já vem com muitos arquivos de cabeçalho, mas você também pode escrever os seus. Os arquivos de cabeçalho tem a extensão .h

4. `main() /* função principal – inicio do programa */`

A linha 4 indica a definição da função **main**. Um programa em C é formado por várias funções. Quando o programa é executado main é a primeira função a ser chamada, assim todo programa deve ter uma função main, que também é conhecida como “programa principal”.

As chaves das linhas 5 e 8 delimitam um bloco de comandos. No caso, no nosso programa elas estão delimitando o programa principal (função main).

```
7. printf("Meu primeiro programa em C \n"); /*função para escrever na tela*/
```

A função **printf** exibe o conteúdo do seu argumento na tela. No caso a mensagem “Meu primeiro programa em C”. Esta função faz parte do arquivo de cabeçalho (biblioteca) stdio.h. Veremos mais detalhes sobre essa função posteriormente. A caracter especial \n significa uma diretiva para indicar que se deve “pular uma linha”.

Todo comando na linguagem C deve ser finalizado com um ; (ponto-e-vírgula).

7. system ("PAUSE");

Para executar um programa para console, uma janela do DOS é aberta e, quando o programa acaba, ela é automaticamente fechada. Dessa forma, não é possível ver o resultado da execução. Para evitar essa situação utiliza-se a função **system("PAUSE")**; que vai evocar o comando PAUSE do sistema operacional, dessa forma, esse comando faz o programa ficar esperando que uma tecla seja pressionada para poder continuar. Esse comando faz parte da biblioteca stdlib.h, por isso foi preciso incluir essa biblioteca no programa.

Se você não entendeu algum dos conceitos ditos acima não se preocupe, apenas aceite (só por enquanto!) e no futuro você entenderá o porquê disso tudo.

Você pode dar uma olhada em uma aula dirigida de introdução ao Dev-C++ em: <http://www.ime.usp.br/~mac2166/devcpp/devcppintro/index.html>

4.1.5 Estrutura de um Programa em C

Um programa em C seguirá sempre a seguinte estrutura:

```
/* Comentário com a descrição do programa */
/* Diretivas de Pré-processamento */
#include <biblioteca.h>
#define .... /* declaração de constantes */
/* Declaração de variáveis globais */
/*Protótipos de Funções*/
Tipo função1 (declaração de parâmetros){
```

```

/*declarações locais*/
/*comandos*/
}
int main( ) /* programa principal */
{
    /* declarações e comandos */
    return 0; /* término normal da função main */
}

```

Tudo isso será armazenado em um arquivo com a extensão .c. Essa é apenas uma orientação geral, vamos mostrar no decorrer deste fascículo, como preencher cada parte dessa estrutura.

Regras gerais para um programa em C:

- Cada bloco de instruções é delimitado por { e } que são equivalentes ao inicio e fim da linguagem algorítmica.
- Após cada comando dentro de um bloco de instruções, deve-se colocar um ; (ponto e vírgula).
- Todo programa em C deve ser identado para que possa ser lido com mais facilidade.
- Todo programa começa a ser executado pela função main (que é o programa principal).

4.1.6 Comentários

Em C, comentários podem ser escritos em qualquer lugar do programa para facilitar o entendimento do mesmo. Os comentário iniciam com o símbolo /* e se estendem até aparecer o símbolo */ e podem se estender por mais de uma linha. Por Exemplo:

```
/* Esta é uma linha de comentário em C */
```

Os comentários servem, principalmente, para documentação do programa e são ignorados pelo compilador, portanto não irão afetar o programa executável gerado. Ou seja, eles permitem a inclusão de observações que auxiliam a compreensão do que está sendo executado pelo programa.

DICA: Você deve procurar SEMPRE iniciar os programas com um comentário contendo a descrição do programa. Para isso, você pode usar o recurso do DevC++ Editar/Inserir/**Comentário de cabeçalho**, que insere automaticamente as seguintes linhas em um programa:

```
/*
Name:
Copyright:
Author:
Date:
Description:
*/
```

Ou seja, já deixa pronto o comentário para inserir as seguintes informações, em seqüência: o nome do programa, informações sobre direitos autorais, o nome do autor do programa, a data de criação do programa e uma breve descrição do objetivo do programa, ou seja, o que o programa faz ou para que serve.

4.1.7 Diretivas de Compilação

Em C, existem comandos que são processados durante a compilação do programa. Estes comandos são, genericamente, chamados de diretivas de compilação. Estes comandos informam ao compilador C, basicamente, quais bibliotecas devem ser anexadas ao programa executável e quais são as constantes simbólicas usadas no programa (o `#define` que explicaremos mais na frente).

A diretiva `#include` diz ao compilador para incluir na compilação do programa outros arquivos. Geralmente, estes arquivos contêm bibliotecas de funções ou rotinas do usuário. Isso é bastante útil porque a linguagem C é bastante extensa e possui várias bibliotecas de funções disponíveis para executar as mais diversas tarefas. Toda vez que chamamos uma função que não foi declarada no nosso programa fonte, estamos usando uma função de uma biblioteca que está disponível na linguagem.

Por exemplo, no nosso primeiro programa em C (seção 4.1.4), usamos a função `printf` para imprimir a mensagem na tela. Essa função faz parte da biblioteca `stdio.h` disponível no C. `stdio` é a abreviação de Standard Input Output que é a biblioteca padrão de Entrada (leitura

do teclado e/ou arquivos) e Saída (escrita na tela e/ou arquivos). Sempre que precisamos usar uma biblioteca, nós devemos inserir no programa uma diretiva de compilação `#include` para que o compilador inclua durante a compilação as definições daquela biblioteca. Por isso, como a função `printf` requer a biblioteca `stdio`, devemos incluir no início do nosso fonte a linha `#include <stdio.h>`. Da mesma forma, existem outras funções que necessitam de outros arquivos de cabeçalho (bibliotecas). Para utilizar estas outras funções devemos incluir outras bibliotecas. A sintaxe da instrução `#include` é a seguinte:

```
#include <biblioteca.h>
```

Onde: biblioteca é o nome da biblioteca a ser incluída na compilação do seu programa fonte. Todo arquivo biblioteca termina com `.h` (indicando que é um arquivo de cabeçalho, do inglês header), da mesma forma que os arquivos fonte em C terminam com `.c`. A Tabela 1 apresenta algumas das principais bibliotecas da linguagem C padrão (ANSI C).

Onde: biblioteca é o nome da biblioteca a ser incluída na compilação do seu programa fonte. Todo arquivo biblioteca termina com `.h` (indicando que é um arquivo de cabeçalho, do inglês header), da mesma forma que os arquivos fonte em C terminam com `.c`. A Tabela 1 apresenta algumas das principais bibliotecas da linguagem C padrão (ANSI C).

Tabela 1 – Principais Bibliotecas da Linguagem ANSI C

Arquivo	Descrição
<code>stdio.h</code>	Funções de entrada e saída (I/O)
<code>string.h</code>	Funções de tratamento de strings
<code>math.h</code>	Funções matemáticas
<code>ctype.h</code>	Funções de teste e tratamento de caracteres
<code>stdlib.h</code>	Funções de uso genérico

Também pode-se usar a forma: `#include "nome_arquivo"`, que serve, normalmente, para incluir algum arquivo que tenha sido criado pelo próprio programador ou por terceiros e que se encontre no diretório atual, ou seja, no mesmo diretório do programa que está sendo compilado.

Note que a instrução `#include` não é terminada com ponto-e-vírgula

como acontece com outros comandos. Na verdade, qualquer linha de um programa em C que for iniciada com # não pode ter um ponto-e-vírgula para terminá-la.

4.1.8 Palavras Reservadas

Palavras reservadas são identificadores associados aos comandos que compõem a linguagem de programação. Por isso, essas palavras não podem ser usadas para referenciar variáveis ou constantes, elas são de uso restrito da linguagem C (comandos, estruturas, declarações, etc.). No C padrão ANSI existem 32 palavras reservadas. São elas:

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

Como já mencionado anteriormente, a Linguagem C diferencia letras maiúsculas e minúsculas. Logo, é preciso atenção na hora de escrever as palavras reservadas, porque, por exemplo, **int** é uma palavra reservada da Linguagem C, mas **Int** ou **INT** não são.

4.1.9 Identificadores

A Linguagem C chama o que é usado para referenciar ou nomear variáveis, funções, rótulos e vários outros objetos definidos pelo usuário de *identificadores*. Para criação dos nomes de identificadores devem-se considerar as seguintes regras:

- Os identificadores devem começar por uma letra ('A'..'Z', 'a'..'z') ou pelo underscore ('_'). O resto do identificador deve conter apenas letras, underscores ou dígitos (0 - 9), não pode conter outros caracteres especiais, tais como \$, @, &, etc.
- Apesar de ser permitido, deve-se evitar o uso do '_' (underscore)

no primeiro caractere do identificador de uma variável, pois este tipo de identificador é de uso do sistema operacional

- Em C, os identificadores podem ter até 32 caracteres. Na verdade, apenas os primeiros 32 caracteres de um nome de identificador são significativos. Isso quer dizer que duas variáveis com os 32 primeiros caracteres em comum, diferindo somente no 33º, são consideradas iguais.
- Como estamos repetindo diversas vezes, em C, letras maiúsculas e minúsculas são tratadas como diferentes e distintasumas das outras. Por isso, contador, Contador e CONTADOR serão considerados três identificadores distintos.
- Não se deve usar acentuação no nome dos identificadores, nem fazer uso do cedilha (ç) ou de algum sinal de pontuação.
- Um identificador não pode ter o mesmo nome de uma palavra reservada.
- Deve-se procurar usar nomes significativos para o contexto do problema sendo resolvido nos identificadores. Evitando identificadores tais como: x, y e z.
- Um identificador de variável ou constante não deve ter o mesmo nome de uma função – tanto uma função que você tenha escrito como uma função pertencente a uma das bibliotecas da linguagem C.

Alguns exemplos de identificadores válidos são: nota1, media, nome, VALOR_MAXIMO e cnpj_cliente.

Alguns exemplos de identificadores que não válidos são: 1dia, matriz-1, média, cnpj.cliente, nome&sobrenome. Se prestar atenção, cada um desses identificadores viola alguma das regras listadas anteriormente.

4.1.10 Constantes

Constantes são identificadores que não podem ter seus valores alterados durante a execução do programa. Elas são usadas em expressões para representar vários tipos de valores. A maneira como cada constante é representada depende do seu tipo. Por exemplo, constantes de caracteres são envolvidas por aspas simples, por exemplo, 'a' e 't'. Constantes inteiras são especificadas como

números sem a parte de ponto flutuante, por exemplo, 10, -3 e 45. Constantes de ponto flutuante requerem o ponto decimal, seguido pela parte fracionária do número, por exemplo, 10.45 e -2.97. A seguir comentaremos um pouco mais sobre cada tipo de constante da linguagem C, uma vez que essa linguagem possui regras rígidas para determinar como devem ser escritos estes valores.

Constantes inteiras

Uma constante inteira é um número de valor inteiro, ou seja, sem a parte decimal ou de ponto flutuante. De uma forma geral, constantes inteiras são seqüências de dígitos que representam números inteiros, precedidos ou não por um sinal (lembre que, quando o sinal for representado, não se deve separar ele do valor numérico!). Em C, os números inteiros podem ser escritos no formato *decimal* (base 10), *hexadecimal* (base 16) ou *octal* (base 8) – lembra dos sistemas de representação numérica?¹

Atenção

 ¹ Se não lembra dos sistemas numéricos usados na informática, você pode dar uma olhada em:
<http://www.forum.unidev.com.br/phpbb2/viewtopic.php?t=37334>,
<http://bitsebites-sistemasnumericos.blogspot.com/2008/05/converso-de-bases.html> ou http://pt.wikiversity.org/wiki/L%C3%B3gica_Digital/Sistemas_num%C3%A9ricos.html

Uma constante inteira *decimal* é formada por uma seqüência de dígitos decimais: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Se a constante tiver dois ou mais dígitos, o primeiro não pode ser 0 senão o compilador considerará esta constante como octal e não decimal. Exemplos válidos: 0, 5, 23450 e -30. Alguns exemplos inválidos de constantes decimais são:

- 2.5 ou 3,2 ou 2.345,00 (Não é possível usar ponto decimal ou vírgula)
- - 345 (Não é possível colocar um espaço entre o sinal e o valor numérico.)
- 23 (Não é possível usar notação de expoentes.)
- 033 (se o número tem mais de um dígito e o primeiro dígito é 0, ele não é uma constante decimal, mas sim uma constante octal)
- 212-22-33 (use de um caracter ilegal: -)

Uma constante inteira *hexadecimal* é formada por uma seqüência de dígitos decimais: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F (ou a, b, c, d, e, pois os dígitos hexadecimais podem ser maiúsculos ou minúsculos). As letras de A a F representam, respectivamente, os valores 10, 11, 12, 13, 14 e 15. Uma constante hexadecimal deve começar por **0x** e tal qual as constantes inteiras decimais, não podem

conter pontos ou vírgulas. Alguns exemplos válidos são: 0x0, 0x3, 0x4f5a, 0x2FFE e 0xAaFf. Alguns exemplos inválidos de constantes hexadecimais são:

- 0x3. e 0x1,e (não se deve usar ponto ou vírgula)
- 0x ff (não se deve usar espaço em branco no meio do número)
- FFEE (toda constante hexadecimal deve começar com 0x)
- 0Xfg34 (uso de um caracter ilegal: g, que não existe entre as constantes hexadecimais)

Uma constante inteira *octal* é formada por uma seqüência de dígitos octais: 0, 1, 2, 3, 4, 5, 6, 7. A constante octal deve ter o primeiro dígito igual a 0 (zero) para que o compilador a identifique como tal. Aqui, também não podem ser usados pontos, vírgulas ou espaços em branco. Alguns exemplos válidos são: 00, -04, 033, 02633 e -0101. Alguns exemplos inválidos de constantes octais são:

- 010. e 02,3 (não se deve usar ponto ou vírgula)
- 0 6 (não se deve usar espaço em branco no meio do número)
- 45 (toda constante octal deve começar com o dígito zero)
- 01784 (uso de um caracter ilegal: 8, uma vez que as constantes octais só vão até o número 7)

O uso de bases diferentes da decimal não é muito freqüente, mas pode haver casos que esse uso seja necessário. Por isso, o C oferece essa opção. Agora, cuidado com a escrita das constantes inteiras: nunca escreva 015 achando que o C vai compilar isto como se fosse 15. Na linguagem C, 015 é diferente de 15! Visto que a primeira é uma constante octal e a segunda uma constante decimal!

Constantes de ponto flutuante

As constantes de ponto flutuante são números reais (não inteiros), representados na base 10, que possuem um ponto decimal e, opcionalmente, um expoente. Quando a constante é seguida por um expoente, a notação é conhecida como “notação científica”². Neste caso, o x10 é substituído por e ou E. Por exemplo, o número 1.23e4 representa 1.23 x 10⁴ ou 12300.

O ponto decimal da constante de ponto flutuante não pode ser substituído por uma vírgula e essa constante pode ser precedida ou

Atenção

O uso de bases diferentes da decimal não é muito freqüente, mas pode haver casos que esse uso seja necessário. Por isso, o C oferece essa opção. Agora, cuidado com a escrita das constantes inteiras: nunca escreva 015 achando que o C vai compilar isto como se fosse 15. Na linguagem C, 015 é diferente de 15! Visto que a primeira é uma constante octal e a segunda uma constante decimal!

Atenção

² Detalhes sobre notação científica em:

http://pt.wikipedia.org/wiki/Notação_científica

não por um sinal. Cada constante de ponto flutuante é considerada ser do tipo *double*. São exemplos válidos: .95, 0.28, +23.45e-10, 123.45, 2.5, -3.8, 123.45E+10.

A forma de representação de um número real em C é bastante flexível. Por exemplo, o número 314 pode ser representado por qualquer uma das seguintes formas: 314.0, 3.14e2 +3.14e+2, 31.4e1, .314E+3 ou 314e0

Constantes de caracteres

Uma constante caracter é uma letra, número ou símbolo colocado entre aspas simples. Embora sejam visualizados como letras e símbolos as constantes caracteres são armazenadas internamente pelo computador como um número inteiro entre 0 e 255. Por exemplo, o caracter 'A' tem valor 65. Esses valores numéricos dos caracteres estão padronizados na chamada Tabela ASCII (*American Standard Code for Information Interchange Table*). A tabela ASCII pode ser encontrada no site: (<http://pt.wikipedia.org/wiki/ASCII>) e também estará disponível no ambiente Moodle da disciplina. Alguns exemplos de constantes caracteres são: 'a', '9' (aqui é o algarismo nove e não o número nove), 'Z', '\$', '+', ' ' (espaço em branco) e '{' .

Constantes de caracteres especiais (ou Constantes de barra invertida)

Certos códigos de controle da tabela ASCII (como o nova linha, representado pelo '\n') ou caracteres especiais (como a aspa simples ') possuem representações diferenciadas na linguagem C. Esta representação chama-se seqüência de escape representada por uma barra invertida (\) e um caracter. Estas seqüências são mostradas na Tabela 2 e elas são utilizadas como qualquer outro caractere, apenas tendo um efeito diferenciado. Entretanto, elas não podem ser inseridas diretamente através do teclado, pelo usuário.

Tabela 2 – Constantes de Barra Invertida

Código	Significado	Código	Significado
\b	Retrocesso (backspace)	\f	Alimentação de formulário
\n	Nova linha	\r	Retorno de carro
\t	Tabulação horizontal	\"	Aspas duplas
'	Apóstrofo	\0	Caractere Nulo
\\"	Barra invertida	\v	Tabulação vertical
\a	Sinal sonoro	\N	Constante octal
\xN	Constante hexadecimal		

Um exemplo de uso seria o \n que usamos no nosso primeiro programa em C: printf("Meu primeiro programa em C \n");

Esse fragmento de código imprime na tela a frase “Meu primeiro programa em C“ e depois pula para uma nova linha.

Constantes strings (cadeias de caracteres)

Uma constante string é um conjunto de qualquer número de caracteres colocados entre aspas duplas, como no exemplo impresso no nosso primeiro programa “Meu primeiro programa em C \n”. Cuidado para não confundir string com caracter. Uma constante caracter simples fica entre dois apóstrofos ou aspas simples, como, por exemplo, ‘B’. Já uma string de somente uma letra ficaria entre aspas duplas, como em “B”. Alguns exemplos de strings são: “Sandra Siebra”, “O valor da média é: “, “ ” (um espaço em branco), “10” (seria string e não valor numérico) e “Valor em reais = R\$ ”.

Embora as instruções do ANSI C usem apenas os caracteres do conjunto padrão da Tabela ASCII, as constantes caracter e string podem conter caracteres do conjunto estendido ASCII, que englobam, entre outras coisas, os caracteres acentuados: é, ã, ç, ü, í, ô, etc.



A linguagem C insere, automaticamente, ao final de cada string um caracter nulo ('\0').

Existem alguns caracteres que têm um significado especial (os que fazem parte da tabela 2). Entre esses, o apóstrofo, as aspas duplas e a barra invertida quando precisam ser impressos, devem ser usados antecedidos pelo caracter escape (a barra invertida). Portanto, para poder imprimir uma barra invertida deve-se usar \\, para imprimir

aspas duplas, deve-se usar \" e para imprimir apóstofro, deve-se usar \'. Por exemplo, se desejássemos imprimir a frase “O símbolo usado é “ (aspas) e não outro”. Usaríamos a string: “O símbolo usado é \" (aspas) e não outro”

A aspa dupla no meio da cadeia de caracter não indica o fim da cadeia e, sim, que deve ser impresso o símbolo “. Isso é identificado justamente porque a aspa foi precedida pelo caractere de escape.

Constantes Simbólicas

Constantes, em linguagens de programação, são identificadores aos quais se atribui um certo valor constante. Em outras palavras, constantes são declarações usadas para fixar um valor a um determinado identificador que ficará constante durante a execução de um programa. Ao contrário das variáveis, não é possível fazer qualquer tipo de atribuição para constantes.

Quando o identificador da constante é referenciado dentro do código do programa, é utilizado nas operações o valor atribuído a este identificador. Ou seja, se for definida a constante simbólica PI com o valor 3.1415926536, quando for encontrado no código do programa o identificador PI, será utilizado em seu lugar o valor 3.1415926536. Por exemplo, se no código do programa existisse o seguinte comando: **area = PI * raio * raio;** o compilador iria interpretar a instrução como se fosse escrita assim: **área = 3.1415926536 * raio * raio;** Entendeu?

Você pode definir constantes simbólicas em qualquer programa na linguagem C. A definição será feita na área de diretivas de pré-processamento (logo no início do programa) e terá a seguinte sintaxe:

```
#define NOME_CONSTANTE valor_associado_a_constante
```

Onde: **#define** é uma diretiva de compilação que diz ao compilador para trocar as ocorrências de **NOME_CONSTANTE** pelo **valor_associado_a_constante**. Geralmente, para diferenciar das variáveis convencionais, os nomes de constantes são escritos todos em maiúscula e, se necessário, separados por um unserscore (_). Por exemplo:

```
#define PI 3.1415926536
#define SINAL "aberto"
```

```
#define ON 1
#define OFF 0
```

Observe que, na definição de uma constante, não há o “;” (ponto e vírgula) no final. Se for colocado, este fará parte do valor associado à constante. O uso da diretiva **#define** não se restringe apenas a definição de constantes simbólicas. Ela também pode ser usada para definir macro instruções. Porém, isto está fora do escopo da nossa disciplina e, por isso, não entraremos nesse mérito.



Questão

Será que no lugar de uma constante simbólica, eu não poderia declarar uma variável convencional, que recebesse um valor inicial? Por exemplo, no lugar da constante PI, que apresentamos anteriormente, será que eu não poderia declarar uma variável de um tipo que recebesse um valor de ponto flutuante, tal como: **float pi = 3.14159;** e, depois, na execução do programa fazer uso dessa variável convencional: **area = pi * area * area;** ????

Bem, poder fazer isso, você poderia. Porém, este tipo de instrução teria duas desvantagens: Primeiro, uma variável do tipo **float** reserva 4 bytes de memória desnecessariamente (porque o valor associado, não seria modificado). Segundo, esta instrução é executada mais lentamente, pois o processador precisa acessar a memória para verificar qual é o valor de pi. Já uma constante, antes da execução, já é substituída no código por seu valor associado.

4.1.11 Variáveis

Variáveis são locais da memória, identificados, onde dados são armazenados e recuperados, durante a execução de um programa (tal qual na linguagem algorítmica). Elas devem ser declaradas antes de serem usadas, com a especificação do tipo de conteúdo válido. O nome das variáveis deve obedecer à regra de nome de identificadores.

Dados são atribuídos às variáveis através de funções de leitura de dados (onde os dados são capturados de algum dispositivo de

entrada de dados) ou através do operador de atribuição (=), que será apresentando mais a frente nesta unidade.

Você deve escolher nomes para as variáveis que indiquem o valor que ela vai armazenar ou a função que ela vai ter dentro do programa. Por exemplo: **nota1**, **media**, **qntde_valores**, **total**, **taxa_imposto**. Normalmente, por convenção, os nomes de variáveis são escritos em letra minúscula. Em alguns nomes de variáveis, podemos combinar duas palavras, para indicar melhor o dado que vão ser armazenados nas mesmas. Essas duas palavras ou serão unidas por um subscrito (_), por exemplo, **taxa_imposto**. Ou podem ser escritas juntas, mas com a primeira letra da segunda palavra em maiúscula, por exemplo: **taxalImposto**. Lembre que o caracter espaço em branco não pode ser usado no nome de variáveis.

4.1.12 Tipos de Dados Básicos (ou Tipos Primitivos)

Toda variável em C deve ter um tipo, que vai definir os valores que podem ser armazenados na mesma. C possui cinco tipos de dados chamados de básicos ou primitivos que estão especificados na Tabela 3. No ANSI C, cada um desses tipos de dados possui um intervalo de valores permitidos, que também pode ser visualizado na mesma tabela.

Tabela 3 - Tipos de Dados Básicos do C

Tipo	Tamanho (em bits)	Intervalo de valores	Descrição
char	8	-128 a 127	O tipo caracter representa letras, dígitos e símbolos e, geralmente, é armazenado como um código (código numérico da tabela ASCII).
int	16	-32768 a 32767	Representa um número inteiro (sem ponto flutuante). O intervalo do conjunto que pode ser representado, normalmente, depende da máquina em que o programa está rodando.
float	32	3,4E-38 a 3,4E+38	Tipo que representa um número de ponto flutuante de precisão simples (7 dígitos). São conhecidos como números reais.
double	64	1,7E-308 a 1,7E+308	Tipo que representa um número em ponto flutuante de precisão dupla (15 dígitos).
void	0	sem valor	Este tipo serve para indicar que um resultado não tem um tipo definido. A aplicação deste tipo será vista posteriormente.

Qualquer outro tipo de dados é baseado nestes tipos primitivos. A divisão nestes tipos de dados se deve, basicamente, ao número de bytes reservados para cada dado. Porém, a quantidade de bytes

e a faixa de valores desses tipos de dados variam de acordo com o processador e com a implementação do compilador C.

4.1.13 O Tipo String

Ao contrário de outras linguagens, string não é um tipo de dado básico ou primitivo da linguagem C. Strings são seqüências de caracteres utilizadas para o armazenamento de texto. Em outras palavras, strings são simplesmente vetores de caracteres (char). Para definir a string usamos:

```
char nomeString[tamanhoString];
```

onde nomeString é o nome da variável do conjunto de caracteres e tamanhoString é o tamanho do conjunto. Uma particularidade das strings, em C, é que elas possuem um caracter terminador, o qual delimita o final do seu conteúdo, que é o caracter '\0' (lê-se "contrabarra zero"). Desta forma, o tamanho da string deve ser definido com um caracter a mais do que será efetivamente necessário para o programa.

O conteúdo de uma string pode ser inicializado por uma seqüência de caracteres entre aspas duplas ou o valor da string pode ser lido usando a função scanf (que veremos mais na frente nesta unidade). Em ambos os casos, o compilador C coloca, automaticamente, o '\0' no final. Por exemplo:

```
char estado[3] = "RS"; /* declara uma conjunto de 3 caracteres chamado estado, no qual cada posição do conjunto terá um dos seguintes valores: 'R', 'S' e '\0' */
```

```
char cidade[100] = "Passo Fundo"; /* declara um conjunto de 100 caracteres chamado cidade */
```

```
char nome[] = "Fulano de Tal"; /* quando o tamanho do conjunto de caracteres não é definido e um valor é atribuído ao mesmo, o conjunto assume o tamanho a string atribuída, por exemplo, neste caso aqui, o conjunto de caracteres teria tamanho 14 – letras e espaços da string atribuída + o caracter '\0' */
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13
F	u	I	a	n	o		d	e		t	a	I	\0

Portanto, é bom observar duas coisas no exemplo acima: para armazenar um nome de 13 letras precisa-se de um vetor de 14 posições (porque temos de deixar o espaço do '\0'. Todo vetor

na linguagem C começa da posição zero. Logo, se o vetor tem 14 posições, a numeração das posições vai de 0 a 13.

Se o string for montado caracter por caracter é necessário acrescentar “manualmente” o caracter ‘\0’ no final. Se isto não for feito, o conjunto de caracteres não será visto como um string pelas funções de manipulação do mesmo.

4.1.14 Modificadores de Tipos de Dados

Modificadores podem ser aplicados aos tipos primitivos da linguagem C. Estes modificadores são palavras que alteram o tamanho ou o significado do conjunto de valores que o tipo pode representar. Isso pode permitir um melhor aproveitamento de espaço na memória. Por exemplo, um modificador permite que possam ser armazenados números inteiros maiores. Um outro modificador obriga que só números sem sinal possam ser armazenados pela variável, tornando desnecessário guardar o bit de sinal do número, fazendo com que somente números positivos sejam armazenados no tipo modificado. O resultado prático é que o conjunto de tipos da linguagem C, praticamente, dobra de tamanho com o uso desses modificadores.

Os modificadores de tipo do C são quatro: **signed**, **unsigned**, **long** e **short**. Com exceção do tipo void e do tipo float, os demais tipos de dados primitivos podem ter modificadores precedendo-os. Porém, atenção:

- Ao float e ao void não se pode aplicar nenhum modificador
- Ao double pode-se aplicar apenas o long (que aumenta o número de bytes para armazenamento de dados).
- Os quatro modificadores podem ser aplicados aos números inteiros. A intenção é que short e long devam prover tamanhos diferentes de inteiros.
- O modificador unsigned serve para especificar variáveis sem sinal. Este modificador pode ser aplicado aos tipos int e char e faz com o bit de sinal não seja usado, ou seja, o tipo passa a ter um bit a mais, porém só vai poder armazenar números positivos.
- O modificador signed pode ser aplicado aos tipos int e char. Porém, o uso de signed com int é redundante, porque o inteiro,

já é um valor que leva em conta o sinal.

É possível combinar estes modificadores de diversas maneiras. A Tabela 4 lista todos os tipos de dados definidos no padrão ANSI C, com seus respectivos tamanhos em bits e sua faixa de valores possíveis.

Uma variável char pode ser usada, além de para armazenar codificações da tabela ASCII, como um “pequeno” inteiro com intervalo de –128 a 127, e no lugar de um inteiro quando a situação não requer números grandes.

4.1.15 Declaração de Variáveis

Os dados que são processados pelos programas são armazenados na memória em variáveis. Em C, é obrigatória a declaração das variáveis antes do seu uso no programa. Na declaração, é definido o nome da variável (seu identificador), o seu tipo e, opcionalmente, seu valor inicial. O nome da variável vai seguir as regras de nomes de identificadores, anteriormente explicada (seção 4.1.9). O tipo da variável vai definir o tamanho em bytes que ela ocupará na memória do computador e o intervalo de valores que ela poderá armazenar. E o valor inicial indicará o valor que essa variável vai ter gravado na memória, no início da execução do programa, podendo esse valor variar no decorrer da execução.

Tabela 4 - Tipos de dados definidos pelo Padrão ANSI C

Tipo	Tamanho (em bits)	Intervalo
char	8	-128 a 127
unsigned char	8	0 a 255
signed char	8	-128 a 127
int	16	-32768 a 32767
unsigned int	16	0 a 65535
signed int	16	-32768 a 32767
short int	16	-32768 a 32767
unsigned short int	16	0 a 65535
signed short int	16	-32768 a 32767
long int	32	-2147483648 a 2147483647
signed long int	32	-2147483648 a 2147483647
unsigned long int	32	0 a 4294967295
float	32	3,4E-38 a 3,4E+38
double	64	1,7E-308 a 1,7E+308
long double	80	3,4E-4932 a 1,1E+4932

A declaração das variáveis deverá ser feita antes de qualquer

comando executável. A forma geral da declaração de variáveis é:

Tipo lista_de_variáveis;

onde, uma lista de variáveis é uma seqüência de nomes de variáveis separadas por vírgulas e tipo é um dos tipos primitivos ou modificados disponíveis na linguagem C. Vamos olhar alguns exemplos de declaração:

- int contador;
- char letra;
- unsigned int num1, num2, num3;
- float raio, diametro;
- double salario;
- char nome[45];

No exemplo acima, **contador** foi declarado como uma variável inteira. Assim, ele pode armazenar valores inteiros de -32768 até 32767. Do mesmo modo, **letra** foi declarada como variável caracter podendo receber valores de -128 até 127 ou qualquer caracter do conjunto padrão ASCII. As variáveis **num1**, **num2** e **num3** foram declaradas como unsigned int, logo só poderão armazenar números inteiros positivos de 0 a 65535. As variáveis **raio** e **diametro** foram declaradas como ponto flutuante do tipo float (precisão de 7 dígitos) e **salario** como ponto flutuante do tipo double (precisão de 15 dígitos). Por fim, a variável **nome** foi declarada como um conjunto de 45 caracteres (ou seja, como uma string).

A declaração de variáveis é feita, em geral, dentro de uma rotina. Por exemplo, a rotina principal main() ou no início de um bloco de código. Deste modo, se diz que está se fazendo uma declaração de *variáveis locais*. Estas variáveis só têm validade dentro do bloco de código ou rotina no qual são declaradas, isto é, só a função à qual elas pertencem sabe da existência destas variáveis, dentro do bloco no qual foram declaradas. Funções distintas podem ter variáveis locais com o mesmo nome. Vamos dar um exemplo de variáveis locais.

1. int main(){
2. float raio, area; // declaração de variáveis da função main()
3. raio = 2.5;
4. area = 3.14 * raio * raio;

```
5. }
```

Neste exemplo, as variáveis area e raio foram declaradas como variáveis locais a função main() – porque estão sendo declaradas dentro do main() - do tipo float (linha 2). Assim, o processador faz a alocação de dois espaços (endereços) de 4 bytes cada para armazenar as informações, um para cada variável. Na linha 3, o processador coloca no endereço alocado para raio o valor 2.5. Depois, na linha 4, o processador coloca o resultado do cálculo da expressão (19.625) na variável chamada area. Todas as variáveis são visíveis dentro de qualquer ponto do método main().

Também são consideradas variáveis locais as variáveis declaradas na lista de parâmetros de uma função. Pois, apesar destas variáveis receberem valores externos, elas são conhecidas apenas pela função onde estão declaradas.

É possível fazer a declaração de variáveis fora de uma rotina ou bloco de código. Neste caso, diz-se que se está fazendo uma declaração de variáveis globais.

Mas o que são variáveis globais? Onde elas são declaradas?

As variáveis globais são definidas fora de qualquer função (inclusive da função main()) e estarão disponíveis para serem utilizadas em todo o programa. Pode-se dizer que, como elas estão fora de todas as funções, todas as funções as vêem. Vamos dar um exemplo para ilustrar os tipos de variáveis.

```
1. #include <stdio.h>
2. int contador; // variável global. Está fora de todas as funções e blocos
3. int funcao1(int j) {
4.     /* aqui viria o código da função
5.     ...
6. }
7. }
8. int main()
9. {
10.    char teste;
11.    int i;
12.    for (i=0; i<100; i=i+1) // um laço estilo para...faça
13.    { /* Bloco do for */
```

```

14.         float valor;
15.         funcao1(i);
16.     }
17.     /* ... */
18.     return(0);
19. }
```

Nesse exemplo, a variável **contador** é uma variável global e é acessível de qualquer parte do programa (seja no main() ou dentro da função funcao1). A variável inteira **j** é um exemplo de declaração de variável na lista de parâmetros de uma função (a função funcao1). Ela é uma variável local e visível apenas dentro da funcao1. As variáveis **teste** e **i**, só existem dentro de main(), isto é são variáveis locais a função main. A variável float valor é um exemplo de uma variável de bloco de código, isto é, ela somente é conhecida dentro do bloco do for, porque foi declarada dentro dele, apesar de pertencer à função main.

As regras que regem onde uma variável é válida chamam-se *regras de escopo da variável*. Há mais alguns pontos que devem ser ressaltados:

- Duas variáveis globais não podem ter o mesmo nome.
- O mesmo vale para duas variáveis locais de uma mesma função. Já duas variáveis locais, de funções diferentes, podem ter o mesmo nome sem perigo algum de conflito. Pois sua existência será em blocos diferentes.
- Se uma função declarar uma variável local com o mesmo nome de uma variável global, esta variável local ocultará/substituirá aquela global, que não poderá mais ser acessada. Veja um exemplo no código a seguir.

```

1. int num1, num2; /* 2 variáveis globais do tipo int */
2. int main(void)
3. {
4.     int valor, num1; /* 2 variáveis locais do tipo int */
5.     double total; /* 1 variável local do tipo double */
6.     num1 = 5;
7.     /* A atribuição acima será feita à variável local num1 e não à variável global de mesmo nome */
8.     return(0);
```

```
9.      }
```

4.1.16 Inicialização de variáveis

Quando se faz a declaração de uma variável está se determinando que tipo de dado ela vai receber. É possível, em C, declarar uma variável e já armazenar nela um valor inicial. Chamamos este procedimento de inicialização de variável. Isto é importante, pois quando o C cria uma variável ele não a inicializa. E o que isso significa na prática? Bem, isto significa que até que um primeiro valor seja atribuído à nova variável, ela tem um valor indefinido (lixo!) e, por isso, não pode ser utilizada para nada. Sendo assim, nunca presuma que uma variável declarada vale zero ou qualquer outro valor! Antes de utilizar uma variável em uma operação, tenha certeza que algum valor foi atribuído a ela, sob pena de que as operações possam resultar em valores incorretos.

A inicialização de uma variável pode ser feita através de uma atribuição à mesma de uma constante, de uma expressão ou função. A sintaxe para a inicialização de variáveis é:

```
tipo var1 = valor1 [, var2 = valor2, ...];
```

onde tipo é o tipo de dado da variável, var1 é o nome da variável a ser inicializada e valor1 é o valor inicial da variável. Você pode, opcionalmente, inicializar mais de uma variável na mesma linha. Vamos apresentar alguns exemplos:

```
int a=10, b=a+20, c=abs(b);
char letra='Z';
float porcentagem = 13.5;
char titulo[30] = " Essa é uma string";
```

Nos exemplos acima, a, b e c foram declaradas variáveis do tipo int. O valor inicial de **a** é 10 (constante numérica inteira), o de **b** é 30 (resultado da expressão a+20) e o **c** também é 30 (resultado da função abs, que retorna o valor absoluto, sem ponto decimal, da variável). A variável **letra** foi declarada como char e inicializada com 'Z'. Do mesmo modo **porcentagem** foi declarada como variável do tipo float com valor inicial 13.5. E, finalmente, a variável **titulo** foi declarada

como um conjunto de 30 caracteres e recebeu, como conteúdo inicial, “Essa é uma string”.

4.1.17 Operadores e Expressões

Um programa tem como característica fundamental a capacidade de processar dados. Processar dados significa realizar operações com estes dados. As operações a serem realizadas com os dados podem ser determinadas por operadores ou funções. Os operadores podem ser de atribuição, aritméticos, de atribuição aritmética, incrementais, relacionais, lógicos e condicionais. Já uma expressão é um arranjo de operadores e operandos. A cada expressão válida atribui-se um valor numérico (como, por exemplo, $4 + 6$ é uma expressão cujo valor é 10) ou lógico (como, por exemplo, $5 > 7$ cujo valor é falso). Nas subseções a seguir discutiremos cada um dos operadores e expressões.

Operador de Atribuição

A operação de atribuição é a operação mais simples do C. Ela é o ato de armazenar um valor em uma variável. Uma atribuição possui a forma: **identificador = expressão**, onde identificador é o nome da variável que recebe o valor da expressão. E a expressão é uma expressão válida ou outro identificador. Atribuir um valor de uma expressão a uma variável significa calcular o valor daquela expressão e copiar o valor calculado para uma determinada variável.

Observe-se que o operando esquerdo deve ser um identificador de variável, isto é, não pode ser uma constante ou expressão. Observe, também, que o símbolo de atribuição em C é o símbolo de igualdade da matemática ($=$). Na linguagem algorítmica que usamos, anteriormente, o operador de atribuição era representado pela seta \leftarrow .

Sempre antes de usar uma variável esta deve ter sido declarada. Você não pode atribuir algo a uma variável se ela não tiver sido declarada. Bom, vamos dar agora alguns exemplos:

```
int a,b;
float f,g;
char h;
a = 10;
b = -15;
f = 10.0;
```

```
h = 'A';
a = b;
g = (b+f+a);
```

Note que é possível usar constantes em uma expressão, mas não é possível atribuir valores a elas. Por exemplo, o comando de atribuição **1 = a;** seria inválido. A regra é: uma constante nunca fica do lado esquerdo de uma atribuição.



1. Cuidado para não confundir o operador de atribuição (=) com o operador relacional de igualdade (==), que será visto posteriormente.
2. Sempre atribua a uma variável algo que seja do tipo da mesma. Por exemplo, se a variável é inteira, atribuía uma constante inteira ou uma expressão cujo resultado seja um número inteiro. Pois, se os dois operandos de uma atribuição não são do mesmo tipo, o valor da expressão ou operador da direita será convertido para o tipo do identificador da esquerda. Por exemplo:

```
int num1;
float num2;
num1 = 5; // valor de num é 5
num2 = num1; // valor de num2 é 5.0
```

No exemplo acima, não houve perda de informação porque o valor inteiro foi atribuído a um valor de ponto flutuante que possui um número de bytes na memória maior que o do valor inteiro. Porém, quando o operando da esquerda é de um tipo que utiliza uma quantidade menor de bytes na memória que o operando da direita, podem ocorrer alterações dos valores convertidos e consequente perda de informação. Por exemplo:

```
int num1;
float num2 = 98.251; // valor de num2 é 98.251, um valor de ponto flutuante
num1 = num2; // como o valor inteiro não tem a parte de ponto flutuante, o valor de num2
é truncado, // para poder ser armazenado em num1, havendo assim, perda de informação.
```

```
Dessa
// forma, o valor de num1 é 98
```

Deste modo, as conversões potencialmente perigosas (onde há possibilidade de perda de informação), da direita para a esquerda são: char <- int <- float <- double

Observe que o compilador C ao encontrar esta operação não gera nenhum erro de compilação. Assim, essa perda de informação pode gerar um erro semântico que pode passar despercebido pelo programador inexperiente.

Atribuição múltipla.

É possível atribuir um valor a muitas variáveis em uma única instrução. A esta operação dá-se o nome de atribuição múltipla. A sintaxe da atribuição múltipla é seguinte:

tipo var1 = [var2 = ...] expressão;

onde var1, var2, ... são os identificadores de variáveis e expressão é uma expressão válida. Observe que na atribuição múltipla, as operações ocorrem da direita para a esquerda, isto é, inicialmente o valor de expressão é atribuído a var2 e, depois, o valor de var2 é atribuído a var1. Vamos exemplificar para deixar mais claro. Suponha a seguinte atribuição:

num1 = num2 = num3 = 5;

Note que as três variáveis receberão o mesmo valor e a atribuição múltipla será feita da direita para a esquerda. Dessa forma, num3 recebe 5. Posteriormente, num2 recebe o valor à sua direita, que é o valor atribuído a num3, ou seja, 5. Finalmente, num1 recebe, também, o resultado à sua direita, que é o valor atribuído a num2, ou seja, 5.

Operadores Aritméticos

Existem nove operadores aritméticos em C (vide Tabela 5). Quatro deles estão relacionados às operações aritméticas elementares: adição, subtração, multiplicação e divisão. Existe ainda um operador (%) chamado operador de módulo cujo significado é o resto da divisão inteira (equivalente ao MOD que foi explicado na linguagem algorítmica, se você não recorda, dê uma olhadinha no assunto “operadores aritméticos” da unidade 2). Existem ainda os operadores

+ e - unários que equivalem ao sinal de positivo e negativo de um número e dois operadores unários (que só envolvem uma variável) novos: ++ e --.

Tabela 5 - Operadores Aritméticos

Operador	Descrição	Prioridade	Exemplo
+	Mais unário (sinal de positivo)	0	+5 (valor positivo)
-	Menos unário (sinal de negativo)	0	- 5 (valor negativo)
++	Incremento Unário	1	int x = 5; X++; //equivale a x = x + 1; // aqui x passaria a ser 6
--	Decremento Unário	1	int x = 5; X--; //equivale a x = x - 1; // aqui x passaria a ser 4
*	Multiplicação	2	4 * 5 -> 20
/	Divisão	2	20/4 ->5
%	Resto da Divisão Inteira	2	20% 2 -> 0 e 5% 2 -> 1
+	Soma	3	4 + 5 -> 9
-	Subtração	3	5 - 2 -> 3

O C possui operadores unários e binários. Os unários agem sobre uma variável apenas, modificando ou não o seu valor e retornam o valor final da variável. Os binários usam duas variáveis e retornam um terceiro valor, sem alterar as variáveis originais.

Os operandos da Tabela 5 podem ser utilizados com qualquer tipo de dados numéricos, exceto o resto da divisão inteira, que deve ser utilizada exclusivamente com o tipo *inteiro (int)*. Uma restrição ao operador de divisão (/) é que o denominador deve ser diferente de zero. Se alguma operação de divisão por zero for realizada ocorrerá um erro de execução do programa (denominado *run-time error*) e o programa será encerrado. Outra observação sobre o operador / (divisão) é que quando ele é aplicado a apenas variáveis inteiros, nos fornece o resultado da divisão inteira (ou seja, a parte decimal é ignorada), porém, quando é aplicado a alguma variável de ponto flutuante, ele nos fornece o resultado da divisão “real”, ou seja, com a parte de ponto flutuante. Por exemplo:

```
int a = 20, b=3;
```

```
float z = 20.0;
```

a / b → 6 (como as duas variáveis são inteiros, o resultado é truncado para ser um número inteiro)

z / b → 6.66 (como a variável z é float, o resultado não é mais truncado)

O operador `++` é unário e equivale ao incremento de uma variável, ou seja, a adicionar o valor 1 à variável. Dessa forma $x++ \rightarrow x = x + 1$. E o operador `--` equivale ao decremento de um número, ou seja, a subtrair o valor 1 de um número. Assim sendo, $x-- \rightarrow x = x - 1$. Esses operadores alteram a variável sobre a qual estão sendo aplicados. Um detalhe é que esses operadores podem ser pré-fixados ou pós-fixados. A diferença é que, quando são pré-fixados, eles incrementam e retornam o valor da variável já incrementada. Por exemplo:

```
x=23;
```

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

Teremos, no final, **y=24** e **x=24**. Como isso ocorreu? `++x` incrementa **x ANTES** de atribuir o valor dele a **y**. Logo, o **x** passou a 24, antes da atribuição e foi o valor 24 que foi atribuído a **y**. É como se o comando fosse dividido em:

```
x = x + 1;
```

```
y = x;
```

Quando esses operadores são pós-fixados eles retornam o valor da variável sem o incremento e depois incrementam a variável. Observe, no exemplo abaixo, a diferença, com relação ao exemplo anterior:

```
x=23;
```

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

Teremos, no final, **y=23** e **x=24**. Isso ocorreu porque `x++` só vai incrementar o valor de **x** DEPOIS de atribuir o valor dele a **y**. Logo, **y** recebe o valor de **x**, que é 23 e depois da atribuição o **x** é incrementado, passando ao valor 24. É como se o comando fosse dividido em:

```
y = x;
```

```
x = x + 1;
```

Percebeu a diferença? É importante que se conheça exatamente o efeito sutil da colocação do operador, pois isto pode enganar o programador inexperiente e, com isso, podem-se obter resultados indesejados. Esses operadores costumam ser usados para o controle de estruturas de repetição.



Atenção

Em C caracteres são armazenados na memória como números inteiros e, por isso, operações aritméticas são permitidas com tipos char. Mas que valores numéricos seriam usados?????

Seriam usados os valores correspondentes ao caracter na tabela ASCII. Vamos dar alguns exemplos. Observe a Tabela 6, a expressão 'A' + 1 seria interpretada internamente como 65 + 1, uma vez que o código da letra A na tabela ASCII é 65. Dessa forma, o resultado seria 66, que equivale à letra B na tabela ASCII.

Tabela 6 – Operações Aritméticas com Caracteres

Expressão	Conversão segundo a tabela ASCII	Caracter resultante
'A' + 1	65 + 1 -> 66	'B'
'A' + 'B'	65 + 66 -> 131	'â'
'A' + 32	65 + 32 -> 97	'a'

Expressões Aritméticas

Uma expressão aritmética é aquela que faz uso de operadores aritméticos binários e valores numéricos. A sintaxe da expressão é: **operando1 operador operando2** onde o operador pode ser a soma, subtração, multiplicação, divisão ou resto da divisão inteira. E os operandos devem ser constantes ou variáveis numéricas. Por exemplo: 1 + 2, a - 4.0, b *c, valor/taxa, num%2.

Quando mais de um operador se encontram em uma expressão aritmética, as operações são efetuadas, uma de cada vez, respeitando as regras de precedência ou prioridade dos operadores (especificadas na Tabela 5). Entre operadores de mesma precedência as operações são efetuadas da esquerda para a direita. Observe a ordem de prioridade das expressões a seguir, a partir dos valores obtidos:

$1 + 2 - 3 \rightarrow 0$ (primeiro, realizou-se a soma e, depois, a subtração)

$24 - 3 * 5 \rightarrow 9$ (primeiro, realizou-se a multiplicação e, depois, a subtração)

$4 - 2 * 6 / 4 + 1 \rightarrow 2$ (primeiro, realizou-se a multiplicação, depois, divisão, seguida da subtração e, por fim, a soma.).

$6 / 2 + 11 \% 3 * 4 \rightarrow 11$ (primeiro, realizou-se a divisão, depois a operação de resto da divisão inteira, seguida da multiplicação e, por último, a soma)

A ordem de precedência dos operadores pode ser quebrada usando-se parênteses: (). Os parênteses são, na verdade, operadores de mais alta precedência e são executados primeiro. Parênteses internos são executados primeiro que parênteses externos. Por exemplo, observe algumas das expressões anteriores, só que, agora, fazendo uso de parênteses para alterar a prioridade dos operadores. Veja que o resultado obtido com o uso dos parênteses pode ser diferente, uma vez que a ordem de execução das operações é alterada.

$(24 - 3) * 5 \rightarrow 105$ (primeiro, realizou-se a subtração e, depois, a multiplicação)

$(4 - 2 * 6) / 4 + 1 \rightarrow -1$ (primeiro, realizou-se a multiplicação, depois, a divisão, seguida da subtração e, por último, a soma)

Operadores de Atribuição Aritmética

Muitas vezes queremos alterar o valor de uma variável realizando alguma operação aritmética com ela. Como, por exemplo: $i = i + 1$ ou $val = val * 2$. Embora seja perfeitamente possível escrever estas instruções, foi desenvolvido na linguagem C algumas instruções otimizadas com o uso dos chamados operadores de atribuição aritmética.

Na tabela 7, podem ser vistos os operadores de atribuição aritmética, a expressão a qual equivalem e um exemplo de uso dos mesmos. Nesta tabela, considere que var é o identificador de uma variável e exp é uma expressão válida). Considere que x e y são variáveis inteiros e que $x = 5$ e $y = 7$.

Atenção

O uso desses operadores deixa o código menos legível. Dessa forma, não recomendo que sejam utilizados com freqüência. Você mesmo pode se confundir no uso deles!

Tabela 7 - Operadores de Atribuição Aritmética

Operador	Equivalência	Intervalo
<code>var += exp;</code>	<code>var = var + exp;</code>	$x += 1 \rightarrow 6 (x = x + 1, \text{ assim, } x = 5 + 1)$ $x += y \rightarrow 12 (x = x + y, \text{ assim, } x = 5 + 7)$
<code>var -= exp;</code>	<code>var = var - exp;</code>	$x - y \rightarrow -2 (x = x - y, \text{ assim, } x = 5 - 7)$
<code>var *= exp;</code>	<code>var = var * exp;</code>	$x * y \rightarrow 35 (x = x * y, \text{ assim, } x = 5 * 7)$
<code>var /= exp;</code>	<code>var = var / exp;</code>	$x / y \rightarrow 0 (x = x / y, \text{ assim, } x = 5 / 7,$ veja que é uma divisão inteira, logo a parte de ponto flutuante é truncada, por isso, o resultado é zero)
<code>var %= exp;</code>	<code>var = var % exp;</code>	$x \% y \rightarrow 5 (x = x \% y, \text{ assim, } x = 5 \% 7)$

Fazendo uso dos operadores de atribuição aritmética, a operação aritmética é primeiramente realizada e, depois, o valor resultante é atribuído a variável do lado esquerdo. Os operadores de atribuição aritmética têm precedência menor que os operadores aritméticos.

Operadores Relacionais

Operadores relacionais (vide Tabela 8) verificam a relação de magnitude ou igualdade entre dois valores. Em outras palavras, os operadores relacionais são utilizados em expressões condicionais para a comparação do valor de duas expressões ou variáveis.

Tabela 8 - Operadores Relacionais

Operador	Significado	Prioridade
<code>></code>	Maior que	0
<code>>=</code>	Maior ou igual a	0
<code><</code>	Menor que	0
<code><=</code>	Menor ou igual a	0
<code>==</code>	Igual a	1
<code>!=</code>	Não igual a (diferente de)	1

Os operadores `>`, `>=`, `<` e `<=` têm a mesma prioridade entre si e são de maior prioridade com relação ao `==` e `!=`. Os operadores aritméticos têm maior prioridade que os operadores relacionais. Operadores relacionais de mesma prioridade são avaliados da esquerda para a direita.

Os operadores relacionais são binários, ou seja, relacionam duas expressões ou variáveis e o seu resultado sempre é verdadeiro (valor 1) ou falso (valor 0).

Atenção

Nas linguagens de programação, os valores verdadeiro e falso podem ser representados de diversas maneiras. Uma das maneiras mais comum é representar verdadeiro por true e falso por false e ter uma variável do tipo lógico (booleana) para manipular esses valores. Porém, em C o valor falso é representado por 0 (zero) e o valor verdadeiro por qualquer valor diferente de 0. Não esqueça disso!

Operadores Lógicos

Os operadores lógicos definem as maneiras como as expressões condicionais podem ser conectadas umas às outras. Por exemplo, podemos querer testar, ao mesmo tempo, se um aluno tem média

maior ou igual a 7.0 e se ele tem taxa de presença em sala de aula maior que 75%. Nesse caso, precisaríamos fazer uso de operadores lógicos. Os operadores lógicos em C, apesar de mudarem a simbologia (vide Tabela 9), têm o mesmo significado dos operadores lógicos da linguagem algorítmica e, consequentemente, a mesma tabela verdade (vide Tabela 10). Ou seja, o resultado da operação lógica `&&` será 1 (verdadeiro) se somente se os dois operandos forem 1 (verdadeiro), caso contrário o resultado é 0 (falso). O resultado da operação lógica `||` será 0 (falso) se somente se os dois operandos forem 0 (falso), caso contrário o resultado é 1 (verdadeiro). E o resultado da operação lógica `!` será 0 se o operando for 1, e 1 se operando for 0.

Tabela 9 - Operadores Lógicos

Operador em C	Operador na Linguagem Algorítmica	Significado
<code>&&</code>	E	Conjunção
<code> </code>	OU	Disjunção
<code>!</code>	NÃO	Negação

A sintaxe de uso dos operadores lógicos:

`expr1 && expr2`

`expr1 || expr2`

`!expr1`

onde `expr1`, `expr2` são expressões quaisquer. Observe que o operador de negação é unário e os outros dois são binários. Observe, também, que, em C, os operadores lógicos atuam sobre expressões de quaisquer valores. Para estes operadores todo valor numérico diferente de 0 (falso) é considerado 1 (verdadeiro).

O Operador `&&` tem prioridade sobre o operador `||`. E, estes dois, têm prioridade menor que os operadores relacionais. O operador `!` tem a mesma prioridade que os operadores incrementais (`++` e `--`).



Atenção

Se não estiver lembrando deste assunto, por favor, revise “Operadores Lógicos” da Unidade 2.

Tabela 10 – Tabela Verdade das Operações Lógicas

p	q	<code>!p</code>	<code>p & q</code>	<code>p q</code>
falso	falso	verdadeiro	falso	falso
falso	verdadeiro	verdadeiro	falso	verdadeiro
verdadeiro	falso	falso	falso	verdadeiro
verdadeiro	verdadeiro	falso	verdadeiro	verdadeiro

Operador Condisional

O operador condicional (`? :`), também chamado de operador ternário (por fazer uso de três operandos), é usado em expressões condicionais. Uma expressão condicional pode ter dois valores diferentes dependendo de uma condição de controle.

Sintaxe: condição ? expressao1 : expressao2

onde expressao1 e expressao2 são duas expressões quaisquer ou valores válidos e condição é uma expressão lógica que será avaliada primeiro. Se o valor de condição for 1, isto é, verdadeiro, então a expressão condicional assumirá o valor de expressao1. Caso contrário, assumirá o valor de expressao2. Essa expressão condicional é equivalente a uma estrutura de decisão simples:

se condição **então**

 expressao1

senão

 expressao2

fimse

Vamos ver alguns exemplos a seguir. Suponha que i, j e k são variáveis do tipo int com valores i = 1, j = 2 e k = 3. Qual seria o resultado das expressões condicionais a seguir?

Vamos ver alguns exemplos a seguir. Suponha que i, j e k são variáveis do tipo int com valores i = 1, j = 2 e k = 3. Qual seria o resultado das expressões condicionais a seguir?

i ? j : k (como i é igual a 1, seu valor é interpretado como verdadeiro, por isso, essa expressão condicional retorna o valor de j que é 2)

(j > i) ? ++k : --k (como a condição é verdadeira, a expressão a ser executada é `++k`. Dessa forma, o k é incrementado antes de ser retornado, sendo o valor resultado 4)

(k == i) && (k != j) ? i + j : i - j -1 (como a condição é falsa, uma vez que falso `&&` verdadeiro. F, a expressão a ser executada é `i-j`, dessa forma, o valor de retorno é -1)

O operador condicional tem baixa prioridade, precedendo, apenas, os operadores de atribuição aritmética.

Atenção

O uso desse operador deixa o código menos legível. Dessa forma, não recomendo que seja utilizado. Uma vez que você mesmo pode se confundir com relação ao valor resultante.

Expressões Lógicas

A chave para a flexibilidade de um algoritmo é a tomada de decisões através da avaliação de condições de controle. Uma condição de controle é uma expressão lógica que é avaliadas como verdadeira ou falsa. Uma expressão lógica é construída com operadores relacionais e lógicos. A sintaxe das expressões lógicas é:

expr1 operador expr2

onde, expr1 e expr2 são duas expressões numéricas quaisquer e operador é um dos operadores relacionais. As expressões lógicas podem ser conectadas umas às outras através de um operador lógico.

(expr1 opRelacional expr2) opLogico (expr3 opRelacional expr4)

Ao contrário de outras linguagens, como mencionado anteriormente, em C não existem tipos lógicos, portanto, o resultado de uma expressão lógica é um valor numérico: uma expressão avaliada verdadeira recebe o valor 1 e uma expressão lógica avaliada falsa recebe o valor 0. Vamos dar alguns exemplos.

Suponha que i e j são variáveis do tipo inteiro (int), sendo i = 5 e j = -3. As variáveis x e y são do tipo float, sendo x=7.3 e y=1.7. Que valor resultaria de cada uma das expressões lógicas abaixo?

i == 7 → 0

x != y → 1

i > x → 0

6 >= i → 1

i < j → 0

y <= 5.9 → 1

Lembrando que os operadores aritméticos têm prioridade sobre os operadores relacionais, vamos avaliar as expressões lógicas a seguir, supondo que m e n são variáveis do tipo inteiro sendo m=4 e n=1.

m + n == 5 → 1

m != 2 * n → 1

6 >= n → 0

$m + n \leq n * n \rightarrow 0$

Agora, conectando as expressões lógicas com operadores lógicos e supondo $a=0$, $b=1$ e $c=2$, todos números inteiros, vamos avaliar as expressões abaixo (lembre que os operadores relacionais têm prioridade sobre os operadores lógicos).

$a \&& b \rightarrow 0$

$c > b \mid\mid a < c \rightarrow 1$

$a + b \&& !(c - b) \rightarrow 0$

$!b \&& c \mid\mid a \rightarrow 0$

Cuidado com as prioridades dos operadores das expressões acima!

Prioridade entre os operadores do C

A Tabela 11 mostra a ordem de prioridade de todos os operadores estudados nesta unidade. Os de maior prioridade estão no topo da tabela e os de menor prioridade estão na base da tabela.



Uma dica

Você não precisa saber toda essa tabela de prioridades decorada. Porém, se ficar na dúvida, é aconselhável que ao escrever o seu código, você tente isolar as expressões com parênteses, para tornar o seu programa mais legível e deixar clara a prioridade dos operadores.

Tabela 11 - Prioridade dos Operadores em C

Operadores	Categoria
()	Parênteses
++ -- ! * / %	Aritmético, lógico
+ - (binários)	Aritmético
< > <= > =	Relacional
= = !=	Relacional
& &	Lógico
	Lógico
? :	Expressão Condicional
= += -= *= /=	Atribuição e Atribuição Aritmética

4.1.18 Entrada e Saída de Dados

Para atender às três fases básicas de qualquer algoritmo ou programa (entrada, processamento e saída), toda linguagem de programação deve oferecer meios para leitura e escrita de dados. Ou seja, para que um programa torne-se minimamente funcional é preciso que ele receba dados do meio externo (através do teclado, mouse, portas de comunicação, leitores de código de barra, etc.) e emita o resultado de seu processamento de volta para o meio externo (através do monitor, da impressora, do alto-falante, etc.).

Em C, existem muitas funções pré-definidas que tratam desta troca de informações. São as funções de entrada e saída do C. Para termos acesso à biblioteca que contém estas funções, o programa deve conter, em seu início, a diretiva de pré-compilação:

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

As principais funções para entrada e saída (leitura e escrita) de valores, em C, estão definida em `<stdio.h>`. O nome deste arquivo originou-se do termo inglês “standard input/output”, que significa “entrada e saída padrão”. Vamos detalhar essas funções nas seções a seguir.

4.1.19 Impressão Formatada de Dados

A função `printf` (print formated) é uma função da biblioteca de entrada e saída padrão utilizada para escrever valores numéricos, caracteres e strings. Ela pode receber diversos parâmetros, sendo que o primeiro deverá ser uma string, chamada string de controle. Somente será exibido o que for solicitado dentro desta string. A string de controle deve aparecer entre aspas duplas e nela podem existir dois tipos de informações: caracteres comuns e especificadores de formato. Tudo o que aparecer nesta string que não for um especificador de formato será impresso literalmente, ou seja, do jeito que estiver escrito. Justamente por isso, esta função é dita de saída formatada, uma vez que os dados de saída podem ser formatados da maneira que desejarmos (alinhados, intercalados com símbolos, etc). Cada especificador de formato encontrado nesta string será substituído pelo parâmetro seguinte pertencente à lista de argumentos da função, na ordem que aparecerem. A estrutura básica da função `printf()` é dada a seguir:

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

Os especificadores de formato que podem fazer parte da string de controle são simbolos, representados por um % mais um outro caracter, que indicam que um valor vai ser escrito pela função, na posição correspondente da string de controle. Eles indicam, também,

o tipo e/ou a base numérica do dado a ser escrito.

Cada valor a ser impresso pelo printf deve ter um especificador de formato correspondente dentro da string de controle. Se faltarem especificadores, os últimos valores não serão exibidos. Se sobrarem especificadores, serão exibidos valores indeterminados (lixo de memória). A Tabela 12. apresenta os principais especificadores de formato utilizados com a função printf. Por exemplo, para exibir um valor inteiro, usa-se o %d. Da mesma forma, para exibir um valor em ponto flutuante, pode-se usar %f.

Tabela 12 – Especificadores de Formato para a Função printf()

Código	Significado
%c	Exibe um caractere
%d	Exibe um inteiro em formato decimal
%i	Exibe um inteiro
%e	Exibe um número em notação científica (com e minúsculo)
%E	Exibe um número em notação científica (com E maiúsculo)
%f	Exibe um ponto flutuante em formato decimal
%g	Usa %e ou %f, o que for menor
%G	O mesmo que %g, só que um E maiúsculo é usado se o formato %e for escolhido
%o	Exibe um número em notação octal
%s	Exibe uma string
%u	Exibe um decimal sem sinal
%x	Exibe um número em hexadecimal com letras minúsculas
%X	Exibe um número em hexadecimal com letras maiúsculas
% %	Exibe um sinal de %
%p	Exibe um ponteiro

Vamos dar uma olhada em uma série de exemplos:

```
printf("Olá, Mundo!");
```

Saída: Olá, Mundo!

```
printf("linha 1 \nlinha 2 ");
```

Saída: linha 1

linha 2

Observe que na primeira instrução, a saída é exatamente igual à string de controle. Já na segunda instrução, a impressão se deu em duas linhas. Isto se deve ao \n que representa o código ASCII para quebra de linha. Vamos dar exemplos, agora, de saída formatada e os respectivos resultados.

Suponha que idade seja uma variável do tipo inteiro contendo o valor 18.

```
printf("Tenho %d anos de vida",idade);
```

Saída: Tenho 18 anos de vida.

Veja que o especificador de formato %d (que indica um valor inteiro em formato decimal) é substituído pelo conteúdo da variável idade, na posição em que o especificador aparece.

```
printf ("Teste %% %%");
```

Saída: Teste % %

O símbolo de porcentagem (%) é reservado para especificar o tipo a ser impresso. Logo, se desejamos imprimir o próprio símbolo temos de usar %% (vide Tabela 12)

```
printf ("%f",40.34534);
```

Saída: 40.34534

Para imprimir valores de ponto flutuante usamos o %f. O default dele é imprimir números com até seis casas decimais.

```
printf ("Um caractere %c e um inteiro %d",'D',120);
```

Saída: Um caractere D e um inteiro 120

```
printf ("%s e um exemplo","Este")
```

Saída: Este e um exemplo

```
printf ("%s%d%%","Juros de ",10)
```

Saída: Juros de 10%

```
printf("A letra , %c\n", 'A'); ou printf("A letra , %c\n", 65);
```

Saída: A letra A

Os dois comandos acima imprimiram a letra A. O especificador de formato %c indica que deve ser impresso um caractere. Se um número é passado, ele pega o caractere correspondente ao número na tabela ASCII. No caso o valor da letra A na tabela ASCII é 65.

```
printf ("O valor decimal %d em Hexadecimal: %X e em Octal: %o", 107, 107, 107);
```

Saída: O valor decimal 107 em Hexadecimal: 6B e em Octal: 153

`%d` imprime o número na notação decimal convencional, `%X` imprime o número usando a notação hexadecimal e `%o` usando a notação octal.

Suponha que `pi=3.14159` e `raio=2.0031`, ambas variáveis do tipo `float`.

```
printf("A area do circulo e %e\n", 2 * pi * raio);
```

Saída: A área do circulo e 1.258584e+001

O `%e` faz com que o valor de ponto flutuante seja impresso em notação científica. A diferença de `%e` para `%E` é apenas na impressão da letra E, que no primeiro caso é minúscula e no segundo maiúscula.

Suponha que `troco` e `pagamento` sejam variáveis do tipo `float` cujos valores são, respectivamente, 12.3 e 15.0.

```
printf("Total: %.2f \nDinheiro: %.2f \nTroco: %.2f", tot, din, (din-tot));
```

Saída: Total: 12.30

Dinheiro: 15.00

Troco: 2.70

O que é esse `.2` depois do `%f`? Ele significa que o número de ponto flutuante deve ser escrito com duas casas decimais de precisão.

Depois do sinal `%` do especificador de formato e antes do caracter que indica o especificador de formato, podem ser usados alguns modificadores OPCIONAIS cuja sintaxe é a seguinte: `% [flag] [tamanho] [.precisão] caracter_do_especificador_de_formato`

Onde:

`[flag]` especifica a formatação da saída

- → alinhamento à esquerda.

- + → conversão de sinal (saída sempre com sinal: + ou - impresso)

`<espaço>` → conversão de sinal positivo para espaço em branco (saídas negativas com sinal, positivas sem sinal)

[tamanho] especificação de tamanho ou quantidade de dígitos a serem impressos. Em outras palavras, o tamanho do campo indica quantas posições da saída serão utilizados para exibir o valor correspondente à um especificador de formato. Por exemplo, %5d indica que o valor a ser impresso vai ocupar 5 posições na tela. Por padrão, as posições em excesso são preenchidas com brancos e o valor é alinhado à direita do campo. Sempre que o tamanho do campo for insuficiente para exibir o valor solicitado, este tamanho é ignorado e o número é impresso por completo. Desta forma, um valor nunca é truncado. Valores possíveis:

n → pelo menos n dígitos serão impressos (dígitos faltantes serão completados por brancos).

0n → pelo menos n dígitos serão impressos (dígitos faltantes serão completados por zeros).

Exemplos:

```
int valor = 2580;
int valor = 2580;
printf("Valor justificado para dir: %8d\n", valor );
printf("Valor justificado para esq: %-8d\n", ano );
Saída: Valor justificado para dir: 2580
Valor justificado para esq: 2580
```

No primeiro caso, dizemos que queremos imprimir o valor 2580 como inteiro em oito posições. O alinhamento default da impressão é à direita. Se quisermos imprimir à esquerda, devemos usar o flag – (menos) antes do tamanho do número.

[.precisão] especificador de precisão, ou seja, da quantidade de dígitos à direita do ponto decimal. É mais usado para configurar valores de ponto flutuante. Valores possíveis:

(nada) → padrão: 6 dígitos para reais.

.0 → nenhum dígito decimal.

.n → são impressos n dígitos decimais, ou seja, casas depois do ponto decimal.

Por exemplo, o formato **%10.2f**, exibe o valor em ponto-flutuante com duas casas decimais, dentro de um campo de tamanho 10.

Quando as notações de tamanho e de precisão são aplicadas ao tipo string, elas passam a indicar, respectivamente, o tamanho total de casas a serem ocupadas e o número de caracteres que serão realmente impressos. Por exemplo:

```
printf(" %10.3s \n", "Alo Mundo!");
```

Saída: Alo

Foram alocadas 10 casas para imprimir a string (por isso, ela está deslocada para a direita), mas apenas três casas foram realmente impressas. O restante das casas ficou com espaços em branco.

Como mencionado, todos esses modificadores são opcionais (por isso, na sintaxe estão expressos entre colchetes []) e, deles, o mais utilizado é o modificador de precisão, quando desejamos imprimir números de ponto flutuante.



Atenção

Uma observação importante é que muitos compiladores C tratam o especificador de formato %i como idêntico a %d. No entanto, para criar-se novos programas, use o especificador %d, pois %i é um legado do passado, e os compiladores futuros talvez deixem de aceitá-lo.

A função printf retorna um valor inteiro representando o número de caracteres que foram realmente impressos. De repente, você pode em algum programa precisar desse valor. Exemplo:

int retorno = printf("%10.3s", "Alo Mundo!"); → retornaria 10

4.1.20 Leitura Formatada de Dados

A função scanf() (scan formated) é a principal função de entrada de dados da biblioteca de entrada e saída padrão da Linguagem C. Ela pode ser usada para ler virtualmente qualquer tipo de dado inserido por meio do teclado. Porém, ela costuma ser mais utilizada para a leitura de números inteiros ou de ponto flutuante (variáveis int, float e double). A forma geral da função scanf() é: **scanf (“string de controle”, lista de argumentos);**

A função lê da entrada padrão (em geral, teclado) uma lista de valores que serão formatados pela string de controle e armazenados nos endereços das variáveis da lista de argumentos. Tal qual no printf(), a string de controle deverá fazer uso de especificadores de formato que vão dizer à função scanf() qual o tipo de dado que deve ser lido em seguida. Como na função printf(), os códigos de formato da função scanf() devem ser correspondidos na ordem com os endereços das variáveis sendo recebidas na lista de argumentos. Os especificadores de formato estão listados na Tabela 13.

Tabela 13 – Especificadores de formato da função scanf()

Código	Significado
%c	Lê um único caractere
%d	Lê um decimal inteiro
%i	Lê um decimal inteiro (não pode ser octal ou hexadecimal)
%u	Lê um decimal sem sinal
%e	Lê um número em ponto flutuante em notação científica, com sinal opcional
%f	Lê um número em ponto flutuante com ponto opcional
%g	Lê um número em ponto flutuante com expoente opcional (double)
%o	Lê um número em base octal
%s	Lê uma string. Atenção: espaço em branco finaliza entrada
%x	Lê um número em base hexadecimal
%p	Lê um ponteiro ou apontador

Os especificadores de formato **d**, **i**, **o**, **u** e **x** podem ser precedidos pela letra **h**, para indicar que um valor do tipo short ao invés do tipo int deve ser lido na lista de argumentos. Ou podem ser precedidos pela letra **l** (ele) para indicar que um valor long vai ser lido na lista de argumentos. De forma semelhante, os especificadores de formato **e**, **f** e **g** podem ser precedidos pela letra **l** (ele) para indicar que um valor do tipo double será lido, ao invés do tipo float.

As variáveis a serem lidas que fazem parte da lista de argumentos, devem ser precedidas pelo símbolo **&**, exceto quando as variáveis forem do tipo string. Essa é uma diferença fundamental que existe entre o scanf() e o printf(). No caso de scanf, os argumentos são os endereços das variáveis que irão receber os valores lidos. Já no printf, os argumentos são as próprias variáveis (com seu respectivo valor) que estão lá para serem impressas. Exemplos:

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

Esse comando espera que dois valores inteiros sejam digitados

no teclado. O primeiro valor será armazenado na variável num1 (ou melhor, no endereço de memória com esse nome) e o segundo valor na variável num2. Os valores serão armazenados diretamente nos endereços indicados por &num1 e &num2, respectivamente. Outro exemplo

```
int indice;
float valor;
scanf("%d %f", &indice, &valor);
```

No comando acima, estariam sendo lidos dois valores, um valor do tipo inteiro (que seria armazenado no endereço referenciado pela variável índice) e um valor do tipo float (que seria armazenado no endereço referenciado pela variável valor).

```
int main()
{
    char nome[5];
    printf("Digite um nome: ");
    scanf("%s", nome);
    printf("Olá %s\n", nome);
}
```

Para ler strings se utiliza o especificador de formato %s. Observe que, como mencionado anteriormente, não foi usado o símbolo & antes do nome da string. Isso porque a string não é um tipo primitivo e sim uma coleção de caracteres.



A função “scanf” acrescenta o ‘\0’ automaticamente após o último caractere. Por exemplo: se for digitado o nome “ANA”, o conteúdo do vetor definido no código em C acima seria:

nome	0	1	2	3	4
	A	n	a	\0	

A função “scanf” lê strings apenas até ser lido um enter ou espaço em branco. Logo, não pode ser usada para ler palavras ou nomes

compostos. Por exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    char nome [40]; /* uma string de 40 caracteres */
    /* Solicitação de entrada de dados */
    printf ("Por favor , qual o seu nome ?\n");
    scanf("%s", nome);
    printf ("Oi %s, tudo bem? \n", nome );
    system("PAUSE");
    return 0;
}
```

Considere que durante a execução desse programa, o usuário digitou quando o nome foi solicitado à string: *Sandra Siebra*. Porém, como o scanf só lê até um espaço em branco, ele só armazenaria o valor *Sandra*. Dessa forma, a mensagem que seria impressa pelo printf seria: *Oi Sandra, tudo bem?* E agora, como leríamos o nome todo? Na próxima seção, saberemos essa resposta! Aguarde!

Qualquer caractere que não seja um espaço em branco ou um especificador de formato que faça parte da string de controle da função scanf() será considerado separador de valores e, por isso, será lido e descartado. Por exemplo:

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

faz com que a função scanf() espere que seja fornecido um número inteiro, depois, uma vírgula (que será descartada, não será armazenada) e, finalmente, que seja fornecido o outro número inteiro. Apenas os dois números inteiros serão efetivamente gravados. Agora, cuidado com esse uso! Porque, quando se faz uso de um separador de valores, se o caractere especificado como separador não for encontrado na leitura (não for digitado pelo usuário), a função scanf() terminará, ou seja, não continuará fazendo a leitura, encerrando o programa com erro. Logo, para o comando scanf especificado acima, o usuário teria de digitar, por exemplo: 46,50 (um número inteiro, uma vírgula, outro número inteiro). Observe que a vírgula serve apenas

como separador dos números.

Suponha o seguinte comando: **`scanf("%c%c%c", &var1, &var2, &var3);`** Se o usuário digitasse a seguinte entrada: x y

Que valores seriam gravados nas variáveis var1, var2 e var3???

Espaços em branco são considerados entradas válidas, dessa forma, em var1 seria armazenado o caracter x, em var2 seria armazenado um espaço em branco e em var3 o caracter y.

4.1.21 Outras Formas de Entrada e Saída e Dados

Entrada e Saída de caracter individual: `getchar()` e `putchar()`

Para ler e escrever caracteres do teclado as funções de entrada e saída mais simples são, respectivamente, *getchar* e *putchar*, que pertencem, também, a biblioteca *stdio.h*.

A função *getchar* (*get character*) lê um caracter da entrada padrão (em geral, o teclado) e retorna o caracter digitado, que pode ser atribuído para uma variável do tipo char. É necessário que um <enter> seja teclado, após o caracter a ser lido, para que ele seja copiado para a variável do tipo char. Por exemplo:

```
char letra;
letra = getchar();
```

Esta função, também, pode ser utilizada simplesmente para dar uma pausa no programa até que o usuário tecle <enter>. Neste caso simplesmente descarta-se o valor de retorno, não atribuindo-o a nenhuma variável.

Se forem digitados vários caracteres, estes ficarão armazenados no buffer de entrada até que a tecla <enter> seja pressionada. Então, cada chamada da função *getchar()* lerá um dos caracteres que estiver armazenado no buffer.

A função *putchar()* (*put character*) imprime um caracter individual na saída padrão (em geral, o monitor de vídeo do computador).

Um exemplo completo para ilustrar o uso do *getchar* e do *putchar* pode ser visto a seguir.

```

#include <stdio.h> // inclui as bibliotecas do C necessárias para o programa
#include <stdlib.h>
int main ()
{
    char letra; // declara uma variável do tipo caracter
    printf ("Entre com uma letra: \n"); // imprime na tela a mensagem especificada
    letra = getchar (); // lê um caracter após ser pressionada a tecla <enter>
    printf ("O caractere lido foi: "); // imprime na tela a mensagem
    putchar (letra); // imprime na tela o caracter armazenado na variável letra
    system("PAUSE"); // espera uma tecla ser digitada para encerrar o programa
    return 0; // indica que o programa main acabou sem erros (valor zero)
}

```



Atenção

Observe que, normalmente, quando algum dado é fornecido pelo teclado termina-se a digitação com a tecla <enter>. No entanto, o <enter> é considerado um caracter válido e, dependendo de como o próximo dado estiver sendo lido, o <enter> pode acabar tomando o lugar dele. Ou seja, o <enter> em alguns casos, pode vir a ser um problema! Vamos analisar o código abaixo. Por favor, digite, compile e execute no DEV-C++ o programa e veja o que acontece.

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main ()
4. {
5. char letra; // declara um caracter
6. int num; //declara um inteiro
7. printf("Digite um numero inteiro: "); //imprime a frase
8. scanf("%d", &num); //lê um valor do tipo inteiro e armazena no endereço de num
9. printf("Digite uma letra: \n"); //imprime a frase
10. letra = getchar(); //lê um caracter após ser digitado <enter>
11. printf("O numero lido e: %d\n", num); //imprime a frase e um valor do tipo inteiro
12. printf("O caracter lido e: %c\n", letra); //imprime a frase e um valor do tipo caracter
13. system("PAUSE"); // espera uma tecla ser digitada para encerrar o programa
14. return 0;
15. }

Use como valores para teste o número 10 e a letra S. E aí, consegui digitar os dois valores? O que aconteceu durante a execução? Seu resultado foi o mesmo apresentado na Figura 7?

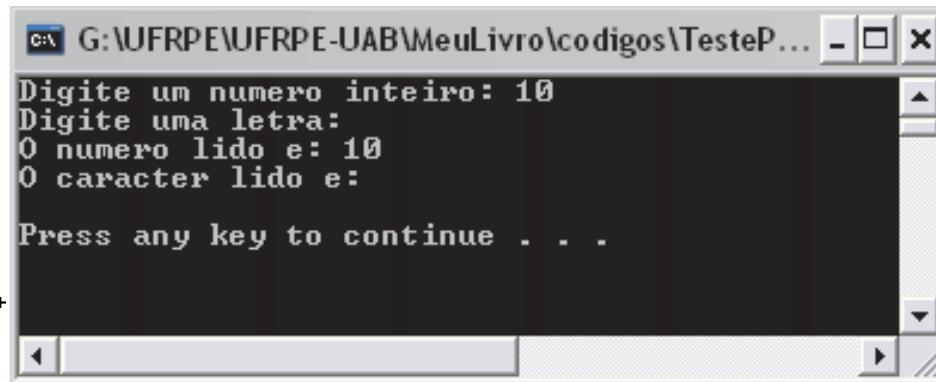


Figura 7 - Janela de execução de um programa no Dev-C+

Leitura de Códigos do Teclado: getch() e getche()

Estas funções fazem a leitura de códigos de teclado. Estes códigos podem representar teclas de caracteres (tais como: A, B, s, 9, 0, *) ou teclas de comandos (tais como, <enter>, <delete>, <Page Up>, <F1>). Não é necessário que um <enter> seja pressionado para que o valor digitado seja lido. Essa é uma das principais diferenças para os outros comandos de leitura. Outra diferença é que essas funções não pertencem a biblioteca stdio.h, mas sim a biblioteca conio.h

Ao ser executada, a função getch() (get character) aguarda que uma tecla (ou combinação de teclas) seja pressionada, recebe do teclado o código correspondente a essas teclas e retorna este valor, sem imprimi-lo na tela (ou seja, o usuário não vê o que foi digitado).

A função getche()(get character and echoe) faz a mesma coisa do getch(), com a diferença que ele escreve na tela (ecoa na tela), quando possível, o caractere digitado. Exemplo:

```

1. #include <stdio.h>
2. #include <stdlib.h>
3. int main()
4. {
5.     char nome[60]; // declara uma string de 60 caracteres
6.     printf("Informe o seu nome: "); //imprime a frase
7.     gets(nome); //lê a string digitada, inclusive com os espaços em branco
8.     printf("O seu nome é: %s \n", nome); //imprime a frase e a string
9.     system("PAUSE"); // espera uma tecla ser digitada para encerrar o programa
10    return 0; //indica final do programa sem erros
11}
```

Imprimindo strings: uso do puts

A função puts imprime uma string e é simples de usar, pois basta passar como parâmetro para a função a string a ser impressa. A impressão sempre vai terminar e passar para uma próxima linha, como se tivesse sendo utilizado um comando printf("%s\n", frase). Vamos dar um exemplo:

Atenção

Não é possível imprimir uma string de texto + uma variável string ao mesmo tempo usando um único puts. Por isso, o programa acima precisou fazer uso de 3 puts distintos para imprimir a frase, uma vez que o valor da variável nome, precisou ser impressa no meio da frase.

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

int main ()
{
    char nome [40]; /* uma string de 40 caracteres */
    /* Solicitação de entrada de dados */
    printf ("Por favor , qual o seu nome ?\n");
    scanf("%s", nome);
    printf ("Oi %s, tudo bem? \n", nome );
    system("PAUSE");
    return 0;
}
```

4.1.22 Aprenda Praticando

Bem, finalmente, começamos a linguagem C e você começou a ter contato com uma linguagem de programação estruturada legítima. Vamos agora dar um exemplo completo de implementação de algoritmo seqüencial que é o que dá para ser implementado com o que vimos até agora na linguagem C.

Construa um algoritmo para ler o nome, número de horas trabalhadas e número de dependentes de um funcionário de uma empresa. Depois, calcule o que for necessário e imprima na tela: o nome do funcionário, seu salário bruto, o total de descontos que ele sofrerá e seu salário líquido. Considere que:

- A empresa paga R\$10 por hora trabalhada e tem um adicional de R\$40 por dependente que o funcionário tenha.
- O salário bruto é o valor total dos ganhos do funcionário
- Sobre o salário bruto são descontados 8 % de INSS e 5% de IR (imposto de renda).

- O salário líquido é o valor do salário bruto, menos os descontos.

```
// primeiro você tem de lembrar de incluir as duas bibliotecas padrão mais utilizadas
#include <stdio.h>
#include <stdlib.h>

int main()
{
    // aqui são feitas as declarações das variáveis. Lembre que vamos precisar de uma variável
    // para cada valor que precisa ser lido (nome, horasTrab, dependentes) e uma variável para
    // cada valor a ser calculado (salarioBruto, salarioLiquido, inss e ir)

    char nome[45];
    float horasTrab, salarioBruto, salarioLiquido, inss, ir;
    int dependentes;

    // a primeira coisa a fazer é pedir ao usuário para digitar os valores de entrada
    printf ("\nDigite o nome do funcionário: ");
    gets(nome); // lê a string do nome
    printf("\nDigite a quantidade de horas trabalhadas: ");
    scanf("%f", &horasTrab); // lê um valor de ponto flutuante
    printf("\nDigite o número de dependentes do funcionário: ");
    scanf("%d", &dependentes);

    // com os valores lidos, podemos calcular agora o salário bruto do funcionário
    // multiplicando o valor em reais de cada coisa, pela sua quantidade
    salarioBruto = (10 * horasTrab) + (40 * dependentes);
    // o INSS é 8% do salário bruto
    inss = (8*salarioBruto)/100;
    // o IR é 5% do salário bruto
    ir = (5*salarioBruto)/100;
    // o salário líquido é o salário bruto subtraído dos descontos dos impostos
    salarioLiquido = (salarioBruto - inss) - ir;
    // depois de tudo calculado, imprimimos os resultados
    printf("Nome Funcionário %s: \n", nome);
    printf("Salário Bruto: R$ %.2f\n", salarioBruto);
    printf ("Descontos: R$ %.2f \n", (ir+inss));
    printf ("Salário Líquido: R$ %.2f \n", salarioLiquido);
    system("PAUSE");
    return 0;
}
```

Vamos agora comparar a construção de um algoritmo na linguagem

algorítmica e na linguagem C.

Faça um programa para ler um preço unitário, uma quantidade comprada e calcular e exibir o preço total a pagar.

Linguagem Algorítmica	Linguagem C

Consegue mapear as diferenças entre os dois códigos?



Saiba Mais

Para saber mais sobre a linguagem C você pode consultar qualquer livro que fale sobre a linguagem. Destaco a chamada “bíblia do C” que é o livro:

SCHILD'T, Herbert. **C completo e total.** Tradução Roberto Carlos Mayer. São Paulo: Makron Books, 3^a.edição, 1996.

Porém, há muitos materiais na Internet (apostilas, tutoriais, cursos, etc). Aqui vão alguns links que consultei antes de escrever esse material e que podem lhe ajudar:

http://www.lsi.cefetmg.br/~rogerio/ApostilaC_UFMG.pdf

http://www.coltec.ufmg.br/~info/cursoC_olinux/curso_c_htm.htm

<http://www.ericolisboa.eng.br/cursos/apostilas/lingc/index.htm>

<http://equipe.nce.ufrj.br/adriano/c/apostila/indice.htm>

<http://www.inf.pucrs.br/~pinho/LaproI/>

http://www.inf.ufrgs.br/~fernando/Ensino/040_pdfs/20061_apoio.html

http://www.ccuec.unicamp.br/treinamento_int2004/lingc/index.htm

http://br.geocities.com/sdiasneto/c_bas/index.htm

<http://www.numaboa.com.br/informatica/c/>

Existe um WIKI na Internet chamado Casa dos Programadores, para que pessoas que trabalhem com programação possam compartilhar experiências. Lá há tutoriais, apostilas, fóruns de discussão, artigos, entrevistas, etc

<http://casadosprogramadores.com.br/menu.html>

Também existem algumas vídeo aulas de C no YouTube que podem ser interessantes para você:

<http://br.youtube.com/watch?v=YiQOBPBxb00>

http://br.youtube.com/watch?v=yYkKBO_f23k

<http://br.youtube.com/watch?v=C1b8gI3qxPU>

<http://br.youtube.com/watch?v=y9Ut4GhQrUg>



Atividade de Estudo

Construa programas usando a linguagem C para resolver os problemas abaixo. Lembre das boas práticas de programação!

- 1) Leia o nome e o salário de um funcionário e exiba com resultado o nome do mesmo com o salário reajustado em 12%.
- 2) Leia 2 notas, calcule e exiba a média ponderada dessas notas. Considere que a nota1 tem peso 6 e a nota2 tem peso 4.
- 3) Faça um algoritmo que receba um valor em dólar e transforme o mesmo em reais, sabendo que a taxa de conversão é: 1 dólar = R\$ 1.70.
- 4) Leia uma distância em km entre dois pontos e o preço da gasolina em reais. Depois, exiba quantos litros de gasolina o carro irá consumir e quanto será o gasto em reais, sabendo que o carro em questão consegue percorrer 12 km com um litro de gasolina.
- 5) Leia as variáveis inteiros n1 e n2 e troque o valor destas variáveis. Isto é, n1 deve ficar com o valor de n2 e n2 deve ficar com o valor de n1. Exiba os valores antes e depois da troca.
- 6) Faça um programa que leia um valor inteiro no formato decimal e escreva, na tela, este mesmo valor nas bases hexadecimal e octal.
- 7) Faça um programa capaz de ler um valor de ponto flutuante e escrevê-lo com apenas uma casa decimal.
- 8) Sabendo que os argumentos da função printf podem ser expressões ($a+b$, a/b , $a*b$, $3*a...$) e não apenas argumentos, faça um programa para ler um valor inteiro e escrever os valores de seu triplo, seu quadrado, e a sua metade.
- 9) Escreva um programa que leia o nome de um aluno e suas três notas e imprima o nome do aluno e a média aritmética das notas.
- 10) Faça um programa que utilize a função gets para ler duas strings de tamanho máximo igual a 20 e, em seguida, escrever as memsas uma ao lado da outra, no formato:

```
% string1 / string2%
```



Resumo

C é uma linguagem de programação compilada de propósito geral, estruturada, de alto e baixo nível, e padronizada (padrão ANSI C), criada em 1972, por Dennis Ritchie, no AT&T Bell Labs. Desde que essa linguagem foi usada para desenvolver uma nova versão do sistema operacional UNIX (que foi originalmente escrito em Assembly), ela tornou-se popular e passou a ser uma das linguagens de programação mais usadas. A linguagem C influenciou muitas outras linguagens, especialmente C++, que foi originalmente desenvolvida como uma extensão para C.

O ambiente de implementação em C que vamos utilizar é o Dev-C++ devido ao fato do mesmo ser gratuito, fácil de instalar e ter um ambiente integrado para edição, compilação e execução de programas, além de ferramentas para depuração de erros.

A linguagem C possui diversas bibliotecas que contêm funções pré-definidas, utilizadas nos programas para ações específicas. A biblioteca desejada pode ser inserida e utilizada nos programas pelo pré-processador a partir do uso da diretiva **#include**.

O C possui quatro tipos básicos de constantes: inteiros, de ponto flutuante, caracteres e strings. Constantes inteiros e de ponto flutuante representam números de um modo geral. Caracteres e strings representam letras e agrupamentos de letras (palavras).

Variáveis são nomeações de endereços de memória onde podem ser armazenados valores. A declaração de variáveis informa ao processador quais são os nomes utilizados para armazenar dados variáveis e quais são os tipos usados. Deste modo o processador pode alocar (reservar) o espaço necessário na memória para a manipulação destas variáveis. Os tipos de dados usados em C podem ser: **int**, **float**, **double**, **void** e **char**. Esses tipos de dados podem ser alterados pelos modificadores: **signed**, **unsigned**, **long** e **short**.

String (um vetor de caracteres) não é um tipo primitivo e é definido como um array de caracteres: **char nomeString[tamanho]**; Lembre que em C não existe o tipo lógico. Os valores verdadeiro e falso são

representados pelos valores inteiros 1 e 0, respectivamente.

Há três lugares nos quais podemos declarar variáveis. O primeiro é fora de todas as funções do programa. Estas variáveis são chamadas variáveis globais e podem ser usadas a partir de qualquer lugar no programa. O segundo lugar no qual se pode declarar variáveis é no início de um bloco de código. Estas variáveis são chamadas locais e só têm validade dentro do bloco no qual são declaradas, isto é, só a função à qual ela pertence sabe da existência desta variável, dentro do bloco no qual foram declaradas. O terceiro lugar onde se pode declarar variáveis é na lista de parâmetros de uma função. Mais uma vez, apesar de estas variáveis receberem valores externos, estas variáveis são conhecidas apenas pela função onde são declaradas.

C possui operadores aritméticos, de atribuição, de atribuição aritmética, relacionais, lógicos e um operador condicional. Além disso, pode fazer uso de expressões lógicas e aritméticas.

Para impressão de dados em C podem ser usadas as funções: **printf** (para impressões em geral), **puts** (para imprimir string) e **putchar** (para imprimir um caracter). Para entrada de dados em C podem ser usadas as funções: **scanf**(leitura principalmente de valores numéricos), **getchar** (para leitura de caracter após ser teclado <enter>), **getch** e **getche** (para leitura de caracter sem necessidade de teclar <enter>) e **gets** (para leitura de string).

As funções printf e scanf são formatadas e necessitam fazer uso de especificadores de formato na sua string de controle. Existe um especificador para cada tipo de dado.

Toda vez se for alternar entre leituras de dados usando scanf e qualquer uma das outras formas de leitura de dados, deve-se usar a função de leitura de buffer de entrada de dados: **fflush(stdin)**; para que não se faça a leitura de um valor (por exemplo, um <enter>) indesejado.

Na próxima unidade daremos continuidade a linguagem C, falando dos comandos de controle de fluxo (condicionais e de repetição). O que lhe dará mais recursos para implementar mais tipos diversificados de programas. Até lá!

Introdução à Programação

Sandra de Albuquerque Siebra



FASCÍCULO 5



Universidade Federal Rural de Pernambuco

Reitor: Prof. Valmar Corrêa de Andrade

Vice-Reitor: Prof. Reginaldo Barros

Pró-Reitor de Administração: Prof. Francisco Fernando Ramos Carvalho

Pró-Reitor de Extensão: Prof. Paulo Donizeti Siepierski

Pró-Reitor de Pesquisa e Pós-Graduação: Prof. Fernando José Freire

Pró-Reitor de Planejamento: Prof. Rinaldo Luiz Caraciolo Ferreira

Pró-Reitora de Ensino de Graduação: Profª. Maria José de Sena

Coordenação de Ensino a Distância: Profª Marizete Silva Santos

Produção Gráfica e Editorial

Capa e Editoração: Allyson Vila Nova, Rafael Lira, Alesanco Azevedo e Aline Fidelis

Revisão Ortográfica: Ivanda Martins

Ilustrações: Allyson Vila Nova

Coordenação de Produção: Marizete Silva Santos



Sumário

Fascículo 5 - Linguagem de Programação C	6
Objetivos do Fascículo 5	6
Conteúdo Programático do Fascículo 5	6
5.1 Unidade 7 – Estruturas de Controle de Fluxo de Dados.....	8
5.1.1 Estrutura Condicional Simples.....	9
5.1.2 Estrutura Condicional Composta	10
5.1.3 Estrutura Condicional Composta Aninhada	11
5.1.4 Estrutura Condicional Múltipla	13
5.1.5 Estrutura de Repetição Contada	18
5.1.6 Estrutura de Repetição Condicional com Teste no Início	22
5.1.7 Estrutura de Repetição Condicional com Teste no Final	25
5.1.8 Comando Break.....	27
5.1.9 O Comando continue.....	29
5.2 Unidade 8 – Tipos Estruturados de Dados.....	37
5.2.1 Tipos Estruturados Homogêneos Unidimensionais (Vetores)	37
Declaração de Vetores	38
Referência aos Elementos de um Vetor.....	39
Limites do Vetor.....	41

Inicialização de vetores	42
Definindo o Tamanho do Vetor com Constantes	42
Lendo Valores para os Elementos de um Vetor	43
5.2.2 Manipulação de Strings	44
Funções Pré-Definidas para manipulação de Strings	45
5.2.3 Tipos Estruturados Homogêneos Bidimensionais (Matrizes)	51
Declaração de Matrizes	52
Referência aos Elementos de Uma Matriz	52
Matrizes de strings	54
Inicialização de Matrizes	55
Lendo Valores para os Elementos de uma Matriz	58
5.2.4 Tipos Estruturados Homogêneos Multidimensionais	59
5.2.5 Tipos Estruturados Heterogêneos (Registros ou Estruturas)	61
Declaração	61
Declarando Variáveis do Tipo da Estrutura Definida	64
Referenciando Elementos em uma Estrutura	65
5.2.6 O Comando <code>typedef</code>	67
Vetores de Estruturas	68

Fascículo 5 - Linguagem de Programação C

Carga horária do Módulo: 90 horas

Objetivos do Fascículo 5

- Apresentar os componentes básicos de um programa utilizando uma linguagem de programação estruturada
- Apresentar uma ferramenta para compilação de programas em uma linguagem estruturada
- Desenvolver a habilidade de programação fazendo uso de variáveis, operadores, expressões, estruturas seqüenciais, estruturas de repetição, tipos estruturados de dados (homogêneos e heterogêneos) e modularização de programas
- Incentivar a prática de desenvolvimento de programas de forma a aprimorar no aluno a sua lógica de resolução de problemas

Conteúdo Programático do Fascículo 5

Unidade 6 – Noções Básicas da Linguagem C (10 horas)

- Ambiente de Programação C
- Comentários, palavras reservadas e identificadores
- Constantes e Variáveis
- Tipos de dados
- Declaração de Variáveis
- Operadores e Expressões
- Entrada e Saída de Dados
- Estruturas Seqüenciais

Unidade 7 – Estruturas de Controle de Fluxo (7 horas e meia)

- Estruturas Condicionais
- Estruturas de Repetição

Unidade 8 – Tipos Estruturados de Dados (10 horas)

- Tipos de Dados Homogêneos: vetores e matrizes
- Manipulação de Strings
- Tipo de Dados Heterogêneo: registros

Unidade 9 – Subalgoritmos (10 horas)

- Modularização: funções, procedimentos, passagem de parâmetros e recursão

Unidade 10 – Desenvolvimento de Projeto (7 horas e meia)

- Bibliotecas e Funções Úteis
- Guia para o desenvolvimento de projetos
- Exemplos Práticos

Cada unidade conterá exercícios para fixação do assunto apresentado, assim como indicações de leituras adicionais. A Unidade 6 exigirá 10 horas de dedicação, uma vez que você estará começando com o ambiente de programação que usa a linguagem C. A unidade 7 exigirá 7 horas e meia. As unidades 8 e 9 exigirão um pouco mais: 10 horas de dedicação e lhe restarão 7 horas e meia para desenvolvimento do projeto final da disciplina, para o qual você encontrará diversas orientações na Unidade 10. Tenha em mente que muitos dos conceitos que serão vistos nessas unidades, você já utilizou em linguagem algorítmica (com exceção de registros e recursão), logo, a novidade será implementar os conceitos na linguagem C.

Mais do que nunca você vai precisar praticar. Logo, arregace as mangas e vamos lá!

5.1 Unidade 7 – Estruturas de Controle de Fluxo de Dados

Com o que foi visto até agora sobre a linguagem C, só é possível fazer programas usando uma estrutura seqüencial. Ou seja, as instruções vão sendo executadas de cima para baixo e a próxima instrução só é executada após a anterior ser encerrada. Porém, como já visto na parte de linguagem algorítmica, é necessário fazer uso de estruturas que proporcionem ou a execução condicional de blocos de comandos ou a repetição de blocos de comandos, para que seja possível implementar programas mais complexos. Estas estruturas podem ser chamadas de estruturas de controle de fluxo de dados e esta unidade é dedicada ao estudo das mesmas.

Basicamente, existem dois tipos de estruturas de controle de fluxo de dados: as estruturas condicionais (também chamadas de decisão ou de seleção) e as estruturas de repetição.

As estruturas condicionais permitem executar um, entre dois ou mais blocos de instruções dependendo da veracidade de uma condição. Ou seja, a execução de um ou mais comandos dependerá de se uma condição é satisfeita ou não. Estas estruturas podem ser classificadas em: condicional simples, condicional composta, condicional composta aninhada e condicional múltipla (ou de seleção múltipla).

As estruturas de repetição permitem que um bloco de comandos seja executado, repetidamente, uma quantidade controlada de vezes ou enquanto uma determinada condição não é alcançada. As estruturas de repetição também são conhecidas como Laços ou Loops e podem ser classificadas em dois tipos: e repetição contada, repetição condicionada com teste no início e repetição condicionada com teste no final.

Em todas as estruturas desta unidade, existe pelo menos uma expressão que faz o controle de qual bloco de instruções será executado ou quantas vezes ele será executado: é a chamada condição de controle. Uma condição de controle é uma expressão lógica ou aritmética cujo resultado pode ser considerado verdadeiro ou falso.



Atenção

Você já estudou sobre essas estruturas na Unidade 2 (Linguagem Algorítmica), seções 1.2.8 e 1.2.9. Lá você encontra mais detalhes do funcionamento de cada estrutura dessas. Se necessário, revise o assunto!



Atenção

Como mencionado na unidade 6, a linguagem C **não possui** variáveis ou constantes lógicas, possui somente expressões numéricas, assim quando uma expressão numérica se encontra em uma condição de controle, ela será considerada falsa se seu valor for igual a zero e verdadeira se seu valor for diferente de zero. Por exemplo, suponha que `var1 = 5` e `var2 = 0` e observe os resultados das condições lógicas da Tabela 1.

Condição	Media	Aparece na tela

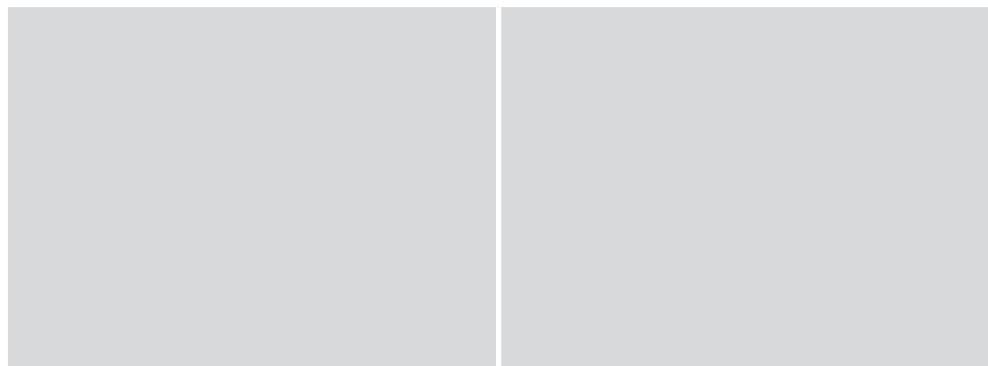
Tabela 1 - Exemplo

Veja na Tabela 1 que, caso não seja utilizada uma comparação como condição, mas apenas alguma variável numérica, este valor é avaliado: se o valor da variável é 0 (zero), ele é considerado falso. Qualquer outro valor da variável é considerado verdadeiro.

Dessa forma, nesta unidade, quando for dito que uma condição é falsa significa dizer que ela é igual a zero. E quando for dito que uma condição é verdadeira significa dizer que ela é diferente de zero. Vamos detalhar agora cada uma das estruturas de controle de fluxo de dados na linguagem C.

5.1.1 Estrutura Condicional Simples

Na estrutura condicional simples, uma única expressão lógica é avaliada, para decidir se o bloco de instruções será executado ou não. Vamos comparar a sintaxe da estrutura em linguagem algorítmica (que foi vista na seção 1.2.8) e em linguagem C.

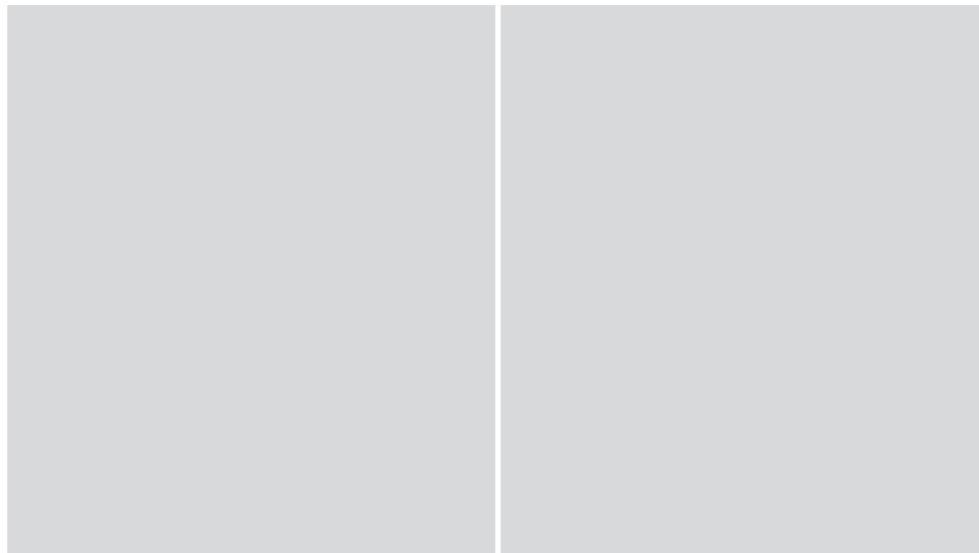


Em ambos os casos, se a **condição** for verdadeira (em C, equivale a dizer se a condição for diferente de zero, lembra?) serão executados os um ou mais comandos dentro do bloco (<comando-1>...<comando-N>). Na linguagem C a condição deve obrigatoriamente ser delimitada por um par de parênteses. Se houver mais de um comando a ser executado, obrigatoriamente, eles devem estar delimitados por um par de chaves. Porém, se houver um único comando, as chaves são opcionais. Cada comando dentro da estrutura deve ser finalizado por um ponto e vírgula, como já explicado anteriormente. Vejamos alguns exemplos:

```
// com apenas um comando as chaves são opcionais
if (idade < 18)
    printf ("Pessoa menor de idade!");
// com mais de um comando, as chaves são obrigatórias, porque são
// delimitadores de bloco.
if (idade < 18){
    printf ("Pessoa menor de idade!");
    printf("A pessoa só tem %d anos", idade);
}
```

5.1.2 Estrutura Condicional Composta

Na estrutura condicional composta, a condição irá determinar qual comando ou bloco de comandos será executado dentre os listados. Ou seja, o controle de qual bloco será executado será decidido por uma condição (expressão lógica ou numérica). Vamos comparar a sintaxe da estrutura em linguagem algorítmica (que foi vista na seção 1.2.8) e em linguagem C.



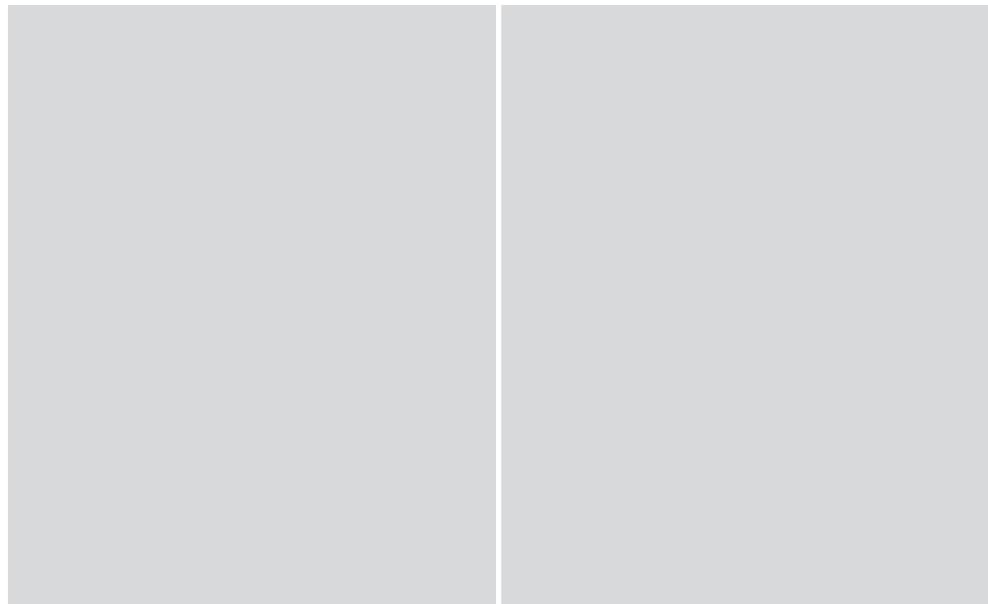
Nesta estrutura, se a condição for verdadeira (diferente de zero) o primeiro bloco de comandos <comando-A1> ... <comando-AN> será executado. Caso contrário, o segundo bloco de comandos <comando-B1> ... <comando-BN> será executado. A parte do comando formada pelo **else** não é obrigatória. É importante nunca esquecer que, quando usamos a estrutura if-else, estamos garantindo que uma das duas declarações será executada. Nunca serão executadas as duas ou nenhuma delas. Vejamos agora um exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int num;
    printf ("Digite um numero: ");
    // lê um número inteiro
    scanf ("%d",&num);
    // O número é par se o resto da divisão (%) por 2 é igual a zero
    // como só tem um comando dentro do if e dentro do else, as
    // chaves são opcionais
    if (num % 2 == 0)
        printf("O número %d é par!", num);
    else
        printf("O número %d é ímpar!", num);
    system("pause");
    return 0;
}
```

5.1.3 Estrutura Condicional Composta Aninhada

A instrução condicional if pode ser encadeada dentro de outro if mais externo. Na verdade, podemos criar estruturas condicionais

uma dentro da outra, seja no bloco executado se a condição for verdadeira (`if`) ou no bloco executável se a condição for falsa (`else`). O encadeamento destas instruções também é chamado de aninhamento. Vamos dar uma olhada na sintaxe desta estrutura em linguagem algorítmica (que foi visto na seção 1.2.8) e em linguagem C.



Nesta estrutura, o programa começa a testar as condições começando pela 1 e continua a testar até que ele ache uma expressão cujo resultado dê diferente de zero (ou seja, verdadeiro). Neste caso, ele executa o bloco de comandos correspondente. Observe que só um bloco de comandos será executado entre todos, ou seja, só será executado o bloco de comandos equivalente à primeira condição que der diferente de zero. O último bloco de comandos é o que será executado no caso de todas as condições darem zero (ou seja, falso).

Porém, ele é opcional, ou seja, você pode querer que, se nenhuma condição for verdadeira, NADA seja executado. Vamos dar uma olhada em um exemplo.

```

#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int num;
    printf ("Digite um numero: ");
    scanf ("%d",&num); // lê o número
    if (num > 0) // checa se o número é positivo
    {
        printf ("\nNúmero positivo!");
        printf ("\n %d > 0", num);
    }else if (num < 0){ // checa se o número é negativo
        printf ("O numero e negativo");
        printf ("\n %d < 0", num);
    }else{ // se não for positivo, nem negativo, só pode ser zero
        printf ("O numero e igual a zero.");
    }
    system("pause"); //uma pausa para ver o resultado da execução
    return(0); // para finalizar o programa sem erros
}

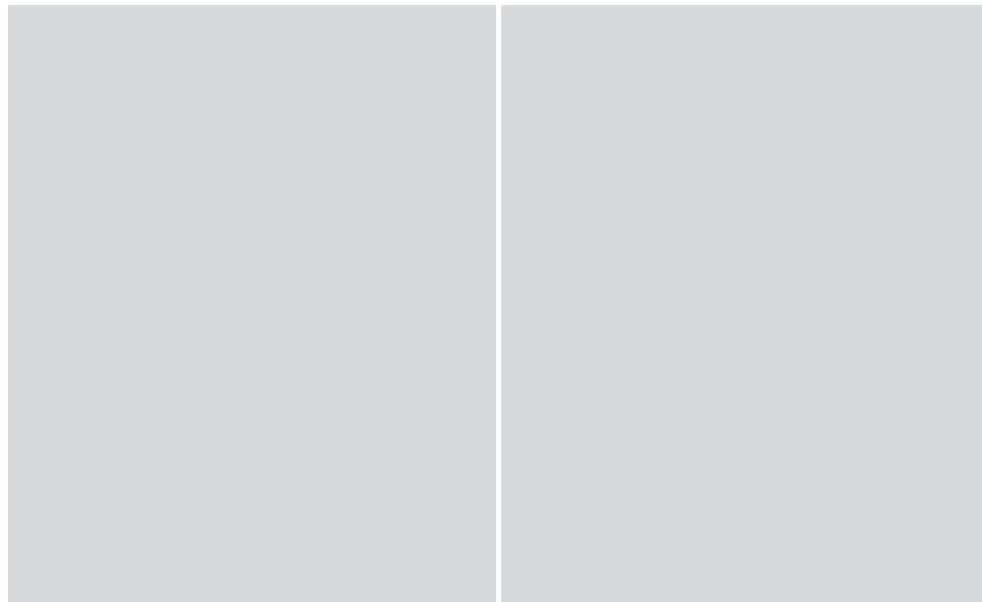
```

Atenção

Você deve tomar cuidado para deixar claro a qual **if** um determinado **else** está ligado. Para isso, é importante **SEMPRE** identar os seus programas

5.1.4 Estrutura Condicional Múltipla

A estrutura condicional múltipla ou de decisão múltipla é equivalente à estrutura condicional composta aninhada (seção 5.1.3), que permite a execução de opções mutuamente exclusivas. Esta estrutura permite a execução de um bloco de comandos a partir de pontos diferentes, conforme o resultado de uma variável ou expressão inteira de controle. O resultado da expressão ou o valor da variável é comparado ao valor de cada um dos rótulos e os blocos de comandos são executados a partir desde rótulo. Os rótulos devem ser constantes inteiras diferentes entre si. Vamos dar uma olhada na sintaxe desta estrutura em linguagem algorítmica (que foi visto na seção 1.2.8) e em linguagem C.



Como anteriormente mencionado, o seletor na linguagem C deve ser uma variável ou expressão inteira. O rotulo1, rotulo2 e os outros rotulos que existirem devem ser constantes inteiros. E como essa estrutura funciona?

1. A expressão ou variável é avaliada;
2. O resultado da expressão ou variável é comparado com os valores das constantes inteiros que devem estar no lugar dos rótulos dos comandos case;
3. Quando o resultado da expressão ou variável for igual a uma das constantes, a execução se inicia a partir do comando associado com esta constante.
4. A execução do bloco de comandos continua até o fim do comando switch (seqüencialmente) ou até que um comando break seja encontrado;
5. Caso o resultado da expressão ou variável for diferente de todos os valores das contantes, o <bloco-comandos-extra> associado ao comando default é executado. O comando default é opcional, e se ele não aparecer nenhum comando será executado.

O comando break é um dos comandos de desvio da linguagem C. O break é usado dentro do comando switch para interromper a execução da seqüência de comandos e pular para o comando seguinte ao comando switch. O uso do break é importante, porque se ele não for usado, todas as intruções a partir do rótulo selecionado serão executadas seqüencialmente.

O **switch...case** é particularmente útil quando se tem um conjunto de instruções que se deve executar em ordem, porém se pode começar em pontos diferentes das instruções. Um caso em que é muito utilizado este tipo de estrutura é na seleção de opções de menu. Esta estrutura também é útil quando a mesma variável precisa ser comparada diversas vezes com valores diferentes. Nestes casos, um switch pode substituir muitos comandos **if**. Vamos ver um exemplo de uso do **switch...case** a seguir em um programa para imprimir o número da semana correspondente ao número digitado.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int dia;
    printf("Informe o dia da semana (números entre 1-7): ");
    scanf("%d", &dia); // lê o número do dia da semana
    switch(dia){
        case 1: // se o número lido for igual a 1
            printf("Domingo\n");
            break;
        case 2: // se o número lido for igual a 2
            printf("Segunda-feira\n");
            break;
        case 3: // se o número lido for igual a 3
            printf("Terça-feira\n");
            break;
        case 4: // se o número lido for igual a 4
            printf("Quarta-feira\n");
            break;
        case 5: // se o número lido for igual a 5
            printf("Quinta-feira\n");
            break;
        case 6: // se o número lido for igual a 6
            printf("Sexta-feira\n");
            break;
        case 7: // se o número lido for igual a 7
            printf("Sábado\n");
            break;
        default: // se o número lido não for igual a nenhuma das constantes anteriores
            printf("Dia inválido\n");
    }
}
```

```

    }
    system("pause"); // para dar uma pausa para ver o resultado da execução
    return(0); // para finalizar o programa sem erros
}

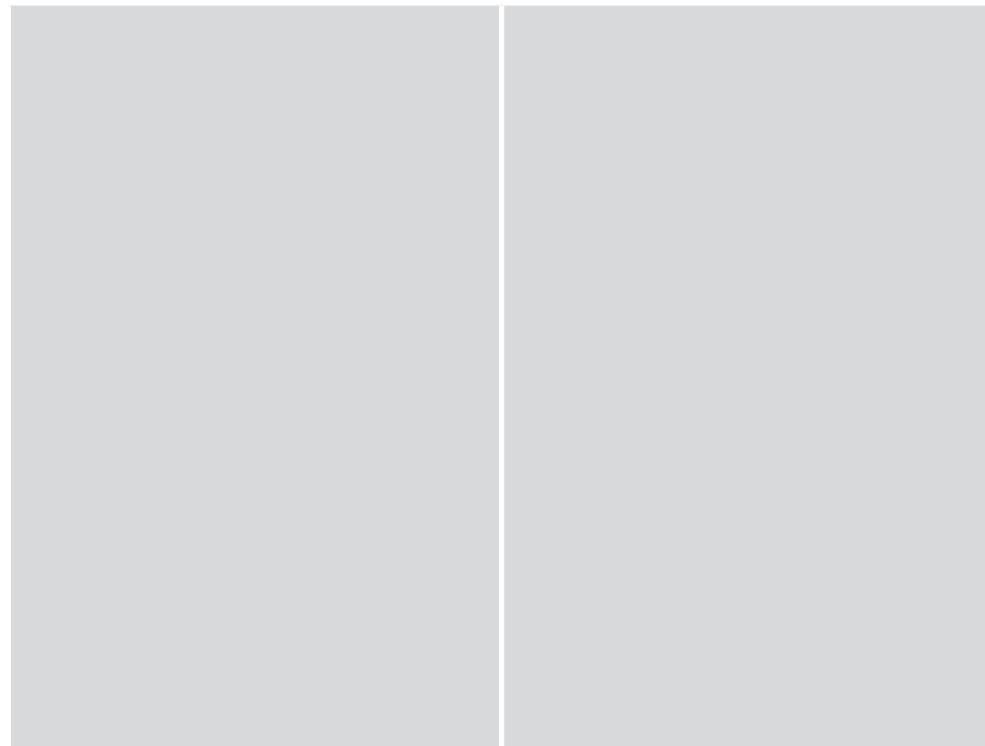
```

Podemos fazer uma analogia entre o **switch...case** e a estrutura **if-else-if** (condicional composta aninhada) apresentada anteriormente. A diferença fundamental é que a estrutura **switch...case** aceita apenas constantes numéricas, já o **if-else-if**, aceita como condição qualquer tipo de expressão. Porém, o **switch...case** permite um código mais “limpo” (mais claro) que o **if-else-if**. Veja um exemplo a seguir do mesmo problema implementado com os dois tipos de estrutura.



Atenção

O comando **break**, faz com que o **switch** seja interrompido assim que um dos blocos de comandos seja executado. Porém, 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 cuidado para não obter resultados indesejados!



Vamos dar um exemplo. No último exemplo do **switch...case** apresentado, qual seria o valor da variável **b** se a variável **a** fosse igual a **2**? Seguindo a estrutura **case**, teríamos que o **switch** testaria o valor de **a** e entraria no **case 2**. Dessa forma, a variável **b** receberia o valor **3**. Porém, e se não existissem os **breaks** dentro de cada **case** do comando, qual seria valor de **b** ao final da execução?

```

switch (a) {
    case 1:
        b=1;
    case 2:
        b=3;
    case 3:
        b=5;
    case 4:
        b=7;
    default
        b=0;
}

```

Da mesma forma que o anterior, a **variável a** seria testada e se executaria o bloco de comando do **case 2**. Porém, como não existem **breaks**, o programa não pararia por aí, sairia executando, seqüencialmente todos os outros comandos abaixo deste ponto. Dessa forma, a **variável b** terminaria com o **valor zero**.

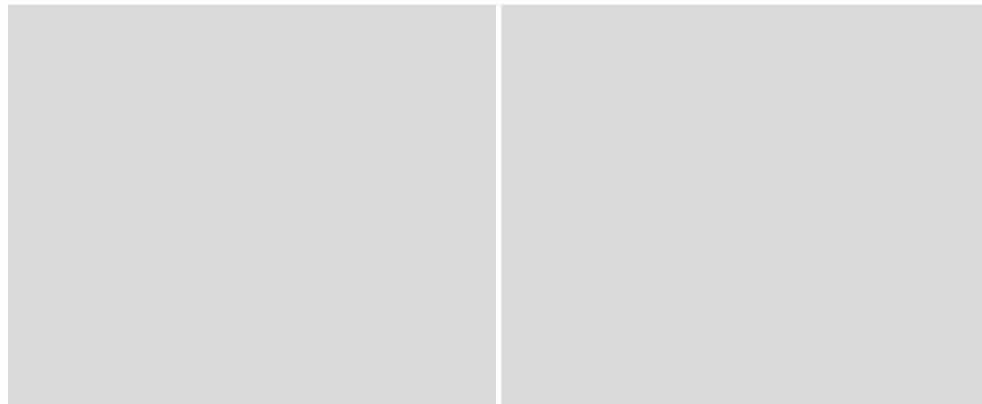


Há alguns pontos importantes que devem ser mencionados sobre o comando **switch**.

- O resultado da expressão ou variável deve ser um tipo enumerável, por exemplo, o tipo int. Também podem ser usados tipos compatíveis com int, isto é, expressões com resultados tipo char podem ser usadas. Lembre que o char armazena o código do caractere na tabela ASCII.
- Notar que caso não apareça um comando de desvio (como o break), todas as instruções seguintes ao teste **case** que teve sucesso serão executadas, mesmo as que estejam relacionadas com outros testes **case**.
- O comando switch só pode testar igualdade.
- Não podem aparecer duas constantes iguais em um **case**.
- Não esqueça dos dois pontos (:) depois do comando **default**.

5.1.5 Estrutura de Repetição Contada

Esta estrutura é utilizada, normalmente, para criar um laço que contém um conjunto de comandos que será executado um número fixo de vezes. É útil quando se conhece previamente o número de vezes que se deseja executar um determinado conjunto de comandos. Vamos dar uma olhada na sintaxe desta estrutura em linguagem algorítmica (que foi visto na seção 1.2.9) e em linguagem C.



Em C, a estrutura é composta por 3 partes separadas por ponto-e-vírgula. A primeira parte (**inicialização**) é onde a(s) variável(is) de controle do laço tem seu valor inicial definido. Pode-se inicializar mais de uma variável neste ponto, separando-as por vírgula. Se esta parte do comando for omitida (ficar em branco), as variáveis manterão os valores atribuídos anteriormente no programa.

A segunda parte (**condição**) contém um teste que será executado ao início de cada rodada no laço para checar se a repetição deve ser encerrada. Se esta condição é verdadeira, executa-se mais uma vez o bloco de comandos dentro do **for**. Se a condição for falsa, o laço é finalizado. Ou seja, de qualquer forma, o laço executa enquanto uma determinada condição for verdadeira. Esta condição é, em geral, uma expressão lógica. Por exemplo: `i <= 100` ou `cont > 0`. Se a condição for omitida, ela será sempre considerada verdadeira, assim sendo o laço é executado indefinidamente (**loop infinito**), a não ser que seja interrompido por um `break`. O comando `break` vai quebrar/parar o **loop infinito** e o programa continuará sua execução normalmente, a partir do primeiro comando após o laço **for**. O loop infinito tem a forma:

```
for (inicialização; ;incremento){  
    <bloco de comando>;
```

}

Vamos dar um exemplo a seguir de um programa que contará quantas vezes o laço será executado até que uma letra S seja digitada.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int cont;
    char letra;
    for (cont=1;;cont++) // laço for sem condição de parada – loop infinito
    {
        printf("Digite uma letra - <para sair digite S> ");
        scanf("%c", &letra); //lê a letra
        if (letra == 'S')
            break; // se foi digitado S, saí do laço, caso contrário continua no loop
    }
    printf("\nO laço executou %d vezes. \n", cont);
    system("pause");
    return(0);
}
```

Na terceira parte da estrutura (**atualização**), deve ser fornecido um comando (para cada variável de controle do laço) que atualize o valor da variável. Normalmente, é utilizado incremento ou decremento, mas pode-se utilizar qualquer expressão que modifique o valor da variável, dependendo do problema que se deseja resolver. Se se desejar modificar mais de uma variável neste ponto, deve-se separá-las por vírgulas.

Em linhas gerais, quando todas as partes do comando **for** estão presentes, a inicialização da variável de controle é feita e depois a condição é testada. Se a condição for falsa nada mais é feito e abandona-se o laço **for**. Se a condição for verdadeira é executado o bloco de comandos, é feita a atualização da variável de controle e volta-se a testar a condição. O laço fica repetindo estas operações até que a condição seja falsa. Veja um exemplo a seguir. Este programa imprime os vinte primeiros números inteiros na tela.



Atenção

O laço **for** na linguagem C é bastante flexível. Temos acesso à inicialização, à condição e ao incremento. Qualquer uma destas partes do laço **for** pode ser uma expressão qualquer do C, desde que ela seja válida. Ou, também, pode ser omitida. Vamos ilustrar a seguir alguns casos que você pode vir a encontrar que envolvem o laço **for**.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int cont;
    for (cont=1; cont<=20; cont++)
        printf ("%d ",cont);
    system("pause");
    return(0);
}
```

Observe que, no exemplo acima, usou-se o operador de incremento dentro da terceira parte (**atualização**) do laço **for**. Esta é a forma usual de se fazer o incremento (ou decremento) em um loop **for**, por ser mais compacta. Note, também, que não foram usadas chaves dentro do comando **for**. Isso se deve ao fato de só existir uma única linha de comando dentro do mesmo. E, quando só há um único comando, o uso de chaves (delimitador de bloco) é opcional.

- Uso de constantes definidas pelo usuário na condição do laço


```
for (cont = 1; cont < MAX; cont++)
```

 $\{ \dots \}$
- Uso do retorno de uma função no teste da condição. Além disso, foi usada na atualização uma atribuição aritmética **+=** (o contador será incrementado de dois em dois).


```
for (cont = 1; cont < BusqueNumeroMax(); cont+=2)
```

 $\{ \dots \}$
- Loop sem conteúdo - é aquele no qual se omite o bloco de comandos do laço **for**. Sua forma geral é (atenção ao ponto e vírgula!): **for** (inicialização;condição;atualização); Uma das aplicações desta estrutura é gerar tempos de espera, dar uma breve pausa na execução do programa, de acordo com o valor a ser contado. Veja o exemplo abaixo:

```
#include <stdio.h>
int main ()
{
    long int i;
```

```

printf("\a"); // Imprime o caracter de alerta (é um beep, um sinal sonoro)
// O laço abaixo faz o programar esperar 100.000.000 iteracoes, antes de
// continuar a execução
for (i=0; i<1000000000; i++);
    printf("\a"); // Imprime outra vez o caracter de alerta (é um beep, um sinal sonoro)
return(0);
}

```

- Laço **for** com mais de um comando tanto na **inicialização**, quanto na **atualização**.

```
for ( i=1, j = 10; i <= 10; i++, j += 10)
```

```
    printf ("i = %d, j = %d\n", i, j);
```

Como já utilizado na linguagem algorítmica, o laço **for** pode ser aninhado com outros laços **for**. Esta construção, geralmente, é mais comum quando estamos trabalhando com matrizes. Porém, ela pode aparecer em outros tipos de problemas, tal qual a criação de uma tabuada. Veja o exemplo a seguir.

```

#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int i, j;
    printf(" Imprime tabuada de multiplicacao .\n");
    // o primeiro laço pega a primeira coluna da tabuada
    for (i=1 ; i <10 ; i++)
    {
        printf (" Tabuada de %d\n", i);
        // o segundo laço, varia a segunda coluna da tabuada, gerando a mesma
        for (j=1; j <10; j++)
        {
            printf ("%d x %d = %d\n", i, j, (i * j));
        }
    }
    system("pause");
    return 0;
}

```

Para exemplificar, vamos mostrar a primeira saída do programa (execução de uma rodada só do laço mais externo):

Imprime a tabuada de multiplicacao.

Tabuada de 1

1 x 1 = 1

1 x 2 = 2

1 x 3 = 3

1 x 4 = 4

1 x 5 = 5

1 x 6 = 6

1 x 7 = 7

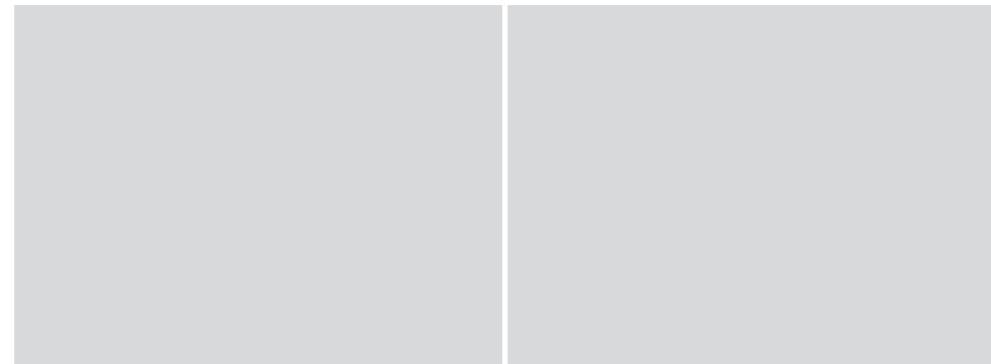
1 x 8 = 8

1 x 9 = 9

5.1.6 Estrutura de Repetição Condicional com Teste no Início

Laço que verifica antes de cada execução, se é “permitido” executar o trecho do algoritmo. O bloco de comandos é repetido enquanto uma dada condição permanecer verdadeira. Ou seja, é feito um teste no início do laço e o bloco de comandos só vai ser executado se a condição for verdadeira (ou seja, diferente de zero). Quando a condição for falsa (igual a zero), o laço não será mais executado.

Como a condição é avaliada no início de cada execução do laço, o conjunto de comandos não será executado nenhuma vez se, logo que entrar no laço, a condição for falsa. Esse tipo de estrutura costuma ser utilizada quando não sabemos de antemão quantas vezes o laço irá ser executado. Vamos dar uma olhada na sintaxe desta estrutura em linguagem algorítmica (que foi visto na seção 1.2.9) e em linguagem C.



Podemos ver que a estrutura **while** testa uma condição logo no início. Se esta for verdadeira os comandos serão executados, retornando ao teste da condição assim que todo o corpo do laço tenha sido executado. Caso a condição seja falsa a repetição é terminada sem a execução do bloco. Vamos dar alguns exemplos.

```
/* Este programa verifica se os números digitados são positivos ou negativos. É lida uma
quantidade indeterminada de números, até que seja digitado o valor zero. */

#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int num;
    // faz uma leitura fora do laço, porque precisa que num tenha algum valor
    printf("Digite um número. Se desejar, digite zero para sair .\n");
    scanf("%d", &num);
    // com o valor de num lido, faz o teste da condição para saber de entra
    // ou não no laço
    while (num != 0){
        // como a estrutura do if só tem um comando em cada parte, não usa chaves
        if (num>0)
            printf("\n O número %d é positivo\n", num);
        else
            printf("\n O número %d é negativo\n", num);
        printf("Digite um número. Se desejar, digite zero para sair .\n");
        scanf("%d", &num); // faz uma nova leitura, agora dentro do laço
    }
    system("pause");
    return 0;
}
```

```
// O programa a seguir fica preso no laço até que a tecla S seja digitada.
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    char tecla = '\0'; // inicializou o caracter com um espaço em branco
    // como tecla tem um valor inicial, é possível entrar no laço, pois a
    // condição vai poder ser testada
    while (tecla != 's'){
        printf("Aperte alguma tecla. Para parar a execução digite
a letra s \n");
        scanf("%c", &tecla); // faz uma nova leitura, agora
        //dentro do laço
    }
    system("pause");
    return 0;
}
```

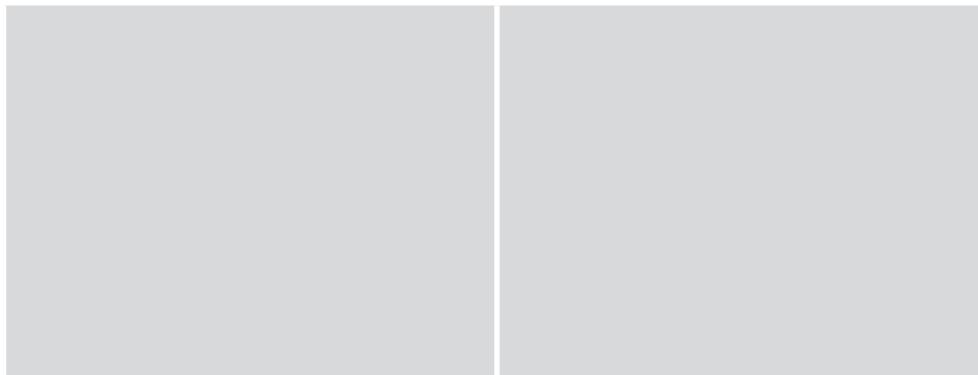


Assim como no caso do laço **for**, podemos fazer um **loop infinito** usando o laço **while**. Para tanto basta colocar uma condição eternamente verdadeira (diferente de zero) e depois usar um **break** para sair, de acordo com algo que ocorreu dentro do laço. Veja um exemplo a seguir.

```
/* O programa a seguir fica preso no laço até que a tecla S seja digitada.
Uma forma diferente de implementar, usando loop infinito */
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    char tecla;
    // 1 equivale a uma condicao sempre verdadeira
    while (1){
        printf("Aperte alguma tecla. Para parar a execução digite
a letra s \n");
        scanf("%c", &tecla); // faz uma leitura de caracter
        if (tecla == 's')
            break; //usamos o break para sair do laço,
            //quando for conveniente
    }
    system("pause");
    return 0;
}
```

5.1.7 Estrutura de Repetição Condicional com Teste no Final

Esta estrutura pode ser utilizada quando um trecho de código precisa ser executado diversas vezes, repetidamente, enquanto uma determinada condição for verdadeira. Logo, o funcionamento é bastante parecido ao da estrutura anterior **while**. Porém, os comandos contidos no interior deste laço serão executados pelo menos uma vez, já que a condição de continuidade do laço só é testada **no final** do bloco de comandos. Vamos dar uma olhada na sintaxe desta estrutura em linguagem algorítmica (que foi visto na seção 1.2.9) e em linguagem C.



Podemos observar pela análise da sintaxe do comando **do...while** que o bloco de comandos é executado uma primeira vez e, após a execução do bloco, a condição é avaliada. Se a condição for verdadeira (diferente de zero), o bloco de comandos é executado outra vez, caso contrário, a repetição é finalizada. Vamos exemplificar.

```
/* o programa a seguir fica "preso" no laço até que um número positivo seja digitado */
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int num;
    printf("Digite um número positivo: ");
    do{
        scanf("%d", &num); // lê o número
        // testa se o número é positivo
        if(num<=0)
            printf("O número digitado não é positivo! Digite um número positivo:\n");
    }
}
```

```

}while(num <= 0); // enquanto o número não for positivo, fica preso no laço
system("pause");
return(0);

```



Mesmo que a estrutura tivesse apenas um comando, é uma boa prática deixar as chaves para delimitar o bloco **do...while**. Atenção, também, que o ponto-e-vírgula no final da estrutura é obrigatório.

A grande novidade no comando **do...while** é que ele, ao contrário dos laços for e while, garante que o bloco de comandos será executado pelo menos uma vez. Um dos usos dessa estrutura é em menus, nos quais você quer garantir que o valor digitado pelo usuário seja válido. Vamos dar um exemplo.

```

#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int opcao;
    do
    {
        printf ("\n\nEscolha uma das opções a seguir pelo
número:\n");
        printf ("1 - Cadastrar Cliente\n");
        printf ("2 - Consultar Cliente\n");
        printf ("3 - Remover Cliente\n");
        scanf("%d", &opcao);
        //se o que foi digitado for diferente das opções válidas
        //fica preso no laço
    } while ((opcao<1)|| (opcao>3));

    switch (opcao) //depois de digitada uma opção válida, viria para cá
    {
        case 1:
            // executar o cadastro
            break;
        case 2:
            // executar a consulta
    }
}

```

```

break;
case 3:
    // executar a remoção
    break;
}
system("pause");
return(0);
}

```



Atenção

É interessante notar que a mesma estrutura lógica pode ser implementada usando as estruturas **for** ou **do...while**. Por exemplo, as duas estruturas abaixo são plenamente equivalentes:

<pre> // outros comandos num++; // incrementa a num }while(num <= 100); </pre>	<pre> int num; for(num = 0; num <= 100; num++){ // outros comandos } </pre>
--	---

5.1.8 Comando Break

Você já usou o comando **break** em duas ocasiões nesta unidade: interrompendo os **cases** do comando **switch** e interrompendo loops infinitos (apresentamos um exemplo com o laço **for** e outro com o laço **while**). 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 tipo de estrutura de repetição (**for**, **while** ou **do...while**). Neste último caso, o laço é abortado independente de a condição ser verdadeira ou falsa, pois o **break** interrompe-o incondicionalmente. O **break** faz com que a execução do programa continue na primeira linha seguinte ao laço ou ao bloco que está sendo interrompido. A sintaxe deste comando é simplesmente: **break;** e os exemplos de uso do mesmo podem ser vistos na explicação sobre o **switch...case** e na explicação sobre **loops infinitos**.



No caso de existirem dois laços aninhados, o **break** causará a saída APENAS do laço mais interno. Veja um exemplo a seguir.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int i, cont;
    for(i=0; i<100; ++i)
    {
        for(cont=1; cont < 100; cont++)
        {
            printf("\n %d", cont);
            if(cont==10)
                break;
        }
    }
    system("pause");
    return 0;
}
```

O código acima imprimirá os números de 1 a 10 cem vezes na tela. Toda vez que o **break** é encontrado, o laço **for** mais interno é interrompido (dessa forma, a variável **cont** NUNCA irá até o valor 100) e o controle é devolvido para o laço **for** mais externo.

Outra observação importante é que, se houver um comando **break** dentro de uma estrutura **switch...case** que está dentro de uma estrutura de repetição, o **break** afetará apenas os dados relacionados com o **switch...case** e não o laço em que o **switch...case** estiver. Por exemplo:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int i, opcao;
    for(i=0; i<100; ++i)
```

```

{
    printf("\nDigite 1 ou 2");
    scanf("%d", &opcao);
    switch (opcao) //depois de digitada uma opção válida, viria para cá
    {
        case 1:
            // executar algum comando
            break;
        case 2:
            // executa algum comando
            break;
        default
            printf("\nValor inválido!");
    }
}
system("pause");
return 0;
}

```

No programa acima, se o valor fornecido para a variável **opcao** fosse um (1), seriam executados os comandos dentro do **case 1** e, depois, por causa do **break**, a execução da estrutura **switch...case** seria encerrada. Porém, o laço **for** continuaria executando enquanto **i** fosse menor do que 100.

5.1.9 O Comando **continue**

O comando **continue** só funciona dentro de estruturas de repetição e pode ser visto como sendo o oposto do **break**. Quando o comando **continue** é encontrado, o laço pula para a próxima iteração, sem ser encerrado, ao contrário do que acontecia no comando **break**. Em outras palavras, este comando pode ser utilizado dentro de laços para ignorar os demais comandos que ainda não foram executados dentro do mesmo, abaixo do **continue**. Quando o **continue** é encontrado, a seqüência de execução do laço passa para o teste da condição do laço atual para checar se inicia ou não uma próxima iteração. Sintaxe: **continue**:

Vamos dar um exemplo para deixar as coisas mais claras. O programa a seguir recebe uma opção do usuário. Se esta opção for

inválida, o **continue** faz com que o fluxo seja desviado de volta ao início do laço. Caso a opção escolhida seja válida, o programa segue normalmente sua execução.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int opcao;
    while (opcao != 4)
    {
        printf("\n\n Escolha uma opção entre 1 e 3. Para sair digite 4. ");
        scanf("%d", &opcao);
        // se a opção digitada for inválida, ou seja, maior do que quatro ou menor do que
        // um, não continue executando, volte ao começo do laço
        if ((opcao > 4)|| (opcao <1))
            continue; // volta ao começo do laço
        // só continuaria a execução a partir daqui, se o valor da variável opcao fosse válido
        switch (opcao)
        {
            case 1:
                printf("\n Escolheu a primeira opcao..");
                break; // cada break, sai apenas da execução do switch
            case 2:
                printf("\n Escolheu a segunda opcao..");
                break;
            case 3:
                printf("\n Escolheu a terceira opcao..");
                break;
            case 4:
                printf("\n Hasta la vista baby! Encerrando o programa... ");
                break;
        }
    }
    system("pause");
    return 0;
}
```

Atenção

No caso do uso do **continue** dentro de um laço **for**, o contador seria incrementado antes da condição ser testada.

Você entendeu o uso do **continue**? Se sim, qual seria o resultado impresso com a execução do programa a seguir?

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int cont=0;
    do {
        cont++;
        if((cont > 50) && (cont < 70))
            continue;
        printf("%d ",cont);
    } while (cont < 100);
    system("pause");
    return 0;
}
```

Pense...

O resultado seria a impressão dos números de 1 a 50 e de 70 a 100. O intervalo de 51 a 69 não seria impresso, por causa do **continue**. E aí, acertou?



Vamos mostrar agora um exemplo completo de resolução de resolução de problema, para que, depois, você possa iniciar os exercícios.

Problema: Faça um programa na linguagem C que funcione como uma calculadora, com as quatro operações básicas: adição, subtração, multiplicação e divisão. Para isso, deve ser pedido ao usuário o primeiro número, o sinal da operação e o segundo número. Dê uma mensagem se o sinal digitado não for um dos quatro presentes na calculadora e lembre que não existe número divisível por zero. A calculadora deve ser executada indefinidamente, até que seja digitada a opção **s** (sair).

/* sempre no início do programa, você deve incluir as bibliotecas necessárias a execução do mesmo */

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

```

#include <stdlib.h>
int main () // o programa principal da linguagem C
{
    /* primeiro vamos fazer a declaração das variáveis necessárias ao programa.
    Declaramos dois operandos e uma variável para armazenar o resultado da operação.
    Todos são do tipo float, que é uma variável de ponto flutuante */
    float num1 , num2, resultado;
    int imprimir; // variável para indicar se o resultado deve ser impresso ou não
    char operador; // caracter para representar a operação da calculadora
    printf ("\n ***** CALCULADORA *** \n");
    do{
        printf ("\nQual a operacao que deseja realizar? ( +, -, *, / ou s para sair)\n");
        fflush(stdin); // para limpar o buffer do teclado antes de ler o caracter
        scanf("%c", &operador); // lê primeiro o operador
        if (operador != 's') // se o operador não indicar que se quer sair do programa
        {
            // parte-se para a leitura dos operandos
            printf ("Por favor entre com o primeiro operando\n");
            scanf("%f", &num1);
            printf ("Por favor entre com o segundo operando\n");
            scanf("%f", &num2);
            // a cada leitura, se suponhe que o valor do resultado deve ser impresso
            // lembre que 1 significa verdadeiro.
            imprimir = 1;
            /* depois de ler os operandos, vai ser usada uma estrutura condicional
            aninhada para selecionar qual a operação a se realizada, entre as operações
            disponíveis na calculadora. Se o operador não for igual a nenhum dos
            especificados, uma mensagem apropriada é impressa */
            if (operador == '+')
                resultado = num1 + num2 ;
            else if (operador == '-')
                resultado = num1 - num2 ;
            else if (operador == '*')
                resultado = num1 * num2 ;
            else if (operador == '/')
            {
                if (num2 == 0.0)
                {
                    printf(" Operacao Inválida! Divisão por zero! \n");
                    imprimir = 0; // não imprime resultado
                }
                else
                    resultado = num1 / num2 ;
            }
        }
    }
}

```

```

    }
else
{
    printf (" Operador inválido!\n");
    /* se o operador é inválido o valor o resultado não
       deve ser impresso */
    imprimir = 0;
}

/* imprime o resultado das operações se o valor da variável imprimir
   ainda for 1, ou seja, se o operador usado era válido. Observe que é usada
   a impressão formatada %.2f que é para imprimir um número de ponto
   flutuante com duas casas decimais */

if (imprimir == 1)
    printf ("\n%.2f %c %.2f = %.2f", num1, operador, num2 , resultado);
} // fim do teste de se é s para sair
}while(operador != 's');
system("pause"); // para pausar para o resultado ser visto
return 0; // para encerrar o programa sem erros
}

```



Conheça Mais

Para saber mais sobre a linguagem C você pode consultar qualquer livro que fale sobre a linguagem. Destaco a chamada “bíblia do C” que é o livro:

SCHILD, Herbert. **C completo e total**. Tradução Roberto Carlos Mayer. São Paulo: Makron Books, 3^a.edição, 1996.

Você também pode olhar os links indicados na unidade anterior



Atividade

Construa programa na linguagem C para resolver os problemas a seguir. Lembre das boas práticas de programação!

- 1) Faça **um** programa para ler uma série de nomes e idades de pessoas, sem usar vetor, (até que seja digitada uma idade igual

a ZERO). Depois, exiba qual o nome e a idade da pessoa mais velha, qual o nome e a idade da pessoa mais nova e qual média de todas as idades lidas.

- 2) Elaborar um programa para ler três números inteiros e, sem utilizar vetor, escrevê-los em ordem crescente.
- 3) Elaborar um programa para imprimir todas as letras (de A a Z). Utilize um comando for para percorrer todas as letras.
- 4) Faça um programa que escreva na tela todos os caracteres ASCII e seus respectivos códigos decimais e hexadecimais.
- 5) Faça um programa para imprimir todos os números ímpares que façam parte de um intervalo. O valor inicial e final do intervalo devem ser fornecidos pelo usuário (ou seja, lidos dentro do programa).
- 6) Faça um programa para imprimir os N primeiros números da série de Fibonacci: 1, 1, 2, 3, 5, 8, 13,... O valor N deve ser fornecido pelo usuário.
- 7) Um órgão de pesquisa deseja efetuar um mapeamento das condições econômicas das pessoas de uma determinada região e efetuou uma pesquisa perguntando Nome, idade, sexo, se trabalha, e qual o valor do salário que recebem por mês. Elaborar um algoritmo para determinar:
 - A quantidade de pesquisas realizadas por sexo (quanto homens e quantas mulheres foram entrevistados)
 - A quantidade pessoas menores de 18 anos que trabalham
 - O salário médio dos homens
 - O salário médio das mulheres
 - Nome e salário da pessoa que trabalha e que tem a menor renda
 - Quantidade de pessoas que não trabalham
 - A média de salário dos entrevistados
 - A média de idade dos entrevistados.

Considere:

Nome = “FIM” indica que já foram lidos todos as pesquisas, ou

seja, é a condição de parada do programa.

Sexo vai ser informado como M ou F;

- 8) Leia um número inteiro e positivo, calcule e imprima o resultado do seu fatorial. Lembre que o fatorial de um número N qualquer é igual a $1 \times 2 \times 3 \times \dots \times N - 1 \times N$. Ex: $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$. Lembre também que $0! = 1$.



Resumo

Nesta unidade você estudou como representar estruturas condicionais (ou de decisão) e estruturas de repetição (também chamadas de laços ou loops) na linguagem C. Estas estruturas já tinham sido estudadas antes quando foi apresentada a linguagem algorítmica.

As estruturas condicionais podem ser de quatro tipos:

condicionais simples:

```
if(condição)
{...}
```

condicionais compostas:

```
if(condição)
{...}
else
{...}
```

condicionais compostas aninhadas - quando colocamos estruturas condicionais, dentro de estruturas condicionais e de decisão ou escolha múltipla

switch (variável inteira ou expressão inteira)

```
{
    case valor1:
        // comandos
        break;
    ....
    case valorN:
        // comandos
        break;
```

```
default // esse bloco é executado apenas se nenhum case foi executado
// comandos
break;
}
```

Já as estruturas de repetição podem ser de dois tipos: condicionais ou contada. As condicionais podem realizar o teste no início:

```
while (condição)
{...}
```

Ou realizar o teste da condição no final:

```
do{
...
}while(condição);
```

Essas estruturas costumam ser usadas quando não se sabe quantas vezes será preciso repetir um determinado bloco de comandos. A estrutura de repetição contada é usada quando se sabe o número de repetições e é representada pelo comando for.

```
for (inicialização; condição; incremento ou atualização)
{ ... }
```

Adiconalmente você viu a explicação do comando **break**, que encerra a execução do bloco de comandos onde esteja (seja uma estrutura switch ou um laço) e do comando **continue** (que só é utilizado dentro de laços e que força o laço a passar para a sua próxima execução).

No próximo módulo, vamos estudar os tipos estruturados de dados. Eles servem para agrupar dados do mesmo tipo ou de tipos diferentes em conjuntos. Depois. veremos como são representados na linguagem C os vetores e as matrizes, além de estudar uma estrutura nova: os registros. Até lá!

5.2 Unidade 8 – Tipos Estruturados de Dados

Em muitas aplicações queremos trabalhar com conjuntos de dados que podem ou não ser semelhantes em tipo. Por exemplo, o conjunto das médias dos alunos de uma turma, o conjunto dos nomes dos jogadores de um time de futebol, o conjunto dos dados de um funcionário dentro da empresa, etc. Nestes casos, seria conveniente poder colocar estas informações dentro um mesmo conjunto, referenciado pelo mesmo nome e poder acessar cada elemento deste conjunto por meio de um número (chamado índice).

Em programação, estes conjuntos são formados pelas *estruturas de dados homogêneas* (conjuntos que todos os elementos são de um mesmo tipo) e pelas *estruturas de dados heterogêneas* (conjunto cujos dados são de tipos diferentes).

As estruturas de dados homogêneas podem ser unidimensionais (os chamados vetores ou *arrays*, em inglês), bidimensionais (as chamadas matrizes) ou multidimensionais.

Já as estruturas de dados heterogêneas serão a novidade desta unidade, porque você ainda não trabalhou com as mesmas. Elas são representadas pelos registros que ganham o nome de estruturas na linguagem C.

Complementando ainda o assunto deste capítulo, apresentaremos um tipo especial de vetor: as strings. Strings são vetores de caracteres, como já definimos antes (na unidade 6), porém vamos dar mais detalhes e apresentar algumas funções que o C tem prontas para manipular strings.

Você vai precisar de 10 horas para estudar esse assunto e exercitar o mesmo. Logo, mãos à obra!



Atenção

Você já estudou sobre estruturas de dados homogêneas na Unidade 4 (Estruturas de Dados Homogêneas), só que elas foram implementadas em linguagem algorítmica. De qualquer forma, lá você encontra mais detalhes sobre cada estrutura dessas. Se necessário, revise o assunto!

5.2.1 Tipos Estruturados Homogêneos Unidimensionais (Vetores)

As estruturas de dados homogêneas correspondem a conjuntos de elementos de um mesmo tipo, representados por um único nome. Esse tipo de estrutura é classificada como unidimensional quando necessita

de apenas um índice apara identificar cada elemento do conjunto. Por exemplo, a estrutura de dados chamada **médias**, referencia um conjunto de posições de memória capazes de armazenar números de ponto flutuante, que podem ser acessados por um índice.

medias	0	1	2	...	48	49

Cada elemento do vetor **medias** seria refenciado através de um índice. Em C, os índices começam no valor zero. Por exemplo: se desejássemos pegar a média da posição 2, usaríamos o nome do vetor e, entre colchetes, o índice desejado: **medias[2]**, daí conseguíramos pegar o valor **9.5**. Agora vamos ver como representar e acessar vetores em C.

Declaração de Vetores

Em C, um vetor é um conjunto de variáveis de um mesmo tipo que possuem um nome único que o identifica e vários índice de referência. A sintaxe para a declaração de um vetor é a seguinte: **tipo nome[tamanho];** onde:

tipo é o tipo dos elementos que serão armazenados no vetor. Por exemplo: **int, float, double, char ...**

nome é o nome identificador do vetor. As regras de nomenclatura de vetores são as mesmas usadas em variáveis (seção 3.1.9).

tamanho é o tamanho do vetor, isto é, o número de elementos que o vetor poderá armazenar.

Vamos dar alguns exemplos de declarações de vetor:

```
int idade[35]; /* declara um vetor chamado idade, onde cada elemento do vetor será do tipo int e o vetor terá capacidade para armazenar 35 valores */.
```

```
float medias[50]; /* declara um vetor chamado medias – que é o que está desenhado no começo desta seção - onde cada elemento do vetor será do tipo float e o vetor terá capacidade para armazenar 50 valores */.
```

```
char nome[60]; /* declara um vetor chamado nome onde cada elemento do vetor será do tipo char e o vetor terá capacidade para armazenar 60 caracteres. Como vimos anteriormente, um vetor de caracteres é chamado de string (mais detalhes na seção 3.3.2) */.
```

Quando declaramos um vetor, estamos reservando espaço de

memória para os elementos que serão armazenados no mesmo. A quantidade de memória (em bytes) usada para armazenar um vetor pode ser calculada como:

$$\text{quantidade de memória alocada} = \text{tamanho do tipo} * \text{tamanho do vetor}$$

Assim, nos exemplo anteriores, a quantidade de memória utilizada pelos vetores, na ordem especificada será a seguinte:

O vetor `idade` reservará 70 bytes. Isso porque os números inteiros costumam ocupar 2 bytes. Assim sendo, como o vetor tem 35 posições, temos que: $\text{qntde memória} = 2 \times 35 = 70$ bytes.

O vetor `medias` reservará 200 bytes. Isso porque os números de ponto flutuante (float) ocupam 4 bytes. Assim, sendo, como o vetor tem 50 posições, temos que: $\text{qntde memória} = 4 \times 50 = 200$ bytes.

Por fim, o vetor `nome` reservará 60 bytes. Isso porque os caracteres costumam ocupar 1 byte. Assim sendo, como o vetor tem 60 posições, temos que: $\text{qntde memória} = 1 \times 60 = 60$ bytes.

Referência aos Elementos de um Vetor

Cada elemento de um vetor é referenciado pelo nome do vetor seguido de um índice inteiro entre colchetes: `nomeVetor[índice]`. Veja alguns exemplos:

```
int pos = 7;
float vetor[10]; // declaração de vetor
vetor[3] = 5.57;
vetor[pos] = 8.6;
vetor[2.5] = 3.9; /* → DARIA ERRO! Pois 2.5 não é um índice válido! Não é um número inteiro positivo */
```

Os índices em C começam do valor 0 e não 1. Ou seja, o elemento `idade[2]` não é o segundo elemento do vetor e, sim, o terceiro. Em outras palavras, o primeiro elemento do vetor tem índice 0 e o último tem índice `tamanho-1`. O índice de um vetor deve ser um número inteiro positivo.

Atenção

Ao escrever um programa utilizando vetores, a complexidade do código fica independente do tamanho do vetor. Por exemplo, a lógica de um programa que manipula um vetor de 10 posições é a mesma da de um programa que manipula um vetor de 1000 posições. O que vai mudar é apenas a quantidade de elementos manipulados.

**Saiba Mais**

Para percorrer todas as posições de um vetor, o que você faria? Suponha um vetor **idades** de 5 posições de números inteiros. Como você preencheria cada posição do vetor? Seria assim?

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int idades[5]; // declaração de um vetor de 5 posições de números inteiros
    // para preencher o vetor idade, poderíamos fazer o seguinte.
    idades[0] = 2;
    idades[1] = 4;
    idades[2] = 6;
    idades[3] = 8;
    idades[4] = 10;
    system("pause");
    return(0);
}
```

Para esse vetor pequeno, poderia até ser. Mas e se esse vetor tivesse 100 posições? Você iria escrever os comandos 100 vezes, para os índices de 0 a 99? Claro que não!

Para percorrer todas as posições de um vetor, deve-se usar alguma estrutura de repetição. Geralmente, utiliza-se a estrutura **for**, porque sabemos o valor inicial e final dos índices do vetor. Por exemplo, para percorrer o vetor **idades** anteriormente apresentado, fazendo a mesma coisa do programa acima de maneira mais elegante, poderíamos fazer o seguinte:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int idades[5]; // declaração de um vetor de 5 posições de números inteiros
    int indice, valor;
    valor = 2; // valor inicial para preencher o vetor
```

```

for (indice=0; indice<5;indice++) // laço for para percorrer o vetor
{
    idades[indice] = valor; ←
    valor = valor * 2;
}
system("pause");
return(0);
}

```

Atenção

Todos esses exemplos fizeram uso de posições válidas do vetor. Mas suponhamos o vetor int idades[5]; O que aconteceria se você tentasse acessar uma posição que não existisse nele como, por exemplo, a posição idades[7] ?

LIMITES DO VETOR

A linguagem C não realiza verificação de limites em vetores. Logo, quando definimos um vetor **int idades[5]**, o compilador reserva o espaço de memória equivalente a 5 variáveis inteiras, ou seja, 10 bytes. Quando tentamos acessar um elemento que ultrapassa o limite do vetor (como o exemplo dado **idades[7]**), não ocorre nenhum erro de compilação! Porém, estaremos acessando uma região de memória que não pertence ao vetor. Logo, poderemos obter resultados inesperados. Isso porque, pode ser que esta posição esteja sendo usada por uma outra variável do programa ou contenha informação inútil (lixo de memória). Observe que acessar uma área de memória fora do espaço destinado a um vetor pode destruir informações reservadas de outras variáveis.

Assim sendo, na linguagem C, devemos ter cuidado com os limites de um vetor. Porque, embora na sua declaração, tenhamos definido o tamanho de um vetor, o C não faz nenhum teste de verificação de acesso a um elemento dentro do vetor ou não, logo, a responsabilidade de fazer esse controle é do próprio programador.

Atenção

Se você não tiver atenção com os limites de validade para os índices do vetor, você corre o risco de ter variáveis sobreescritas ou de ver o computador travar. O ruim é que estes erros são difíceis de detectar, podem passar despercebidos, uma vez que o compilador não gera nenhuma mensagem de erro. A solução mais adequada é sempre avaliar os limites de um vetor antes de manipulá-lo, já que é tarefa do programador fazer com que os índices dos vetores estejam sempre dentro dos limites estabelecidos na declaração do mesmo.

Inicialização de vetores

Assim como podemos inicializar variáveis (por exemplo: `int pos = 5;`), podemos inicializar vetores. A sintaxe para a inicialização dos elementos de um vetor é a seguinte:

`tipo nome[tamanho] = {lista de valores válidos};`

onde:

lista de valores válidos é uma lista, separada por vírgulas, de valores para serem armazenados em cada um dos índices do vetor. Os valores devem ser do **tipo** especificado para o vetor. Veja exemplos a seguir.

```
int numdiasMes[12]={31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};  
float notas[4] = {9.0, 7.5, 4.2, 8.5};  
char vogais[5] = {'a', 'e', 'i', 'o', 'u'};
```

Se houver menos valores que o tamanho especificado para o vetor, os elementos restantes serão preenchidos com zero. Se houver mais valores que o tamanho especificado para o vetor, o compilador pode acusar um erro ou, pelo menos, dar uma advertência (warning) sobre este ocorrido.

No caso de inicializar o vetor com valores, o **tamanho** do mesmo pode ser omitido na declaração. Pois, se nenhum **tamanho** for fornecido para inicializar o vetor, o compilador contará a quantidade de itens da “lista de valores válidos” e o fixará como o **tamanho** do vetor. Por exemplo:

`int nums[] = {10, 20, 30, 40, 50, 60};`

O vetor `nums` seria criado com o tamanho 6, porque existem seis elementos na lista de valores válidos.

Atenção

Em C, não há como inicializar um elemento no meio de um vetor, sem inicializar todos os elementos anteriores ao mesmo tempo.

Definindo o Tamanho do Vetor com Constantes

Para facilitar e deixar mais fácil de modificar o código de um programa que manipula vetores, é possível especificar o tamanho do vetor em uma constante simbólica.

Declaramos uma constante simbólica com a diretiva `#define` no cabeçalho do programa (isso foi visto na seção 3.1.10, na Unidade 6)

e, depois, usamos essa constante para definir o tamanho do vetor, na declaração do mesmo. E qual a vantagem disso?

Usando a definição do tamanho do vetor em uma constante simbólica, podemos alterar o número de elementos do vetor mais facilmente, bastando para isso apenas alterar o valor da constante e recompilar o programa. Além disso, esta constante passa a ser usada nas referências ao vetor, por exemplo, na condição de parada do laço **for**. Veja um exemplo a seguir.

```
#define TAM 10 //definição da constante TAM com valor 10
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int idades[TAM]; // declaração de um vetor com o tamanho TAM
    int indice, valor;
    valor = 2; // valor inicial para preencher o vetor
    // a constante TAM também vai ser usada na condição de parada do laço for
    for (indice=0; indice<TAM;indice++)
    {
        idades[indice] = valor;
        valor = valor * 2;
    }
    //laço para imprimir os elementos do vetor
    for (indice=0; indice<TAM;indice++)
    {
        printf("\n %d", idades[indice]);
    }
    system("pause");
    return(0);
}
```

Veja que, agora, para alterar o tamanho do vetor e a forma de percorrê-lo, só seria necessário alterar o valor da constante na primeira linha de código (por exemplo, **#define TAM 500**) e, depois, recompilar o programa. Prático, não?

Lendo Valores para os Elementos de um Vetor

Para fazer a leitura de valores do teclado e armazenar diretamente dentro do vetor, usam-se as funções de entrada de dados

convencionais, tais como o `scanf`. Lembrando de usar, no caso do `scanf`, o operador de endereço & antes do nome da referência a cada elemento do vetor. Veja um exemplo a seguir.

```
#define TAM 10 //definição da constante TAM com valor 10
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int indice;
    int nums[TAM];
    for (indice=0; indice<TAM;indice++)
    {
        printf("\nDigite o valor do elemento %d: ", indice);
        → scanf("%d", &nums[indice]);
    }
    for (indice=0; indice< TAM;indice++)
    {
        printf("\n o valor é %d", nums[indice]);
    }
    system("pause");
    return(0);
}
```

Na seção a seguir detalharemos um tipo especial de vetor: o vetor de caracteres, por ele ser tratado como um novo tipo, chamado **string** e possuir funções especiais que auxiliam na sua manipulação.

5.2.2 Manipulação de Strings

O vetor de caracteres, também conhecido como **string**, é um dos tipos de dados mais importantes da linguagem C. Isto porque ele é usado para manipular texto, palavras, nomes e frases. A sintaxe para a declaração de uma string é:

```
char nomeString [tamanho];
```

Como, por exemplo:

```
char nome[60];
```

Esta declaração cria o vetor de caracteres chamado **nome**, que

também pode ser interpretado como a variável do tipo **string** chamada **nome**, permitindo um comprimento de até 50 caracteres.

Vale lembrar que o último caractere de uma string deve ser o caractere nulo ('\0'), ou seja, no vetor acima declarado, teríamos 59 caracteres para trabalhar livremente, porque um deles, obrigatoriamente seria o caractere nulo. Sabendo disso, a declaração a seguir, seria válida?

```
char nome[5] = "Paulo";
```

Não, não seria, porque não haveria espaço reservado no vetor suficiente para as 6 variáveis ('P', 'a', 'u', 'l', 'o' e a terminação do caractere nulo '\0'). Assim sendo, devemos sempre lembrar que o tamanho da string deve incluir o '\0' final. Logo, para armazenar uma cadeia de 10 caracteres deve-se reservar um vetor de 11 caracteres, para poder acomodar o caracter nulo.

Em C, as constantes do tipo **string** são definidas como uma lista de caracteres entre aspas duplas. Como por exemplo, "Ana Maria", "Sandra", "Igor", "A", "123", etc.

Atenção

Como uma string é um vetor de caracteres, cada elemento da mesma pode ser acessado individualmente, como em qualquer vetor, através do uso de um índice entre colchetes. Por exemplo, se tivéssemos a declaração: `char nome[10] = "Sandra";` poderíamos acessar `nome[3]`, que obteríamos o carácter 'd' (lembre que todo vetor começa da posição zero).

Funções Pré-Definidas para manipulação de Strings

A biblioteca `<string.h>` fornece um conjunto de funções para manipulação de strings. É muito útil fazer uso dessas funções, principalmente considerando algumas limitações das strings por serem um vetor:

- Strings não podem ser atribuídas a outras strings com o operador de atribuição (=), embora possam ser inicializadas com constantes do tipo string (vindas entre aspas duplas).
- Strings não podem ser comparadas com os operadores relacionais (==, !=, >, >=, <, <=). Isso, porque a string é um vetor e a comparação entre vetores, deve ser feita elemento a elemento. Por isso, para comparar strings, vamos fazer uso de uma função adequada.

Na sequência será apresentado um resumo do conjunto de funções disponibilizados pela biblioteca `<string.h>`.

A função `strcpy()` - copia a string_origem para a string_destino.
Sua sintaxe é:

```
strcpy (string_destino, string_origem);
```

A string_destino deve ter espaço suficiente para armazenar a string_origem. É essa a função que deve ser empregada para a atribuição de strings, no lugar do operador normal de atribuição (=). Valor de retorno: a função retorna o endereço do primeiro caracter da string_destino. Vejamos um exemplo.

```
#include <stdio.h>
#include <string.h> // Tem de lembrar de incluir essa biblioteca
#include <stdlib.h>
int main ()
{
    char nome1[50], nome2[50]; // Declara duas strings de tamanho 50
    printf ("\nDigite uma string:");
    gets (nome1); // Lê uma string
    strcpy (nome2, nome1); // Copia a string nome1 para a string nome2
    printf ("\n As duas strings são %s e %s", nome1, nome2);
    strcpy (nome1,"Copiando uma constante string"); // Copia a frase para nome1
    printf ("\n A string nome1 depois de modificada é: %s \n", nome1);
    system("pause");
    return(0);
}
```

No código acima, você pode observar que é possível copiar uma string para outra e, também, uma constante string (frase entre aspas duplas), para uma variável string.

A função strcpy() - copia no máximo n caracteres de uma string_origem para uma string_destino. Não coloca o '\0' no final da string_destino, a não ser que tenha conseguido copiar a string_origem completa. Sua sintaxe é:

strcpy (string_destino, string_origem, n);

onde:

string_destino: String que irá receber o conteúdo

string_origem: String cujo conteúdo será copiado.

n: Número máximo de caracteres a serem copiados. Deve ser um valor inteiro.

Valor de retorno: a função retorna o endereço do primeiro caracter da string_destino.

Por exemplo, suponha duas variáveis do tipo string chamadas nome1 e nome2. Suponha, também, que nome1 possui o valor “Sandra Siebra”. Agora se fosse usado o comando **strncpy(nome2, nome1, 6)**; O que seria copiado para nome2? Apenas o valor “Sandra”, que são os primeiros 6 caracteres da string origem nome1.

A função strcat() - concatena string_origem ao final de string_destino. Em outras palavras, essa função copia o conteúdo da string_origem para o final da string_destino. O primeiro caracter da string_origem substitui o caracter nulo ('\0') da string_destino. A string de origem permanecerá inalterada. Sua sintaxe é:

```
strcat (string_destino, string_origem);
```

onde:

string_destino: String que irá receber, no seu final, o conteúdo da string_origem. Ela deve ter tamanho suficiente para armazenar o conteúdo atual mais o conteúdo da string_origem.

string_origem: String cujo conteúdo será acrescentado ao final da outra. Permanecerá inalterada, após a execução da função.

Valor de retorno: a função retorna o endereço do primeiro caracter da string destino. Vejamos um exemplo.

```
#include <stdio.h>
#include <string.h> // Tem de lembrar de incluir essa biblioteca
#include <stdlib.h>
int main ()
{
    char nome1[30], nome2[50]; // Declara duas strings
    printf ("\nDigite uma string: ");
    gets (nome2); // Lê uma string
    strcpy (nome1,"O valor lido é: "); // Copia a frase para nome1
    strcat (nome1, nome2); // Copia a string nome2 para o final da string nome1
    printf("\n %s \n", nome1);
    system("pause");
    return(0);
}
```

A função strncat() - copia os n primeiros caracteres da string_origem para o final da string_destino. O primeiro caracter da string_origem substitui o caracter nulo ('\0') da string_destino. A string de origem permanecerá inalterada. Sua sintaxe é:

strncat (string_destino, string_origem, n);

onde:

string_destino: String que irá receber, no seu final, o conteúdo da string_origem. Ela deve ter tamanho suficiente para armazenar o conteúdo atual mais o conteúdo da string_origem.

string_origem: String cujo conteúdo será acrescentado ao final da outra. Permanecerá inalterada, após a execução da função.

n: Número máximo de caracteres a serem copiados. Deve ser um valor inteiro.

Valor de retorno: a função retorna o endereço do primeiro caracter da string destino.

Por exemplo, suponha duas variáveis do tipo string chamadas nome1 e nome2. Suponha, também, que nome1 possui o valor “Sandra”. E nome2 possui o valor “Siebra”. Se fosse usado o comando **strncat(nome1, nome2, 2)**; Qual seria o valor final de nome1? Seria “SandraSi”, que seria o resultado da concatenação dos 2 primeiros caracteres da string nome2, no final da string nome1.

A função strcmp() - Compara lexicograficamente duas strings. Se as duas strings forem idênticas, a função retorna zero. Se elas forem diferentes, a função retorna um valor diferente de zero. Esta função deve ser utilizada em substituição aos operadores relacionais no caso de comparação entre strings. Sua sintaxe é:

strcmp (string1, string2);

onde:

string1 e string2: as duas strings a serem comparadas.

Valor de retorno: a função retorna

- um valor menor que zero se a string1 for menor que a string2;
- zero se as strings forem iguais;
- um valor maior que zero se a string1 for maior que a string2.

Quando se trata de comparação de strings, maior e menor não se refere ao tamanho, mas à posição em ordem alfabética das letras que compõem a string. A ordem alfabética é baseada na tabela ASCII. Por exemplo, ‘C’ é maior do que ‘B’, que, por sua vez, é maior do que ‘A’. A palavra “ana” é menor do que a palavra “ano”. A palavra “arroz” é

Atenção

Cuidado ao comparar letras maiúsculas com letras minúsculas! Na tabela ASCII, as letras maiúsculas possuem um valor menor que as letras minúsculas. Por exemplo, o caractere ‘Z’ é menor do que o caractere ‘a’ (se olhar o valor numérico de Z e de a na tabela ASCII você observará isso). Outro exemplo é a palavra “ana” que é maior do que a palavra “Ano”. Tome cuidado, também, porque espaços e outros caracteres especiais são considerados na comparação e, devido a isso, algumas vezes, a comparação não fornece o resultado que se espera de uma ordenação alfabética.

maior do que a palavra “anel”.

Vejamos um exemplo.

```
#include <stdio.h>
#include <string.h> // Tem de lembrar de incluir essa biblioteca
#include <stdlib.h>
int main ()
{
    char nome1[30], nome2[30]; // Declara duas strings de tamanho 30
    printf ("\nDigite uma string: ");
    gets (nome1); // Lê a string nome1
    printf ("\nDigite outra string: ");
    gets (nome2); // Lê a string nome2
    // Compara as duas strings
    if (strcmp(nome1, nome2) > 0) // se o resultado for maior do que zero
        printf ("\n\nA primeira string é maior do que a segunda\n");
    else if (strcmp(nome1, nome2) == 0)
        printf ("\n\nAs duas strings são iguais.\n");
    else
        printf ("\n\nA primeira string é menor do que a segunda.\n");
    system("pause");
    return(0);
}
```

Implemente o programa acima e teste com algumas palavras, para poder entender melhor o que acabamos de explicar.

A função strncmp() – Faz a mesma coisa do comando anterior, porém, só compara lexicograficamente os n primeiros caracteres das duas strings. Sua sintaxe é:

strncmp (string1, string2);

onde:

string1 e string2: as duas strings a serem comparadas.

n: quantidade de caracteres a serem comparados.

Valor de retorno: o mesmo da função strcmp (explicado anteriormente).

Exemplo, suponha duas variáveis do tipo string chamadas nome1 e nome2. Suponha, também, que nome1 possui o valor “anel”. E nome2

possui o valor “anta”. Se fosse usado o comando **strcmp(nome1, nome2, 2)**; Qual seria o resultado da comparação? Ela daria que as strings eram iguais. Porque se estaria comparando apenas os 2 primeiros caracteres das duas strings, que, realmente, são iguais.

A função strlen() - retorna o comprimento da string fornecida. O terminador nulo ('\0') não é contado. Isto quer dizer que, de fato, o comprimento da string armazenada deve ser um a mais que o inteiro retornado por strlen(). Cuidado! Não confunda o tamanho total do vetor de caracteres com o tamanho atual da string. Esta função retorna quantas posições do vetor estão realmente preenchidas e não o tamanho total do vetor. Sua sintaxe é:

strlen(string);

onde: string: a string que terá seu tamnho calculado.

Valor de retorno: retorna o número de caracteres da string. Vamos dar um exemplo:

```
#include <stdio.h>
#include <string.h> // Tem de lembrar de incluir essa biblioteca
#include <stdlib.h>
int main ()
{
    char nome[30]; // Declara duas strings de tamanho 30
    int tam;
    printf ("\nDigite uma string: ");
    gets (nome); // Lê a string nome
    tam = strlen(nome); // Pega o tamanho da string nome e copia na variável tam
    printf ("\n\nA string digitada %s tem tamanho %d.\n", nome, tam);
    system("pause");
    return(0);
}
```

Atenção

Espaços em branco fazem parte da string e, por isso, são contabilizados tal qual letras e algarismos.

A função atoi() - converte uma string para um valor numérico inteiro. Essa função é usada quando se deseja converter para valores numéricos, números que foram lidos como string. Por exemplo, “12”. Como foi lido como string, não seria possível aplicar operadores aritméticos a esse número. Porém, uma vez convertido para um valor numérico 12, seria possível usá-lo, normalmente, como qualquer outro valor numérico.

Sua sintaxe é:

atoi(string);

onde: string: a string a ser convertida.

Valor de retorno: retorna um número inteiro correspondente ao valor armazenado na string.

A função atof() - converte uma string para um valor numérico de ponto flutuante. Tal qual a função anterior, é usado para converter números lidos no formato de string para valores numéricos (só que, agora, de ponto flutuante). Sua sintaxe é:

atof(string);

onde: string: a string a ser convertida.

Valor de retorno: retorna um valor double correspondente ao conteúdo armazenado na string.

Atenção

Recomendamos fortemente que você execute os códigos fornecidos como exemplo de cada função de manipulação de strings, para que possa assimilar melhor o que cada uma delas faz.

5.2.3 Tipos Estruturados Homogêneos Bidimensionais (Matrizes)

A estrutura de dados homogênea bidimensional, também conhecida como matriz (vide Figura 1), necessita de dois índices (um para a linha e outro para a coluna) para localização de um elemento dentro do conjunto. Existem ainda estruturas multidimensionais, onde é utilizado um índice para localizar o elemento em CADA uma das dimensões da estrutura, conforme visto na linguagem algorítmica (Unidade 4).

M	0	1	2
0	m_{00}	m_{01}	m_{02}
1	m_{10}	m_{11}	m_{12}
2	m_{20}	m_{21}	m_{22}

Figura 1 - Exemplo de valores das posições de uma matriz

Declaração de Matrizes

Em C, uma matriz é um conjunto de variáveis de um mesmo tipo que possuem um nome único e cada elemento dela é identificado por dois índices (o de linha e o de coluna). A sintaxe para a declaração de uma matriz é a seguinte:

```
tipo nome[No. linhas][No. Colunas];
```

onde:

tipo é o tipo dos elementos que serão armazenados na matriz. Por exemplo: int.

nome é o nome identificador da matriz. As regras de nomenclatura de matrizes são as mesmas usadas em variáveis (seção 3.1.9).

Nº. Linhas é a quantidade de linhas que a matriz terá.

Nº. Colunas é a quantidade de colunas que a matriz terá.

Vamos dar alguns exemplos de declaração de matrizes:

```
int M[3][3]; /* declara uma matriz quadrada chamada M, de 3 linhas por 3 colunas, ou seja, com capacidade para armazenar 9 valores inteiros. Essa é a declaração da matriz ilustrada na Figura 1 */.
```

```
float numeros[30][3]; /* declara uma matriz chamada numeros de 30 linhas por 3 colunas. Ou seja, em cada linha haverá espaço para 3 números de ponto flutuante. */.
```

Poderíamos interpretar as matrizes como vetores de vetores. Ou seja, uma matriz é um vetor em que seus elementos são vetores. Como assim? Veja, por exemplo, a declaração:

int conjunto[5][6]; ela poderia ser interpretada como a matriz **conjunto** equivale a 5 vetores, com cada um deles tendo até seis elementos do tipo inteiro.

Referência aos Elementos de Uma Matriz

Cada elemento de uma matriz é referenciado pelo nome da matriz seguido de um índice que representa a linha e outro que representa a coluna, entre colchetes:

```
nomeMatriz[índice linha][índice coluna].
```

Veja alguns exemplos:

```
float notas[10][2]; // declaração da matriz
```

Aqui é declarada uma matriz de 10 linhas e 2 colunas. Agora, observe que, como os índices em C começam no valor 0, os índices de linha da matriz variarão de **0** a 9 e os índices das colunas, variarão de 0 a 1. Os índices devem ser números inteiros positivos.

notas[3][1] = 5.57;

O comando acima vai gravar o valor 5.57 na terceira linha (linha 2), primeira coluna (coluna 0). Isso porque os índices em C começam do valor 0 e não 1. É importante ressaltar que o índice da esquerda indexa as linhas e o da direita indexa as colunas.

Para preencher ou percorrer todas as posições de uma matriz, devem-se usar dois laços **for** aninhados. O laço mais externo irá percorrer as linhas da matriz e o laço mais interno irá percorrer as colunas de cada linha. Dessa forma, o laço mais interno vai variar mais rapidamente do que o do laço mais externo. Por exemplo, para preencher a matriz notas anteriormente declarada, poderíamos fazer o seguinte:

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    float notas[10][2]; // declaração da matriz
    int linha, coluna;
    for (linha=0; linha < 10; linha++) // laço for para percorrer as linhas
    {
        for (coluna=0; coluna < 2; coluna++) // laço for para percorrer as colunas
        {
            // vai colocar 0.0 em todas as posições da matriz
            notas[linha][coluna] = 0.0; ←
            printf(" %.2f", notas[linha][coluna]);
        }
        printf("\n"); //pula uma linha, cada vez que acaba de imprimir as colunas
    }
    system("pause");
    return(0);
}
```

Atenção

Tal qual na manipulação de vetores, é tarefa do programador manter os índices da matriz dentro da faixa estipulada, pois o C não vai fazer nenhum controle sobre isso.

Observe que o laço **for** mais externo anda da linha 0 a linha 9 (já que a condição de parada é “enquanto linha < 10”). A cada linha do

laço mais externo são percorridas todas as colunas, usando o laço for mais interno, que vai variar da coluna 0 a coluna 1 (já que a condição de parada é “enquanto coluna < 2”). Execute o programa acima para poder entender melhor.

Matrizes de strings

Como sabemos, uma string é um vetor de caracteres. Dessa forma, uma matriz de strings equivale a termos um vetor de vetores de caracteres. Simplificando, no fundo, uma matriz de strings é equivalente a uma matriz bidimensional de caracteres ou a um vetor de strings. Dessa forma, podemos encarar a declaração de uma matriz de strings como sendo:

```
char nomeMatriz[quantidade_de_strings][comprimento_das_strings];
```

onde:

nomeMatriz é o nome da variável do tipo matriz;

quantidade_de_strings é a quantidade de strings que será armazenada na matriz;

comprimento_das_strings é o tamanho das strings em termos da quantidade de caracteres que cada uma delas terá. Todas as strings da matriz deverão ter, no máximo, esse comprimento.

Por exemplo:

```
char nomes[5][45];
```

Essa declaração cria uma matriz de strings chamada nomes. Isso equivale a dizer que a matriz terá 5 strings de, no máximo, 45 caracteres de comprimento. Em outras palavras, 5 vetores de no máximo 45 caracteres.



E, como acessar uma string individual dentro de uma matriz?

Fácil. É só usar apenas o primeiro índice da matriz. Ou seja, para acessar uma determinada string faça: **nomeMatriz[índice]**; Vamos dar um exemplo: considere a declaração anteriormente feita da matriz nomes. Para acessar a segunda string gravada nesta matriz, você

poderia fazer: **nomes[1]**. Com isso, você pegaria a string gravada na segunda linha da matriz (lembre que os índices começam do zero!) e essa string teria, no máximo, 45 caracteres de comprimento.

Vamos dar um exemplo mais completo: o programa a seguir vai ler 3 strings de, no máximo, tamanho 50 e, depois, imprimir as strings lidas na tela.

```
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    char nomes[3][50]; // declaração da matriz de strings
    int indice;

    /* faz apenas um laço for para percorrer as linhas da matriz, que vão ser equivalentes
    a ir pegando string por string da matriz */

    for (indice=0; indice < 3; indice++)
    {
        printf ("\n\n Digite um nome: ");
        gets(nomes[indice]); //lê string por string e vai colocando dentro da matriz
    }
    printf ("\n\n Os nomes que você digitou foram:\n");
    for (indice=0; indice < 3; indice++)
    {
        printf ("%s\n",nomes[indice]); //imprime as strings lidas
    }
    system("pause");
    return(0);
}
```

Inicialização de Matrizes

Da mesma forma que os vetores podem ser inicializados, as matrizes também podem. A sintaxe para a inicialização dos elementos de uma matriz é a seguinte:

tipo nome[No. Linhas][No. Colunas] = {lista de valores válidos};

onde:

lista de valores válidos é uma lista composta por valores (do mesmo tipo da matriz) entre chaves e separados por vírgula. Os

valores devem ser dados na ordem em que serão colocados na matriz. Vejamos alguns exemplos a seguir.

```
int numeros[3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
```

Este primeiro exemplo demonstra a inicialização de uma matriz de números inteiros chamada `numeros`. Considerando as dimensões da matriz a inicialização seria feita da seguinte forma: 1, 2, 3 e 4 ficaria na primeira linha da matriz; 5, 6, 7 e 8 na segunda linha e 9, 10, 11 e 12 na terceira e última linha. Ou seja, três linhas, cada uma delas com quatro elementos.

Caso a matriz seja de caracteres, como mencionado anteriormente, é equivalente a ter um vetor de strings. Dessa forma, a inicialização poderia se dar da seguinte forma:

```
char nomes[5][15] = {"Ana", "Igor", "Maria da Luz", "Alice", "Jonas"};
```

Neste caso, a matriz seria inicializada da seguinte forma:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
0														
1														
2														
3														
4														

Às vezes, para deixar a inicialização mais clara, levando em conta que a matriz pode ser considerada um vetor de vetores, pode-se separar os grupos de informações de cada linha da matriz entre chaves e separar esses grupos de informações por vírgula. Veja dois exemplos a seguir:

```
int numeros[4][5] = { { 1, 2, 3, 4, 5}, { 6, 7, 8, 9, 10}, {11, 12, 13, 14, 15}, {16, 17, 18, 19, 20} };
```

Uma matriz chamada `numeros` que equivale a ter 4 vetores com 5 números inteiros em cada um.

```
float notas[3][3] = {{8.5, 6.0, 7.5}, {6.0, 2.5, 5.0}, {10.0, 8.0, 9.5}};
```

Uma matriz chamada `notas` que equivale a ter 3 vetores com 3 números de ponto flutuante em cada um.

Podemos, também, inicializar matrizes das quais não foi especificado a quantidade de linhas. O compilador C, neste caso,

vai verificar o tamanho da lista de valores válidos, o No. de colunas definido e, assim, vai deduzir o tamanho da dimensão que falta da matriz. Isto ocorre na hora da compilação e não poderá mais ser mudado durante o programa. Por exemplo:

```
int matriz [][]2 = {1, 2, 3, 4, 5, 6, 7, 8, 9,10};
```

Neste exemplo, como se tem 10 números e eles devem ser colocados em duas colunas por linha, o compilador deduzirá que o valor não especificado para as linhas deverá ser 5. Para que a matriz contenha 5 linhas com dois números em cada (atendendo a quantidade total de números da lista de valores válidos).

Definindo o Tamanho da Matriz com Constantes

Para facilitar e deixar mais fácil de modificar o código de um programa que manipula matrizes, é possível especificar as dimensões da mesma através de constantes simbólicas, tal qual foi feito com os vetores. Veja um exemplo a seguir.

```
#define NLINHAS 5 // constante para definir o número de linhas da matriz
#define NCOLUNAS 10 // constante para definir o número de colunas da matriz
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int numeros[NLINHAS][NCOLUNAS]; // declaração da matriz
    int linha, coluna;
    // as constantes também vão ser usadas na condição de parada do laço for
    for (linha=0; linha < NLINHAS; linha++) // percorre as linhas
    {
        for (coluna=0; coluna < NCOLUNAS; coluna++) // percorre as colunas
        {
            // vai colocar zero em todas as posições da matriz
            numeros[linha][coluna] = 0;
            printf(" %d", numeros[linha][coluna]);
        }
        printf("\n"); //pula uma linha, cada vez que acaba de imprimir as colunas
    }
    system("pause");
    return(0);
}
```

Atenção

No caso das matrizes, você não pode deixar de especificar todos os tamanhos (o No. de linhas e de colunas), senão o compilador não conseguirá definir o tamanho apropriado para a matriz, devido à variedade de opções. Dessa forma, sempre deve ser especificado, ao menos, o número de colunas da matriz.

Veja que, agora, para alterar as dimensões da matriz e a forma de percorrê-la, seria necessário apenas trocar o valor das constantes que estão nas primeiras linhas do código e, depois, recompilar o programa.

Lendo Valores para os Elementos de uma Matriz

Para fazer a leitura de valores do teclado e armazenar diretamente dentro da matriz, usam-se as funções de entrada de dados apropriadas, tais como o scanf ou gets (no caso de leitura de strings). Lembre-se de usar, no caso do scanf, o operador de endereço & antes do nome da referência a cada elemento da matriz. Quando falamos de “matrizes de strings” já mostramos um exemplo de leitura usando o comando gets. Agora, vamos dar um exemplo usando o scanf.

```
#define NLINHAS 5 // constante para definir o número de linhas da matriz
#define NCOLUMNAS 10 // constante para definir o número de colunas da matriz
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int numeros[NLINHAS][NCOLUMNAS]; // declaração da matriz
    int linha, coluna;
    // laços para fazer a leitura dos elementos da matriz
    for (linha=0; linha < NLINHAS; linha++) // percorre as linhas
    {
        for (coluna=0; coluna < NCOLUMNAS; coluna++) // percorre as colunas
        {
            printf("\nDigite o elemento da linha %d, coluna %d: ", linha, coluna);
            → scanf("%d", &numeros[linha][coluna]);
        }
    }
    // laços para imprimir os elementos da matriz
    for (linha=0; linha < NLINHAS; linha++) // percorre as linhas
    {
        for (coluna=0; coluna < NCOLUMNAS; coluna++) // percorre as colunas
        {
            printf(" %d", numeros[linha][coluna]);
        }
        printf("\n"); //pula uma linha, cada vez que acaba de imprimir as colunas
    }
}
```

```

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

```

5.2.4 Tipos Estruturados Homogêneos Multidimensionais

Agora, só para ilustrar, a criação de estruturas homogêneas multidimensionais, vamos apresentar como criá-las em C. A sintaxe para criação dessas estruturas seria:

Tipo nomeEstrutura [tam_dim1][tam_dim2] ... [tam_dimN];

Onde:

[tam_dim1][tam_dim2] ... [tam_dimN] seriam os tamanhos de cada dimensão da estrutura multidimensional.

Por exemplo: **int números[2][3][4];** seria a declaração de uma estrutura tridimensional.

Em termos de manipulação, uma estrutura multidimensional funciona, basicamente, como as matrizes. Porém, deve-se usar um índice para cada dimensão da estrutura.

Para percorrer a estrutura, cada dimensão teria um laço **for** exclusivo. Por isso, ao se adicionar uma nova dimensão, deve-se criar uma nova variável contadora e um novo laço **for** para manipulá-la. Ou seja, é criado um laço **for** para cada dimensão a ser percorrida e a referência ao elemento da estrutura é feita pela quantidade de índices correspondente a quantidade de dimensões.

Outra coisa é que, assim como os vetores e matrizes, as estruturas multidimensionais podem ter os tamanhos das dimensões especificados através de constantes simbólicas. Vejamos um exemplo a seguir, fazendo uso de uma estrutura tridimensional.

```

#define DIM1 3 // constante para definir o tamanho da primeira dimensão
#define DIM2 5 // constante para definir o tamanho da segunda dimensão
#define DIM3 2 // constante para definir o tamanho da terceira dimensão
#include <stdio.h>
#include <stdlib.h>

```

```

int main ()
{
    int numeros[DIM1][DIM2][DIM3]; // declaração da estrutura tridimensional
    int d1, d2, d3; //um contador para cada dimensão
    // as constantes também vão ser usadas na condição de parada do laço for
    for (d1=0; d1 < DIM1; d1++) // percorre a dimensão 1
    {
        for (d2=0; d2 < DIM2; d2++) // percorre a dimensão 2
        {
            for (d3=0; d3 < DIM3; d3++) // percorre a dimensão 3
            {
                // vai colocar zero em todas as posições da matriz
                numeros[d1][d2][d3] = 0;
                printf(" %d", numeros[d1][d2][d3]);
            }
            printf("\n");
        }
        printf("\n");
    }
    system("pause");
    return(0);
}

```

Para inicializar diretamente uma estrutura multidimensional, a forma geral é:

```
tipo nomeEstrutura[tam_dim1][tam_dim2]...[tam_dimN] = {lista_valores_válidos};
```

Onde:

lista_valores_válidos é uma lista composta por valores (do mesmo tipo da estrutura) entre chaves e separados por vírgula.

Vamos dar um exemplo. Para deixar mais claro o exemplo, agrupamos os valores com chaves, de acordo com a quantidade de elementos de cada dimensão:

```

int tabela[2][3][2] =
{
    {{10,15}, {20,25}, {30,35}},
    {{40,45}, {50,55}, {60,65}}
};

```

Observando o exemplo acima, temos que **tabela** é uma estrutura tridimensional, composta por 2 vetores, de 3 subvetores, com 2 elementos em cada um deles.

No caso de estruturas multidimensionais, é necessário especificar todas as dimensões, podendo deixar sem especificação apenas a primeira dimensão, para que o compilador possa deduzir o tamanho da mesma e reservar memória de maneira adequada. Porém, não aconselhamos deixar nenhuma dimensão sem especificação.

5.2.5 Tipos Estruturados Heterogêneos (Registros ou Estruturas)

Até agora, tanto em linguagem algorítmica, quanto em C, só trabalhamos com estruturas homogêneas, ou seja, aquelas em que cada posição estava armazenado um mesmo tipo de dado. Porém, é muito útil em programação poder armazenar na mesma estrutura, tipos de dados diferentes. Por exemplo, digamos que você queira representar a ficha de um funcionário de uma empresa. Essa ficha poderia conter dados do funcionário como: a matrícula (número inteiro), nome (tipo string), salário (número de ponto flutuante), entre outras informações, que não poderiam ser representadas com estruturas homogêneas. Por isso, precisamos das estruturas heterogêneas.

Em várias linguagens, as estruturas heterogêneas costumam ser chamadas de **registros**. Porém, em C, elas são chamadas simplesmente de **estruturas (struct)**.

Uma estrutura é um conjunto de variáveis agrupadas sob um identificador único, sendo que estas variáveis podem ser de tipos de dados diferentes. A estrutura serve para organizar, agrupar, de forma lógica, algum dado cujo valor é composto por mais de uma variável, como o exemplo da ficha do funcionário, anteriormente, citado. O agrupamento das informações facilita a manipulação dos dados armazenados nestas estruturas.

Uma estrutura pode incluir outras estruturas além de variáveis simples. Considere que, normalmente, os elementos que compõem a estrutura devem ter alguma relação semântica entre si. Como por exemplo, no caso da ficha do funcionário, todas as variáveis se referem a dados do mesmo funcionário.

Declaração

Em C, as estruturas são definidas com a palavra reservada **struct**, normalmente, no início do programa (antes do programa principal **main**). Também é possível definir as estruturas em um arquivo separado (arquivo com extensão .h), que será incluído (usando a diretiva **#include**) no programa que necessitar usar as mesmas. A sintaxe da declaração é:

```
struct identificadorEstrutura {
    tipo nomeElemento1;
    tipo nomeElemento 2;
    ...
    tipo nomeElementoN;
};
```

Onde:

identificadorEstrutura - deve ser um nome único que identificará o tipo estrutura como um todo. Ele será utilizado posteriormente para definição de variáveis que devem ter o formato da estrutura definida.

Entre o par de chaves ({}), deve-se declarar todas as variáveis que irão compor a estrutura. Essas variáveis são chamadas elementos ou membros da estrutura. A declaração desses elementos segue as regras para declaração de variáveis normais em C (**tipo nomeElemento;**), exceto que não podem ser inicializados. Pode-se, inclusive, declarar vetores ou outras estruturas previamente definidas como elementos da estrutura sendo declarada. O tamanho total de uma estrutura, em bytes, é igual à soma do tamanho de todos os seus elementos.

Para ilustrar, vamos dar um exemplo de declaração de estrutura para criação da ficha do funcionário que mencionamos anteriormente, supondo que gostaríamos de armazenar as seguintes informações sobre o funcionário: matrícula, nome, departamento onde está alocado e salário.

```
struct ficha {
    int matricula;
    char nome [45];
    char depto [35];
    float salario;
```

```
};
```

Onde:

A palavra chave **struct** inicia a declaração da estrutura.

Ficha é o nome do identificador da estrutura.

Entre as chaves fica a declaração dos elementos que vão compor a estrutura. Para cada elemento deve ser especificado o tipo do mesmo, por exemplo int e o nome do mesmo, por exemplo matricula.

Estruturas podem conter outras estruturas como elementos. Por exemplo, vamos definir uma estrutura chamada endereço.

```
struct est_endereco{
    char rua[50];
    int numero;
    char compl[10];
    char bairro[20];
    char cep[10];
};
```

Agora vamos modificar a estrutura ficha, anteriormente definida, para que ela inclua o endereço do funcionário.

```
struct ficha {
    int matricula;
    char nome [45];
    char depto [35];
    float salario;
    struct est_endereco endereco;
};
```

Fazendo um esquema gráfico, para ver se você entende melhor, a estrutura ficha, especifica um tipo que terá o seguinte formato:

matricula	nome	depto	salario	endereço				

Ou seja, a estrutura ficha, tem um campo chamado endereço que

Atenção

Para usar uma estrutura X, na definição de outra estrutura Y, a estrutura X tem de ter sido definida antes da estrutura Y, uma vez que Y vai usá-la. Por exemplo, no caso acima, para definir a estrutura ficha, uma vez que ela faz uso da estrutura est_endereco, a estrutura est_endereco teria, obrigatoriamente, de ter sido definida antes da estrutura ficha.

não é um tipo simples, mas sim outra estrutura que tem seus próprios elementos (conforme a definição da estrutura `est_endereco`).



Saiba Mais

Uma vez que eu já declarei a estrutura, eu já posso fazer uso da mesma? Não, não pode ainda. Isso porque a definição da estrutura, sozinha, não é o suficiente para que ela possa ser utilizada.

A declaração da estrutura é apenas um modelo de como estruturas deste tipo devem ser construídas. Por exemplo, a estrutura `ficha`, é apenas o modelo dos dados que uma ficha deve ter. Porém, ainda não foi alocada na memória, nenhum espaço para armazenar valores, já que nenhuma variável do tipo da estrutura definida, foi criada. Para isto, é necessária a declaração de variáveis (quantas forem necessárias) para armazenamento dos valores. Veremos como fazer isso a seguir.



Atenção

Observando que a definição de uma struct é como se fosse a declaração de um novo tipo, a declaração das variáveis do tipo da estrutura, é equivalente a declaração das variáveis normais de um programa em C (ex: int num1, num2);.

Declarando Variáveis do Tipo da Estrutura Definida

Para a declaração de variáveis do tipo da estrutura definida, usa-se a seguinte sintaxe:

```
struct identificadorEstrutura nomeVar1, nomeVar2, ... , nomeVarN;
```

Por exemplo, para definirmos variáveis do tipo da estrutura `ficha` anteriormente definida, faríamos: `struct ficha func1, func2;`

Nesta declaração duas estruturas do tipo `ficha` foram criadas. Esta declaração vai alocar espaço para armazenar os dados de dois funcionários.

Para agilizar o processo de declaração das variáveis do tipo da estrutura, pode-se fazer a declaração delas, logo após a criação da estrutura, antes do ponto e vírgula final. Veja um exemplo a seguir.

```
struct ficha {
    int matricula;
    char nome [45];
```

```

char depto [35];
float salario;
struct est_endereco endereco;
}func1, func2;

```

Atenção

O que é usado dentro do programa é a variável do tipo estrutura e não o nome do identificador da estrutura. Por exemplo, nos exemplos anteriores, o que seria usado no programa seriam as variáveis func1 ou func2 e não o identificador do tipo estrutura ficha (porque este é o nome do tipo e não de uma variável).

Referenciando Elementos em uma Estrutura

Para referenciar um elemento da estrutura usa-se o nome da variável do tipo da estrutura, seguido por um ponto (.) e, depois, o nome do elemento interno da estrutura que se deseja referenciar. Veja alguns exemplos de referência a elementos da estrutura.

```
func1.matricula = 12345;
```

O comando acima armazenaria na matricula da variável estrutura de nome func1, o valor 12345.

```
gets(func1.nome);
```

O comando acima leria uma string e a armazenaria na variável da estrutura func1 chamada nome.

**Saiba Mais**

E agora, se eu quisesse atribuir o valor “Av. Caxangá” ao nome da rua onde mora o funcionário func1, como eu faria? Uma vez que é uma estrutura, dentro de outra estrutura?

func1

matricula	nome	depto	salario	endereco				

Quando você referencia a variável **func1**, você está se referindo a estrutura toda (todos os elementos que a compõem). Se você quer pegar só o campo **rua**, você primeiro tem de acessar o elemento chamado **endereco** dentro da estrutura **func1** (ou seja, **func1.endereco**). Porém o elemento **endereco** não é um tipo simples. Ele é composto de vários elementos também (rua, numero, compl, bairro

Atenção

Também é possível atribuir o conteúdo de uma estrutura completa a outra estrutura do mesmo tipo, diretamente, sem ser necessário atribuir elemento por elemento. Para isto, é só usar o operador de atribuição convencional. Por exemplo, uma vez que a variável func1 estivesse preenchida, poderíamos fazer: func2 = func1;. Dessa forma as duas variáveis iriam referenciar a mesma estrutura na memória, ou seja, iriam ter os mesmos dados em cada um dos seus campos.

e cep). Daí como você quer apenas o elemento **rua**, você acessa apenas ele fazendo o seguinte: **func1.endereco.rua**

Observe que cada elemento da estrutura que se deseja acessar vai ser separado por um ponto, até chegar ao elemento desejado, que tem de ser um tipo simples ou uma string. O operador ponto (.) associa os elementos da esquerda para a direita, ou seja, do mais externo (que é o nome da variável) para o mais interno. Assim sendo, para atribuir o valor “Av. Caxangá” ficaríamos com o seguinte: **strcpy(func1.endereco.rua, “Av. Caxangá”);**

Agora, vamos dar um exemplo completo, usando tudo que foi definido sobre estruturas.

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

// declaração das estruturas
struct e_endereco{
    char rua[50];
    int num;
    char bairro[30];
};

struct aluno{
    int matricula;
    char nome[50];
    struct e_endereco endereco; // aluno tem um campo do tipo e_endereco
} alu1, alu2; //declaração de duas variáveis do tipo da estrutura

// programa principal
int main ()
{
    //vai fazer a leitura de dados para todos os elementos da estrutura alu1
    printf("\nDigite a matrícula do aluno: ");
    scanf("%d", &alu1.matricula);
    fflush(stdin); // para limpar o buffer do teclado
    printf("\nDigite o nome do aluno: ");
    gets(alu1.nome);
    printf("\nDigite o nome da rua onde o aluno mora: ");
    gets(alu1.endereco.rua); // lê o dado de uma estrutura interna a outra
    fflush(stdin);
    printf("\nDigite o número da casa ou do prédio onde o aluno mora: ");
    scanf("%d", &alu1.endereco.num);
    fflush(stdin);
```

```

printf("\nDigite o nome do bairro onde o aluno mora: ");
gets(alu1.endereco.bairro);
// vai copiar a estrutura da variável alu1, para a variável alu2
alu2 = alu1;
// vai imprimir os dados de alu2, para comprovar a cópia
printf("\n\nO aluno %s tem matrícula %d", alu2.nome, alu2.matricula);
printf("\nEle mora na rua %s, No. %d", alu2.endereco.rua, alu2.endereco.num);
printf("\nBairro: %s \n", alu2.endereco.bairro);
system("pause");
return(0);
}

```

5.2.6 O Comando **typedef**

O comando **typedef** permite ao programador definir um novo nome (apelido) para um tipo existente ou definido pelo usuário. Isso pode auxiliar na documentação do código, permitindo nomes mais descriptivos aos tipos de dados padrões. Sua sintaxe é:

```
typedef nome_original novo_nome;
```

onde:

nome_original é o nome de qualquer tipo de dado existente no C ou que já tenha sido declarado pelo usuário.

novo_nome é o novo nome ou apelido dado para o **antigo_nome**.

É importante ressaltar que o novo nome definido é uma opção, não uma substituição ao nome original do tipo existente. Sendo assim, você poderá usar qualquer um deles: o apelido definido ou o nome original do tipo.

Vamos exemplificar. Suponha que você quer criar um novo nome para o tipo **int**, para ficar mais fácil de lembrar o que ele representa. Logo, você poderia fazer:

```
typedef int inteiro;
```

Agora, no código do programa, podemos usar para declarar uma variável ou o nome do tipo original (**int**) ou o apelido definido (**inteiro**). Por exemplo:

```
inteiro numero;
```

Uma das aplicações mais úteis para o **typedef** é simplificar a declaração de variáveis do tipo de uma estrutura definida. Por exemplo, vejamos a estrutura produto.

```
struct produto{
    int codigo;
    char descricao[50];
    float preco;
};
```

Para declarar uma variável do tipo produto, deve-se usar uma declaração tal como:

```
struct produto prod;
```

Observe que essa declaração exige o uso de dois identificadores: a palavra reservada **struct** e o nome da estrutura **produto**. Para simplificar isso, o comando **typedef** pode ser aplicado à declaração da estrutura aluno. Veja o exemplo a seguir:

```
typedef struct produto{
    int codigo;
    char descricao[50];
    float preco;
} tproduto ;
```

Observe que um apelido foi dado a **struct produto**. Ele agora pode ser chamado de **tproduto**. Dessa forma, agora podemos declarar variáveis deste tipo estrutura usando apenas o novo apelido:

```
Tproduto prod;
```

Observe que, através do uso do comando **typedef** é criado um apelido para o tipo estrutura, que é composto por um único nome (**tproduto**). Assim sendo, não é mais necessário usar a palavra chave **struct** para declarar uma variável do tipo estrutura.

Vetores de Estruturas

Muitas vezes, uma única estrutura não é suficiente para resolver um problema. Veja o caso da ficha do funcionário. Uma empresa não vai ter apenas uma única ficha de funcionário, mas uma ficha para

cada um dos funcionários da empresa. Dessa forma, é necessário um conjunto de fichas. Ou seja, um vetor de estruturas.

Vetores onde os elementos são estruturas podem ser declarados da mesma forma que os outros tipos de vetores já estudados.

struct nomeEstrutura nomeVetor[tamanho];

onde:

nomeEstrutura – é o nome que identifica a estrutura criada;

nomeVetor – é o nome do vetor de estruturas;

tamanho – é o tamanho que vai ter o vetor de estruturas, ou seja, quantas estruturas vão poder ser criadas.

Por exemplo,

```
struct produto{  
    int codigo;  
    char descricao;  
    float preco;  
};
```

```
struct produto estoque[30];
```

No exemplo acima, está sendo criado um vetor de 30 posições (do índice 0 ao índice 29), onde cada posição vai conter uma estrutura do tipo produto, ou seja, uma estrutura com código, descrição e preço, conforme ilustrado na Figura 2.

Figura 2 - Vetor de Estrutura chamado estoque

Para acessar uma posição do vetor de estruturas, utiliza-se o nome do vetor, o índice do mesmo que se pretende acessar e depois, separados por pontos, o nome do campo da estrutura desejado.

Por exemplo, para atribuir um valor ao código do produto da segunda posição do vetor estoque, anteriormente definido, usariamos:
estoque[1].codigo = 11;



Atenção

Lembre que vetores em C começam com o índice zero.

Para se imprimir a descrição do primeiro produto do vetor estoque, usaríamos: **printf("%s\n", estoque[0].descricao);**



Aprenda Praticando

Agora vamos dar alguns exemplos de programas para lhe ajudar na resolução das atividades propostas.

Problema 1: SOBRE VETORES - Faça um programa em C para ler a idade de 20 pessoas e, depois, imprimir apenas as idades que estiverem acima da média das idades lidas. A partir da leitura do enunciado temos:

ENTRADA: as idades de 20 pessoas

PROCESSAMENTO: calcular a média de idades

SAÍDA: as idades que estiverem acima da média.

```
#define TAM 20 //definição da constante TAM com valor 20, porque são 20 pessoas
```

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

```
#include <stdlib.h>
```

```
int main ()
```

```
{
```

```
    int idades[TAM]; // declaração de um vetor com o tamanho TAM
```

```
    int indice; // variável para controlar o laço for
```

```
    float media; // variável para acumular a média das idades
```

```
    media = 0; // zera a variável, para poder acumular as idades, para calcular media
```

```
    // primeiro fazemos um laço para ler as idades e jogar dentro do vetor idades
```

```
    // a constante TAM vai ser usada na condição de parada do laço for
```

```
    for (indice=0; indice<TAM; indice++) {
```

```
        printf("\nDigite uma idade: ");
```

```
        // lê a idade e joga dentro do vetor idades,
```

```
        // na posição indice
```

```
        scanf("%d", &idades[indice]);
```

```
        // acumula na variável média, a idade lida
```

```
        media = media + idades[indice];
```

```
}
```

```
    /*quando sai do laço, calcula a média das idades, dividindo a soma das idades
acumulada na variável média, pela quantidade de pessoas, que está na constante TAM */
```

```
    media = media/TAM;
```

```

printf("\nA média de idades é %.2f", media); //imprime a media de idades
// percorre denovo o vetor para imprimir as idades que estão acima da média calculada
for (indice=0; indice<TAM; indice++) {
    if (idades[indice] > media)
        printf("\n %d", idades[indice]);
}
system("pause");
return(0);
}

```

Problema 2: SOBRE MATRIZES - Faça um programa em C para ler uma matriz 4 x 4 contendo números inteiros. Depois imprima a soma de cada linha da matriz e, ao final, a soma de todos os elementos da matriz. A partir da leitura do enunciado temos:

ENTRADA: matriz 4 x 4 de números inteiros

PROCESSAMENTO: calcular a soma de cada linha da matriz e a soma de todos os elementos da matriz

SAÍDA: o valor da soma de cada linha e o valor total da soma de todos os elementos da matriz.

Logo, vamos começar o algoritmo criando a matriz. Vamos precisar de duas variáveis, uma para acumular a soma das linhas e outra para acumular a soma total dos valores. Precisaremos também de dois laços, um para preencher a matriz e outro para calcular a soma das linhas e a soma total.

```

#define NLINHAS 4 // constante para definir o número de linhas da matriz
#define NCOLUMNAS 4 // constante para definir o número de colunas da matriz
#include <stdio.h>
#include <stdlib.h>
int main ()
{
    int numeros[NLINHAS][NCOLUMNAS]; // declaração da matriz
    int linha, coluna, somaLinha, somaTotal;
    // laços para fazer a leitura dos elementos da matriz
    for (linha=0; linha < NLINHAS; linha++) // percorre as linhas
    {
        for (coluna=0; coluna < NCOLUMNAS; coluna++) // percorre as colunas
        {
            printf("\nDigite o elemento da linha %d, coluna %d: ", linha, coluna);

```

```

scanf("%d", &numeros[linha][coluna]);
}

somaTotal = 0; //zera a variável que vai fazer a soma dos elementos da matriz
// laço para realizar as devidas somas
for (linha=0; linha < NLINHAS; linha++) // percorre as linhas
{
    // a cada nova linha a variável que acumula a soma das linhas deve ser zerada
    somaLinha = 0;
    for (coluna=0; coluna < NCOLUMNAS; coluna++) // percorre as colunas
    {
        // soma cada um dos elementos da mesma linha da matriz
        //veja que aqui só se varia o valor da coluna
        somaLinha = somaLinha + numeros[linha][coluna];
    }
    // quando sai do laço que anda nas colunas, terminou a soma de uma linha toda.
    // Por isso, imprimimos o resultado da soma dessa linha e acumulamos na
    // variável somaTotal o valor da soma dessa linha
    printf("\nA soma da linha %d, é igual a %d", linha, somaLinha);
    somaTotal = somaTotal + somaLinha;
}
// quando sair de todos os laços, imprime o resultado da somaTotal
printf("\n\nA soma de todos os valores da matriz é %d. ", somaTotal);
system("pause");
return(0);
}

```

Problema 3: REGISTROS - Faça um programa em C para ler várias fichas de DVD contendo o código, o nome, o gênero e o tempo de duração em minutos, até que seja digitado um código igual a ZERO. Considere que o gênero do DVD vai ser representado pela siglas ACO = ação, ROM = romance, COM = comédia, POL = policial, FAR = faroeste, FIC = ficção científica, MUS = musical e DOC = documentário. Depois, faça um programa para consultar os dvds de um determinado gênero fornecido pelo usuário, imprimindo o nome e o tempo de duração em minutos dos DVDs encontrados. Considere que o estoque da loja tem, no máximo 100 DVDs. A partir da leitura do enunciado temos:

ENTRADA: várias fichas de DVDS (até que seja digitado um código zero)

PROCESSAMENTO: consultar os dvds de um determinado gênero

SAÍDA: o nome e o tempo de duração dos DVDs que sejam do gênero consultado.

```
#define TAM 100 // constante para definir a quantidade máxima de DVDs para criar o vetor
#include <stdio.h>
#include <stdlib.h>
#include <string.h> // para poder usar a função de comparação de strings
// declaração da estrutura_DVD, usamos o typedef para dar um apelido à estrutura
typedef struct estrutura_DVD{
    int codigo;
    char titulo[60];
    char genero[20];
    int duracao;
}tDVD;
// programa principal
int main ()
{
    //declaração do vetor de estruturas de tamanho TAM
    tDVD acervo[TAM];
    char generoDesejado[20];
    int indice; // o índice para caminhar no vetor
    int qtde; // contador para contar a quantidade de dvds lidos
    int encontrados; //contador para a quantidade de dvds
    indice = 0; //o indice começa da posição zero do vetor
    qtde = 0;
    encontrados = 0;
    //vai fazer a leitura dos dados dos DVDs em um laço infinito
    while(1){
        printf("\nDigite o código do DVD: ");
        scanf("%d", &acervo[indice].codigo);
        if (acervo[indice].codigo != 0) // se não foi digitado zero, lê os dados
        {
            fflush(stdin); // para limpar o buffer do teclado
            printf("\nDigite o título do DVD: ");
            gets(acervo[indice].titulo);
            fflush(stdin);
            printf("\nDigite o gênero do DVD, sendo: ACO = ação, ROM = romance,
COM = comédia, POL = policial, FAR = faroeste, FIC = ficção científica,
```

```

MUS = musical e DOC = documentário ");
gets(acervo[indice].genero);
fflush(stdin);
printf("\nDigite a duração em minutos do filme: ");
scanf("%d", &acervo[indice].duracao);
fflush(stdin);
qtde = qtde + 1; // incrementa a quantidade de dvds lidos
//depois de ler todos os dados, passa para o próximo indice do vetor
indice = indice + 1;
}

else //se foi digitado zero, sai do laço, usando o break
break;
} // fim do laço while
// agora vai consultar os DVDs pelo gênero
fflush(stdin);
printf("\n\n Digite o gênero de DVDs que gostaria de consultar");
gets(generoDesejado);
// usa a variável quantidade no laço, para facilitar
for(indice=0; indice < qtde; indice++)
{
    // usa a função de comparação de strings para checar o genero
    if(strcmp(acervo[indice].genero, generoDesejado) == 0) // se são iguais
    {
        printf("\nDVD: %s, Duração: %d min \n", acervo[indice].titulo,
        acervo[indice].duracao);
        encontrados = encontrados + 1; // contabiliza os dvds encontrados
    }
}
if(encontrados == 0)
printf("\n\n Infelizmente, não há nenhum DVD do gênero desejado");
system("pause");
return(0);
}

```



Para saber mais sobre a linguagem C você pode consultar qualquer livro que fale sobre a linguagem. Destaco a chamada “bíblia do C” que é o livro:

SCHILD'T, Herbert. **C completo e total**. Tradução Roberto Carlos Mayer. São Paulo: Makron Books, 3^a.edição, 1996.

Você também pode olhar os links indicados na unidade 6.



Atividade de Estudo

Construa programa na linguagem C para resolver os problemas a seguir. Lembre das boas práticas de programação!

- 1) Leia um vetor chamado NUMEROS de 10 números inteiros e exiba qual o maior e o menor número desse vetor.
- 2) Estenda a questão anterior para, a partir do vetor NUMEROS completamente preenchido, criar os vetores VP (que será um vetor dos números pares que fazem parte do vetor NUMEROS) e VI (que será um vetor dos números ímpares que fazem parte do vetor NUMEROS).
- 3) Leia duas matrizes A e B (5 X 5) de números inteiros e, a partir delas, gere a matriz SOMA (correspondente a soma das duas matrizes: A + B) e a matriz SUBTRACAO (correspondente a subtração entre as duas matrizes: A - B).
- 4) Leia uma matriz N (2X3) de números inteiros e positivos e gere uma matriz C, do mesmo tamanho, de caracteres, onde cada posição C[linha, coluna] terá o valor “P” ou “I” dependendo de se o número da posição correspondente em N[linha, coluna] for PAR ou IMPAR, respectivamente.
- 5) Faça um programa que leia quatro palavras pelo teclado e armazene cada palavra em uma string. Depois, concatene todas as strings lidas em uma única string. Por fim, imprima a string concatenada na tela.
- 6) Escreva um programa que leia uma frase de até 80 caracteres do teclado e depois imprima quantos caracteres foram lidos.
- 7) Escreva um programa que leia uma frase do teclado e, em seguida, um caracter. O programa deve procurar este caracter na frase lida e, depois, imprimir quantas vezes esse caracter apareceu na frase. **Obs:** nesta questão NÃO USE funções da biblioteca de strings do C. Ache outra alternativa!

- 8) Escreva um programa que leia uma linha do teclado e imprima todas as vogais encontradas no texto e o total de vezes que elas aparecem que cada uma delas aparece. Obs: Tamanho máximo da linha deve ser 40 caracteres.
- 9) Escreva um programa que leia uma matriz 3x3 de caracteres. Essa matriz deve conter apenas caracteres 0 ou X. Depois, informe as posições (linha, coluna) que contêm somente um dos dois caracteres. O caractere a ser procurado deve ser lido do teclado.
- 10) Escreva um programa que leia um conjunto de nomes para uma matriz de strings e, depois, imprima apenas os nomes que começam com um caractere fornecido pelo usuário. Assuma que os nomes serão lidos somente em letras maiúsculas. Assuma, também, que os nomes terão, no máximo, 50 caracteres e que ao todo, serão lidos 10 nomes.
- 11) Considere que uma empresa precisa armazenar os seguintes dados de um cliente:
 - Ano de nascimento;
 - Nome completo com no máximo 50 caracteres;
 - Renda mensal do cliente;
 - Se possui ou não carro.Devem ser lidos esses dados de clientes até que o ano de nascimento seja 0. No máximo, serão lidos 50 clientes. Faça uso de estruturas para resolver esse problema. Depois de ler todos os dados, informe:
 - quantas pessoas são maiores de idade (2008 – ano de nascimento para saber a idade), ou seja, têm mais de 18 anos.
 - o nome das pessoas que tem renda mensal superior a 1000 reais
 - a renda mensal média de todos os clientes.
 - Quantos clientes possuem carro
 - Quantos clientes nasceram entre 1960 (inclusive) e 1980 (exclusive).
- 12) Fazer um programa em C para armazenar em um vetor de

estruturas, os dados dos funcionários de uma empresa (nome, endereço, salário). Onde endereço deve conter (o bairro e o cep). O programa deve ler os dados dos funcionários e imprimi-los. Em seguida, o programa deve aumentar todos os salários em 10% e imprimir uma lista com os nomes dos funcionários e seus novos salários. Considere que existem apenas 10 funcionários.



Desafio

Escreva um programa em C para manipular o nome e o telefone de várias pessoas e gravar essas informações em um vetor de estruturas. O programa deve apresentar um menu de opções com:

- 0 – Adicionar uma pessoa
- 1 – Listar todas as pessoas
- 2 – Consultar uma pessoa pelo nome
- 4 – Sair

Esse vetor de estrutura deve ter apenas 10 elementos e deve fornecer uma mensagem de erro, caso o usuário tente adicionar mais pessoas que o tamanho máximo permitido.



Resumo

Esta unidade tratou da declaração, inicialização e manipulação de estruturas de dados homogêneas e heterogêneas.

As estruturas de dados homogêneas possuem o mesmo tipo de dados em todas as posições e são classificadas, de acordo com a quantidade de índices usados para acessar os elementos da mesma, em: unidimensionais (um único índice), bidimensionais (dois índices) ou multidimensionais (três ou mais índices).

As estruturas unidimensionais são também chamadas de vetores. Em C, para declarar uma estrutura como essa você especifica: **tipo nomeVetor[tamanho]**; Esses vetores são normalmente percorridos usando uma estrutura de repetição `for`. E o tamanho do vetor pode

ser definido usando uma constante simbólica, definida pela diretiva **#define**.

Um cuidado especial que devemos ter é o controle dos índices dos vetores, matrizes e estruturas multidimensionais, porque o C não faz esse controle, ele fica a cargo do programador. Ou seja, se um vetor é definido com 10 posições, não vai ser dado nenhum erro de compilação se a posição 11 for acessada. Porém, os resultados podem ser imprevisíveis, porque essa posição, não faz parte do vetor e pode estar sendo usada por outra variável.

Um tipo especial de vetor é o vetor de caracteres, que é também chamado de string. Devemos ficar atentos para o fato de que as strings têm como marcação do seu último caractere, o caractere nulo '\0'. As strings em C possuem várias funções pré-definidas que pertencem a biblioteca `<strings.h>` que podem ser usadas para manipulá-las.

Também foram estudadas nesta unidade as estruturas de dados heterogêneas, representadas em C pelas estruturas (struct). Elas servem para agrupar em um nome único, tipos de dados diferentes, tais como os que compõem uma ficha de funcionário (matrícula, nome, idade, salário). As estruturas são bastante úteis e, geralmente, usadas dentro de vetores (vetores de estruturas).

Na próxima unidade, vamos estudar nosso último assunto: as funções! Funções são as unidades de modularização de um programa! Depois desse assunto, você estará apto a implementar o projeto da disciplina. Logo, vamos lá! Até mais!

Introdução à Programação

Sandra de Albuquerque Siebra



FASCÍCULO 6



Universidade Federal Rural de Pernambuco

Reitor: Prof. Valmar Corrêa de Andrade

Vice-Reitor: Prof. Reginaldo Barros

Pró-Reitor de Administração: Prof. Francisco Fernando Ramos Carvalho

Pró-Reitor de Extensão: Prof. Paulo Donizeti Siepierski

Pró-Reitor de Pesquisa e Pós-Graduação: Prof. Fernando José Freire

Pró-Reitor de Planejamento: Prof. Rinaldo Luiz Caraciolo Ferreira

Pró-Reitora de Ensino de Graduação: Profª. Maria José de Sena

Coordenação de Ensino a Distância: Profª Marizete Silva Santos

Produção Gráfica e Editorial

Capa e Editoração: Allyson Vila Nova, Rafael Lira e Alesanco Azevedo

Revisão Ortográfica: Ivanda Martins

Ilustrações: Diego Almeida

Coordenação de Produção: Marizete Silva Santos



Sumário

Fascículo 6 - Linguagem de Programação C	6
Objetivos do Fascículo 6	6
Conteúdo Programático do Fascículo 6	6
 6.1 Unidade 9 – Modularização.....	8
6.1.1 Funções	8
Definição de funções	9
Alguns Exemplos de Funções	12
Localização da Especificação da Função	14
Hierarquia de Funções	17
Ativando uma Função.....	17
6.1.2 Escopo de Variáveis	20
6.1.3 Passagem de Parâmetros	22
Passagem de Parâmetros por Valor	22
Passagem de Parâmetros por Referência	23
Vetores e Matrizes como Argumentos de Funções	28
6.1.4 Recursividade	31
 6.2 Unidade 10 – Desenvolvimento de Projetos	38
6.2.1 Ponteiros	38

6.2.2 Etapas para Desenvolvimento de um Programa.....	45
6.2.3 Orientações Gerais para Desenvolvimento de Projetos	49
6.2.4 Funções que podem ser Úteis.....	56
6.2.5 Refinamentos Sucessivos	60
6.2.6 Erros Comuns na Implementação de Programas	61

Fascículo 6 - Linguagem de Programação C

Carga horária do Módulo: 45 horas

Objetivos do Fascículo 6

- Apresentar os componentes básicos de um programa utilizando uma linguagem de programação estruturada
- Apresentar uma ferramenta para compilação de programas em uma linguagem estruturada
- Desenvolver a habilidade de programação fazendo uso de variáveis, operadores, expressões, estruturas seqüenciais, estruturas de repetição, tipos estruturados de dados (homogêneos e heterogêneos) e modularização de programas
- Incentivar a prática de desenvolvimento de programas de forma a aprimorar no aluno a sua lógica de resolução de problemas

Conteúdo Programático do Fascículo 6

Unidade 6 – Noções Básicas da Linguagem C (10 horas)

- Ambiente de Programação C
- Comentários, palavras reservadas e identificadores
- Constantes e Variáveis
- Tipos de dados
- Declaração de Variáveis
- Operadores e Expressões
- Entrada e Saída de Dados
- Estruturas Seqüenciais

Unidade 7 – Estruturas de Controle de Fluxo (7 horas e meia)

- Estruturas Condicionais
- Estruturas de Repetição

- Desenvolvimento e Depuração de Programas

Unidade 8 – Tipos Estruturados de Dados (10 horas)

- Tipos de Dados Homogêneos: vetores e matrizes
- Manipulação de Strings
- Tipo de Dados Heterogêneo: registros

Unidade 9 – Modularização (10 horas)

- Funções
- Escopo de Variáveis
- Passagem de Parâmetros
- Recursão

Unidade 10 – Desenvolvimento de Projeto (7 horas e meia)

- Algumas informações adicionais sobre Ponteiros
- Bibliotecas e Funções Úteis
- Guia para o desenvolvimento de projetos

Cada unidade conterá exercícios para fixação do assunto apresentado, assim como indicações de leituras adicionais. A Unidade 6 exigirá 10 horas de dedicação, uma vez que você estará começando com o ambiente de programação que usa a linguagem C. A unidade 7 exigirá 7 horas e meia. As unidades 8 e 9 exigirão um pouco mais: 10 horas de dedicação e lhe restarão 7 horas e meia para desenvolvimento do projeto final da disciplina, para o qual você encontrará diversas orientações na Unidade 10. Tenha em mente que muitos dos conceitos que serão vistos nessas unidades, você já utilizou em linguagem algorítmica (com exceção de registros e recursão), logo, a novidade será implementar os conceitos na linguagem C.

Mais do que nunca você vai precisar praticar. Logo, arregace as mangas e vamos lá!

6.1 Unidade 9 – Modularização

A modularização é uma forma de facilitar a construção de grandes programas, através de sua divisão em pequenas etapas, que são os chamados subprogramas, subrotinas ou módulos, de forma a minimizar a complexidade de resolução. Dessa forma, ao invés de escrever um programa grande, escrevem-se vários programas menores que são integrados para resolver o problema proposto.

Para problemas não-triviais, é desejável que se implemente a solução para cada subproblema como um subprograma que terá um certo grau de independência. E, dependendo da complexidade do problema a resolver, um subproblema pode sofrer um novo processo de decomposição em problemas ainda menores, gerando mais subprogramas, até que o programador julgue que um nível aceitável de divisão do problema foi atingindo, facilitando, assim, sua resolução.

Além de facilitar a resolução de um problema grande e completo, fazendo uso de subprogramas é possível fazer o reaproveitamento de código, uma vez que será possível utilizar um subprograma (ativá-lo) quantas vezes forem necessárias, eliminando, assim, a necessidade de escrever o mesmo código diversas vezes, em situações repetitivas. Além disso, a modularização facilita a concepção e manutenção dos programas. Detalhes sobre outras vantagens de usar modularização podem ser lidas na Unidade 5.

O módulo de um programa por onde começa a execução recebe o nome de programa principal e os outros módulos são os subprogramas propriamente ditos, que são executados sempre que ocorre uma chamada dos mesmos. No caso da linguagem C, a modularização de um programa é toda implementada a partir do desenvolvimento de **funções**. Inclusive, até o nosso programa principal (o main), é uma função também! Por isso, é justamente sobre funções e tópicos relacionados a elas (tais como: passagem de parâmetros, escopo de variáveis, etc) que discutiremos nessa unidade.

6.1.1 Funções

Funções (também chamadas de rotinas ou subprogramas) são segmentos de programa que executam uma determinada tarefa

Atenção

Você já estudou sobre modularização e funções na Unidade 5 (Subalgoritmos), só que tudo foi visto relacionando com a linguagem algorítmica. De qualquer forma, lá você encontra mais detalhes sobre modularização, funções, passagem de parâmetros e escopo de variáveis. Se necessário, por favor, revise o assunto!

específica. A segmentação ou modularização de um programa permite que cada segmento seja escrito, testado e revisado individualmente, sem alterar o funcionamento do programa como um todo.

O código de uma função é privativo a ela mesma e não pode ser acessado por nenhum comando em uma outra função, exceto por meio de sua ativação. Além disto, todas as funções estão no mesmo nível de escopo, isto é, não é possível definir uma função na parte interna de outra função.

Função não é novidade para você. Tanto porque já falamos sobre esse assunto na Unidade 5 em linguagem algorítmica, quanto porque, desde que começamos a estudar a linguagem C (na Unidade 6), já fizemos uso de várias funções existentes nas bibliotecas-padrão do C, tais como: printf(), scanf(), gets, getch(), getchar, putchar(), entre outras.

Na verdade, um programa em C, geralmente, consiste de várias pequenas funções ao invés de poucas de um tamanho maior. E nós podemos tanto usar funções pré-definidas (existentes nas bibliotecas), quanto definir as nossas próprias funções que serão chamadas de funções do usuário ou rotinas do usuário.

Definição de funções

Para definir uma função no C devemos utilizar a seguinte sintaxe:

```
tipoRetorno nomeFuncao (declaração_de_parâmetros)
{
    // declaração de variáveis locais
    //corpo_da_função;
    return valorRetorno;
}
```

Onde:

tipoRetorno é o tipo de variável que a função vai retornar (por exemplo: **int**, **float**, **char**, etc). O default é o tipo **int**, ou seja, em qualquer função para a qual não for declarado o tipo de retorno, ele será considerado do tipo **int**. Ressaltamos que existe apenas um valor de retorno para funções em C. Não é possível especificar mais de um tipo de retorno ou retornar mais de um valor. Agora, se a função não

for retornar nenhum valor para o programa que a chamou, devemos definir o seu retorno como do tipo **void**. Em inglês, **void** significa vazio e, no caso de funções, ela indica que a função não tem um retorno especificado (algo como “retorno vazio”). Assim sendo, uma função que não retorna nada teria a seguinte sintaxe:

```
void nomeFunção (declaração_de_parâmetros)
{ ... }
```

Neste tipo de função sem retorno, o comando **return** não é definido, porque ele não é mais necessário na função.

nomeFuncao – é o identificador da função. Ele não pode ser igual ao de nenhuma outra função existente e deve seguir as regras para criação de identificadores em C.

declaração_de_parâmetros é uma lista com os parâmetros que serão recebidos pela função para que esta seja ativada. Essa lista vai ter a seguinte forma:

```
tipo1 nome1, tipo2 nome2, ..., tipoN nomeN
```

Observe que, para cada uma das variáveis de entrada, o seu respectivo **tipo** deve ser especificado. É nesta declaração de parâmetros que informamos ao compilador quais serão as entradas da função (assim como informamos a saída da função quando especificamos o **tipoRetorno**). Os parâmetros podem ser passados por valor ou por referência (trataremos disso ainda nesta unidade, mais a frente). Essas entradas da função serão manipuladas no corpo da função. Ressaltamos que, podem existir funções que acontecer da função não receber valores, ficando a mesma sem uma lista de parâmetros.

Declaração de variáveis locais para realizar o seu processamento, a função pode precisar de variáveis locais, que só serão válidas e visíveis durante a execução da função. Elas são declaradas nessa área, no formato:

```
tipo1 nome1;
tipo2 nome2, nome3;
```

corpo da função é onde as entradas são processadas, cálculos são feitos e saídas ou retornos são produzidos.

return valorRetorno; e essa linha de comando indica que a função será encerrada e o **valorRetorno** será retornado pela função, para quem a chamou. Ou seja, a instrução **return** transfere o controle para o programa chamador da função e tem duas finalidades: determinar o fim lógico da função sendo executada e retornar o **valorRetorno** da função.

Quando um **return** não é usado (no caso, por exemplo, das funções com o tipo de retorno **void**), a função termina sua execução e retorna ao código que a chamou, após executar o último comando que faz parte do seu corpo e encontrar a chave final (que fecha a função). Ressaltamos que uma função pode ter vários comandos **return**, porém, apenas um deles será executado, ou seja, a função é finalizada assim que encontrar a primeira declaração **return**. Por exemplo:

```
int testaParidade (int num)
{
    if (num%2==0) // Testa se a variável num é divisível por dois
        return 1; // se for divisível, ou seja, se for par, retorna 1
    else
        return 0; // se não for divisível, ou seja, se for ímpar, retorna 0
}
```

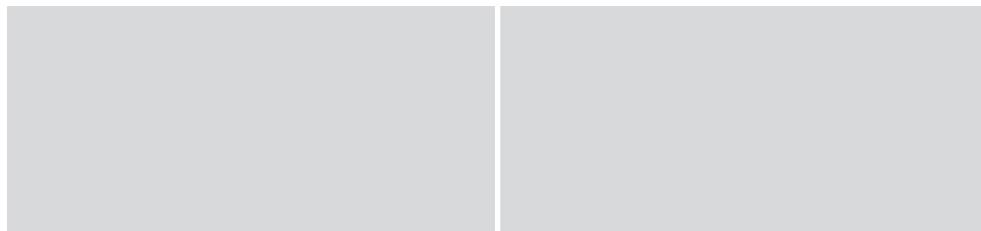
Atenção

É importante lembrar que o **valorRetorno** da instrução **return** tem que ser compatível com o **tipoRetorno** declarado para a função.

Atenção

Observe que a função **main** é uma função. A função **main** pode ser declarada como **void** (sem retorno) ou como retornando um número inteiro. Só devemos declarar a função **main** retornando um valor inteiro, se desejarmos que o sistema operacional receba um valor de retorno da função **main**. Se for esse o caso, devemos nos lembrar da seguinte convenção: se o **main** retornar zero (a gente vem usando isso: **return 0;**), significa que ele terminou, normalmente, sem erros. Porém, se a função **main** retornar um valor diferente de zero, significa que o programa teve um término anormal (com erros). Dessa forma,

as duas declarações a seguir seriam válidas:



Alguns compiladores C poderão apresentar problemas durante a compilação do segundo formato do main (**void main**), especificando que a função **main** sempre deve retornar um valor inteiro. Se isto ocorrer, basta fazer a função **main** retornar um inteiro (primeiro exemplo). O Dev-C++ aceita os dois formatos, sem problemas.

Alguns Exemplos de Funções

Exemplo 1: função com parâmetros, tipo de retorno e declaração de variável local definidos, para calcular a media aritmética de dois números de ponto flutuante.

```
float calcularMedia(float nota1, float nota2) // definição da função
{
    float media; //declaração de variáveis locais
    media = (nota1 + nota2) / 2; //corpo da função, faz o cálculo da média
    return media; // retorna a média calculada para o programa chamador da função
}
```

No exemplo acima, definimos uma função chamada **calcularMedia** que precisa receber valores para dois parâmetros do tipo ponto flutuante (nota1 e nota2). Dentro da função é declarada uma variável local (só vista dentro da função) chamada **media**. Os parâmetros e variável local são usadas para o cálculo da média, dentro do corpo da função. Depois que a média é calculada, ela é retornada ao programa chamador da função através da instrução **return**. Na verdade, a instrução **return** termina a execução da função e retorna o valor da variável **media**.

Exemplo 2: uma função sem retorno e sem parâmetros, apenas para imprimir algo na tela.

```
void imprimirAlgo() // definição da função
{
```

```
printf("\n\nUm exemplo de função sem parâmetro ou retorno \n");
}
```

A função acima tem tipo de retorno void, não possui parâmetros, nem declaração de variáveis locais. Ela foi implementada apenas para exibir uma mensagem na tela.

Exemplo 3: uma função sem retorno e com lista de parâmetros, para imprimir se o número é par ou ímpar.

```
void testaPar (int num) // definição da função
{
    if (num%2==0) // checa se o número é par e imprime o resultado
        printf("\n\nO número é par);
    else
        printf("\n\nO número é ímpar);
}
```

A função acima não tem retorno (void), nem variáveis locais. Porém, recebe um parâmetro do tipo inteiro e avalia se ele é par ou ímpar (usando o operador % que é o resto da divisão inteira), imprimindo o resultado dentro da própria função.

Exemplo 4: uma função com retorno e sem nenhum parâmetros, para devolver quantas letras tem o nome de uma pessoa (sendo sempre a mesma pessoa, visto que não se recebe nenhum parâmetro).

```
int contaLetrasNome ( ) // definição da função
{
    return (strlen("Sandra Siebra")); // retorna a quantidade de letras do nome
}
```

A função acima tem tipo de retorno int, não possui parâmetros, não possui declaração de variáveis locais e vai sempre retornar a quantidade de letras do mesmo nome (“Sandra Siebra”), usando a função strlen (que conta a quantidade de caracteres de uma string).

Exemplo 5: uma função com retorno e com parâmetros, para realizar um cálculo. O detalhe é que essa função usa mais de um return dentro dela.

```
int calcular(int num1, int num2, char operacao) // definição da função
```

```
{
    switch (operacao) // teste do seletor operacao
    {
        case '+':
        {
            return (num1+num2); // retorna a soma
            break;
        }
        case '-':
        {
            return (num1-num2); // retorna a subtração
            break;
        }
    }
}
```

A função acima tem tipo de retorno int, possui uma série de parâmetros declarados (num1, num2 e operacao) e não possui variáveis locais. Ao executar o comando switch, apenas um dos cases será executado. Ao ser executado, o case retorna o valor calculado para o programa chamador da função através do uso do return.

Localização da Especificação da Função

As funções declaradas pelos usuários podem ser especificadas em três locais diferentes: antes ou depois da função main (chamada de programa principal) ou em um arquivo separado. Vamos detalhar, a seguir, cada uma dessas três possibilidades.

1) Corpo da função antes do programa principal

Uma função pode ser definida antes do programa principal. Dessa forma, a localização da função seria a seguinte:

Declaração da função ANTES do main()

```
# includes...
// declaração de variáveis globais
tipoRetorno nomeFunção (lista_de_parâmetros)
{
    // Declaração de variáveis locais
    // corpo da função
    return valorRetorno;
```

```

}

int main()
{
    // programa principal
    // declaração de variáveis locais ao main
    // comandos
    // chamada da função em algum ponto do programa
    return 0;
}

```

Atenção

Para fins didáticos,
é esta a localização
das funções que
iremos adotar.

2) Corpo da função depois do programa principal

Quando escrevemos a definição de uma função **depois** do programa principal, ou seja, depois do **main()**, torna-se necessário incluir um **protótipo** da função chamada antes do programa principal. Protótipos são declarações de funções, isto é, você deixa indicado o formato da função irá usar. Um protótipo tem o seguinte formato:

```
tipoRetorno nomeFunção (declaração_de_parâmetros);
```

onde **tipoRetorno**, **nomeFunção** e a **declaração-de-parâmetros** devem ser os mesmos que você pretende usar quando realmente for escrever a função.

Para poder compilar a função **main()**, onde são chamadas quase todas as funções, o compilador precisa saber, com antecedência, quais são os tipos de retorno e quais são os parâmetros das funções. Essas informações são fornecidas pelo protótipo das funções, uma vez que elas só serão criadas/implementadas, depois do **main()**. Dessa forma, o uso de protótipos possibilita que o compilador possa checar se a função está sendo chamada da maneira adequada (por exemplo, se os parâmetros dos tipos corretos e na quantidade correta estão sendo enviados) e possa gerar o código do programa corretamente. Vamos dar um exemplo.

```

#include <stdio.h>
#include <stdlib.h>
float calcularQuadrado (float num); //protótipo da função antes do main()

int main () // o programa principal
{

```

Declaração da função DEPOIS do main()

```

float numero, resultado;
printf ("\n Digite um número qualquer: "); //lê o número
scanf ("%f", &numero);
resultado = calcularQuadrado (numero); //chamada da função
printf ("\n\nO quadrado de %.2f é %.2f \n",numero, resultado); //imprime resultado
system("pause");
return 0;
}

// declaração da função completa após o main()
float calcularQuadrado (float num)
{
    return (num*num);
}

```

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

3) Corpo da função escrito em um arquivo separado do arquivo que contém o programa principal

Em C, como em algumas outras linguagens, é permitido que o usuário crie uma função em **um arquivo** e um programa que a chame em outro **arquivo distinto**. Esta facilidade permite a criação de **bibliotecas de usuário**. Esse tipo de biblioteca nada mais é do que um conjunto de arquivos contendo funções escritas pelo usuário, da mesma forma que as bibliotecas natives do compilador C são um conjunto de arquivos contendo funções que são pré-definidas, ou seja, que já vêm com o compilador C.

Quando escrevemos a uma função em um **arquivo separado** do programa principal devemos **incluir** este arquivo no conjunto de **diretivas de compilação** (a parte dos **#include**) do programa principal. Esta diretiva instrui o compilador a incluir na compilação do programa principal o(s) outro(s) arquivo(s) que contém a criação das funções criadas pelo usuário. Na diretiva **#include**, indicamos entre aspas duplas o caminho de localização do arquivo onde está definida a função chamada. Por exemplo: **#include "c:\ip\testes\novas.h"**

O arquivo **novas.h** conterá as funções que poderão ser usadas no programa principal. Se a biblioteca estiver no mesmo diretório do

arquivo do programa principal, não é necessário indicar o caminho da nova biblioteca. Por exemplo `#include "funcoes.h"`

Essa opção de localização de função é mais usada, quando determinadas funções são importantes em vários programas distintos. Porque daí vale mais a pena declarar essas funções em um arquivo separado, para que elas possam ser usadas em vários lugares, sem precisar criar tudo denovo.



Um arquivo de biblioteca do usuário poderá conter a definição de uma ou mais funções. Em geral, quando o arquivo possui apenas uma função ele é nomeado com o mesmo nome da função e extensão `*.cpp` ou `*.c`. Por exemplo, poderíamos definir a função `calcularMedia()` no arquivo `calcularMedia.c` ou `calcularMedia.cpp`. Porém, quando um arquivo possui a definição de mais de uma função, ele deve ser nomeado com a extensão `*.h`. Por exemplo, se desejássemos criar um conjunto de funções estatísticas para usar em vários programa (tais como `calcularDesvioPadrao()`, `calcularMedia()`, `valorMaximo()`, `valorMinimo()`, etc) poderíamos implementar essas funções e gravá-las em um arquivo chamado de `estatistica.h`

Hierarquia de Funções

A criação das funções deve ser feita seguindo uma certa ordem, uma certa hierarquia: da função mais independente para a que depende das anteriores. Em outras palavras, sempre é possível que um programa principal chame uma função e, esta, por sua vez, chame outra função, que chama outra função e, assim, sucessivamente. Dessa forma, devemos ter o cuidado de escrever a função **chamada antes** da sua respectiva função chamadora. Isso é necessário para que o compilador conheça o formato da função antes de ela ser efetivamente utilizada.

Ativando uma Função

As funções podem ser chamadas ou ativadas, dentro do corpo do programa principal, o `main()` ou de dentro de outras funções. Quando

**Atenção**

Diferente de muitas linguagens, em C, se uma função retorna um valor, você não precisa necessariamente aproveitar este valor. Se você não fizer nada com o valor de retorno de uma função, ele será descartado.

Isto ocorre, o fluxo de execução do programa sendo executado é desviado para executar a função e, após seu término (por exemplo, quando uma instrução **return** é encontrada), a execução continua a partir do ponto seguinte à chamada da função.

Para ativar uma função é preciso saber quantos, quais, em que ordem estão e de que tipos são os parâmetros da função, bem como qual é o tipo do retorno da mesma. Como as funções podem retornar valores, podemos aproveitar esses valores para fazer atribuições, ou mesmo para usá-los em expressões condicionais. Nesses casos, a função é executada e, ao seu término, o trecho do comando que a chamou é substituído pelo valor retornado pela mesma dentro da expressão ou atribuição em que se encontra e a avaliação da expressão ou atribuição prossegue normalmente.

Vejamos, a seguir, dois exemplos de ativação de funções: o primeiro chama a função de dentro de uma expressão e o segundo chama a função usando atribuição.

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

//declaração da função
int checarIdade(int id) // checa se a pessoa é maior de idade ou não
{
    if (id >= 18)
        return 1;
    else
        return 0;
}

int main () // o programa principal
{
    int idade;
    puts("Digite a idade da pessoa: ");
    scanf("%d", &idade);
    // chamada da função dentro da expressão da estrutura condiciona IF
    if (checarIdade(idade)==1) // se retornar verdadeiro (1) a pessoa é de maior
        printf("\nA pessoa e' maior de idade \n");
    else
        printf("\nA pessoa e' menor de idade \n");
    system("pause");
    return 0;
}
```

}

Observe no exemplo acima que a função foi chamada dentro de uma expressão que faz parte de uma estrutura condicional **if**. O **if** só será avaliado, depois que a função for executada, pois ele precisa do valor de retorno da mesma para poder realizar a comparação. Outra forma de chamar uma função é usando uma **atribuição**, vejamos um exemplo.

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

float calcularMedia(float n1, float n2) // criação da função
{
    float media;
    media = (n1 + n2) / 2;
    return(media); // retorno da função
}

int main () // o programa principal
{
    float nota1, nota2, med;
    puts("Digite a primeira nota: ");
    scanf("%.f", &nota1);
    puts("Digite a segunda nota: ");
    scanf("%.f", &nota2);
    // chamada da função, o retorno dela é atribuído a variável med
    med = calcularMedia(nota1, nota2); // devem ser passados os parâmetros apropriados
    printf("\nA media do aluno e' %.2f", med);
    system("pause");
    return 0;
}
```

Observe, no exemplo acima, a função foi chamada/ativada dentro de uma atribuição. Nisso a função é executada e o valor que ela retornar será atribuído a variável **med** do programa principal.

6.1.2 Escopo de Variáveis

Atenção

Em blocos diferentes podemos ter variáveis locais de mesmo nome. Por exemplo, podemos ter quantos blocos quisermos com uma variável local chamada numero e não haverá nenhum problema com relação a isso. A razão é que cada variável local só será visível, só será válida, dentro do seu próprio bloco.

Em C, uma variável só pode ser **usada** após ter sido **declarada**, uma vez que é necessário reservar uma posição de memória para armazenar os valores atribuídos a ela. O escopo de uma variável define o **âmbito de validade** dela. Em outras palavras, o escopo define onde variáveis e funções são reconhecidas. Neste contexto, as variáveis podem ser classificadas como: **locais**, **globais** ou **parâmetros**, de acordo com o local dentro do programa onde forem declaradas.

Variáveis Locais: Uma variável é dita local, se a mesma só têm validade dentro do bloco no qual foi declarada (por exemplo, dentro de uma função, dentro de uma estrutura de repetição **for**, etc). Isto significa que qualquer variável local pode ser acessada e modificada apenas dentro de um bloco. O espaço de memória alocado para esta variável é **criado** quando a execução do bloco é iniciada e **destruído** quando ele é encerrado. Só para lembrar: um bloco começa quando abrimos uma chave e termina quando fechamos a chave.

Variáveis Formais ou Parâmetros Formais: Um parâmetro formal é considerado uma variável local da função onde ele foi declarado, com a diferença de que ele não é criado dentro da função, mas faz parte da **lista de parâmetros** da mesma. Desse modo, o parâmetro formal tem validade apenas dentro da função onde é declarado.

Daí, você poderia me perguntar, qual a diferença então entre um parâmetro formal e uma variável local da função, **for a o local de criação?** Bem, a grande diferença é que parâmetros formais precisam receber valores externos, que lhe são passados no momento da ativação da função. E variáveis locais não recebem valores externos.

Variáveis Globais: Uma variável é dita *global*, se for declarada **fora** de qualquer bloco do programa, inclusive do **main()**. Uma variável global pode ser usada (acessada e modificada) dentro de qualquer função ou bloco de comando. O espaço de memória alocado para esta variável é criado no momento de sua declaração e **destruído** apenas quando o programa é encerrado.

Atenção

Evite, ao máximo, o uso de variáveis globais, pois elas apresentam

os seguintes problemas: 1) Ocupam memória o tempo todo (as locais só ocupam memória enquanto estão sendo usadas); 2) Tornam o programa mais difícil de ser entendido e mais complicado de ser modificado, uma vez que podem ocasionar a mudança acidental do valor de uma variável, uma vez que ela pode ser acessada e modificada de qualquer lugar.



Atenção

Se uma variável global e uma variável local possuem o mesmo nome, todas as referências ao nome da variável, dentro do bloco onde a variável local foi declarada, dizem respeito à variável local e não têm efeito algum sobre a variável global.

Vamos dar um exemplo agora de um código com todos os tipos de variáveis.

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

int teste = 10; // Variável Global: pode ser acessada e manipulada em qualquer parte do programa

float calcularMedia(float n1, float n2) // criação da função
{
    float media; // Variáveis Locais: podem ser acessadas e manipuladas APENAS nos blocos onde foram criadas.

    media = (n1 + n2) / 2;
    printf("\n acesso a variável global dentro da função: %d", teste);
    return(media); // retorno da função
}

int main () // o programa principal
{
    float nota1, nota2, med; // Parâmetros Formais: podem ser acessados e manipulados APENAS dentro da função

    puts("Digite a primeira nota: ");
    scanf("%f", &nota1);
    puts("Digite a segunda nota: ");
    scanf("%f", &nota2);
    // chamada da função, o retorno dela é atribuído a variável med
    med = calcularMedia(nota1, nota2); // devem ser passados os parâmetros apropriados
    printf("\nA media do aluno e' %.2f", med);
    printf("\n acesso a variável global no main: %d", teste);
    system("pause");
}
```

Variável Global:
pode ser acessada e manipulada em qualquer parte do programa

Parâmetros
Formais: podem ser acessados e manipulados APENAS dentro da função

Variáveis Locais:
podem ser acessadas e manipuladas APENAS nos blocos onde foram criadas.

```

    return 0;
}

```

Observando o código acima **teste** é uma *variável global*, porque foi definida fora de qualquer bloco de comando. Por isso, esta variável pode ser acessada e manipulada de qualquer lugar. Além disso, ela vai ficar alocada na memória do computador até o final do programa.

As variáveis **n1** e **n2** foram definidas na lista de parâmetros da função **calcularMedia**, logo elas são *parâmetros formais* e só podem ser manipuladas e acessadas dentro desta função. A variável **media** é uma variável local da função **calcularMedia**. E, finalmente, as variáveis **nota1**, **nota2** e **med** são variáveis locais à função **main()**, só podendo ser manipuladas dentro da mesma.

6.1.3 Passagem de Parâmetros

Quando são definidos parâmetros formais para a função será preciso passar argumentos como parâmetro para que a função seja ativada. Esses argumentos podem ser passados para uma função de duas maneiras: por valor (ou por cópia) ou por referência (ou endereço). Vamos detalhar cada uma dessas maneiras a seguir.

Passagem de Parâmetros por Valor

Em C, essa é a forma default (padrão) de passagem de parâmetro (a única que usamos até agora na linguagem C). Ela funciona da seguinte maneira: quando um parâmetro é passado por valor, o valor do argumento do momento da ativação da função é avaliado e COPIADO para o parâmetro correspondente da lista de parâmetros da função. Ou seja, o argumento é apenas um valor de entrada para o parâmetro correspondente.

Dessa forma, qualquer alteração feita, dentro da função, no parâmetro não vai afetar o argumento do programa chamador da função. Isto, porque, o que está sendo manipulado dentro da função é uma cópia do argumento. Lembre que já explicamos isso com detalhes e com ilustrações na **Unidade 5 (Subalgoritmos)**.

Todas as funções implementadas até agora fazem uso de passagem de **parâmetro por valor**. Ou seja, todas as funções

manipularam apenas uma cópia do argumento que foi passado para os parâmetros da função, sem causar alterações nesses argumentos dentro do programa chamador da função.

Passagem de Parâmetros por Referência

Passagem de parâmetros por referência (também chamada de por endereço) é utilizada quando desejamos alterar o valor do argumento passado por quem ativa a função. Na verdade, na passagem de parâmetros por referência, toda alteração feita no parâmetro vai alterar o argumento do programa chamador associado a este parâmetro. Assim, quando a passagem é por referência, isto significa que o parâmetro é de *entrada e saída*.

Quando a passagem é por referência, o que é fornecido ao parâmetro do subalgoritmo não é uma cópia do valor do argumento, mas um apontador para a sua *localização na memória do computador* (ou seja, um endereço de memória). Dessa forma, quando é utilizada uma passagem por referência, o parâmetro e o argumento referenciam o mesmo endereço de memória e, consequentemente, qualquer modificação feita no parâmetro será refletida no argumento.

Para fazer uma passagem de parâmetros por referência, devemos declarar seus parâmetros formais como sendo **ponteiros**. Mas, o que é um **ponteiro**? Ele é um tipo de dado capaz de armazenar uma referência a um endereço de memória. Ou seja, ele não armazena um valor inteiro, de ponto flutuante ou caracteres, mas sim, endereços de memória.

Antes de continuar explicando a passagem por referência, vamos dar alguns detalhes úteis sobre ponteiros, também chamados de apontadores, que vão lhe ajudar a entender melhor esse tipo de passagem de parâmetro.

Para fazer uso de um ponteiro é necessário especificar o tipo de dados para o qual ele deve ser capaz de apontar. Isso porque cada tipo de dado aloca uma quantidade diferente de posições de memória e o ponteiro precisa saber disso. Por exemplo, um ponteiro para **char** guarda referência para um único byte na memória onde será armazenada uma variável do tipo **char**. Um ponteiro para inteiro guarda referência para uma posição de memória com dois bytes, onde será armazenada uma variável do tipo **int** (veja a Figura 1). Observe que na figura o ponteiro não guarda um valor, mas sim um endereço

de memória de uma variável do tipo inteiro (no caso, uma invenção de um endereço **#1Yn2**). O ponteiro não poderia, em hipótese alguma, armazenar o endereço de uma posição de memória que contivesse outro tipo de dado fora inteiro (por exemplo, não poderia guardar o endereço de uma variável do tipo **char**).



um endereço de memória de alguma variável existente no nosso programa. Mas, como saber a posição na memória de uma variável do nosso programa? Calma, não é complicado e inclusive já usamos isso quando fizemos uso da função **scanf**. Para pegar o endereço de memória de uma variável do nosso programa basta usar o operador **&**. Vamos dar um exemplo.

Primeiro vamos declarar e inicializar uma variável do tipo inteiro. Quando a variável é criada, é alocada uma posição de memória batizada de **contador** e nessa posição é gravado o valor de inicialização 10.

int contador = 10;  #1X89 Contador
10

 #1X45 pt

#1X45 pt #1X89 Contador
#1X89 10

Veremos mais alguns detalhes sobre ponteiros na **Unidade 10**. Porém, o que foi visto até aqui já é suficiente para entender a passagem de parâmetros por referência. Logo, voltando ao assunto da passagem por referência, para que as mudanças no parâmetro sejam refletidas nos argumentos do programa chamador, os parâmetros da função devem ser declarados como ponteiros. E, no momento da ativação da função, ou seja, quando a função for chamada, não se deve passar os valores dos argumentos, mas sim seus endereços de memória. Vamos dar um exemplo a seguir de um programa que faz uso de função para trocar dois valores inteiros.

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

// observe que os parâmetros são agora PONTEIROS

void trocar(int *n1, int *n2) ←

    //declaração de variável local
    /*lembre que quando usamos um asterisco na frente do nome do apontador estamos
    nos referindo ao conteúdo da posição armazenada pelo apontador e não ao endereço*/
    aux = *n1;
    *n1 = *n2;
    *n2 = aux;

}

int main () // o programa principal
{
    int valor1, valor2;
    puts("Digite o primeiro valor: ");
    scanf("%d", &valor1); // observe que o scanf usa o endereço da variável valor1
    puts("Digite o segundo valor: ");
    scanf("%d", &valor2);
    printf("\nOs valores ANTES da troca. Valor1 = %d e Valor2 = %d", valor1, valor2);
    // chamada da função, veja que são passados os endereços das variáveis como
    // parâmetro para isso, foi usado o & na frente do nome das variáveis
    trocar(&valor1, &valor2); ←
}
```

```

float calcularMedia(float n1, float n2) // criação da função
{
    float media;
    media = (n1 + n2) / 2;
    printf("\n acesso a variável global dentro da função: %d", teste);
    return(media); // retorno da função
}

int main () // o programa principal
{
    float nota1, nota2, med;
    puts("Digite a primeira nota: ");
    scanf("%f", &nota1);
    puts("Digite a segunda nota: ");
    scanf("%f", &nota2);
    // chamada da função, o retorno dela é atribuído a variável med
    med = calcularMedia(nota1, nota2); // devem ser passados os parâmetros apropriados
    printf("\nA media do aluno é' %.2f", med);
    printf("\n acesso a variável global no main: %d", teste);
    system("pause");
    return 0;
}

```

Observe que o que está acontecendo no programa acima é que passamos para a função **trocarr** o endereço das variáveis **valor1** e **valor2**, o que está indicado pelo uso do & na frente dos nomes destas variáveis no momento da ativação da função. Estes endereços são copiados para os ponteiros **n1** e **n2**, especificados na declaração da função como ponteiros.

Dentro da função **trocarr**, com o uso do *, acessamos o conteúdo da posição armazenada pelos ponteiros, modificando-o. Esse conteúdo, nada mais é do que os valores armazenados em **valor1** e **valor2**. Logo, os valores originais estão sendo modificados.

Indicamos também no programa, que a função de leitura de dados **scanf** que nós já vínhamos utilizando há um certo tempo, já fazia uso de passagem de parâmetros por referência. O motivo disso é que a função **scanf** precisa alterar as variáveis que passamos para ela, precisa gravar na posição de memória dessas variáveis os valores que vão ser lidos.

Em um mesmo programa podemos fazer uma passagem por valor e outra por referência. Veja o exemplo abaixo.

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

// observe que o primeiro parâmetro é passado por valor e o segundo por referência
void incrementar(int n1, int *n2) ←
{
    n1 = n1 + 1;
    *n2 = *n2 + 1;
}

int main () // o programa principal
{
    int valor1, valor2;
    puts("Digite o primeiro valor: ");
    scanf("%d", &valor1); // observe que o scanf usa o endereço da variável valor1
    puts("Digite o segundo valor: ");
    scanf("%d", &valor2);
    printf("\nOs valores ANTES do incremento: Valor1 = %d e Valor2 = %d", valor1, valor2);
    /* como primeiro argumento da função passamos o valor da variável valor1 (que será copiado para n1). Já como segundo parâmetro, passamos o endereço da variável valor2 */

    incrementar(valor1, &valor2); ←
    printf("\nOs valores DEPOIS do incremento: Valor1 = %d e Valor2 = %d \n", valor1, valor2);
    system("pause");
    return 0;
}
```

O argumento **valor1** será associado ao parâmetro **n1** e será feita uma passagem de parâmetro por valor, ou seja, uma cópia do conteúdo de **valor1** será passada para **n1**. Dessa forma, o conteúdo de **valor1** não é modificado, continuando com o seu valor original.

Já o argumento **valor2** tem seu endereço copiado para o parâmetro **n2**, que foi declarado como ponteiro. Dessa forma, está sendo feita uma passagem por referência. Dessa forma, a modificação feita dentro da função em **n2**, vai afetar, diretamente, a variável **valor2**, alterando seu valor original. Ficou claro? Espero que sim!

Atenção

 Em C, todo parâmetro é passado por valor, exceto quando indicado que a passagem de parâmetros é por referência, através do uso de apontadores.

Vetores e Matrizes como Argumentos de Funções

É possível passar vetores ou matrizes como argumentos de funções. No entanto, essa passagem tem um comportamento diferente dos outros tipos de variáveis. Ao se passar uma variável comum como parâmetro para uma função, cria-se uma cópia dessa variável e a função atua sobre essa cópia sem poder modificar a variável original que foi passada (passagem de parâmetro por valor).

Porém, quando se passa um vetor ou matriz como parâmetro para uma função, não se cria uma cópia, em vez disso, é passado o endereço do primeiro elemento do vetor ou matriz. A consequência dessa forma de passagem é que os elementos do vetor ou da matriz que forem modificados na função serão, efetivamente, modificados no programa chamador da função. Em outras palavras, vetores e matrizes sempre são passados para funções por referência.

A seguir, veja um exemplo de passagem de vetor como parâmetro em um programa onde o vetor é lido e a soma dos seus elementos é calculada.

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

// observe que o primeiro parâmetro é passado por valor e o segundo por referência
void incrementar(int n1, int *n2)
{
    n1 = n1 + 1;
    *n2 = *n2 + 1;
}

int main () // o programa principal
{
    int valor1, valor2;
    puts("Digite o primeiro valor: ");
    scanf("%d", &valor1); // observe que o scanf usa o endereço da variável valor1
    puts("Digite o segundo valor: ");
    scanf("%d", &valor2);
    printf("\nOs valores ANTES do incremento: Valor1 = %d e Valor2 = %d", valor1, valor2);
    /* como primeiro argumento da função passamos o valor da variável valor1 (que será
       copiado para n1). Já como segundo parâmetro, passamos o endereço da variável
       valor2 */
    incrementar(valor1, &valor2);
    printf("\nOs valores DEPOIS do incremento: Valor1 = %d e Valor2 = %d \n", valor1, valor2);
    system("pause");
}
```

```

return 0;
}

```

Na definição da função, é especificado como parâmetro um vetor de números inteiros. Observe que não é preciso especificar a dimensão do vetor, que ele vai pegar a dimensão do vetor que for passado como argumento no momento da ativação da função. Porém, cuidado! Lembre-se de que cabe ao programador assegurar que um elemento fora da faixa de valores do vetor será acessado. O compilador C não checará isso.

Vamos agora dar um exemplo de passagem de uma matriz como parâmetro em um programa para ler e imprimir os valores de uma matriz

```

#define NLINHAS 2 //declaração de constantes
#define NCOLUMNAS 2
#include <stdio.h>
#include <stdlib.h>

// definição da função
void imprimirMatriz(int matriz[][NCOLUMNAS]) // veja que não é necessário definir a
primeira dimensão
{
    int i, j;
    printf("\n A matriz digitada foi:\n");
    for (i = 0; i < NLINHAS; i++) {
        for (j = 0; j < NCOLUMNAS; j++) {
            printf(" %d", matriz[i][j]); //imprime a matriz
        }
        printf("\n"); // pula uma linha só para separar as linhas da matriz
    }
}

int main()
{
    int i, j;
    int numeros[NLINHAS][NCOLUMNAS]; //definição de uma matriz de números inteiros
    for (i = 0; i < NLINHAS; i++) {
        for (j = 0; j < NCOLUMNAS; j++) {
            printf("Digite o elemento [%d,%d] da matriz \n", i, j);
            scanf("%d",&numeros[i][j]); //le valores para preencher a matriz
        }
    }
}

```

```

}

printf("\n\n\n");
imprimirMatriz(numero);
//chamada da função
printf("\n");
system("pause");
return 0;
}

```

Quando uma matriz é passada com parâmetro para uma função é necessário indicar todas as dimensões com exceção da primeira. Dessa forma, o parâmetro **matriz [][2]** é visto como uma matriz de primeira dimensão desconhecida. Assim **matriz [][2]** é uma matriz de **N x 2** elementos (2 porque NCOLUNAS é igual a 2, neste programa). Quando uma matriz é passada como argumento na ativação da função, a dimensão que falta é completada. Novamente ressaltamos que compete ao programador assegurar que nunca será acessado um elemento fora das dimensões reais da matriz.

6.1.4 Recursividade

Na linguagem C, assim como em outras linguagens de programação, assim como uma função pode chamar outra função, ela também pode chamar a si própria. Uma função assim é chamada função recursiva.

Recursão é o processo pelo qual uma função chama a si mesma, repetidamente, um número finito de vezes. Este recurso é muito útil em alguns tipos de programas chamados de programas recursivos. Esses tipos de programas costumam ser muito utilizados na área de Inteligência Artificial, assim como na resolução de problemas que envolvem simulação numérica, busca e manipulação de árvores (uma estrutura de dados não-linear que você estudará na disciplina de algoritmos e estruturas de dados), entre outros.

Teoricamente, qualquer algoritmo pode ser implementado de forma recursiva (se precisar ser executado mais de uma vez) e qualquer algoritmo recursivo pode ser implementado de forma linear (como os que implementamos até agora).

Uma função que calcule o fatorial de um número inteiro positivo N é um bom exemplo de uma função recursiva. Vamos explicar isso



Atenção

Você deve ter cuidado ao implementar funções recursivas. A primeira coisa que deve ser providenciada é um critério de parada. Este critério vai determinar quando a função deverá parar de chamar a si mesma. Isto impede que o programa fique em um loop infinito com a função ativando a si mesma infinitamente, sem parar.

melhor: a definição de fatorial é:

0! = 1

n! = n * (n-1) * (n-2) * ... * 1, dessa forma, podemos definir que: n! = n * (n-1)!

Ou seja, o fatorial de um número pode ser calculado multiplicando ele pelo fatorial de seu antecessor. Tendo isso em mente, podemos implementar uma função recursiva para calcular o fatorial. Veja o exemplo a seguir.

```
#include <stdio.h>
#include <stdlib.h>
// definição da função
int factorial(int num)
{
    if (num != 0)
        // chama recursivamente a função factorial, mudando o
        // parâmetro para (num - 1)
        return (num * factorial(num - 1)); ←
                                            // retorna 1 porque o fatorial de zero é um
}
int main()
{
    int numero;
    printf("\n\nDigite um número inteiro positivo: ");
    scanf("%d", &numero);
    printf("\nO fatorial de %d é %d \n", numero, factorial(numero));
    system("pause");
    return 0;
}
```

No corpo da função observe que, quando o valor do parâmetro **num** for diferente de zero, a função **factorial** chama a si mesma, cada vez com um valor menor (n-1, n-2... até que a variável **num** seja zero). Ou seja, **num ser igual a zero** é o critério de parada para esta função. A figura 3 ilustra as chamadas recursivas da função, supondo que o valor que veio como parâmetro foi o número 3. Observe que o **main()** chama a função solicitando o cálculo do fatorial do número 3. Essa função, como o número é diferente de zero, executa o comando **return (n * factorial(n - 1));** para poder executar o **return**, como há uma função na expressão, a função é executada primeiro. Dessa forma, é chamada a mesma função, recursivamente, só que agora passando o valor 2 (**factorial(2)**) e assim por diante, até chegar no fatorial de zero, que não chama novamente a função. Simplesmente retorna o valor 1. Nas setas da parte de cima da figura, estão ilustrados os retornos de cada chamada da função.



Figura 3 - Ilustrando a execução de uma função recursiva para calcular fatorial do número 3

Implementar funções recursivas nem sempre é trivial. Então, só use se tiver certeza do que está fazendo, ok?



Você tem diversas funções implementadas neste capítulo. Todas elas você pode copiar para o Dev-C++ e compilar e executar para avaliar os resultados. De qualquer forma, aqui, vamos dar mais um exemplo de resolução de problemas.

Problema: Faça um programa em C para ler vários números inteiros (até que seja digitado o valor zero) e imprimir na tela se o número é par ou ímpar. Se o número lido não for positivo, dê uma mensagem apropriada e não julgue o número. Faça uso de função para resolver o problema. A partir da leitura do enunciado temos:

ENTRADA: vários números inteiros e positivos (até que seja digitado o valor zero)

PROCESSAMENTO: testar se os números são pares ou ímpares

SAÍDA: dizer se o número é par ou ímpar ou dar uma mensagem apropriada se o número for negativo.

Uma função deve ter uma finalidade bem definida, no nosso caso, ela deve checar se o número é par ou ímpar. Para a estrutura de repetição, como não temos uma quantidade definida de números para ler, não devemos usar o laço for, mas o laço do...while ou while.

```

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

// definição da função para testar se o número é par ou ímpar
void testarParidade (int num)
{
    if (num%2==0) // Testa se a variável num é divisível por dois
        printf("\n O número e' par"); // se for divisível o número é par
}
  
```

```

else
    printf("\n O número é' ímpar"); // se não for divisível, o número é ímpar
}
int main()
{
    int numero;
    do{ // optei por usar um laço do...while
        printf("\n\nDigite um número inteiro: ");
        scanf("%d", &numero); // lê um número
        if (numero != 0) // se o número não for zero – que é a condição de parada
        {
            if (numero < 0)
                printf("\nO número é negativo, por favor, digite um número positivo!");
            else
                testarParidade(numero); // chama a função se o número for positivo
        }
    }while(numero != 0); // só continua no laço enquanto o número for diferente de zero
    system("pause");
    return 0;
}

```



Saiba Mais

Para saber mais sobre a linguagem C você pode consultar qualquer livro que fale sobre a linguagem. Destaco a chamada “bíblia do C” que é o livro:

SCHILD'T, Herbert. **C completo e total**. Tradução Roberto Carlos Mayer. São Paulo: Makron Books, 3^a.edição, 1996.

Você também pode olhar os links indicados na unidade 6.



Atividade

Construa programas na linguagem C para resolver os problemas abaixo. Lembre das boas práticas de programação!

- 1) Crie um programa que faz uso de função para exibir a tabuada

da soma de um número que deve ser passado como parâmetro. Por exemplo, se vier o número 1, você deve imprimir a tabuada da soma do número 1 (a soma de um com todos os outros números de 0 a 9).

- 2) Escrever um programa que faz uso de uma função para checar se um número inteiro vindo como parâmetro é positivo ou negativo. Apresentar as mensagens apropriadas.
- 3) Faça um programa para ler o nome e duas notas de 25 alunos e exibir, para cada aluno, seu nome, sua média e seu conceito final. Use uma função para definir qual o conceito de cada aluno, conforme a tabela abaixo:

Nota	Conceito

- 4) Faça um programa para ler a altura e o sexo de diversas pessoas (até que seja digitada uma altura igual a ZERO) e que use uma função para calcular o peso ideal de cada pessoa. Para homens, calcular o peso ideal usando a fórmula peso ideal = $72.7 * \text{altura} - 58$ e, para mulheres, peso ideal = $62.1 * \text{altura} - 44.7$.
- 5) Escreva uma função EDivisivel(int a, int b). A função deverá retornar 1 se o resto da divisão de a por b for zero. Caso contrário, a função deverá retornar zero.
- 6) Faça um programa para ler um número e determinar, através de uma função que se este número é primo ou não. A função deve retornar 1 se o número for primo e 0 caso contrário. Lembre que um número é primo se ele só é divisível por ele mesmo e por um. Sugestão: faça uso da função definida na questão anterior.
- 7) Escreva um programa para ler um vetor de 10 posições de números inteiros, usando uma função e, além disso, tenha outra função para descobrir e retornar qual o maior valor gravado dentro desse vetor. Imprima o resultado no programa principal.
- 8) Escreva um programa que armazene em uma matriz três nomes de pessoas e em seguida os imprima. Assuma que o tamanho máximo de cada nome é de 50 caracteres. Faça a leitura da

matriz em uma função e a impressão da mesma em outra função.



Resumo

Funções permitem a modularização, ou seja, dividir um problema em problemas menores, além de permitir a reutilização de trechos de código, evitando duplicação e facilitando a manutenção. Funções, geralmente, executam cálculos ou pesquisas e retornam um único valor, do tipo especificado em sua declaração. As funções têm alta precedência, tendo prioridade sobre qualquer operador e sendo superada apenas pelos parênteses (expressões entre parênteses).

As funções podem receber ou não parâmetros de entrada. Quando não recebem parâmetros de entrada, as funções são definidas como **void**. Quando recebem parâmetros, as funções, na sua chamada (ativação) devem receber os argumentos obedecendo a ordem, ao tipo e a quantidade de parâmetros definidos em sua declaração.

Os parâmetros podem ser passados para as funções de duas maneiras:

- **por valor:** esse método copia o valor de um argumento no parâmetro formal da função. Assim, alterações feitas nos parâmetros da função não tem nenhum efeito nas variáveis usadas para chamá-la.
- **por referência:** nesse método, o endereço de um argumento é copiado no parâmetro. Para isso, na definição da função são usados apontadores (ou ponteiros) e no momento de chamar a função (ativar) são passados os endereços dos argumentos e não os seus valores (através do uso do operador &). Assim, alterações feitas no parâmetro afetam a variável usada para chamar a função.

As variáveis de um programa em C podem ser classificadas como:

- **Variáveis globais** - podem ser manipuladas em qualquer parte do programa e existem enquanto o programa estiver sendo executado. Elas são declaradas fora de qualquer bloco de comandos.

- **Variáveis locais** - só são válidas dentro de um bloco de comandos ou de uma função e existem enquanto o bloco não for finalizado.
- **Parâmetros formais** – são variáveis locais, com a diferença de que fazem parte da lista de parâmetros de uma função e recebem valores externos à função (os argumentos).

Variáveis globais devem ser evitadas, pois elas dificultam a localização de erros e a manutenção dos programas. A nossa próxima unidade é a última. Nela você vai encontrar dicas e informações adicionais que pode lhe ajudar no desenvolvimento do projeto da disciplina. Agora, falta pouco para acabar, espero que a jornada esteja sendo proveitosa para você. Até breve!

6.2 Unidade 10 – Desenvolvimento de Projetos

Olá, essa é nossa última unidade! Nesta unidade iremos apresentar alguns assuntos adicionais que podem lhe ajudar na implementação do projeto da disciplina, assim como vamos dar dicas e sugestões para organizar seus projetos. Vamos lá!

6.2.1 Ponteiros

Atenção

Apresentamos na unidade 9 algumas informações sobre apontadores (ou ponteiros), quando explicamos o assunto “passagem de parâmetro por referência”. Agora, vamos dar alguns detalhes adicionais sobre ponteiros, porém, não chegaremos a nos aprofundar nesse assunto, apresentaremos apenas o essencial.

Como visto na unidade 9, ponteiros são variáveis especiais para armazenar endereços de memória. Neste sentido, estas variáveis apontam para algum endereço da memória de um determinado tipo de dado. Em geral, o ponteiro aponta para o endereço de alguma variável que foi declarada no programa principal. E, quando declaramos um ponteiro, devemos declará-lo com o mesmo tipo (int, char, etc.) do endereço a ser apontado. Ou seja, se queremos um ponteiro para armazenar o endereço de uma variável do tipo char, devemos declarar esse ponteiro como char também.

Vamos dar um exemplo: considere que A é uma variável do tipo inteiro que recebeu o valor inicial 50. Considere, também, que P é um apontador para inteiro. Logo, P seria capaz de armazenar o endereço da variável A. Esse fato está representado na figura 1, onde com a seta queremos indicar que, como P armazena o endereço da variável A, isso equivale a dizer que P “está apontando” para A.

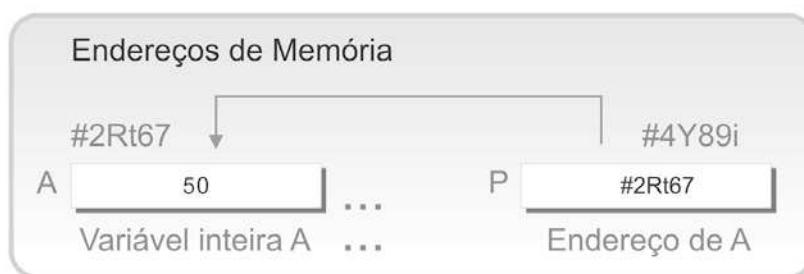


Figura 1 - Ponteiro P guardando endereço da variável inteira A

Para declarar um ponteiro usamos a seguinte forma geral:

`tipoPonteiro *nomeVariavelPonteiro;`

onde:

tipoPonteiro especifica o tipo de dado para o qual o ponteiro é capaz de apontar. Em outras palavras, o ponteiro apenas poderá guardar endereços de memória de dados desse tipo.

O **asterisco (*)** faz o compilador saber que aquela variável não vai guardar um valor mas sim um endereço para aquele tipo especificado.

nomeVariavelPonteiro é o identificador da variável ponteiro. Deve seguir as regras de identificadores especificadas para a linguagem C.

Alguns exemplos são: **float *ponteiroF, *ponteiroZ; int *p;** Nestas declarações, temos dois apontadores para números de ponto flutuante e um apontador para inteiro.

Mas, você poderia me perguntar: por que usar ponteiros? Bom, a principal utilização de ponteiros está relacionada à passagem de parâmetros por referência (vista na unidade 9). Outra utilização bastante interessante é quando se faz uso de alocação dinâmica. Mas o que é alocação dinâmica?



Operações com Ponteiros

Quando trabalhamos com ponteiros, trabalhamos, basicamente com duas coisas:

1. **O endereço de uma variável** – para conhecer o endereço de uma variável é utilizado o operador **&**. Quando usamos esse operador na frente do nome de uma variável, isso significa que queremos o endereço da variável e não o seu valor. Por exemplo, se tivéssemos em um programa uma variável inteira chamada `x` (`int x;`), o comando `&x` se refere ao “endereço de `x`”.

Endereço de X → #1X89 X



2. o conteúdo de um endereço – para isso é utilizado o operador *****. Quando usamos esse operador na frente do nome de um ponteiro, estamos nos referindo ao conteúdo do endereço apontado pelo ponteiro. Por exemplo, suponha a declaração de um apontador para números inteiros `int *ponti;`

#1X45 Ponti



Como apontadores são capazes de armazenar endereços, poderíamos fazer algo como: `ponti = &x;`

Armazenou o Endereço de X → #1X45 Ponti #1X89 X



E, depois, usando o operador ***** poderíamos referenciar o conteúdo do endereço apontado pelo ponteiro `ponti`: `*ponti = 10;` Esse comando equivale a dizer “o conteúdo do endereço apontado por `ponti` (no caso o conteúdo do endereço `#1X89`) deve receber o valor inteiro `10`”. Isso equivale ao endereço de `x`.

receber o valor 10.



Lembre sempre que o operador * só pode ser usado com ponteiros, a fim de referenciar “o conteúdo da posição armazenada no ponteiro”.

Existe uma série de operações (aritméticas, lógicas, etc.) envolvendo ponteiros. Contudo, como este é um curso de introdução à programação, nós só iremos estudar as operações de atribuição entre ponteiros e entre ponteiros e variáveis.

1. Um ponteiro pode receber o endereço de uma variável comum, contanto que o ponteiro seja do tipo da variável. Por exemplo:

```
int *pont; // declaração de um apontador para inteiros
int numero; // declaração de uma variável inteira
pont = &numero; // pont recebe o endereço da variável inteira numero
```



Atenção

Apesar de o símbolo * ser o mesmo, o operador * dos ponteiros não tem nada a ver com o operador * da multiplicação. O operador * usado em ponteiros é unário e, como o operador de endereço &, tem maior precedência com relação a TODOS os operadores aritméticos.

A atribuição só foi possível porque a variável **numero** é do tipo inteiro e **pont** é um apontador para inteiros. Se tentássemos atribuir à **pont** uma variável do tipo **float**, poderia não haver erro de compilação, mas o programa, provavelmente não funcionaria a contento (lembre que inteiros ocupam dois bytes na memória e floats ocupam quatro bytes).

2. Um ponteiro pode receber o valor de outro ponteiro, isto é, pode receber o endereço apontado por outro ponteiro, desde que os ponteiros sejam de mesmo tipo. Desta forma, ambos os apontadores estarão apontando para o mesmo endereço de memória. Por exemplo:

```
int *pont, *pont2; // declaração de um apontador para inteiros
int numero; // declaração de uma variável inteira
pont = &numero; // pont recebe o endereço da variável inteira numero
// pont2 recebe o conteúdo de pont, ou seja, o mesmo endereço que ele armazena, que
// é o endereço da variável numero
```

```
pont2 = pont;
```

3. Um ponteiro pode receber um endereço de memória diretamente. Um endereço é um numero inteiro, em geral, no formato hexadecimal (aquele que começa por 0x.... lembra?). Por exemplo:

```
int *pont;
pont = 0x03F8; // o apontador está recebendo o endereço da porta serial COM1
```

4. A um ponteiro pode ser atribuído um valor nulo (um endereço nulo, porém, mapeável), usando a constante simbólica NULL (que está declarada na biblioteca stdlib.h). Essa constante pode ser atribuída a qualquer tipo de apontador. Por exemplo:

```
int *pont;
char *pontc;
pont = NULL;
pontc = NULL;
```

Lembre que será necessário incluir a biblioteca stdlib.h para poder usar a constante NULL (**#include <stdlib.h>**). O valor NULL é equivalente a 0 (zero) e significa que o ponteiro não aponta para lugar nenhum.

5. Dois ponteiros podem ser comparados (usando-se operadores lógicos) desde que sejam de um mesmo tipo. Por exemplo:

```
int *pontx, *ponty;
int x, y;
pontx = &x;
ponty = &y;
// você pode testar se os apontadores guardam o mesmo endereço,
// ou seja, eles apontam para o mesmo endereço
if (pontx == ponty) ...
// você pode testar se os apontadores guardam endereços diferentes, ou seja,
// se eles apontam para endereços diferentes
if (pontx != ponty) ...
// você também pode testar se um apontador é nulo
if (pontx == NULL) ...
```

Vejamos, agora, um exemplo de uso de apontadores e dos operadores & e *.

```
#include <stdio.h>
#include <stdlib.h>
int main() // o programa principal
{
    int *pont; //declara um apontador para números inteiros
    int numero, conteudo; //declara duas variáveis inteiros
    numero = 10; // inicializa a variável numero com o valor 10
    pont = &numero; // o endereço da variável numero é atribuído ao apontador pont
    conteudo = *pont; // o que está no endereço guardado em pont é copiado em conteudo
    printf("Número = %d\n", numero); //exibe o valor de numero, ou seja, 10.
    printf("Conteúdo = %d\n", conteudo); // exibe o valor de conteudo. Ou seja, 10
    // exibe o conteúdo do endereço armazenado em pont. Ou seja, 10
    printf("O conteúdo gravado no endereço armazenado em pont = %d\n",
*pont);
    // O código de formatação %p serve para exibir um número no formato hexadecimal
    printf("O endereço da variável número = %p\n", &numero); //exibe o endereço de numero
    //exibe o conteúdo de pont, ou seja, o endereço da variável numero
    printf("O endereço armazenado dentro de pont = %p\n", pont);
    //exibe o endereço da variável pont
    printf("O endereço de pont = %p\n", &pont);
    system("pause");
    return 0;
}
```

No exemplo acima, na função **printf**, o código **%p** indica à função que ela deve imprimir um endereço no formato hexadecimal. Vamos mostrar mais um exemplo.

```
#include <stdio.h>
#include <stdlib.h>
int main() // o programa principal
{
    int num, num2; //declara duas variáveis inteiros
    int *pont, *pont2; // declara um apontador para números inteiros
    num = 10; // inicializa a variável num
    printf("O valor de num = %d\n", num);
    pont = &num; // grava em pont o endereço de num
    printf("O conteúdo gravado no endereço armazenado em pont = %d\n",
*pont);
    // a variável num2 recebe o conteúdo da posição de memória armazenada em pont, ou seja,
    // num2 recebe o mesmo valor de num, que é 10.
    num2=*pont;
    printf("O valor de num2 = %d\n", num2);
    *pont = 500; // Muda o conteúdo da posição de memória armazenada em pont
    // dessa forma, o valor de num é, indiretamente, modificado, mas num2
```

```

continua com
// o valor anterior, 10
printf("O valor de num = %d\n", num);
printf("O valor de num2 = %d\n", num2);
pont2 = pont; // os dois apontadores agora estão com o endereço da
variável num
printf("O conteúdo gravado no endereço armazenado em pont2 = %d\n",
*pont2);
system("pause");
return 0;
}

```

Problemas com Ponteiros

Existem alguns cuidados que precisamos ter com a manipulação de ponteiros. Primeiro, não podemos atribuir valores a ponteiros que ainda não apontam para nenhum lugar. Por exemplo:

```

int num = 10;
int *pont;
*pont = num; //o conteúdo do endereço apontador por pont recebe o
valor de num

```

Atenção

O grande problema do caso acima é que, muitas vezes, o programa compilará e rodará. Porém, o resultado será imprevisível. Isto porque o ponteiro **pont** poderá estar apontando para qualquer lugar.

Com um único caso de uso indevido, pode ser que você não veja nenhum efeito estranho. Porém, se você começar a gravar números em posições aleatórias, desconhecidas, no seu computador, não vai demorar muito para travar o micro ou gravar dados por cima da área de outros programas que estejam executando.

O problema é que **pont**, no momento desta última atribuição ainda não aponta para nenhum lugar, ou seja, ele não armazena nenhum endereço, logo, o apontador pode conter apenas “lixo”. Dessa forma, esse comando não terá um bom resultado, porque tentará atribuir um valor a um endereço inválido. A solução para esse tipo de problema com ponteiros é sempre se certificar de que o ponteiro está apontando para algum endereço de memória válido, antes de fazer uso dele.

Outro problema comum é confundir o uso dos operadores ou esquecer algum operador. Por exemplo, considere as seguintes declarações:

```
int *pont; e int num=5;
```

Agora, vamos analisar a atribuição a seguir: **pont = num;**

Essa operação seria inválida, porque apontadores só podem armazenar endereços de memória e não valores. O correto seria: **pont = #**

É preciso lembrar que quando usamos **pont** estamos fazendo referência ao endereço de memória armazenado em **pont**. Já quando usamos ***pont** estamos nos referindo ao conteúdo da posição de memória armazenado em **pont**, ou seja, estamos nos referindo, neste caso, a um número inteiro.

Ponteiros, Vetores e Strings

Em C, o nome de um vetor é tratado como o endereço de seu primeiro elemento. Assim ao se passar o nome de um vetor para uma função está se passando o endereço do primeiro elemento de um conjunto de endereços de memória. Por exemplo, se `vet1` é um vetor, então passar o nome `vet1` como parâmetro, é equivalente a passar o endereço `&vet1[0]`, que é o endereço da primeira posição do vetor `vet1`.

Podemos acessar o endereço de qualquer elemento do vetor do seguinte modo: `&vetor[indice]`, onde `indice` representa o índice do vetor que queremos acessar, lembrando que os vetores começam no índice zero. Vejamos um exemplo.

```
#include <stdio.h>
#include <stdlib.h>
int main() // o programa principal
{
    int vetor [] = { 10, 20, 30, 40, 50 }; // cria um vetor de inteiros e já o
    inicializa com valores
    int *pont1 , *pont2; // declara dois apontadores para números
    inteiros
    pont1 = &vetor[2]; // o apontador recebe o endereço do terceiro
    elemento do vetor
    // dessa forma, quando imprimirmos o conteúdo do endereço, ele vai
    pegar o valor 30
    printf("O conteúdo gravado no endereço armazenado em pont1 =
    %d\n", *pont1);
    pont2 = vetor; // o apontador recebe o endereço do primeiro elemento
    do vetor
    // lembre que quando usamos só o nome do vetor nos referimos a
    posição zero deste vetor
    // logo vai ser impresso, a seguir, o valor 10
    printf("O conteúdo gravado no endereço armazenado em pont2 =
    %d\n", *pont2);
    system("pause");
    return 0;
}
```

Em C, uma **string** é um vetor unidimensional de caracteres, sendo o último destes o caractere especial '`\0`'. Observe que sendo um vetor, uma string segue o mesmo que foi explicado anteriormente para vetores.

6.2.2 Etapas para Desenvolvimento de um Programa

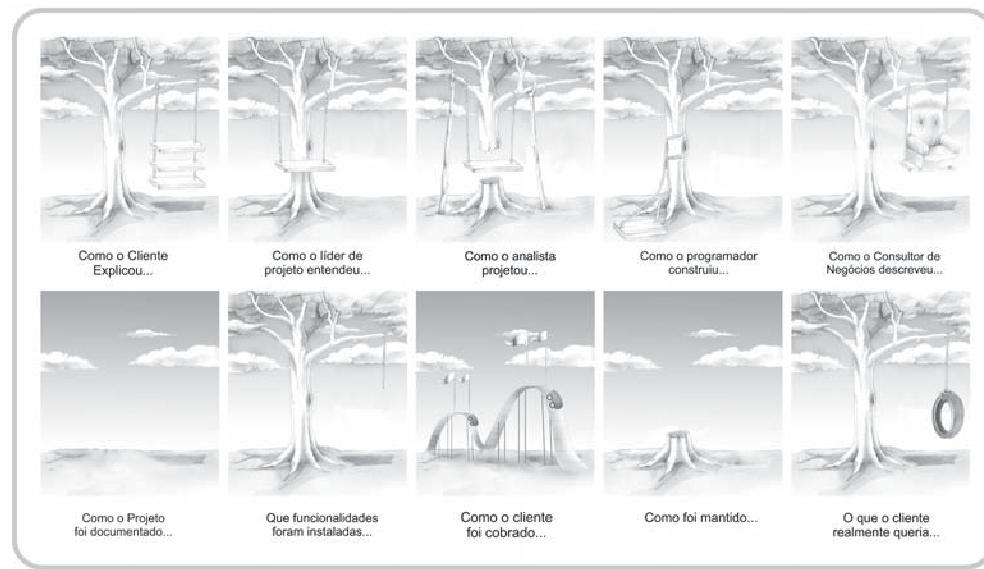
Para implementar qualquer projeto, você tem de, primeiro, entender bem o problema que pretende resolver. Mais à frente no seu curso,

Atenção



Por regra, sempre que em um programa aparece o nome de um vetor, sem o índice e sem os colchetes à sua direita, isto corresponde ao endereço do primeiro elemento daquele vetor. Assim, tal qual no programa acima, para fazer um ponteiro apontar para o primeiro elemento de um vetor, ao invés de usar `pont = &vetor[0];` podemos usar `pont=vetor;`

você vai estudar isso em uma disciplina de análise de sistemas ou de Engenharia de Software. Nessa disciplina existe um assunto chamado “levantamento de requisitos” que é justamente onde você procurar levantar todas as informações necessárias para o desenvolvimento do projeto. Realiza entrevistas, olha documentos, avalia sistemas existentes, etc. Essa etapa é extremamente importante, para que o sistema implementado corresponda às necessidades e expectativas do usuário. Pois, é mais comum do que se pensa, ainda hoje, que sistemas desenvolvidos se distanciem da idéia original almejada pelo cliente (vide Figura 2).



do problema e tentar entendê-lo. Não adianta seguir adiante sem entender o problema bem. Se dúvidas surgem e persistem, melhor consultar o seu tutor ou professor para esclarecer o que tem de ser feito em detalhes.

Uma vez que o que precisa ser feito for entendido, tente esquematizar as funções que precisam ser criadas para resolver o problema. Ou seja, tente dividir o problema em partes passíveis de serem resolvidas (isso pode até facilitar a divisão de trabalho entre o grupo, se for o caso). Por exemplo, se você tem de fazer um algoritmo para manipular uma agenda de endereços e telefones, você, provavelmente, vai precisar inserir, consultar, atualizar e deletar dados da agenda. Cada funcionalidade dessas pode ser implementada como uma função. Pergunte-se: Como o usuário poderá ter acesso a essas funcionalidades? Que funcionalidade deve vir primeiro? Que dados o usuário deve fornecer para a função funcionar? Que informações deverão ser apresentadas ao usuário como resposta?

Depois, esquematize como será o menu do sistema. O que terá no programa principal? Quais as variáveis que serão necessárias? Pense no esqueleto da chamada das funções. Algo como:

```
int main ()
{
    int opcao;
    do
    {
        printf ("\n\nEscolha uma das opções a seguir pelo
número:\n");
        printf ("1 - Cadastrar Pessoa na agenda de
endereços\n");
        printf ("2 - Consultar Por Nome na agenda\n");
        printf ("3 - Remover Pessoa da agenda de endereços\n");
        scanf("%d", &opcao);
        //se o que foi digitado for diferente das opções válidas fica
preso no laço
    } while ((opcao<1)|| (opcao>3));

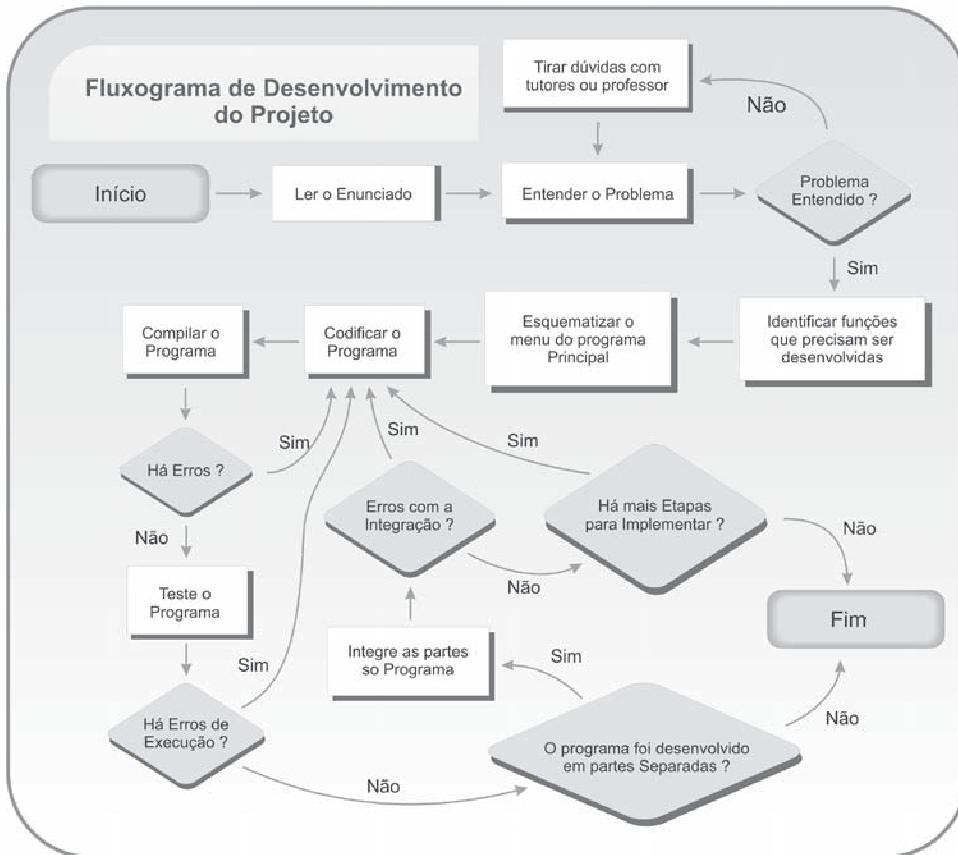
    switch (opcao) //depois de digitada uma opção válida, viria para
cá
    {
        case 1:
            // executar o cadastro
            break;
        case 2:
            // executar a consulta
            break;
        case 3:
            // executar a remoção
            break;
    }
    ...
}
```

Quando tudo estiver especificado, não queira implementar tudo de uma vez (a não ser que o trabalho seja dividido no grupo). Vá por partes. Implemente uma função, depois a teste (chamando-a no programa principal, de preferência, já usando uma estrutura de menus). Quando ela estiver correta e funcionando bem, implemente a seguinte. Isso se chama desenvolvimento incremental, onde você vai construindo o programa por partes, adicionando funcionalidades aos poucos.

Lembre de sempre discutir com seu grupo (se for o caso) ou com seu tutor ou professor as possíveis soluções e as dificuldades que forento durante a implementação. Releia alguma parte do material de estudo se tiver dúvida em um assunto. Pesquise na Internet por algum exemplo de programa ou procure em livros (se tiver acesso aos mesmos). Existe até uma ferramenta da google para pesquisa de códigos em qualquer linguagem: <http://code.google.com/> O detalhe é que essa busca de códigos só está disponível em inglês.

Quando o programa estiver todo pronto, teste ele todo. Se ele foi implementado separado (dividido entre os componentes do grupo), deve vir aí uma etapa de integração, onde tudo que foi feito deve ser integrado em um único programa que tem diversas funções. Neste caso, os testes devem ser ainda mais cuidadosos.

Se você compilar e executar o programa e não ocorrerem erros (nem de compilação, nem de execução), o programa está pronto! Bem, vamos esquematizar em um fluxograma (vide Figura 3) uma sugestão de passo a passo a seguir para o desenvolvimento de um projeto de disciplina de programação. Lembra da notação de fluxogramas? Vamos fazer uso da mesma! Se não lembra, a explicação de fluxograma está na unidade 1 do primeiro fascículo desta disciplina.



6.2.3 Orientações Gerais para Desenvolvimento de Projetos

No processo de escrita de código fonte em uma linguagem de alto nível é interessante que o programador se preocupe não somente em escrever códigos sintaticamente corretos, mas, também, se preocupe com a lógica do programa (ou seja, se ele está funcionando corretamente, de acordo com o seu objetivo) e com organização do texto do programa em si. A vantagem mais evidente de um texto de programa mais organizado é a maior facilidade de entendimento e manutenção posteriores, bem como a maior facilidade na detecção de erros.

Durante minha vida profissional como professora e, também, como desenvolvedora, não foram raros os casos de alunos e programadores que tinham dificuldade em entender o código dos programas desenvolvido por eles mesmos, após algum tempo sem manusear o código. Tudo isso devido à falta de organização, identação e comentários no código.

Neste contexto, vamos dar, a seguir, algumas dicas de como deixar seu programa mais simples, claro, fácil de entender e de dar manutenção. Seguindo essas dicas, você vai melhorar e muito, sua vida de programador.

1. Tente seguir a estrutura do programa na linguagem escolhida

– toda linguagem de programação tem um esqueleto-padrão de definição de programa. Em muitas delas, este esqueleto pode ser modificado em algumas partes (algumas coisas podem trocar de lugar). Porém, quando seguimos o padrão, fica mais fácil de outra pessoa compreender nosso código e de todos sabermos onde localizar cada declaração dentro do programa. Em C, a estrutura-padrão de um programa é a especificada na Figura 4.

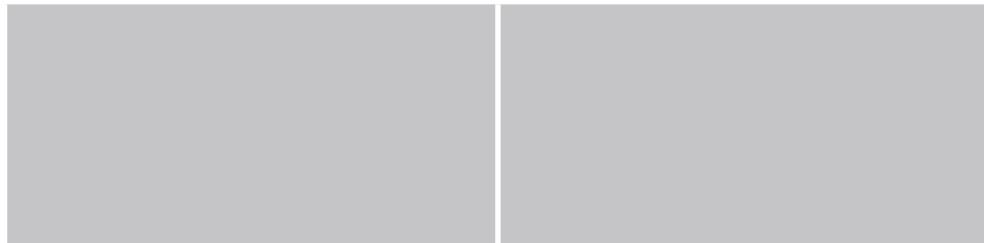


Figura 4 - Estrutura de um programa em C

2. Identifique o seu programa! – se todas as linhas do código-fonte possuírem o mesmo alinhamento na margem esquerda, não ficará claro, a princípio, quais são as estruturas que compõem o programa, onde elas se iniciam e onde terminam, dificultando o entendimento do código e, consequentemente, dificultando sua manutenção.

O uso de identação determina que cada bloco lógico do programa (tais como: estruturas de controle, de repetição, corpo de funções, definição de estruturas, etc) seja marcado por um espaçamento para a direita em relação ao bloco anterior,

tornando mais fácil identificar a estruturação do programa e a sua lógica no que diz respeito às estruturas de controle e aos módulos funcionais. Por exemplo, qual código está mais claro?



3. Alinhe as chaves demarcadoras dos blocos lógicos – Em linguagem C, os blocos lógicos devem ser delimitados por chaves (à exceção dos blocos formados somente por uma instrução e que não são corpo de função). Você tem duas opções para alinhamento das chaves (que pode facilitar você checar se todas estão sendo abertas e fechadas devidamente):

- Você pode alinhar as chaves com a identação do bloco anterior ao novo bloco lógico sendo iniciado, por exemplo:

```
while (num != 0)
{ // chaves alinhadas com o bloco anterior
    x = x + 1;
    num--;
}
```

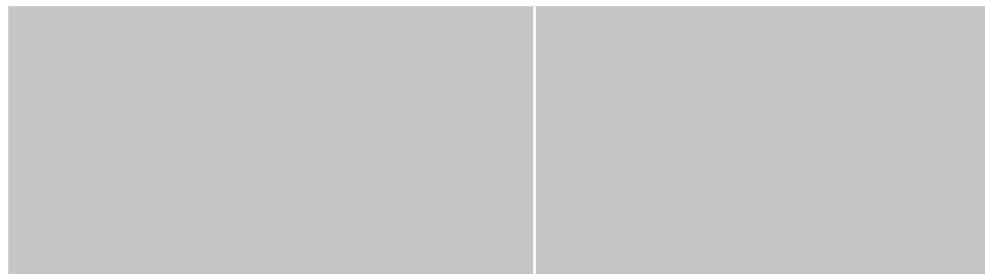
- Você pode alinhar a chave de fechamento com a identação do bloco anterior e posicionar a chave de abertura no final da última instrução do bloco anterior, por exemplo:

```
while (num != 0){ // chave inicial no fim da instrução
    x = x + 1;
    num--;
} // chave final alinhada com o bloco anterior
```

Nos blocos lógicos formados somente por uma instrução o uso de chaves é recomendado, porém não é obrigatório. Caso não se queira utilizar as chaves, uma boa prática consiste em identar a instrução, para indicar que ela pertence àquele bloco. Por exemplo:

```
if (num1 = num2)
    printf("Os números são iguais!");
```

- 4. Tenha cuidado com o fechamento correto das chaves** – uma chave fechada no lugar errado pode provocar uma série de erros de compilação ou resultados indesejados durante a execução do programa. Por exemplo, quando o teste `z = b` será executado?



Observe a diferença entre os dois trechos de programa e como as chaves influenciaram na mudança da resposta.

- 5. Use nomes de variáveis que sejam expressivos** – O identificador (nome) atribuído à uma variável deve expressar, na medida do possível, a finalidade daquela variável. Muitas vezes, nos programas que desenvolvemos, por preguiça ou comodidade, usamos variáveis chamadas A, B, C, X ou Y. Isso pode dificultar (e muito!) o entendimento do programa e do que está sendo armazenado em cada uma dessa variáveis. O identificador da variável deve tentar expressar claramente a sua finalidade. Identificadores compostos por um único caracter não diminuem o tamanho do código executável gerado e devem ser evitados tanto quanto possível. Eles só devem ser utilizados se forem apenas auxiliares em algum cálculo ou processamento ou quando exprimirem, claramente, o objetivo para o qual foram declaradas as variáveis (por exemplo, coeficientes de uma equação matemática). Alguns exemplos de bons identificadores são:

```
int qntde_produtos, totalPessoas;
char nome[45], cpf[11];
float media;
```

- 6. Comente seus programas** - Um programa bem documentado deve conter comentários, principalmente nos pontos onde uma simples leitura do código não é suficiente para se entender a sua

finalidade. Adicionalmente, você pode comentar as declarações de variáveis, pois mesmo quando seus identificadores são adequadamente escolhidos, nem sempre fica clara a finalidade a qual elas se destinam. Em implementações de funções, uma boa prática consiste em se introduzir um comentário, antes da definição da mesma, contendo as seguintes informações:

- nome (identificador) da função
- descrição dos parâmetros (dados de entrada) da função
- descrição do retorno da função (se houver)
- uma breve descrição do funcionamento da função (pra quê ela serve?)

Por exemplo, veja o comentário abaixo, que poderia ser feita antes de uma função para calcular o fatorial de um número.

```
/* ****
Nome Função: factorial
Parâmetros: num – número do qual se calculará o factorial dentro da função
Retorno: valor do factorial de num
Descrição: calcula e retorna o factorial de um número vindo como parâmetro
***** */
```

Comentários podem lhe ajudar a lembrar o que faz cada parte do programa, se você tiver de dar manutenção em um código no qual não mexe há muito tempo. Assim como deixarão o programa, como um todo, mais legível.

7. **Use, sempre que necessário, espaçamentos e quebras de linha** - Expressões envolvendo operadores, tanto lógicos quanto aritméticos, devem conter espaços separando os operandos dos operadores para facilitar a sua leitura. Veja as duas linhas abaixo. Elas representam o mesmo comando. Qual você acha que está mais simples de ler?

aux = num * 2 + 5; ou **aux=num*2+5;**

Em casos nos quais seja necessário quebrar uma expressão ou chamada de função em mais do que uma linha, utilizar uma das seguintes regras: quebrar após uma vírgula ou quebrar após um operador.

Quando uma função `printf` ou equivalente é utilizada e é necessário imprimir uma grande quantidade de caracteres, é interessante utilizar várias chamadas dessa função em sequência. Isso irá evitar que uma única linha de código contenha um número excessivo de caracteres.

8. **Cuidado com o uso dos operadores de incremento (++ e --) e atribuições aritméticas (+=, -=, *=, /=)** – Nem sempre esses operadores são legíveis (fáceis de entender). Logo, seu uso pode dificultar o entendimento do programa. Evite-os sempre que possível.
9. **Lembrar que vetores e matrizes não têm seus limites checados pelo C** – o acesso a uma posição que não pertence a um vetor ou matriz não provoca um erro de compilação no C. Logo, é responsabilidade do programador tomar cuidado para que posições inadequadas não sejam acessadas, provocando resultados imprevisíveis.
10. **Evite usar o operador condicional ? (aquele que usa (condição) ? expressão1 : expressão2)** - Este operador é muito complicado de entender e pode tornar o código bem ilegível. Se `num1 = 10` e `num2 = 20`, qual seria o valor de `max` na expressão abaixo?
 - `max = (num1 > num2) ? num1 : num2`
 - O valor de `max` seria 20. Pois se a condição é verdadeira, a expressão1 é executada, senão a expressão2 é que é executada.
 - Note que o uso desse operador pode ser substituído por uma expressão usando `if..else`, sem problema algum, deixando o código muito mais legível. Por exemplo:

```
if (num1 > num2)
    max = num1;
else
    max = num2;
```

11. **Termine todas as linhas de comandos com ; (ponto e vírgula)** – afinal essa é a marcação para final de comando.
12. **Evite digitar mais de um comando na mesma linha** – pois isso compromete a legibilidade do programa. Por exemplo, já

imaginou um trecho de programa assim? Concorda que ficaria ruim de entender?

```
if (num1 > num2) max = num1; else max = num2;
while(num1 > num2){ printf("Digite o numero");scanf("%d", &num1);}
```

13. **Modularize seu sistema** – não esqueça que é muito mais fácil “dividir para conquistar”. Implementar as funcionalidades em funções tornar o programa mais modular e mais fácil de compreender e manter.
14. **Tenha preocupação em dar mensagens informativas na entrada e na saída de dados** – ou seja, imprima uma mensagem especificando o que deve ser digitado, antes de partir para um comando de leitura de dados, tipo o scanf. Isso deixará seu usuário ciente do que deve ser digitado. Também imprima uma mensagem especificando o dado que será impresso em um printf ou similar. Isso facilitará o entendimento do resultado apresentado pelo programa.
15. **Não tenha vergonha de perguntar, mas saiba definir sua dúvida** – mesmo quando se é o “ás” da programação, pode haver situações em que dúvidas surgem. Nesses casos, você pode recorrer aos seus colegas, aos tutores, professores ou até mesmo partir para tentar tirar uma dúvida em uma lista de discussão na Internet. Afinal, vivemos em um mundo conectado e outras pessoas podem ter a solução para o problema que você está enfrentando agora, e podem estar dispostas a ajudar nas mais diversas listas de discussão. Porém, para poder tirar uma dúvida, você deve ao menos saber especificar qual o problema ou em que ponto tem dúvida. Chegar para alguém como algo do tipo: “eu não sei nada do assunto” ou “não sei nem por onde começar”, é muito genérico e fica difícil alguém tentar lhe ajudar.

- **Aprenda a pesquisar** - Isso mesmo! Muitas vezes, podemos tirar algumas das nossas dúvidas pesquisando na Internet ou em livros. Sabia que muitas pessoas não sabem usar o google!? (não digo chegar ao site do google, mas usar palavras-chaves que realmente possam trazer um resultado apropriado). Aqui vêm algumas dicas de como pesquisar melhor:

Atenção

As funções têm alta precedência, estando abaixo apenas dos parênteses. Ou seja, elas quase sempre são executadas primeiro.

- a. Use palavras significativas na pesquisa e não frases. Por exemplo, procurar por frases do tipo: “eu quero aprender a programar em C”, pode até achar algo útil, mas, geralmente não vai achar. Seria melhor pesquisar por algo como: “programar em C”, melhor ainda, seria usar algo mais específico, de acordo com suas necessidades, tal como: “vetores em C”.
- b. Algumas vezes, é necessário, além de fazer a pesquisa usando a palavra em português, fazer, também, a pesquisa usando a mesma palavra ou tema em inglês (até porque os artigos e publicações mais novos, geralmente, estão neste idioma).
- c. Há várias dicas de como fazer pesquisas no google no site: <http://www.infowester.com/dicasgoogle.php>

16. Algumas dicas para compilação e execução:

- Sempre salve o programa antes de compilar
- Sempre compile o programa antes de executar
- Quando ocorrer um erro de compilação, dê um duplo clique sobre a mensagem de erro para destacar a linha contendo o comando errado no programa (quando possível de ser identificada)
- Verifique também a linha anterior àquela que onde foi acusado erro, porque pode ser ela a responsável pelo erro, especialmente se estiver faltando algum ponto e vírgula ou algum parênteses ou chaves.

6.2.4 Funções que podem ser Úteis

Nesta seção apresentaremos algumas funções existentes nas bibliotecas do C que podem ser úteis para você no projeto ou em alguma outra ocasião. Claro que não apresentamos tudo que há disponível. Apenas as funções que consideramos mais úteis.

Uso de funções

Antes de usar uma função é preciso saber como a função está declarada, isto é, quais são os parâmetros que a função recebe e o que a função retorna. A sintaxe de declaração de uma função é: tipoRetorno nomeFuncao(tipo1 param1, tipo2 param2, ...)

Onde: **tipo1 param1, tipo2 param2, ...** são os tipos dos parâmetros e seus respectivos identificadores. Por exemplo, a função cos() da biblioteca math.h tem a seguinte declaração: double cos(double x); Isso significa que ela precisa receber uma variável de ponto flutuante do tipo double para funcionar e que retornará também uma variável do tipo double.

Classificação de caracteres e conversão

Algumas funções para classificação de caracteres e conversão podem ser encontrados na biblioteca ctype.h Para usar qualquer uma das funções a seguir, você tem de lembrar de incluir esta biblioteca, ou seja, de colocar no início do programa: #include <ctype.h> Vejamos alguns exemplos de funções dessa biblioteca. Suponha que elemento é uma variável do tipo caracter ou um inteiro entre 0-255. E saiba que todas as funções especificadas retornam um número inteiro. Este número é zero se o teste for falso e um outro número qualquer se o teste for verdadeiro.

- **int isalnum(elemento);** - Testa se elemento é um carácter alfanumérico (ou seja, contido em 'A'-'Z', 'a'-'z', '0'-'9'). Retorna 0 se não.
- **int isalpha(elemento);** - Testa se elemento é uma elemento (ou seja, um carácter de 'A'-'Z', 'a'-'z'). Retorna 0 se não.
- **int isdigit(elemento);** - Testa se elemento é um dígito (um valor entre '0'-'9'). Retorna 0 se não.
- **int islower(elemento);** - Testa se elemento é uma elemento minúscula ('a'-'z'). Retorna 0 se não.
- **int ispunct(elemento);** - Testa se elemento é um sinal de pontuação. Retorna 0 se não.
- **int isspace(elemento);** - Testa se elemento é um carácter de 'espaço em branco' (0x09-0x0d, 0x20). Retorna 0 se não.
- **int isupper(elemento);** - Testa se elemento é uma elemento maiúscula ('A'-'Z'). Retorna 0 se não.
- **int isxdigit(elemento);** - Testa se elemento é um dígito hexadecimal ('A'-'F', 'a'-'f', '0'-'9'). Retorna 0 se não.

Dessa biblioteca, as duas funções de conversão interessantes são:

**Atenção**

Lembre que, no printf para imprimir um valor do tipo double usamos o especificador %lf

- **int tolower(elemento);** - Converte elemento para uma letra minúscula, se elemento for uma letra maiúscula. Caso contrário, ele continua com o mesmo valor.
- **int toupper(elemento);** - Converte elemento para uma letra maiúscula, se elemento for uma letra minúscula. Caso contrário, ele continua com o mesmo valor.

Funções matemáticas

Diversas funções matemáticas podem ser encontradas na biblioteca math.h . Para fazer uso de qualquer uma das funções aqui descritas, você tem de lembrar de incluir essa biblioteca no início do programa: #include <math.h>

- **int abs(double num);** - Calcula e retorna o valor absoluto do inteiro num. Por exemplo, abs(5.5) → 5
- **double acos(double x);** - Calcula e retorna o arco cujo coseno é x. O valor retornado está entre 0 e pi. O valor de x deverá estar entre -1 e 1.
- **double asin(double x);** - Calcula e retorna o arco cujo seno é x. O valor retornado está entre -pi/2 e pi/2. O valor de x deverá estar entre -1 e 1.
- **double atan(double x);** - Calcula e retorna o arco cuja tangente é x. O valor retornado está entre -pi/2 e pi/2.
- **double ceil(double num);** - Arredonda o valor de num para cima. Por exemplo, ceil(3.2) → 4.0
- **double cos(double x);** - Calcula e retorna o coseno de x. O valor de x deverá estar em radianos.
- **double exp(double x);** - Calcula e retorna o valor da função exponencial com expoente x.
- **double floor(double num);** - Arredonda o valor de num para baixo. Por exemplo, floor(3.2) → 3.0
- **double log(double x);** - Calcula e retorna o valor do logaritmo natural de x.
- **double log10(double x);** - Calcula e retorna o valor do logaritmo de base 10 de x.
- **double pow(double x, double y);** - Calcula e retorna o valor

de x elevado a y. Ou seja, é a função de potenciação. Por exemplo, pow(2.0, 3.0) → 8.0

- **double sin(double x);** - Calcula e retorna o seno de x. O valor de x deverá estar em radianos.
- **double sqrt(double x);** - Calcula e retorna a raiz quadrada de x. Por exemplo, sqrt(9.0) → 3.0
- **double tan(double x);** - Calcula e retorna a tangente de x. O valor de x deverá estar em radianos.

Posicionamento do cursor na tela

Em modo texto padrão, a tela é dividida em uma janela de 25 linhas e 80 colunas. A função gotoxy() permite posicionarmos o cursor em qualquer posição (posX, posY) da tela. Sendo que a posição (1,1) corresponde ao canto superior esquerdo da tela e a posição (80,25) corresponde ao canto inferior direito da tela.

Como a função printf() escreve a partir da posição do cursor, com o uso desta função, podemos escrever em qualquer posição da tela. Dessa forma, gotoxy() é uma função usada para criação de interfaces com o usuário. Sua sintaxe é: void gotoxy(int pos_x, int pos_y); onde: posX é o número da linha e posY é o número da coluna, onde se deseja posicionar o cursor. Para fazer uso dessa função deve-se incluir a biblioteca conio.h no início do programa

Limpeza de tela

A função clrscr() (clear screen) limpa a janela de tela onde o programa está sendo executado e posiciona o cursor na primeira linha e primeira coluna da janela (canto superior esquerdo da janela). Sua declaração é: void clrscr(void);

A função clreol() (clear to end of line) limpa uma linha desde a posição do cursor até o final da linha, porém não modifica a posição do cursor. Sua declaração é: void clreol(void);

Ambas as funções fazem parte da biblioteca conio.h

Término da Execução do Programa

A função (não instrução) exit() pertence a biblioteca stdlib.h e é uma função que termina a execução do programa.

Normalmente, um programa é termina quando se executa a sua última instrução. Porém, pode-se terminar a execução do programa a qualquer momento com o uso desta função. A função `exit()` tem a seguinte declaração: `void exit(int status)`. Onde o parâmetro `status` da função é um valor inteiro que será passado para o Sistema Operacional, especificando o tipo de saída do programa (se com ou sem erros, por exemplo). Ou seja, o código do `status` é usado para indicar qual condição causou a interrupção do programa. Geralmente, o valor 0 (zero) indica que o programa terminou sem problemas, sem erros. Um valor diferente de 0 indica que houve algum tipo de erro.

6.2.5 Refinamentos Sucessivos

A técnica dos refinamentos sucessivos consiste na divisão de um problema complexo em subproblemas, sucessivamente, até torná-lo suficientemente simples. Essa técnica parte do princípio de que resolver um problema complexo é mais fácil se não precisarmos considerar todos os aspectos do problema simultaneamente.

No 1º Refinamento, o problema é repartido em uma seqüência de subproblemas, onde cada subproblema descreve uma ação mais elementar que a ação do problema. A concatenação das ações dos subproblemas deve produzir a ação total do problema.

Se um subproblema ainda não estiver suficientemente simples, no próximo refinamento, ele será repartido da mesma forma que o problema original e, assim, sucessivamente, até se chegar em um subproblema suficientemente simples. Ou seja, comece pensando de forma mais genérica e vá, aos poucos, detalhando, o programa, até que ele pareça, relativamente, simples.

Cada subproblema pode ser implementado e testado separadamente, facilitando o projeto, a construção, a verificação e a manutenção do programa. Para permitir o uso da técnica de refinamento sucessivo são usadas funções.

Vamos dar um exemplo. Suponha um par de inteiros distintos (A e B). Leia valores para esses números e, depois, coloque-os em ordem crescente, imprimindo, ao final, o par ordenado (A, B).

1º Refinamento – refinamento do problema original

leia os valores A e B

ordene-os // veja que aqui não foi definido o que vai ser feito

imprima os valores A e B

2º Refinamento – refinamento sucessivo de ORDENE-OS

```
// especificação do que vai ser feito na etapa ORDENE-OS
se (A > B) entao
    aux ← A
    A ← B
    B ← aux
fimse
```

6.2.6 Erros Comuns na Implementação de Programas

Para facilitar o debug (ato de procurar erros em um programa) dos programas que você for implementar, vamos dar algumas dicas dos erros mais comuns cometidos pelos estudantes ou programadores principiantes.

Confundir atribuição (=) e comparação (==)

Um erro comum e difícil de detectar é consequência do fato de uma atribuição ser considerada uma expressão em C e, além disso, qualquer expressão ter um valor lógico (onde zero equivale a falso e um valor diferente de zero a verdadeiro). Dessa forma, a instrução: `if (a = b) { ... }`; é sintaticamente correta, mas não equivale ao que se pretende. Nesta instrução, o valor de b é atribuído a variável a. Depois disso, a condição do if é avaliada como verdadeira se esse valor for diferente de 0 (zero).

Porém, provavelmente, o que se pretendia com aquela instrução, seria comparar o valor de a com o valor de b. Ou seja, a instrução correta seria: `if (a == b) { ... }`;

Cometer erros na passagem por referência

Certas funções são escritas esperando um endereço em alguns dos seus argumentos (para ser copiado para um apontador), como apresentado nesta unidade e na unidade anterior. Porém, é comum o programador esquecer-se de fazer corresponder aos parâmetros do tipo apontador, um argumento endereço. Para resolver esse problema, temos de lembrar que se os parâmetros de uma função forem apontadores para qualquer tipo, exceto vetores (que não precisam fazer uso do operador de endereço), é obrigatório o uso do operador `&`, para fornecer o endereço de uma variável para o apontador. Por

exemplo, observe o trecho de código a seguir.

```
...
int numero;
float vetor[100];
// criação da função, que tem como parâmetro dois apontadores
void teste(int *indice, float *num)
{ ... }

...
teste(&numero, vetor); // chamada da função teste() , observe a passagem de parâmetro
// note que para o inteiro é passado o endereço de uma variável inteira
// para o outro apontador, é passado um vetor. Logo, vetores não precisam do operador &
...
```

Esquecer os parênteses na chamada de uma função

Em C, todas as chamadas a funções têm de incluir os parênteses, mesmo quando a função não possui parâmetros. Por exemplo:

```
void teste2() // declaração da função
{ ... }

...
teste2(); // chamada à função teste2()
```

Não ter cuidado com a manipulação dos índices dos vetores e matrizes

Você nunca pode esquecer que, em C, os vetores unidimensionais, bidimensionais ou multidimensionais são declarados com uma ou mais dimensões, mas que o compilador não checa essas dimensões. Assim sendo, todo o cuidado no manuseio dos índices válidos de um vetor de qualquer dimensão está a cargo do programador. Também temos de lembrar que um vetor declarado com N elementos, em qualquer das suas dimensões, tem índices que se situam sempre entre 0 e N-1 (lembre que os indices sempre começam do zero).

Esquecer que C é case sensitive

Ser case sensitive (sensível ao caso), quer dizer que C diferencia caracteres minúsculos e maiúsculos. Esquecer disso é um dos erros mais comuns entre iniciantes na linguagem e pode causar muitos problemas. Isso porque uma palavra reservada passa a não ser

reconhecida se não estiver na grafia correta, tal como float e Float (é correta a primeira forma). E variáveis declaradas podem não ser referenciadas se estiverem com a grafia diferente. Por exemplo, Media, media e MEDIA são três variáveis diferentes em C.

Esquecer de terminar as instruções com ponto-e-vírgula

Todas as instruções deverão ser terminadas com ponto e vírgula.

Atribuir Strings com = e não com strcpy

Strings são tipos especiais de dados, uma vez que representam um vetor de caracteres. Dessa forma, temos de lembrar que, para fazer a atribuição de valores a Strings, devemos usar a função strcpy() e não o operador de atribuição (=), como nos outros tipos.

Esquecer de importar alguma biblioteca

Esse também é um erro muito comum. Quando esquecemos de importar alguma biblioteca, algumas instruções do programa podem passar a não serem reconhecidas. Temos de ter atenção nisso!



Aprenda Praticando

Nesta unidade, não tenho um exemplo para lhe mostar, porque a implementação do projeto envolve quase tudo que estudamos até agora. Logo, se necessário, você pode revisar as unidades anteriores.



Saiba Mais

Cantinho do C

<http://www.geocities.com/cantinhodoc/c.html>

Dez dicas para você se tornar bom programador

<http://www.reiwolf.com.br/10dicasprogramador.htm>

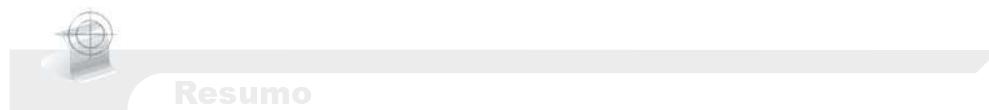
Um curso de C a distância

<http://www.unipacto.com.br/cursos/>
Programando em C (com animações)
<http://informatica.hsw.uol.com.br/programacao-em-c.htm>
Sobre Alocação Dinâmica
<http://mtm.ufsc.br/~azeredo/cursoC/aulas/ca60.html>
http://www.carlosfelgueiras.hpg.com.br/Cursos/LinguagemC/Cap_10.html
<http://www.ead.cpdee.ufmg.br/cursos/C/aulas/ca60.html>
<http://equipe.nce.ufrj.br/adriano/c/apostila/ponte.htm#aloca>
<http://www.inf.ufsc.br/~ine5384-hp/Estruturas.AlocDinamica.html>
<http://www.juliobattisti.com.br/tutoriais/katiaduarte/cbasico009.asp>



O objetivo é que você aproveite o tempo que falta para trabalhar no projeto da disciplina. Dessa forma, recomendamos que você:

1. Forme sua equipe (ou talvez, decida trabalhar sozinho(a)) e comunique aos professores da sua decisão;
2. Baixe do ambiente o enunciado do projeto da disciplina e leia com os seus companheiros de grupo até chegarem a um entendimento do problema e a um consenso de como será realizado o trabalho (divisão do trabalho ou trabalho conjunto).
3. Tire dúvidas sempre que necessário. Não acumule! Há muito a ser feito!



Nesta última unidade, apresentamos uma definição um pouco mais detalhada do que são ponteiros (ou apontadores). Mostramos as operações mais básicas que podem ser realizadas com esse tipo de dado capaz de armazenar um endereço de memória. Foi, também, apresentada a diferença entre os operadores & e *. De forma resumida,

o operador & determina o endereço de uma variável, enquanto que o operador * determina o conteúdo de um endereço. Dessa forma, se temos um apontador chamado pont, teríamos que:

- Usar só o nome do ponteiro fornece o endereço para o qual ele aponta;
- Usar o operador & junto ao nome do ponteiro retorna o endereço do ponteiro;
- Usar o operador * junto ao nome do ponteiro fornece o conteúdo da variável apontada pelo ponteiro.

Adicionalmente, nesta unidade, apresentamos algumas funções que podem ser úteis no dia-a-dia do programador e que fazem parte das bibliotecas que já vêm com o C (padrão ANSI).

Para ajudar no desenvolvimento do projeto, apresentamos as etapas de desenvolvimento do projeto, algumas dicas gerais, erros mais comumente cometidos e apresentamos, rapidamente, a técnica de refinamentos sucessivos.

Pronto, agora é com você! Esperamos que tenhamos lhe ajudado nessa caminhada da “introdução à programação”, porém, nunca esqueça que “programar só se aprende programando”. Esperamos que você tenha tido tempo para estudar, praticar e se dedicar à disciplina e que, talvez, até tomado gosto com essa tal de programação!

Para finalizar, desejo que você seja perseverante, lute, estude, progrida, porque não há nada mais valoroso que você pode ter do que o conhecimento. Quando as coisas ficarem difíceis e você pensar em desistir, lembre, faça como nos programas “divida o problema em partes” e dê um passo de cada vez! Boa Sorte!

Profa. Sandra Siebra

“Quem quiser alcançar um objetivo distante,

tem que dar muitos passos curtos”

(Helmut Schmidt)