Apostila de Estudos de Algoritmos de programação em C
Estudos de casos

Edmundo S. Spoto

UFG/BES

ÍNDICE

CAPÍTULO VIII: ASPECTOS DE IMPLEMENTAÇÃO DE ALGORITMOS

INTRODUÇÃO

Prezados alunos, este material didático abrangerá a teoria e prática de algoritmos de programação de uma forma geral, envolvendo a linguagem de programação C e os aspectos principais da programação e da construção do algoritmo de um problema antes de levarmos para uma linguagem de alto nível (aqui utilizaremos a linguagem C).

Um algoritmo nada mais é do que uma sequência de instruções, escritas ou desenhadas, visando atender a solução de uma tarefa ou um problema. Tais instruções devem ser bem elaboradas de forma a não deixarem dúvidas ou erros de interpretações e que sempre levam aos mesmos resultados. Se o problema for matemático devemos primeiro desenvolver o raciocínio lógico da construção da equação matemática conforme solicitada na descrição do problema, podendo existir um modelo formal de execução ou até mesmo algorítmico que sempre leva ao mesmo resultado. Depois de realizado o algoritmo devemos passar para uma escrita em uma linguagem de alto nível para que o computador siga o raciocínio construído no algor timo e apresente as soluções dos problemas com as entradas sugeridas pelo mesmo.

Desta forma, este material estará induzindo a forma de construção de algoritmos sempre que uma nova atividade surja, o ideal e podermos no futuro resolver todos tipos de problemas de forma mais simples e seguindo sempre a construção do algoritmo conforme o mundo real faria.

O material também apresentará um estudo dos principais aspectos da programação utilizando a linguagem C, estudando seus tipos primitivos, operadores matemáticos e lógicos, bem como comandos de decisão, repetição e outros. Também estaremos fazendo um estudo de tipos de dados homogêneos em C armazenados em estruturas unidimensionais (vetores) e multidimensionais (matrizes), bem como criação de tipos de dados heterogêneos (registros) podendo armazenar nestas mesmas estruturas dadas. O uso de arquivos para gravar a informação em disco sempre que houver necessidades.

Para melhorar a programação serão dadas o uso de modularização com uso de procedimentos e funções, tratando as passagens de parâmetros das variáveis podendo ser passagem por valor ou por referência, conforme a necessidade requerida pelo problema.

Para encerrar o estudo serão apresentados aspectos de desenvolvimentos de algoritmos e programação utilizados no mundo real, com uma visão geral de cada tipo para que possamos classificar quais tipos de programação serão necessárias em determinados problemas existentes.

No final deste material serão colocados diversos problemas para serem estudados pelos estudantes durante o curso. Os resultados serão postos posteriormente pelo professor em um local de download.

Espero que todos tenham um bom aproveitamento neste estudo.

Obrigado!

CAPÍTULO I

INTRODUÇÃO AO CONCEITO DE ALGORITMO

1 – Introdução

Inicialmente para entendermos o que é um algoritmo, vamos supor que temos que explicar os passos de uma tarefa como trocar um pneu de um carro. As etapas da descrição neste caso é primeiro verificar se existe um pneu de reserva. Se não existir não podemos continuar a trocar. Existindo temos que primeiro tirar o pneu no porta-malas ou no local onde ele está colocado, em seguida soltar os parafusos da roda danificada, depois seguir passo a passo cada etapa que nos leva a efetuar a troca até guardar o pneu danificado etc.

O algoritmo de um problema possui 3 etapas fundamentais para serem observadas, inicialmente entender o que o problema solicita realizar, em geral ele das informações que devem ser consideradas na construção do algoritmo em si. Principalmente se houver conversões de valores que serão utilizados e como esses valores devem ser trabalhados no problema. Neste caso separaremos as etapas em: 1. Entradas de Informações; 2. Resolução dos problemas utilizando as dicas dadas pela descrição do problema, esta etapa poderá requerer vários cuidados pois pode envolver uma análise logica entre as variáveis que serão envolvidas, bem como comandos de repetição ou decisão. Em 3. Saídas da informação seguindo a solicitação do problema. Alguns cuidados específicos serão tratados no decorrer de cada capítulo em suas especificidades.

Então pode-se dizer que Algoritmo é um conjunto finito de regras que descrevem uma sequência de etapas visando resolver um determinado problema. Pode-se dizer que é uma sequência de raciocínio, instruções ou operações visando alcançar uma solução para o problema, sendo necessário que os passos sejam finitos e executados sistematicamente até atingir seu objetivo (ASCENCIO, 2012). Esta descrição pode ser realizada com uma descrição "*portugol*" (na linguagem portuguesa) ou até mesmo na linguagem inglesa, ou por uma representação gráfica (fluxograma), onde as atividades dos algoritmos possuem desenhos que representam as etapas de inicialização, comandos de atribuição, comandos aritméticos, comandos de decisão, entre outros. Se tratar de um problema complexo, existem técnicas que devem ser observadas, que é dividir para conquistar (quebrar em pequenos problemas simples até alcançar o objetivo de todo problema existente). Essas técnicas serão observadas em capítulos mas a frente quando for tratado a modularização de um problema.

Para aprender trabalhar com algoritmos é necessário praticar vários tipos de problemas da mesma forma que estudamos matemática (fazendo exercícios). Se apenas estuarmos como fazer pode surgir um problema que exija algo que não foi estudado e termos dificuldades em realizá-lo.

A origem do Algoritmo se deu na idade média, trata-se de uma palavra latinizada, derivada do nome de Al Khowarizmi, matemático árabe do século 9. Ele surgiu da necessidade de fazer cálculos sem o auxílio de ábacos, dedos e outros recursos. Até então, a estrutura dos cálculos esteve associada às ferramentas que havia à mão: pedras sobre o chão, varetas de bambu, a calculadora de manivela, a régua de cálculo e, por fim, a calculadora. É resultado de técnicas de cálculo que levaram séculos para se desenvolver. Também é usado na computação para prever a solução de um problema, antes de levar a solução para uma linguagem de alto nível como C, Pascal, Java, Phyton, etc.

SUGESTÃO DE LEITURA: Um livro bom para ser consultado é o livro de Algoritmos, teoria e prática (Cormen et. all, 2012), e (ASCENCIO, 2012).

2 – O que é um algoritmo e como funciona

Os algoritmos contribuíram para a evolução tecnológica vista nas últimas décadas e são cada vez mais complexos, com o objetivo de entender o comportamento humano na internet, em especial, nas redes sociais. Conhecê-los pode contribuir para melhorar a estratégia digital de uma empresa. Ele antecede a implementação do código, bem como ajuda a melhorar o entendimento do problema antes de colocar em prática em uma linguagem de programação de alto nível.

O algoritmo descreve uma sequência de raciocínio, instruções ou operações que visam alcançar um determinado objetivo. Um exemplo fácil de entender é quando temos que calcular uma equação do segundo grau onde o problema entrará com valores de a, b e c. A equação do 2º Grau é descrita como $ax^2 + bx + c = 0$. O coeficiente a multiplica x^2, o coeficiente b é o número que multiplica x e o coeficiente c é um número real.

Tipos de Algoritmos encontrados na literatura de forma descritiva: Receita de execução de um problema em passos consecutivos ou como uma descrição de um código em pseudo-código (linguagem natural).

Existem vários tipos de algoritmos, que podem representar o mesmo problema através de descrições de etapas descritivas, como uma receita ou um guia de execução para elaborar uma determinada tarefa. No exemplo da Equação do 2º grau pode se descrever no formato de receita, descrevendo as etapas, passo a passo, conforme segue abaixo:

Passo1 – Leia os valores de (a, b, c) sendo números reais (podendo ser números com casas decimais). O valor de “a” não pode ser 0, senão não será uma equação do 2º Grau.

Passo2 – Calcular o valor do Delta, tendo em vista que a formula para calcular os valores das raízes é dada pela Figura 1. Onde a equação dentro da raiz quadrada é denominada de Delta, a qual determina se haverá 2 raízes, 1 raiz ou nenhuma raiz. Delta = b^2 + 4ac

Passo3: Se Delta for > 0 existirão 2 raízes.

X1 = (-b + sqrt(delta)) / 2*a e X2 = (-b - sqrt(delta)) / 2*a

Passo4: Se Delta = 0 - existirá apenas 1 raíz.

X = (-b) / 2*a;

$$X = \frac{-b \pm \sqrt{b^2 - 4 \cdot a \cdot c}}{2 \cdot a}$$

Figura 1 – Cálculo das raízes da equação do 2º Grau

Passo5: Se Delta < 0 como não existe raiz quadrada de valor negativo, não existe raiz no mundo real.

Passo6: Exibir o Resultado final. Imprimir o valor das raízes caso houver ou uma mensagem que não existe raiz no mundo real. E finalizar o algoritmo.

Após analisarmos os algoritmos acima, inicia-se a preparação do programa em uma linguagem de alto nível, ainda não falado neste material, mas em breve mostraremos a montagem do código para a resolução de cálculo das raízes de uma equação do 2º grau.

Existe ainda um tipo de algoritmo que pode ser descrito com um pseudocódigo (representação de um código, mas em linguagem natural) no caso utilizaremos a linguagem em português. No Exemplo dado do cálculo da equação do 2º grau, faremos um algoritmo em pseudocódigo. Na época da programação estruturada, onde iniciávamos com o projeto baseado em fluxo de dados (DFD) no final do projeto eram gerados os pseudocódigos de cada funcionalidade visando deixar preparado para passar para codificação posteriormente.

Na representação do pseudocódigo temos algumas palavras chaves, que agem próximas aos comandos padrões existentes nas linguagens de programação. Vamos mostrar um exemplo de um algoritmo em pseudocódigo para o mesmo problema da Equação do 2º Grau, conforme apresentado na Figura 2.

```
/* Algoritmo de cálculo da equação do 2o Grau
      Autor: Edmundo Sérgio Spoto
      Data: 16/set/2021
*/

var
 a, b, c, delta, x1, x2: reais
inicio
 escreva("a = ");
 leia(a);
 escreva("b = ");
 leia(b);
 escreva("c = ");
 leia(c);
 delta = b*b - 4*a*c;
 se (delta< 0) então
   escreva("Delta = ", delta);
   escreva("Nao existe reizes reais");
 senão se (delta ==0) então
        x1 = (-b/2*a);
        escreva("Delta = ", delta);
        escreva("x1 = x2 = ",x1:6:2);
      senão   x1 = (-b - raizq(delta)/2*a);
           x2 = (-b + raizq(delta)/2*a);
           escreva("Delta = ", delta);
           escreva("x1 = ",x1:6:2);
           escreva("x2 = ",x2:6:2);
      fimse;
 fimse;
fimalgoritmo;
```

Figura 2: Exemplo de um algoritmo em pseudocódigo

Observe que o algoritmo em pseudocódigo é uma fase anterior a codificação, é uma preparação da organização para se iniciar a programação numa linguagem de alto nível.

O terceiro tipo de algoritmo é através de figuras geométricas que pode ser representado todas as instruções de um problema através de um fluxograma.

3 – Descrição de Algoritmos gráficos (fluxograma)

A representação de um algoritmo também pode ser através de um fluxograma, que são desenhos que possuem figuras que representam os tipos de instrução capturadas na interpretação do problema. Para interpretar um fluxograma temos que primeiro entender as formas de cada figura e seus significados.

A ideia básica desta forma de representação dos algoritmos é empregar figuras geométricas na representação de cada instrução (ou passo) que compõe o algoritmo. Existe um conjunto específico de símbolos empregados na elaboração dos fluxogramas. Estes símbolos são definidos pela norma ISO 5807 (ISO, *International Organization for Standardization*). Apresentam-se na Figura 3 como os mais importantes.

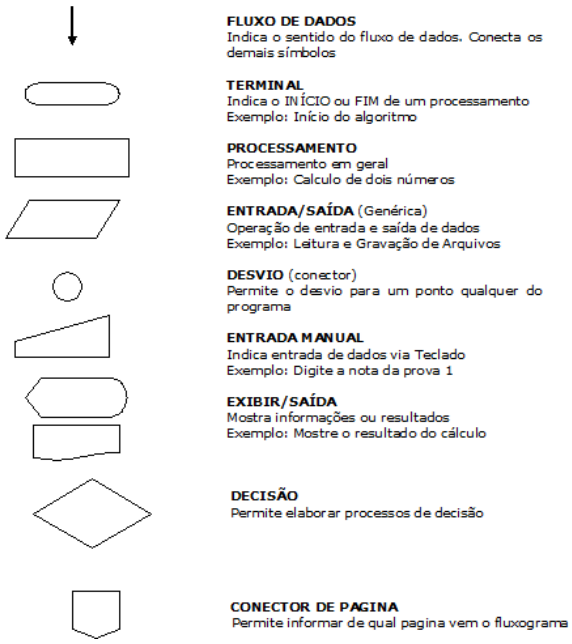


Figura 3: Representação de um fluxograma

(Extraído: https://www.devmedia.com.br/fluxogramas-diagrama-de-blocos-e-de-chapin-no-desenvolvimento-de-algoritmos/28550)

O fluxograma é muito utilizado na engenharia como uma forma de representar os passos de um algoritmo de qualquer tipo de execução de problemas, também podendo ser usado para a computação. Existem réguas específicas que facilitam a confecção de um fluxograma, com as figuras que representam as tapas do fluxograma. No exemplo do cálculo da equação do 2º grau ficaria conforme a Figura 4.

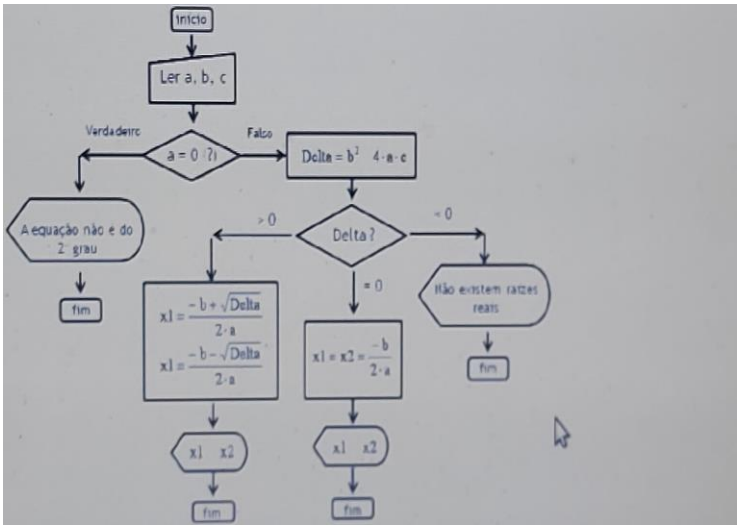


Figura 4: Representação de um fluxograma da equação do 2º grau

(exemplo do Prof. José Gonçalves Cruz - UFSCAR)

Depois de verificado com um caso de teste inicia-se a implementação do código visando escrever o programa em uma linguagem de alto nível para o computador (linguagem C). Um caso de teste é supor uma entrada para a, b e c e verificar seu resultado final e se o fluxo exerce sua execução de maneira correta.

4 – Os principais benefícios

A evolução e aprimoramento dos algoritmos nas últimas décadas contribuíram também com a melhoria dos comandos, e uma evolução natural das linguagens de programação. Um grande benefício de escrever primeiro um algoritmo para a resolução de um ou vários problemas, é uma maneira de documentar as principais funcionalidades de um programa a ser construído. A programação baseada em Objetos por exemplo, teve uma grande evolução a partir da década de 90 com o surgimento da UML (Linguagem de modelagem Unificada) que são diagramas de projetos bem definidos que evoluem desde a fase de análise de projeto a ser construído até sua criação do diagrama de classes. Existem ferramentas de desenvolvimento de projetos que ajudam muito na sua confecção. Da mesma forma todo algoritmo contribui em realizar uma análise antes de colocar no computador o seu código, evitando que algo falhe durante a sua execução.

Nos casos dos algoritmos conforme apresentados anteriormente é importante entender e compreender os conceitos relacionados a eles durante sua criação, focando principalmente nas entradas de dados, os tipos que serão adotados para as variáveis, bem como estabelecer uma forma de conversão entre as variáveis quando esta exigir uma conversão de valores, por

exemplo entra em quilometro e deve ser convertida para metros e ajustando assim todos os parâmetros necessários para o seu melhor funcionamento. Bem como é importante saber como as saídas dos resultados devem ser apresentados aos usuários, adotando uma informação coerente com o esperado por eles.

Alguns algoritmos também podem ter a necessidade de filtrar os seus resultados para diferentes tipos de usuários, essa característica pode exigir diferentes tipos de funções visando atender diferentes níveis de usuários e isso pode contribuir em se pensar em suas execuções ainda na etapa de algoritmo, para depois levar para a codificação. Entre outras formas de segurança da informação que poderão existir em diferentes tipos de algoritmos.

Existem algoritmos que são complexos e precisam ter estratégias de modularização de suas funcionalidades, criando-se etapas construtivas para se atingir um determinado objetivo exigido pelo problema. Desta forma o algoritmo pode contribuir em preparar as funcionalidades principais em etapas que se tornem bem mais fácil seu desenvolvimento, depois interligar essas funcionalidades para montar o algoritmo todo, esse benefício poderá facilitar muito e evitar retrabalhos futuros. Quando trabalhamos com uma estrutura estática ou dinâmica em uma implementação é fundamental termos suas inicializações, etapas de leituras, depois etapas de consultas, etapas de exibição de resultados, etapas de realização de alguma alteração e assim por diante, onde a construção do algoritmo ajuda a quebrar cada etapa dessa em um pequeno algoritmo (procedimento isolado) onde torna o entendimento mais fácil e alcançando o objeto que no seu início era complexo.

Todo problema a ser resolvido pode envolver outras tecnologias (exemplo banco de dados, gerador de imagens, etc) que serão observadas durante a construção dos algoritmos, e tais tecnologias podem ser acoplada como uma caixa preta inicialmente, e posteriormente serão também desenvolvidos e resolvidos para se ajustar ao desenvolvimento do programa como um todo.

Existem vários tipos de algoritmos que podem ser comprados como um serviço de uma necessidade que serão usadas no futuro em nosso programa, como por exemplo carrinho de compra, pagamentos por cartão, segurança de redes, entre outros. Esses não serão comentados neste curso pois o objetivo deste é ajudar o estudante a construir seu próprio algoritmo e posteriormente sua programação.

Agora que já sabemos o que é um algoritmo podemos dizer que os principais benefícios é ajudar na mediação para quem ele se destina e como deve ser realizado seu uso. Cada usuário podem ter críticas diferentes de um mesmo algoritmo, pois é natural ter usuários que

gostam mais de teclas de atalho, outros de ler todas informações, e outros que possuem suas próprias formas de trabalhar. Pode-se também analisar com detalhes todas exigências levantadas antes de sua construção. Dizem que muitos erros podem ser corrigidos ainda em fase de análise do algoritmo e não deixar para serem verificadas posteriormente para após sua implementação.

Outro benefício importante é entender quais tipos de comandos poderão ser utilizados para a execução de um determinado laço ou de várias decisões. A etapa de algoritmo é importante levantar se pode ser utilizado recursividade ou iteratividade antes de por em prática. Alguns perigos hoje levantados por muitas empresas é que existem pouca vontade de se desenvolver algoritmos que precede a implementação, e muitas empresas de desenvolvimento no mercado partem para a implementação do código com poucas análises ou quase nenhuma dos algoritmos realizados.

5 – Tipos de problemas matemáticos

Existem muitos desenvolvedores que possuem um modelo de algoritmos sempre que for desenvolver um problema matemático. É muito comum que isso aconteça dado sua experiência e sua percepção do domínio do conteúdo para se desenvolver. Outros já utilizam mais os comandos e trechos de execuções explorando suas habilidades de programação e pouca de uso de formalismos para resolver seus problemas matemáticos. Pode se dizer inicialmente que os principais cuidados que temos que ter são:

a) Ler o enunciado: Leia atentamente o enunciado do problema, quantas vezes for necessário até compreender todas suas informações passadas;

b) Coletas de Dados: Anote todas informações relevantes do problema, bem como todas equações e adaptações que deverão ocorrer da questão a ser resolvida;

c) Defina um procedimento: depois de ter compreendido o problema e já ter coletado os dados, defina um procedimento de sua execução, podendo quebrar as equações muito complexas em partes se for necessário, ou a sequência que elas devem ser resolvidas até obtenção do resultado.

d) Resolução do problema objeto: aqui é momento de resolução do problema e validação de seu resultado, observando as aproximações exigidas etc.

e) Busque problemas similares: Encontre problemas parecidos e verifique se a estratégia utilizada se aplique para esses problemas;

f) Gere etapas de execução: se houver necessidade pode-se gerar etapas de execução ou criação de funções e pequenos procedimentos que podem facilitar a melhorar o

entendimento de sua execução e obtenção da solução. Em geral isso é necessário quando se tratar de problemas complexos e que envolvam vários tipos de fórmulas difíceis.

g) Elabore as saídas das informações: todo resultado pode ter um formato específico levantado pelos usuários ou proprietários do problema de como eles pretender obter tais informações resultantes. Procure criar saídas mais próximas do mundo real solicitado, assim oberemos mais sucessos com os usuários.

Alguns cuidados ainda com as expressões matemáticas em linguagem C por exemplo que existem operações simples entre variáveis inteiras que retornam apenas resultados inteiros por exemplo A **= 10/3** resultará em um valor inteiro **3** e o esperado seria que seu resultado fosse um valor real com duas casas decimais. Esses problemas iremos abordar na linguagem C quando for trabalhado os tipos primitivos de variáveis. No algoritmos temos apenas que apontar tais necessidades para alertar o programador a arrumar tais problemas.

Outras informações que não são pertinentes levantar nos algoritmos quando trabalharmos com expressões matemáticas é da necessidade do uso ou não da biblioteca de operações matemática em C (math.h). Porém o programador tem que saber quando existe ou não a necessidade de colocar esta biblioteca.

6 – Exemplos e resultados

Vamos listar alguns problemas a serem gerados algoritmos e posteriormente um programa computacional que possa ser resolvido na Linguagem de Programação C. Neste Capítulo somente serão desenvolvidos os algoritmos, mais para frente faremos o uso destes algoritmos para a geração do código para a solução computacional.

P1 - Consumo de energia (grau de dificuldade 1)

Sabendo-se que 100 kW de energia custam 70% do salário mínimo, escreva um algoritmo que leia o valor do salário mínimo e a quantidade de kW gasta por uma residência. Calcule e imprima:

- o valor em reais de cada kW;
- o valor em reais a ser pago pelo consumo da residência;
- o novo valor a ser pago pela residência com um desconto de 10%.

Entrada

O programa deve ler o valor do salário-mínimo e a quantidade de kW gasta por uma residência. Ambos os valores são reais.

Saída

O programa deve imprimir três linhas contento o texto:

- Custo por kW: R$ x.xx
- Custo do consumo: R$ x.xx
- Custo com desconto: R$ x.xx

Caso de Teste

Entrada
81 3.54
Saída
Custo por kW: R$ 0.57 Custo do consumo: R$ 2.01 Custo com desconto: R$ 1.81

P2 - Custo da Lata de Cerveja (dificuldade 1)

Um fabricante de latas deseja desenvolver um programa para calcular o custo de uma lata cilíndrica de alumínio, sabendo-se que o custo do alumínio por m 2 é R$ 100,00.

Entrada

O programa deve ler dois valores na entrada: o raio e a altura da lata. Ambos os valores correspondem a valores em metros. Cada valor ocorre em uma linha diferente na entrada.

Saída

O programa deve imprimir a frase: O VALOR DO CUSTO E = XXX.XX, onde XXX.XX é o valor do custo da lata. Logo após o valor do custo da lata o programa deve imprimir o caractere de quebra de linha.

Observações

• O seu programa deve utilizar a constante π com o valor aproximado de 3.14159.

• O valor total da área de um cilindro é dada por $A\ t = 2A_c + A_l$, onde A_c é a área do círculo, calculada como: $A_c = \pi r^2$ e A_l é a área lateral do cilindro, computada por $A_l = 2\pi ra$, onde r é o raio e a a altura da lata em metros.

Caso de teste

Entrada
0.02 0.09
Saída
O VALOR DO CUSTO E = 1.38

P3- Cálculo do Determinante de uma Matriz Quadrada de Duas Dimensões (dificuldade 1)

Fazer um algoritmo tal que dados os quatro elementos de uma matriz 2 × 2, calcule e escreva o valor do determinante desta matriz.

Entrada

O programa deve ler os quatro elementos a, b, c e d que formam uma matriz quadrada bidimensional. Há um valor por linha de entrada. Cada valor corresponde a um número real (float).

Saída

O programa deve imprimir uma linha contendo a frase: O VALOR DO DETERMINANTE E = X, onde X é o valor do determinante computado pelo seu programa e deve conter no máximo 2 casas decimais.

Após o valor do determinante, o algoritmo deve fazer o cursor avançar para a próxima linha.

Observações

Dada uma matriz quadrada bidimensional M, o determinante de M det(M) é definido por det(M) = a*d – b*c;

$$M = \begin{bmatrix} a & B \\ c & D \end{bmatrix}$$

Caso de teste

Entrada
4 3 5 4
Saída
O VALOR DO DETERMINANTE E = 1.00

P4 – Decolagem de um avião (dificuldade 2)

Escrever um algoritmo que leia a massa (em toneladas) de um avião, sua aceleração (m/s 2) e o tempo (s) que levou do repouso até a decolagem. O algoritmo deve calcular e escrever a velocidade atingida (Km/h), o comprimento da pista (m) e o trabalho mecânico realizado (J) no momento da decolagem.

Dicas

- v = velocidade; a = aceleração; t = tempo;
- m = massa;
- s = espaço percorrido;
- 1 m/s = 3,6 Km/ h;
- $v = a * t$;
- $s = at^2/2$;

• W = trabalho mecânico realizado;
• W = $mv^2/2$;
• Utilize valores do tipo double deve ser lido com "%lf"
• A massa utilizada no trabalho é em Kg

Entrada

O algoritmo deve ler três linhas de entrada. A primeira linha contém um valor do tipo double representando a massa do avião em toneladas. A segunda linha, contém um valor do tipo double correspondente à aceleração de avião. A terceira, linha contém um valor do tipo double correspondente ao tempo em segundos gastos na decolagem.

Saída

O Algoritmo deve escrever três linhas. A primeira, contém a frase: VELOCIDADE = x, onde x é o valor da velocidade do avião em Km/h. A segunda, contém a frase: ESPACO PERCORRIDO = y, onde y corresponde ao espaço em metros percorrido pelo avião durante a decolagem. A terceira linha contém a frase: TRABALHO REALIZADO = z, onde z corresponde ao valor do trabalho em Joules, realizado pelo avião durante a decolagem. Os valores de x, y e z devem ser do tipo *double* e devem conter duas casas decimais e após esses valores deve vir o caractere de quebra de linha, movendo o cursor para a próxima linha.

Soluções:

S1– Algoritmo do Cálculo de consumo de Energia

```
/* Algoritmo do cálculo do Consumo de energia
Autor: Edmundo S. Spoto
Data: 16/09/2021
*/

var salmin, qtdkw, custokwh:real;

inicio
	escreva("salario minimo: ");
	leia(salmin);
	escreva("Quantidade em Kw: ");
	leia(qtdkw);
	custoKwH=(salmin*0.7)/100;
	escreva("Custo por kW: R$ ", custoKwH:6:2);
	escreva("Custo do consumo: R$ ", custoKwH*qtdkw:6:2);
	escreva("Custo com desconto: R$ ", custoKwH*qtdkw*.9:6:2);
fim_algoritmo;
```

S2: Algoritmo do calulo do custo da lata de cerveja

```
/* Algoritmo do calculo de latas de cerveja
autor: Edmundo S Spoto
Data: 16/09/2021
*/
```

```
Contante PI=3.14159
var r, a:real;
var custo: real;

inicio
        escreva("Raio da lata: ");
        leia(r);
        escreva("Altura da lata");
        leia(a);
        custo = 100*( 2*(PI*r*r) + (2*PI*r*a));
        escreva("O VALOR DO CUSTO E = ", custo:6:2);
fim_algoritmo;
```

S3 – Algoritmo do Cálculo de Determinante de matriz quadrada

```
/* Algoritmo que calcula o determinante de uma matriz de ordem 2x2
Autor: Edmundo S. Spoto
Data: 16/09/2021
*/

var a, b, c, d: inteiro;
var det:real;

inicio
    escreva("a: ");
    leia(a);
    escreva("b: ");
    leia(b);
    escreva("c: ");
    leia(c);
    escreva("d: ");
    leia(d);
    det= a*d - b*c;
    escreva("O VALOR DO DETERMINANTE E = ",det:6:2);
fim_algoritmo;
```

S4 – Algoritmo de Decolagem de um Avião

```
/* Algoritmo em pseudocódigo para resolver o problema D decolagem de um avião
Autor: Edmundo S Spoto
Data: 16/set/2021
*/

var massa, a, t, velocidade, trabalho,espaco:Double;

inicio
   //entrada de dados
    escreva("Massa: ");
        leia(massa);
```

```
        escreva("Aceleracao: ");
        leia(a);
        escreva("Tempo: ");
        leia(t);
        // Calculo da velocidade
        velocidade = a*t;
        escreva("VELOCIDADE = ", velocidade*3.6:6:2);  //transformação de m/s para km/h
        espaco=(a*t*t)/2.0;   //calculo do espaço percorrido para a decolagem
        escreva("ESPACO PERCORRIDO = ", espaco:6:2);
        trabalho=(massa*1000*velocidade*velocidade)/2.0; //calculo do trabalho em jaule
        escreva("TRABALHO REALIZADO = ", trabalho:6:2); //escrita do trabalho com 2 casas decimais
fim_algoritmo;
```

CAPÍTULO II
TIPOS DE DADOS E OPERAÇÕES PRIMITIVAS.

1 – Introdução

Nesta sessão iremos estudar os tipos de dados primitivos em C, bem como os operadores matemáticos e lógicos e iniciar nossos estudos na programação na Linguagem C. A linguagem C nasceu na década de 70. Seu inventor, Dennis Ritchie, implementou-o pela primeira vez usando um DEC PDP-11 rodando o sistema operacional UNIX. O C é derivado de uma outra linguagem: o B, criado por Ken Thompson. O B, por sua vez, veio da linguagem BCPL, inventada por Martin Richards.

O C é uma linguagem de programação genérica que é utilizada para a criação de programas diversos como processadores de texto, planilhas eletrônicas, sistemas operacionais, programas de comunicação, programas para a automação industrial, gerenciadores de bancos de dados, programas de projeto assistido por computador, programas para a solução de problemas da Engenharia, Física, Química e outras Ciências, etc. O Linux Sistema Operacional muito utilizado, foi todo construído pela linguagem C. Utilizaremos a estrutura do ANSI C, que é o padrão da Linguagem C. As bibliotecas utilizadas pelos compiladores, são diferentes em Sistemas Operacionais distintos, por isso sempre que migrar a linguagem de um ambiente para outro é necessário recompilar o programa. Quando não houver equivalentes para as funções em outros sistemas, apresentaremos formas alternativas de uso dos comandos.

Para o aprimoramento do uso da linguagem C é importante que o estudante use o máximo de tempo para fazer os exercícios, problemas que serão postados neste material. Só assim o conhecimento e a habilidade na programação em C será melhorada com o tempo.

SUGESTÃO DE LEITURA: Herbert Schildt, C Completo e Total, Ed. Person, 3ª Edição, 1997. Para estudos da Linguagem C.

2 – Tipos de dados primitivos

Todo programa deve conter variáveis que vão conduzir a informação, os valores de entrada, bem como controlar os cálculos e operações que forem necessárias. Para isso tais variáveis devem receber um tipo de dado para tornar seus resultados mais apropriado com as necessidades do problema. A Linguagem C possuem 5 tipos de dados primitivo, conforme mostra a Tabela 1. A Linguagem C é “case sensitiva”, se você chamar uma variável de “A” e quiser utilizar outra variável “a” serão diferentes. Isto significa que letras maiúscula e minúsculas são diferentes. Evite de usar nomes de variáveis com palavras reservadas como

(if, else, do, while, for, end, switch, case, goto, entre outras) que compões os comandos da linguagem.

Tabela 1: Tipos primitivos em C

Palavra Chave	Tipo
char	caracter
int	inteiro
float	real precisão simples
double	real precisão dupla
void	vazio (sem valor)

O padrão ANSI C determina somente um intervalo de valores mínimo para cada tipo de dado.

Modificadores de Tipos

Com exceção de **void**, os outros tipos de dados primitivos podem ter modificadores. Os modificadores alteram o tamanho do tipo de dado ou sua forma de representação. Sua utilização faz com que seja possível adequar-se melhor às necessidades de armazenamento de dados em determinados casos. Veja quais são os modificadores na Tabela 2.

Tabela 2: Tipos de dados e modificadores

Palavra Chave	Tipo
signed	caracter
unsigned	inteiro
long	longo
short	curto

Tabela 3: tipos de dados e seus domínios de valores e número de bytes usados

Palavra chave	Tipo	bytes	Intervalo
char	Caracter	1	-128 a 127
signed char	Caractere com sinal	1	-128 a 127
unsigned char	Caractere sem sinal	1	0 a 255
int	Inteiro	2	-32.768 a 32.767
signed int	Inteiro com sinal	2	-32.768 a 32.767
unsigned int	Inteiro sem sinal	2	0 a 65.535
short int	Inteiro curto	2	-32.768 a 32 767
signed short int	Inteiro curto com sinal	2	-32.768 a 32.767
unsigned short int	Inteiro curto sem sinal	2	0 a 65.535
long int	Inteiro long	4	-2.147.483.648 a 2.147.483.647
signed long int	Inteiro longo com sinal	4	-2.147.483.648 a 2.147.483.647
unsigned long int	Inteiro longo sem sinal	4	0 a 4.294.967.295
float	Ponto flutuante com precisão simples	4	3.4 E-38 a 3.4E+38
double	Ponto flutuante com precisão simples	8	1.7 E-308 a 1.7E+308

long double	Ponto flutuante com precisão dupla longo	16	3.4E-4932 a 1.1E+4932

Todo programa em C deve conter uma função main() que é denominado de função principal, por onde o programa inicia e termina. Pode haver outras funções e procedimentos, para compor seu programa quando houver necessidade. Existem várias bibliotecas que são fundamentais para facilitar nossas operações, como biblioteca de entrada e saída (stdio.h), biblioteca que tratam operações matemáticas (math.h), biblioteca de operações para strings (string.h), biblioteca de funções e operadores básicos (stdlib.h), que devem ser estudadas em separadas para ter um melhor aproveitamento de suas funções e atribuições.

Para gerar as operações de entrada e saída utilizaremos a biblioteca <stdio.h> através das funções scanf() e printf(), que serão utilizadas para manipular as informações de entrada e saída. Todo tipo de variável devem utilizar uma diretiva que está apresentada na Tabela 4.

Tabela 4: Diretivas de manipulação de tipos de dados na leitura e gravação

Identificador	**Significado**
%d	Inteiro decimal
%c	Caractere simples
%f	Real simples decimal
%s	Cadeia de caractere (string)
%e	Ponto flutuante (notação exponencial)
%g	Usa o %f ou o %e mais curto
%ld	Inteiro longo (long)
%lf	Real longo (double)
%u	Inteiro decimal sem sinal
%o	Inteiro Octal sem sinal
%x	Inteiro hexadecimal sem sinal
%i	Usado para inteiro independente se a base é decimal octal ou hexadecimal
%%	Quando pretende imprimir %

A função de escrita contida na biblioteca <stdio.h> é utilizada o **printf**, que pode ser usado para escrever uma frase e no final deve ter um comando de mudança de linha “\n”, ou se quiser tabular alguma saída “\t”, entre outros que são apresentados na Tabela 5.

Tabela 5: Constantes de barras invertidas

comando	significado
\b	Retrocesso ("back")
\f	Alimentação de formulário ("form feed")
\n	Nova linha ("new line")
\t	Tabulação horizontal ("tab")
\”	Aspas
\’	Apóstofro
\0	Nulo (0 em decimal)
\\	Barra invertida
\v	Tabulação vertical
\a	Sinal sonoro ("beep")
\N	Constante octal (N é o valor da constante)
\xN	Constante hexadecimal (N é o valor da constante)

Todo programa em C tem uma estrutura conforme mostraremos na Figura 2.1 iniciando com as bibliotecas no formato #include <stdio.h> que pede para incluir no programa a biblioteca de entrada e saída. Todas as bibliotecas que forem utilizadas devem estar nas primeiras linhas do programa antecedendo todas as demais estruturas de código. Vamos fazer uma equação que é entrar com 3 variáveis inteiras e gerar um resultado em ponto flutuantes com 2 casas decimais, sendo: resultado = (a/b)*(a/c)*PI sendo que PI será um valor constante de 3.141516.

```
#include <stdio.h>   //biblioteca de entrada e saída
#define PI 3.141516  //valor de PI a ser adotado na equação
int main() {
  int a, b, c;
  float resultado;
  printf("Entre com os valores de a, b e c: \n");
  scanf("%d %d %d",&a,&b,&c);  /*observe que para colocar o valor na memória é preciso
                                    passar o endereço da variável "&" */
  resultado =(float)a/b;  /*toda divisão entre dois inteiros retorna inteiro para isso utilizaremos
                            um cast (float) para forçar durante a operação que “a” se transforme
                            em float para retornar float. */
  resultado = resultado*(float)a/c;
  resultado = resultado*PI;
  printf("O resultado da operacao e = %.2f\n",resultado);
```

```
return 0;
}
```

Figura 2.1: Programa que executa operações de inteiro resultando em float.

Observe duas coisas para ler variáveis de tipo simples, devemos usar o endereço representado pelo comando “&” (e comercial) antes de cada variável lida na função scanf. A outra coisa a se observar foi o uso do “cast” que é muitas vezes utilizado em operações e comandos que devemos forçar a transformação de um resultado durante a execução. A divisão entre dois tipos inteiros a/b resultaria em um valor inteiro tipo 10/3 resultaria em 3. Para fazer com que o resultado da operação não perca os valores nas casas decimais devemos usar então o cast (float) antes da operação. Lógico que quando a operação é com numero basta usar 10.0/3 que já iria retornar 3.33, mas por se tratar de variáveis foi necessário o uso do cast “(float)a/b” o mesmo entre “(float)a/c”.

Toda impressão de informações realizadas pelo print é necessário colocar no final da frase ou depois das diretivas de valores um “\n” para que após a escrita na tela o cursor mude de linha, faça isso sempre que necessário.

Pode-se criar também um enumerador, uma enumeração é útil quando se deseja utilizar um conjunto determinado de valores que podem estar associados a uma variável.

Quando tenta-se atribuir a uma variável de um tipo enumeração um valor que não faz parte da própria enumeração, o compilador emite uma mensagem de erro. Uma declaração de variável de um tipo enumeração consiste da palavra reservada ***enum*** seguida de uma lista de identificadores entre chaves, e seguido finalmente pelo nome da variável. Por exemplo:

```
enum  {AZUL, VERMELHO, BRANCO, PRETO} cor;
```

declara a variável `cor` como sendo do tipo enumeração que consiste dos valores constantes `AZUL`, `VERMELHO`, `BRANCO` e `PRETO`; estes são os únicos valores que a variável `cor` pode assumir. Outro exemplo dias da semana como

```
enum  {SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA} dsem;
```

declara a variável dsem como sendo do tipo enumeração que consiste dos valores constantes dos dias da semana postado entre as { }.

Evite de usar nomes com acento ou Ç ou outros tipos que podem dar erros na compilação. Mesmo na escrita de alguma frase não utilize acentuação.

SUGESTÃO DE LEITURA: Stroustrup, J. A Tour of C++,2a Edição, Addison Wesley Edition, 2008. É um livro que apresenta as principais peculiaridades da linguagem C dentro de C++. Stroustrup é

o precursor da linguagem C++ seus livros são bem didáticos e importante para quem pretende seguir na programação em C/C++.

3 – Operadores aritméticos

Os operadores aritméticos são usados para desenvolver operações matemáticas. Na Tabela 6, é apresentada a lista dos operadores aritméticos do C:

Tabela 6: Operadores aritméticos em C

Operadores	**Ação**
+	Soma (inteira e ponto flutuante)
-	Subtração ou troca de sinal (unário)
*	Multiplicação (inteira e ponto flutuante)
/	Divisão (inteira e ponto flutuante)
%	Resto de divisão (de inteiros)
++	Incremento (inteiro e ponto flutuante)
–	Decremento (inteiro e ponto flutuante)

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. A soma é um operador binário pois pega duas variáveis, soma seus valores, sem alterar as variáveis, e retorna esta soma. Outros operadores binários são os operadores - (subtração), *, / e %. O operador - como troca de sinal é um operador unário que não altera a variável sobre a qual é aplicado, pois ele retorna o valor da variável multiplicado por -1. O operador / (divisão) quando aplicado a variáveis inteiras, nos fornece o resultado da divisão inteira; quando aplicado a variáveis em ponto flutuante nos fornece o resultado da divisão "real". O operador % fornece o resto da divisão de dois inteiros. Assim seja o seguinte trecho de código:

```
int a = 17, b = 3;
int x, y;
float z = 17. , z1, z2;
x = a / b; y = a % b;
z1 = z / b;
z2 = a/b;
```

ao final da execução destas linhas, os valores calculados seriam x = 5, y = 2, z1 = 5.666666 e z2 = 5.0 . Note que, na linha correspondente a z2, primeiramente é feita uma divisão inteira (pois os dois operandos são inteiros). Somente após efetuada a divisão é que o resultado é atribuído a uma variável float. Os operadores de incremento e decremento são unários que alteram a variável sobre a qual estão aplicados. O que eles fazem é incrementar ou decrementar, a variável sobre a qual estão aplicados, de 1. Então

```
x++;
x--;
```

são equivalentes a

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

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

```
x=23;
y=x++;
```

teremos, no final,

y=23 e

x=24.

Em

```
x=23;
y=++x;
```

teremos, no final, **y=24** e **x=24**. Uma curiosidade: a linguagem de programação C++ tem este nome pois ela seria um "incremento" da linguagem C padrão. A linguagem C++ é igual a linguagem C só que com extensões que permitem a programação orientada a objeto, o que é um recurso extra. O operador de atribuição do C é o =. O que ele faz é pegar o valor à direita e atribuir à variável da esquerda. Além disto ele retorna o valor que ele atribuiu. Isto faz com que as seguintes expressões sejam válidas:

```
x=y=z=1.5; /* Expressao 1 */
if (k=w) ... /* Expressao 2 */
```

A expressão 1 é válida, pois quando fazemos z=1.5 ela retorna 1.5, que é passado adiante, fazendo y = 1.5 e posteriormente x = 1.5. A expressão 2 será verdadeira se w for diferente de

zero, pois este será o valor retornado por k=w. Pense bem antes de usar a expressão dois, pois ela pode gerar erros de interpretação. Você não está comparando k e w. Você está atribuindo o valor de w a k e usando este valor para tomar a decisão.

Para verificar se você entendeu dessas atribuições e incrementos pré e pós interprete o resultado da sequência de operações a seguir:

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

qual seriam os valores de x, y e z no final?

Inicialmente x = 10, y=10. Na operação z=++x; o valor de z=11 e x=11. Na operação x = -x; o valor de x = -11. A operação y++; y=11. No final na operação x=x+y-(z--); x assumirá o valor x=-11, y=11 e z=10.

Outras operações abreviadas existentes em C e C++ são:

```
int x, y, z;
```

x += y; é o mesmo que x = x+y;

x =z; é o mesmo que x = x*z;

z = y+x; é o mesmo que z = z/(y+x);

x %= z; é o mesmo que x = x%z; operação que retorna o resto da divisão entre x e z.

Com esses operadores matemáticos podemos elaborar algoritmos e programas que com expressões matemáticas em problemas do mundo real. No final deste capítulo apresentaremos vários exercícios com expressões matemáticas onde iremos exercitar nosso conhecimento matemático que é muito importante na programação em C.

SUGESTÃO DE LEITURA: Stroustrup, J. A Tour of C++,2a Edição, Addison Wesley Edition, 2008. e BRASIL, Reyolando M. L. R. F; BLTHAZAR, José Manoel; GÓIS, Wesley. Métodos numéricos e computacionais na prática de engenharias e ciências. São Paulo: Blucher, 2015.

4 – Operadores Relacionais e Lógicos

Os operadores relacionais do C realizam comparações entre variáveis. São apresentados na Tabela 7:

Tabela 7: Relação de operadores Relacionais em C.

Operador	Ação
>	Maior do que
>=	Maior ou Igual a
<	Menor do que
<=	Menor ou Igual a
==	Igual a
!=	Diferente

Os operadores relacionais retornam verdadeiro (1) ou falso (0). Uma forma de verificar o uso dos operadores relacionais é entender que seus usos devem estar em comandos de decisão ou laços, que serão vistos em capítulo mais a frente. Faremos um exemplo onde a operação será colocada nos comandos de escrita, onde serão impressos 1 se for verdadeiro ou 0 se for falsa.

```
#include <stdio.h>

int main() {
  int x,y,z;
  printf("\nEntre com os valores de X, Y, Z: ");
  scanf("%d %d %d",&x,&y,&z);
  printf("\n%d==%d -> resultado %d\n",x,y,x==y);
  printf("\n%d>%d -> resultado %d\n",x,y,x>y);
  printf("\n%d<%d -> resultado %d\n",x,y,x<y);
  printf("\n%d==%d -> resultado %d\n",x,z,x==z);
  printf("\n%d>%d -> resultado %d\n",x,y,x>z);
  printf("\n%d<%d -> resultado %d\n",x,y,x<z);
  printf("\n%d==%d -> resultado %d\n",y,z,y==z);
  printf("\n%d>%d -> resultado %d\n",y,z,y>z);
  printf("\n%d<%d -> resultado %d\n",y,z,y<z);
return 0;
}
```

Pode-se observar que o resultado dos operadores relacionais é sempre igual a 0 (falso) ou 1 (verdadeiro). Para fazer operações com valores lógicos (verdadeiro e falso) temos os operadores lógicos mostrados na Tabela 8:

Tabela 8: Operadores Lógicos em C

Operador	Ação
&&	And (E)
||	OR (OU)
!	Not (NÃO)

Usando os operadores relacionais e lógicos podemos realizar uma grande quantidade de testes. Na Tabela 9, são apresentadas uma tabela verdade entre as combinações dos operadores, e quais resultados se alcançam.

Tabela 9: Tabela Verdade entre os operadores Lógicos em C

P	Q	P ou Q	P E Q
F	F	F	F
F	V	V	F
V	F	V	F
V	V	V	V

Para ilustrar o uso dos operadores lógicos vamos criar um programa onde entraremos com dois valores de A e B, podendo ser 0 (falto) ou 1 (verdadeiro). Como ainda não entramos em comando de decisão faremos no ato da escrita de valores.

```
#include <stdio.h>
int main()
{
 int A, B;
 printf("informe dois n˙meros(cada um sendo 0 ou 1): ");
 scanf("%d%d", &A, &B);
 printf("%d AND %d resultado %d\n", A, B, A && B);
 printf("%d OR %d resultado %d\n", A, B, A || B);
 printf("NOT %d resultado %d\n", A, !A);
 return 0;
}
```

Também existem operadores lógicos denominados de bit a bit. A Linguagem C permite que se faça operações logicas "bit-a-bit" em números. Ou seja, neste caso, um número é representado por sua forma binária e as operações são feitas em cada bit dele. Imagine um número inteiro de 16 bits, a variável i, armazena o valor 2. A representação binaria de i, será: 0000000000000010 (contendo 16 dígitos, sento 14 zeros a esquerda seguido de 1 e 0). Pode-se fazer operações em cada um dos bits deste número. Por exemplo, se fizermos a negação do número (operação binária NOT, ou operador binário ~ em C), isto é, ~i, o número se

transformará em 1111111111111101. As operações binárias ajudam programadores que queiram trabalhar com o computador em "baixo nível". As operações lógicas bit a bit só podem ser usadas nos tipos char, int e long int. Os operadores são:

Tabela 10: Operadores lógicos bit a bit

Operador	Ação
&	AND
|	OR
^	XOR (ou exclusivo)
~	NOT
>>	Deslocamento de bits a direita
<<	Deslocamento de bits a esquerda

Os operadores &, |, ^ e ~ são as operações lógicas bit a bit. Para se utilizar os deslocamentos devemos fazer:

valor>>valor_a_ser_deslocado_a_direita

valor<<valor_a_ser_deslocado_a_esquerda

O valor_a_ser_deslocado_a_direita indica o quanto cada bit será deslocado. Por exemplo, para a variável i anterior, armazenando o número 2:

i << 3; //lembrando que i vale 0000000000000**1**0

i terá a representação binária no valor: 000000000001000, isto significa que o valor armazenado em i passa a ser igual a 16.

Em nossas programações poderá ser útil quando formos trabalhar com valores em binário e tivermos a necessidade de operar com esses operadores.

5 – Domínios dos tipos de dados em C

Antes de prosseguirmos é importante alertar sobre os domínios que cada tipo trabalha, e sempre tomarmos cuidados com o estouro desses domínios. Conforme foi visto na Tabela 3 os intervalos que representam o campo de domínio em que cada tipo de dados trabalha, facilitará para nossas operações matemáticas quando necessários. Para isso quando houver necessidade de um resultado aumentar a capacidade de domínio de seu resultado podemos utilizar tipos maiores, conforme visto.

Algumas transformações de tipos para tipos também são importante ter a preocupação sempre que houver essa necessidade, e ver quais tipos poderemos ter menos perdas

possíveis ao fazer uma adaptação em casos de programas que envolvam banco de dados, ou valores já pré estabelecidos no mercado.

O que acontecerá se usarmos um valor muito grande em uma variável inteira. O compilador C poderá nem dar um alerta durante a compilação, porém o erro do valor surgirá no resultado, saindo um valor não esperado.

O programa a seguir induzirá que um resultado onde estoura o domínio da variável c. A operação é c = (a*a)*(b*b); sendo a = 1000 e b=5000;

```
#include <stdio.h>

int main() {
  int a=1000, b=5000, c;
  c = (a*a)*(b*b);
  printf("Resultado da operacao: %d\n",c);
return 0;
}
```

Resultado da operacao: -1004630016. Observe que o valor do resultado deu negativo o que indica que estourou o limite máximo da variável c que é inteira e vai até $2^{32} – 1$. Que dá 32.767. A operação acima deviria dar 25000000000000. Neste caso o tipo que poderia acomodar esse valor seria float e mandar imprimir em tipo exponencial, potencia de 10, conforme mostra abaixo:

```
#include <stdio.h>

int main() {
  int a=1000, b=5000;
  float c;
  c = (float)(a*a)*(b*b);
  printf("Resultado da operacao: %e\n",c);
return 0;
}
```

Neste caso o resultado seria: Resultado da operacao: 2.500000e+13

Essa dica é importante quando tivermos que trabalhar com valores muito grande e podermos cair em situações que estouraria o domínio das variáveis a serem utilizadas.

6 – Expressões Matemáticas com uso dos operadores

As expressões matemáticas são combinações entre variáveis, constantes e operadores aritméticos. Alguns tipos de necessidades nas expressões matemáticas como raiz quadrada, potencia, seno, coseno, tangente, entre outros, devemos incluir a biblioteca <math.h>. A expressão quando for muito grande temos que tomar cuidado com as preferências de

operadores que são executados antes de outros como * e / precede + e -, e outros comandos podem ter precedência maior. Para evitar isso pode- se fazer uso de ( ) que torna a operação com maior precedência. Outras maneiras de resolver uma equação muito grande é quebrar sua fórmula em pequenas fórmulas mais simples e depois montar com os respectivos resultados. Exemplo: X = (A + B)*3*A/2*C; foi necessário colocar (A+B) entre parênteses por ser uma operação de menor precedência, e poderia dar problemas na operação. Outra maneria poderia ser X = A+B; depois X = X*3*A; e por fim X = X/2*C; que chegaremos ao mesmo resultado sem cometermos erro.

Todo problema que possui uma formula a ser desenvolvida como por exemplo, dado uma equação linear simples resolver os valores entre x e y sendo:

a*x + b*y = c ;

d*x + e*y = f;

Antes de levar para o computador deve ser desenvolvido colocando primeiro em função de x e depois em função de y usando uma das 2 expressões:

x = (c – b*y)/a;

d*(c-b*y)/a + e*y = f;

(d*c-d*b*y)/a + e*y = f;

d*c – d*b*y + a*e*y = a*f;

logo: (a*e*y - d*b*y) = a*f-d*c;

y*(a*e-d*b) = (a*f-d*c);

y = (a*f-d*c)/a*e-d*b); → essa formula iria tirar o valor de y

depois só colocar na primeira formulada

x = (c – b*y)/a;

teremos também o valor de x. portanto na programação só usaremos as formulas y = (a*f-d*c)/a*e-d*b); e x = (c – b*y)/a;

O programa a seguir executa essa equação linear conforme desenvolvemos acima:

```
#include <stdio.h>

int main() {
  int a, b, c, d, e, f;
  float x, y;
  printf("Entre com os valores de a, b, c, d, e, f: \n");
  scanf("%d %d %d %d %d %d",&a,&b,&c,&d,&e,&f);
  y = (float)(a*f-d*c)/(a*e-d*b);
  x = (float)(c - b*y)/a;
  printf("X = %.2f, Y = %.2f\n",x,y);
```

```
  return 0;

}
```

com os valores de entradas abaixo o programa executou da seguinte forma:

```
Entre com os valores de a, b, c, d, e, f:
7  8  12  3  5  9  // valores de entrada
X = -1.09, Y = 2.45  //resultado obtido
```

Desta forma todas as equações matemáticas devem ser refinadas antes de colocar em operação nos programas em C.

Para usar esses cálculos matemáticos em C pode ser utilizado o compilador debugger que tem disponível na internet on line, no link: https://www.onlinegdb.com/ esse compilador é multi linguagem, portanto antes de apertar run veja se a linguagem desejada está acionada. A Figura 2.3 mostra uma imagem desse compilador e no lado direito superior a linguagem que está sendo utilizada.

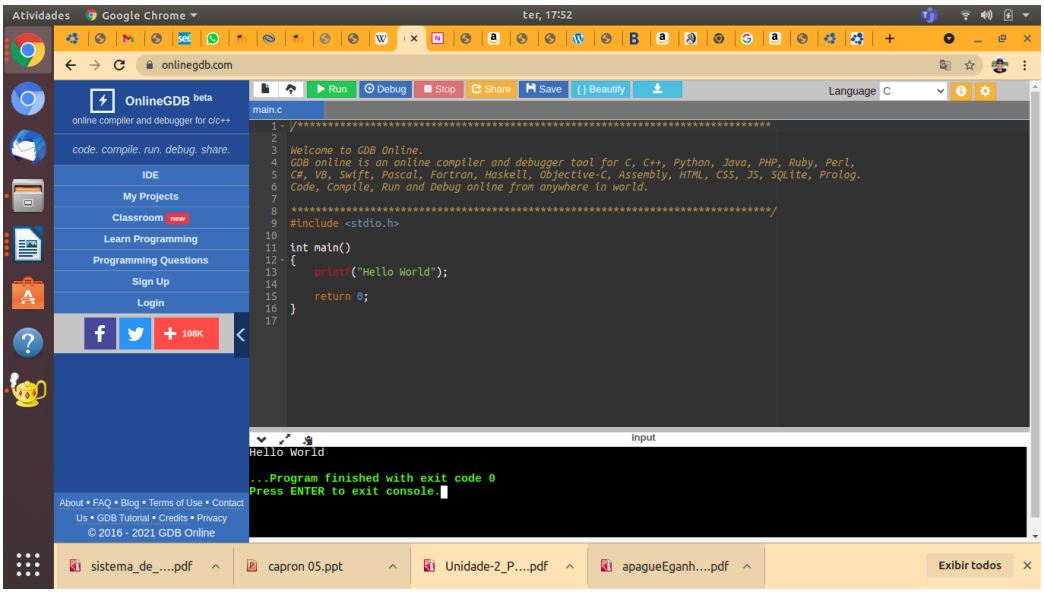


Figura 2.3: Imagem do compilador on-line no site https://www.onlinegdb.com/

Outros compiladores podem ser utilizados como dev_c++ do sistema operacional windows ou geany do sistema operacional linux Ubuntu. Todos possuem uma interface amigável e fácil de se trabalhar. Esse apresentado na Figura 2.3 é um bom compilador por aceitar todas as bibliotecas em C.

7 – Bibliotecas de operações em C

Existem várias bibliotecas em C que devem ser estudadas antes de começar a utilizar. A mais importante já conhecida até agora é a biblioteca <stdio.h> que possui funções de entrada e saída.

Veja algumas amostras simplificadas de interfaces das principais Bibliotecas padrão em C:

- stdlib.h
- stdio.h
- math.h
- string.h
- limits.h
- ctype.h
- time.h
- stdbool.h

Para ter acesso a uma biblioteca, seu programa deve incluir uma cópia do correspondente arquivo-interface. Por exemplo, basta escrever

#include <stdlib.h>

para que o pré processador da linguagem C acrescente uma cópia de stdlib.h ao seu programa. (No sistema Linux, os arquivos de interface ficam, em geral, no diretório /usr/include/; mas você não precisa saber disso.)

A documentação das funções de uma biblioteca deveria, idealmente, estar no arquivo de interface. Na prática, porém, a documentação é mantida à parte. Para consultar a documentação de uma função de biblioteca no sistema Linux, digite man nome_da_função no terminal ou consulte o Function and Macro Index da GNU C Library (https://www.gnu.org/software/libc/manual/html_node/Function-Index.html).

Todas as linguagens possuem bibliotecas, sendo que muitas são bem ricas em conteúdo como o caso de Java, que muitas vezes ajuda em melhorar o desenvolvimento em 80% com muitos pacotes e classes prontas. Em C as bibliotecas também devem ser exploradas pois possuem muitas funções e procedimentos prontos que não precisamos refazer, apenas aprender como utiliza-los.

8 – Exemplos e problemas

Continuando com a resolução de problemas ainda iremos explorar problemas matemáticos que envolvem operações simples ou sem necessidade de laços.

2.A – Custo Final de um Carro

O custo ao consumidor de um carro novo é a soma do custo de fábrica com a porcentagem do distribuidor e dos impostos (aplicados ao custo de fábrica). Supondo que a porcentagem do distribuidor seja de x% do preço de fábrica e os impostos de y% do preço de fábrica, fazer um programa para ler o custo de fábrica de um carro, a percentagem do distribuidor e o percentual de impostos, calcular e imprimir o custo final do carro ao consumidor.

Entrada

O programa deve ler três valores na entrada: o preço de fábrica do carro, o percentual do distribuidor e o percentual de impostos. Cada valor aparece em uma linha de entrada. Todos os valores são do tipo float.

Saída

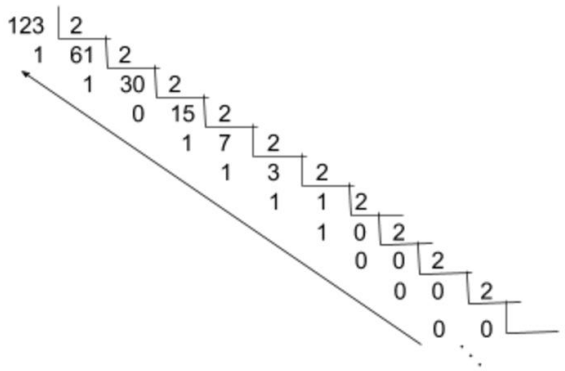
O programa deve imprimir uma linha, contento a frase O VALOR DO CARRO E = Z, onde Z é o valor do preço final do carro ao consumidor. O valor de Z deve ter duas casas decimais. Após imprimir o valor do preço final, o programa deve imprimir o caractere de quebra de linha '\n'.

Caso de Teste

Entrada
25000 12 30
Saída
O VALOR DO CARRO E = 35500.00

2.B – Conversão de Decimal para Binário

Escreva um algoritmo em Linguagem C que leia um número 0 ≤ n ≤ 255 na base decimal e apresente sua representação em binário. Caso o número informado não esteja no intervalo especificado, o programa deve finalizar imprimindo a mensagem "Numero invalido!" na tela. A transformação de um número na base decimal para binária é obtida pela sequência de divisões por 2. O número 123, por exemplo, tem sua representação binária 01111011 porque:



Não é permitido o uso de outras bibliotecas além da stdio.h.

Entrada

O programa deve ler um número inteiro qualquer.

Saída

Caso o número lido esteja fora do intervalo especificado, o programa deve imprimir a mensagem "Numero invalido!" e encerrar. Caso o número lido seja válido, o programa deve apresentar a representação binária de n na tela.

Observação

Neste problema, todos os números binários deverão conter 8 bits. O número zero (em decimal), por exemplo, tem sua representação binária 00000000. O número 1 = 00000001, o 2 = 000000010 e assim por diante.

Caso de teste

Entrada
0
Saída
00000000

Entrada
123
Saída
01111011

2.C – Valor em notas e moedas

Escreva um algoritmo par ler um valor em reais e calcular qual o menor número possível de notas de $R 100, $R 50, $R 10 e moedas de $R 1 em que o valor lido pode ser decomposto. O programa deve escrever a quantidade de cada nota e moeda a ser utilizada.

Entrada

O programa deve ler uma única linha na entrada, contendo um valor em Reais. Considere que somente um número inteiro seja fornecido como entrada.

Saída

O programa deve imprimir quatro frases, uma em cada linha: NOTAS DE 100 = X, NOTAS DE 50 = Y , NOTAS DE 10 = Z, MOEDAS DE 1 = W , onde X, Y , Z e W correspondem às quantidades de cada nota ou moeda necessárias para corresponder ao valor em Reais dado como entrada. Após cada quantidade, o programa deve imprimir um caractere de quebra de linha: ‘\n’.

Caso de Teste

Entrada
46395
Saída
NOTAS DE 100 = 463 NOTAS DE 50 = 1 NOTAS DE 10 = 4 MOEDAS DE 1 = 5

2.D – Distância entre dois pontos

Dados dois pontos A e B, cujas coordenadas A(x 1 , y 1) e B(x 2 , y 2) serão informadas via teclado, desenvolver um programa que calcule a distância entre A e B.

Entrada

O programa deve ler os quatro valores reais correspondendo às coordenadas dos dois pontos : x 1 , y 1 , x 2 , y 2 , nessa ordem, e um valor por linha.

Saída

O programa deve imprimir uma linha contendo a frase: A DISTANCIA ENTRE A e B = X, onde X é o valor da distância entre os dois pontos e deve conter no máximo 2 casas decimais. Após o valor da distância, o programa deve imprimir um caractere de quebra de linha: '\n'.

Observação

A distância entre dois pontos é computada pela fórmula:

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

Caso de teste

Entrada
3 4 5 6
Saída
A DISTANCIA ENTRE A e B = 2.83

Os resultados dos problemas apresentados devem ser retirados e passados ao professor para que ele possa resolver junto com os alunos.

Resultados

2.A

```
#include <stdio.h>
int main(){
	float precoFabrica,percDist, percImposto,precoFinal;
	printf("Entre com o preço de fabrica – percDist – percImposto : \n");
	scanf("%f",&precoFabrica);
	scanf("%f",&percDist);
	scanf("%f",&percImposto);
	printf("O VALOR DO CARRO E = %.2f\n", precoFabrica +percDist/100* precoFabrica+
percImposto/100*precoFabrica);
}
```

2.B

```
#include <stdio.h>

int main() {
	int n;
	printf("Entre com o valor: \n");
	scanf("%d", &n);

	if((n < 0) || (n > 255 )) {
		printf("Numero invalido!");
		return 0;
	}
```

```
        printf("%d", n/128%2);
        printf("%d", n/64%2);
        printf("%d", n/32%2);
        printf("%d", n/16%2);
        printf("%d", n/8%2);
        printf("%d", n/4%2);
        printf("%d", n/2%2);
        printf("%d\n", n%2);
        return 0;
}
```

2.C

```
#include <stdio.h>
int main()
{
   int valor, n100, n50, n10, m1, resto;
       printf(“Entre com o valor:\n”);
        scanf("%d",&valor);
        n100=valor/100;
        resto=valor%100;
        n50=resto/50;
        resto=resto%50;
        n10=resto/10;
        m1=resto%10;
        printf("NOTAS DE 100 = %d\n", n100);
        printf("NOTAS DE 50 = %d\n", n50);
        printf("NOTAS DE 10 = %d\n", n10);
        printf("MOEDAS DE 1 = %d\n", m1);
   return 0;
}
```

2.D

```
#include <stdio.h>
#include <math.h>
int main(){
        printf(“Entre com os pontos X1, Y1, X2, Y2 : \n”);
        float x1,y1,x2,y2,dist;
        scanf("%f",&x1);
        scanf("%f",&y1);
        scanf("%f",&x2);
        scanf("%f",&y2);
        dist=sqrt( (x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
        printf("A DISTANCIA ENTRE A e B = %.2f\n",dist);
}
```

CAPÍTULO III

ELEMENTOS FUNDAMENTAIS DA PROGRAMAÇÃO

1 – Introdução

Neste capítulo iremos estudar os principais fundamentos da programação na linguagem C, sua estrutura principal, linguagem de alto nível, linguagem genéricas e específicas. A linguagem C é uma linguagem que depende muito da plataforma em que ela se propõem a ser projetada. As bibliotecas apesar de serem padrões para todas as plataformas existem algumas diferenças básicas que mudam alguns tipos de funções dependendo da biblioteca.

Um programa de computador é um conjunto de instruções que descreve um algoritmo pré-definido ao qual foi descrito para exercer algumas ações principais de um contexto do mundo real. Essas instruções são descritas com um conjunto de códigos que chamamos de comandos de programação, e que seu padrão representa uma ação em cada tipo de linguagem mudando apenas algumas características dependendo da linguagem. Em C que é nossa linguagem escolhida possuem diversos comandos que possuem um conjunto de regras semânticas que devem ser respeitadas, estruturas lógicas e sintaxe própria. Dizemos então que esse conjunto de comandos e regras formam um programa. O programa deve ser compilado, corrigido sempre que aparecem erros ou alertas, evitando que algo possa surpreender durante a sua execução.

Existem muitas linguagens de programação. Podemos escrever um algoritmo para resolução de um problema por intermédio de qualquer linguagem. A seguir mostramos alguns exemplos de trechos de códigos escritos em algumas linguagens de programação. Exemplo: trecho de um algoritmo escrito em *Pseudo-linguagem* que recebe um número num e escreve a tabuada de 1 a 10 para este valor:

```
    leia num
    para n de 1 até 10 passo 1 faça
        tab ← num * n
        imprime tab
  fim faça
```

Exemplo: trecho do mesmo programa escrito em linguagem C:

```
   scanf(&num);
  for(n = 1; n <= 10; n++){
    tab = num * n;
   printf(”\n %d”, tab);
};
```

Exemplo: trecho do mesmo programa escrito em linguagem Basic:

```
10 input num
20 for n = 1 to 10 step 1
30 let tab = num * n
40 print chr$ (tab)
50 next n
```

Exemplo: trecho do mesmo programa escrito em linguagem Fortran:

```
read (num);
do 1  n = 1:10
     tab = num * n
    write(tab)
10 continue
```

Exemplo: trecho do mesmo programa escrito em linguagem Assembly para INTEL 8088:

```
MOV CX,0
IN AX,PORTA
MOV DX,AX
LABEL:
INC CX
MOV AX,DX
MUL CX
OUT AX, PORTA
CMP CX,10
JNE LABEL
```

Existem linguagens de Baixo Nível (*Assembly*) entre outras que são linguagens voltadas para uso de máquinas usando as instruções do microprocessador do computador. Essas linguagens tornam os programas mais rápidos e ocupam menos espaços de memorias, porém, os programas em *Assembly* tem pouca portabilidade, sendo um código gerado para um tipo de processador não serve para outro. O programa em **Assembly** não são estruturado tornando a programação muito difícil.

Já as linguagens de Alto Nível são voltadas mais para ser humano sendo estruturadas e podendo tornar o código mais legível de entendimento. Elas precisam de compiladores e

interpretadores para gerar os códigos mais próximo do microprocessador. O interpretador faz a interpretação de cada instrução do programa fonte executando-a dentro de um ambiente de programação. Já os compiladores fazem a tradução de todas as instruções do programa fonte gerando um programa executável. Após a compilação o programa executável pode ser executado fora do ambiente de programação, a Linguagem C utiliza compilador, assim como Pascal ou Fortran. Com isso tornam os programas com maiores portabilidades, podendo ser executados em várias plataformas com poucas adaptações.

Todos os programas podem ter vidas longas, existem softwares que já estão em uso a mais de anos podendo sofrer algumas manutenções com o tempo e adaptações. Por isso devemos sempre pensar em fazer um bom software (programa com um conjunto de documentações e funcionalidades bem definidas) visando sua manutenção e correções futuras.

2 – Divisão das etapas de um programa

Um programa em C é constituído de:

• um cabeçalho contendo as diretivas de compilação onde se definem o valor de constantes simbólicas, declaração de variáveis, inclusão de bibliotecas, declaração de rotinas, etc.

• um bloco de instruções ***principal*** e outros blocos de ***rotinas***.

• documentação do programa: comentários que podem estar em várias linhas (usando /* ….*/ ou em uma mesma linha ou após um determinado comando usando o //.

Todo programa segue um padrão e a Linguagem C possui um padrão ANSI e seus caracteres seguem o padrão de caracteres ASCII. Em geral o padrão ASCII possui os caracteres:

Caracteres válidos:

a b c d e f g h i j k l m n o p q r s t u v w x y z

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

1 2 3 4 5 6 7 8 9 0

+ - * / \ = | & ! ? # % ( ) { } [ ] _ ‘ “ . , : < >

Caracteres não válidos:

@ $ ¨ á é õ ç

Os Comentários em C podem ser escritos em qualquer lugar do programa, e são desprezados pelo compilador, ele ajuda no entendimento do programa ou de cada comando utilizado. Para sua identificação pode-se utilizar os comentários entre várias linhas da seguinte forma:

```
/* Este é um comentário
        que podemos escrever em várias linhas  */
```

ou numa mesma linha após um determinado comando: //comentário em uma mesma linha.

As diretivas de compilação são comandos que são processados durante a compilação em C e servem para ajudar a programação do usuário. A diretiva #include inclui outros programas (bibliotecas) que contém funções e outros arquivos. A diretiva #define ajuda a definir constantes que serão usadas no programa. Elas facilitam algumas operações que serão utilizadas no programa.

As declarações de variáveis ou definições de tipos de dados, são usados em seguida dessas diretivas, o que veremos em capítulos mais a seguir. Procure declarar as variáveis de programas no início de cada procedimento que serão utilizados apenas o necessário. As variáveis em C são sensíveis ao contexto uma variável A é diferente de outra variável a .

Procure sempre organizar seu programa em etapas bem definidas evitando confusões de entendimento. Todo programa em C pode ser modularizado, ou seja, quebrado em procedimentos que resolvam pequenas tarefas que são esperadas para atender as especificações do sistema.

3 – Entrada de dados

Todas entradas de dados devem ser utilizadas tipos que possam melhor representar a informação que o programa deseja tratar. As variáveis devem ser declaradas no início do programa podendo ser de vários tipos: int (inteiro), float (real de simples precisão) e conforme já apresentamos no Capítulo 2. Outros tipos não primitivos serão vistos mais a seguir quando houver necessidades de criarmos novos tipos para ajustar as informações que iremos utilizar em nosso programa.

Os comandos de leitura e gravação estão melhor representados em funções das bibliotecas de C. Por isso é muito importante se habituar a utilizar tais funções como scanf,

fgets para ler strings, getchar para leitura de char ou controle de parada, entre outros. Os principais comandos de entrada e saída estão na biblioteca <stdio.h> ou <stdlib.h>.

Todo comando scanf deve ter um código de leitura apresentado no Capítulo 2, por exemplo %d para int, %f para float, %c para char, %lf para long e outros.

Da mesma forma esses comandos serão colocados nos comandos de saída de informação "*printf*" que será comentado no item 5 deste capítulo.

As entradas de informações podem ser por teclado ou por arquivo que veremos em capítulos mais a frente quando formos tratar de arquivos de armazenamentos das informações, para facilitar as operações contínuas do nosso programa.

4 – Etapa de resolução do problema

Após lido todas as informações necessárias de nosso programa, iniciamos a etapa de resolução dos problemas pré-definidos pelos requisitos do software. Essa etapa pode ser distribuída para uso de módulos quando for algo muito complexo, visando quebrar partes da resolução em pequenas partes mais fáceis.

A modularização sempre é necessária quando temos muitas atividades (funcionalidades) a serem resolvidas visando atender as necessidades do usuário. Imaginamos que teremos uma função de leitura dos dados, outra de inserção de valores quando houver necessidade, outras de exibição de resultados, outras de orientações e assim por diante.

Em um programa que devemos resolver um caso do mundo real podemos atender as etapas de resolução de várias maneiras. Muitas vezes procuramos resolver tudo no programa principal (main()) mas devemos evitar que isso aconteça quando nosso programa tiver várias funcionalidades de usuário e que devem ser separadas por etapas. Veremos esse tipo de resolução em capítulos mais a frente.

5 – Saída dos resultados

Como a terceira e a mais importante etapa de um programa são as saídas das informações, visando levar ao usuário todos resultados esperados pelo software, de acordo com a especificação.

Alguns programas guardam seus resultados em relatórios outros mostram na própria tela de saída do programa. Outros guardam em arquivos ou banco de dados, utilizando técnicas mais avançadas da programação. Aqui neste material só iremos ver resultados na

tela ou em arquivos armazenados para uso do mesmo em outros momentos e suas informações somente será lida pelo próprio programa.

Como já vimos o principal comando de saída da informação é o printf por ser mais estruturado e mais fácil de representar o resultado da maneira esperada pelo usuário. Ou fprintf quando formos armazenar em arquivos.

6 – Observações importantes

Procure estruturar sempre seu programa em etapas bem definidas. Documente seu programa sempre que achar necessário. Evite utilizar nomes de variáveis que não significa muito para o que ela representará no seu programa, tipo você deverá usar uma variável para calcular uma equação y = (a*b + a*c)/2*a; neste caso procure declarar as variáveis y, a, b, e c. Utilize os mesmos nomes dados pela equação.

Estude todo conteúdo dado no Capitulo 2 ele será muito utilizado ao decorrer dos próximos capítulos, principalmente os comandos de leitura e gravação, bem como as diretivas utilizadas para mudar de linha, tabulação, entre outros usando o \ .

Faça os exercícios sempre que forem propostos por esse material eles ajudam a reforçar o entendimento e aprendizado dos conteúdos que iremos abordar em cada Capítulo.

Anote sempre as dúvidas e leve para o monitor da disciplina ou professor. Não deixe acumular muitas dúvidas porque isso pode atrapalhar na continuidade do entendimento do restante dos conteúdos.

Referência Bibliográfica para leitura para este capítulo: Ascencio, A. F. e Campus, E.A. V. – Fundamentos da Programação de Computadores: Algoritmos, Pascal, C, C++ e Java – Pearson, 3ª Edição, 2012.

CAPÍTULO IV

ESTRUTURAS DE FLUXO DE CONTROLE DAS INFORMAÇÕES

1 – Introdução

Neste capítulo veremos os comandos que farão mudanças aos fluxos de informação na programação em C. Iniciaremos com os comandos básicos de atribuição, matemáticos e outros simples que são considerados comandos que calculam as necessidades exigidas pelo problema em questão. E outros que ajudam a investigar se seus resultados estão sendo verificados da forma correta com comparações simples e compostas que definiremos por predicativos. Essas comparações são inicialmente colocadas em comandos de decisão como *if-then-else* que iremos ver com detalhes e suas necessidades com exemplos.

Da mesma forma abordaremos exemplos das necessidades do uso do comando *switch case* quando é importante utilizarmos e também outros exemplos de ninhos de *if-then-else* que as vezes se faz necessário.

Após trabalharmos com os comandos de decisão iniciaremos os comandos de laços (repetição) que muitas vezes serão necessários para executarmos sequências de atividades repetidas, com pequenas mudanças. O uso de cada tipo de comando de repetição e quando são importantes utilizarmos o comando *for* ou *while* ou *do-while*. Cada um desses comandos ajuda a exercer uma tarefa, que mesmo que essa tarefa para usar um ou mais comandos, devemos entender a necessidade de cada comando para ter seu uso correto.

Para ilustrar esses comandos que serão estudados nesse capítulo, faremos exercícios e indicaremos outros para que o aluno possa melhorar seus estudos e práticas com os comandos aqui estudados, para que nas etapas mais a frente eles sejam bem trabalhados e praticados.

Uma boa Leitura seria o Livro de Algoritmos – Teoria e Prática de Thomas Cormen, Charles Leiserson e Ronald Rivest, 3ª Edição - 2009.

2 – Comandos de atribuição sequenciais

Um conjunto de comandos em C sequencial denominamos de bloco de comandos, que são tratados como comandos sem mudanças de fluxo de controle dentro de um programa. Quando um comando, desse bloco de comandos sequenciais é executado, apenas ocorrem alterações nas informações, nos valores das variáveis envolvidas no conjunto de comandos que estão nesse bloco.

Para exemplificarmos essa sequência vamos ilustrar o cálculo de uma operação matemática que dado 3 valores de a, b e c, iremos calcular a média aritmética desses valores. Para ilustrar operações de inteiros resultando em valores com ponto flutuante (*real* em pascal)

ou (*float* em C), iremos mostrar como procedemos tais resultados sem perda de casas decimais. Então consideraremos a, b e c valores inteiros e a média valor *float*.

```
01      #include <stdio.h>
02
03      int main( ) {
04       int a, b, c;
05       float media;
06      // entrada de dados
07       scanf("%d %d %d", &a, &b, &c); // a leitura exige o endereço das variaveis a, b e c
08      media = (float)(a+b+c)/3.0;
09      printf("A media entre %d, %d e %d = %.2f\n",a,b,c,media);
10    return 0;
11    }

/*
supondo valores 3 10 4
 o resultado ficaria
 A media entre 3, 10 e 4 = 5.67
 */
```

Observe que na linha 08 foi colocado um (*float*) antes da soma de a, b e c. Este é um comando de forçar que a soma entre 3 inteiros se transforme em *float*, é denominado de *cast*.

Essas operações são necessárias sempre que as operações são entre variáveis de um tipo e temos que transformar em outro tipo de dados. Apesar que nessa operação realizada na linha 8 o divisor foi colocado em ponto flutuante (3.0) e sendo assim não seria necessário o uso do *cast*, vocês podem verificar tirando o *cast* (*float*) nessa linha, o resultado seria o mesmo, porém se colocarmos o divisor por 3 como inteiro, ai seria necessário o uso do *cast*.

Em outras operações usaremos o *cast* como forma de converter o resultado para o tipo da variável que está recebendo a informação.

A sequência de comandos que vai entre a linha 07 até a 09 podem ser colocadas num mesmo bloco de comandos, o bloco só se quebraria se houvesse algum comando de mudança de fluxo de controle como um *if* ou de um *laço*, que veremos mais a seguir.

Um outro exemplo que podemos mostrar para entender o conceito de bloco de comandos (em uma determinada sequência de comandos sem alteração de fluxo de controle) vamos ver um problema como exemplo.

Exemplo 4.1:

Um fabricante de latas deseja desenvolver um programa para calcular o custo de uma lata cilíndrica de alumínio, sabendo-se que o custo do alumínio por m^2 é R$ 100,00.

O programa deve ler dois valores na entrada: o raio e a altura da lata. Ambos os valores correspondem a valores em metros. Cada valor ocorre em uma linha diferente na entrada.

O programa deve imprimir a frase: O VALOR DO CUSTO E = XXX.XX, onde XXX.XX é o valor do custo da lata. Logo após o valor do custo da lata o programa deve imprimir o caractere de quebra de linha '\n'.

Observações

• O seu programa deve utilizar a constante π com o valor aproximado de **3.14159**.

• O valor total da área de um cilindro é dada por $A_t = 2A_c + A_l$, onde A_c é a área do círculo, calculada como: $A_c = \pi r^2$ e A_l é a área lateral do cilindro, computada por $A_l = 2\pi ra$, onde r é o raio e a a altura da lata em metros.

Resultado:

```
01      #include <stdio.h>
02      #define PI 3.14159
03
04      int main() {
05
06      float r, a;
07      float custo;
08
09      scanf("%f", &r);
10      scanf("%f",&a);
11      custo = 100*( 2*(PI*r*r) + (2*PI*r*a) );
12      printf("O VALOR DO CUSTO E = %.2f\n", custo);
13
14      return 0;
15      }
```

```
/* Se as entradas lidas fossem 10.0 10.0
 o resultado seria:
O VALOR DO CUSTO E = 125663.60
*/
```

Observem que os comandos entre as linhas de 09 a 12 são sequências que não afetam o fluxo de controle do programa. Neste caso são considerados como bloco de comandos sequenciais.

A seguir será introduzido comandos de decisão *if-then-else* que muda o bloco de comandos em um programa, deviso a mudança de fluxos de controle que ocorre em um programa conforme a necessidade exigida pelo problema.

3 – Comandos de decisão

O comando if ou if-else em C instrui o programa a tomar uma decisão em seu resultado. O comando é descrito como:

if (predicativo de decisão)

(comando ou comandos;)

else (comando ou comandos;)

Em geral os comandos de decisão simples são tratados no comando if as expressões lógicas que são colocadas no predicativo de decisão, devem resultar em valores booleanos ***true*** (verdadeiro) ou ***false*** (falso) que entra no bloco de comando do ***if*** ou do ***else***.

Vamos ver um exemplo que mostra a mudança de fluxo de controle nos comandos de decisão (if-else).

```
01    #include <stdio.h>
02    int main( ) {
03    char c;
04    printf("digite c ou p\n");
05    scanf("%c",&c);
06    if (c =='c')
07        printf("Voce digitou c\n");
08      else if(c=='p')
09              printf("Voce digitou p\n");
10            else  printf("Voce digitou outra letra diferente de c e p\n");
11    return 0;
12    }
```

Neste exemplo da linha 04 ate a linha 06 pertence ao mesmo bloco de comandos, depois que aparece o comando if, o comando da linha 07 está na decisão verdadeira do comando if da linha 06, o comando da linha 08 está como parte falsa do comando if da linha 06, nesse comando existe outro if que verifica se o digito digitado foi 'p' mudando novamente o fluxo caso sim ou caso não. Se a pessoa digitou p irá executar o comando da linha 09, caso contrário, não tenha digitado nem c e nem p irá executar o comando da linha 10.

Para entender a necessidade de um comando de decisão em um problema vamos fazer um exemplo para ilustrar o uso dos comandos de decisão.

Exemplo 4.2:

Desenvolver um programa que leia um número inteiro e verifique se o número é divisível por três e também é divisível por cinco.

O programa deve ler uma linha contendo um número inteiro na entrada.

O programa deve imprimir uma linha contendo a frase: O NUMERO E DIVISIVEL, se ele for divisível tanto por três quanto por cinco, ou a frase O NUMERO NAO E DIVISIVEL, em

caso contrário. Após imprimir uma das frases, o programa deve imprimir um caractere de quebra de linha: ‘\n’.

Solução:

```
01    #include <stdio.h>
02    int main(){
03    int num;
04    scanf("%d",&num);
05    if((num%3==0) && (num%5==0)){
06          printf("O NUMERO E DIVISIVEL\n");
07    }
08    else
09          printf("O NUMERO NAO E DIVISIVEL\n");
10    }
```

```
/* Se o valor de entrada for 871
o resultado deverá ser: O NUMERO NAO E DIVISIVEL
porém se o valor for 75 o resultado será: O NUMERO E DIVISIVEL
*/
```

Observe que o fluxo de controle muda na linha 05 se o predicativo de comparação ((num%3==0) && (num%5==0)) resultar em verdadeiro a resposta será “ O NUMERO E DIVISIVEL” caso contrário a resposta será da linha 09 "O NUMERO NAO E DIVISIVEL".

Outros exemplos serão colocados na parte de exercícios deste Capítulo para que os alunos possam exercitar seu conhecimento em comandos de decisão simples e compostos.

A seguir vamos ver melhor quando há necessidade de múltiplas decisões colocadas de forma aninhadas em uma sequência construtiva visando interpretar melhor as necessidades do problema.

4 – Comandos de múltiplas decisões aninhadas

Os comandos if-else aninhados, são quando existem uma sequência de decisões para se obter diferentes tipos de resultados. Um exemplo simples, porém, demanda uma decisão aninhada seria determinar qual é o menor valor entre 4 números inteiros a, b, c e d.

Para isso devemos inicialmente ler os valores das variáveis inteiras a, b, c e d, e em seguida iniciar a verificação de qual seria o menor valor entre elas.

```
01    #include <stdio.h>
02    int main() {
03    int a, b, c, d, menor;
04    scanf("%d %d %d %d", &a, &b, &c, &d);
```

```
05
06      if ((a<b)&&(a<c)&&(a<d))
07              menor = a;
08        else if((b<c)&&(b<d))
09              menor = b;
10             else if(c<d)
11                   menor =c;
12                  else menor = d;
13      printf("O menor valore entre a = %d, b = %d, c=%d e d=%d => eh %d\n",
        a,b,c,d,menor);
14
15      return 0;
16     }
```

O exemplo é fácil de entender pois primeiro foi necessário comparar ***a*** com todas as variáveis ***b, c*** e ***d***, (na linha 06) depois se ***a*** não for menor que todas vamos para a 2ª comparação linha 08, que ocorre entre ***b*** com as variáveis ***c*** e ***d***, pois ***b*** já foi comparada com ***a*** na linha 06. E assim caso ***b*** também não for menor que todas, iremos para a 3ª comparação entre ***c*** e ***d*** na linha 10 que encerra o ninho de ***if-else*** necessário para detectar qual será o menor valor entre ***a, b, c*** e ***d***.

Alguns cuidados são necessários quando temos comparações compostas como ocorreu na linha 06 é necessário separar entre parênteses cada item da comparação e depois toda a comparação deve estar entre parênteses, facilitando para o compilador.

5 – Comando Switch case

O comando switch case é um comando que facilita a comparação quando um valor a ser comparado pode ser uma sequência de valores pré-definidos, evitando assim de fazermos um ninho de ***if-else***. Para entender o uso correto do comando switch case vamos supor que queremos ler uma dada sendo ddmmaaaa em um número único inteiro e escrever primeiro se o dia ou mês ou ano for inválido sabendo que os dias de um mês pode ser de 1 a 28 (fevereiro), de 1 a 31 para janeiro, março, maio, julho, agosto, outubro e dezembro e de 1 a 30 para os demais meses. Ex: se entrarmos com a data 10121999 devemos escrever 10 de dezembro de 1999.

Então Faça um programa que leia uma data no formato ddmmaaaa usando um único número inteiro. Escreva a mesma data no formato dia/mês/ano, <dia> de <mês por extenso> de <ano>. O programa deve verificar se o número informado representa uma data válida. Caso

não seja, imprimir na tela a mensagem "Data invalida!". Considere que o ano em questão nunca é bissexto, ou seja, fevereiro tem somente 28 dias.

A entrada será um único número inteiro com 8 dígitos.

O programa deve apresentar uma transição da data no formato “dd do mês por extenso de aaaa (ano)”.

Resultado:

```
01  #include <stdio.h>

02  int main() {
03  int data, d, m, a;
04  scanf("%d", &data);
05
06  if(data < 0 || data/100000000 > 0) {
07          printf("Data invalida!\n");
08          return 0;
09  }
10  d = data / 1000000;
11  m = data % 1000000 / 10000;
12  a = data % 10000;
13
14  if( d < 1 || d > 31 || m > 12 || m < 1 )  {
15          printf("Data invalida!\n");
16          return 0;
17  }
18  if((m==2) && (d>29)) {
19    printf("Data invalida!\n");
20    return 0;
21  } else if((m==2)&&(d==29)) {
22          if(((a%100)==0)&& (((a/100)%4)!=0)) {
23              printf("Data invalida!\n");
24              return 0;
25                  } else if ((a%4)!=0) {
26                                  printf("Data invalida!\n");
27                    return 0;
28                  }
29
30    }
31  if( (m==4 || m==6 || m==9 || m==11) && d>30 ) {
32          printf("Data invalida!\n");
33          return 0;
34  }
35  printf("%d de ", d);
36  switch (m) {
37     case 1: printf("janeiro");
38          break;
39     case 2: printf("fevereiro");
```

```
40              break;
41          case 3: printf("março");
42              break;
43          case 4: printf("abril");
44              break;
45          case 5: printf("maio");
46              break;
47          case 6: printf("junho");
48              break;
49          case 7: printf("julho");
50              break;
51          case 8: printf("agosto");
52              break;
53          case 9: printf("setembro");
54              break;
55          case 10: printf("outubro");
56              break;
57          case 11: printf("novembro");
58              break;
59          case 12: printf("dezembro");
60              break;
61        }
62      printf(" de %d\n", a);
63      return 0;
64      }
/* Se a entrada for 10192019 o resultado deverá sair “Data invalida!”
porém se a entrada for 10082021 a saída deverá ser “10 de agosto de 2021”
*/
```

Observe que esse exemplo tivemos que fazer vários ifs aninhados, e outros isolados e para decidir o mês utilizamos o comando switch case para facilitar nossa operação. Poderia ter sido usado comandos ifs também aninhados, porém ia fica muito grande e mais confuso. No comando ***switch*** case na linha 36 houve a necessidade abrir um conjunto de comandos com o “{“ e fechando com o “}” na linha 61 no final do comando ***switch***. Para cada case utilizado para escrever o respectivo mês de janeiro até dezembro, tivemos que colocar um comando ***break***, que faz com que o case que ele entrar seja realizado e encerrado o comando, não deixando continuar no próximo case. Portanto, o uso do ***break*** nos ***cases*** do ***switch*** é necessário.

Outro uso muito importante do comando ***switch*** é quando temos um programa com várias opções dadas por um menu como por exemplo, inserir um elemento, remover elemento, consultar, exibir todos os elementos etc, que serão muito úteis quando estivermos

desenvolvendo um programa real para um determinado usuário. Neste caso utilizaremos o comando ***switch*** para as opções de escolha do menu.

6 – Comandos de Repetição:

Os comandos de repetição (laços) são muito necessários quando temos atividades repetitivas para serem utilizadas, como por exemplo efetuar o cálculo de uma somatório de valores ou expressões, ou entrar com dados de um vetor ou buscar um valor em um determinado vetor, ou ler os dados de uma matriz ou buscar valores de em uma matriz e assim sucessivamente quando houver essas. Os comandos de repetição também conhecidos como laços (ou do inglês loops) ajudam a executar uma sequência de repetição de um conjunto de comandos quando for necessário. Os laços devem ser finitos, visando atender a especificação do problema. Caso o laço não atenda o programa poderá resultado em um erro de Run Time Error (tempo de execução errada).

Os comandos de repetição em C são for quando temos uma sequência a ser seguida podendo ter na forma crescente ou decrescente conforme a necessidade.

Outros comandos de repetição podem caracterizar como melhor caso (não entrar nenhuma vez ou só 1 vez no laço) ou pior caso (quando tem que entrar todas as vezes no laço) esse é o caso dos comandos **while** ou ***do*** – ***while***. Os quais veremos com exemplos mais a seguir.

6.1 **For**

O comando “**for”** permite que um certo trecho de programa seja executado **um determinado número de vezes**.

A forma do comando **for** é a seguinte:

```
for (comandos de inicialização;condição de teste;incremento/decremento)
{
// comandos a serem repetidos
// comandos a serem repetidos
}
// comandos após o 'for'
```

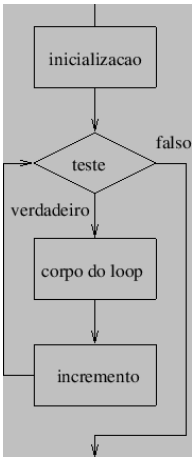


Figura 4.1 – Fluxograma do comando **for**;

O comando **for** executa uma sequência de iterações, podendo ser crescente ou decrescente e tendo um número inicial e um número que finaliza sua sequência de execução em um determinado exemplo.

O comando tem o formato: **for**(***i=0;i<n;i = i+1***) comando; sendo ***i=0*** o comando de inicialização, ***i<n*** a condição de teste e ***i = i+1*** o incremento da variável ***i***.

Neste comando mostra que a variável ***i*** deve ser inteira e declarada no início do procedimento, e ela irá executar o comando n vezes sendo n um valor pré definido ou lido pelo usuário. O comando pode ser um único comando ou um bloco de comandos, quando for mais do que um comando devemos por todos entre “{“ e “}”.

Um exemplo: Dado dois inteiros num e n sendo num o numero inicial de uma sequencia de pares (n) que deverão ser descrito a seguir. Ex: 20 4 os pares a partir de 20 serão 20 22 24 26.

Resultado:

```
01    #include <stdio.h>

02    int main() {
03    int x, y, c;
04    scanf("%d%d",&x, &y);
05    if(x%2 != 0) {
06          printf("PRIMEIRO NUMERO NAO E PAR\n");
07          return 0;
08    }

09    for(c=0;c<y-1;c++) {
10          printf("%d ", x);
11          x += 2;
```

```
12      }
13      printf("%d\n", x);
14      return 0;
15      }
/* Caso a entrada for 20 10 o resultado será: 20 22 24 26 28 30 32 34 36 38
caso a entrada for 21 10 o resultado seria "PRIMEIRO NUMERO NAO E PAR"
*/
```

Neste exemplo na linha 09 caso o primeiro número seja par o laço é executado de 0 até atingir o n-1 e na saída escreve o valor final mudando a escrita do laço que tem espaço e sequencial, com uma escrita única e mudando de linha com o “\n”.

Outros exemplos serão vistos no final deste capítulo com exercícios para o aluno praticar e utilizar os comandos de repetição.

6.2 **While**

O outro comando de repetição é o ***while*** (enquanto (condição) faça:). O formato do comando é ***while***(condição de teste) comando; ou { conjunto de comandos }

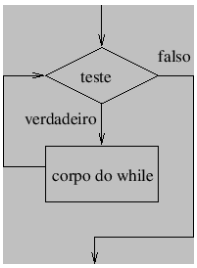


Figura 4.2 – fluxograma do comando **while**

O comando while é muito utilizado quando fazemos uma busca em conjunto de valores e somente para sua execução quando atingir a quantidade de valores existentes ou encontrar o valor neste conjunto. Neste caso podemos dizer que o melhor caso seria encontrar no inicio do conjunto de valores e temos apenas uma comparação executada. No pior caso num conjunto de N valores seria encontrar no final do conjunto e termos n comparações executadas. O caso médio seria (N+1)/2 que é a média entre o melhor caso e o pior caso.

Para ilustrar esse comando vamos fazer o exercício 4.6 a seguir:

Escreva um programa que leia várias linhas contendo cada uma a matrícula de um funcionário (um valor inteiro), seu número de horas trabalhadas, o valor que recebe por hora e calcule o salário desse funcionário. A seguir, mostre o número e o salário do funcionário, com duas casas decimais.

A entrada é formada por varias linhas, cada uma contendo três valores decimais separados um do outro por um espaço em branco. O último valor na linha é seguido pelo

caractere de quebra de linha (\n). A última linha contém uma matrícula igual a zero e não deve ser processada. Ela serve apenas para indicar ao programa o término da entrada de dados.

A saída deve conter para cada linha de entrada a matrícula, um espaço em branco e o salário calculado com duas casas decimais.

Observações:

Para ler uma linha com os três valores, utilize a função scanf: scanf("%d %f %f", &A, &B, &C); e, em seguida, a função getchar(). A função getchar() é usada para consumir o caractere de quebra de linha na entrada.

Resultado do Programa:

```
01    #include <stdlib.h>
02    #include <math.h>
03    #include <stdio.h>

04    int main(){
05    int matricula, horasTrab;
06    double salarioHora, salarioTotal;
07
08    scanf("%d %d %lf", &matricula,&horasTrab,&salarioHora);
09    getchar();
10    while (matricula){  //quando matricula for diferente de 0 é verdadeiro
11            salarioTotal= salarioHora*horasTrab;
12            printf("%d %.2f\n", matricula,salarioTotal);
13            scanf("%d %d %lf", &matricula,&horasTrab,&salarioHora);
14            getchar();
15          }
16    }
/* para a entrada:
2015001 16000 3.23
2015002 16010 3.0
2015003 16009 2.99
0 0 0.0
a saída será:
2015001 51680.00
2015002 48030.00
2015003 47866.91

*/
```

No exemplo dado o comando while usou como teste apenas matricula, o que em C isso é possível pois qualquer valor diferente de 0 (zero) é verdadeiro e 0 (zero) é falso. Seria o mesmo que ter colocado ***while*** (matricula!=0) {….}.

O comando ***while*** pode ter a expressão lógica de comparação simples ou composta, conforme a necessidade que tivermos que exercer tal comparação. Quando chegarmos em algoritmos de busca iremos utilizar expressão lógica de comparação no ***while*** composta.

6.3 Do While

O comando ***do while*** também conhecido como ***repeat until*** do pascal, é bem parecido com o comando while mudando apenas que a comparação ocorre no final do corpo do laço. A Figura 4.3 mostra como seria o fluxograma deste comando.

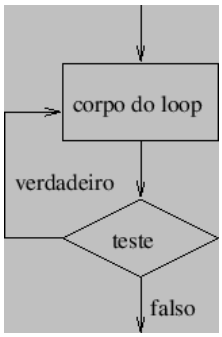


Figura 4.3: fluxograma do comando do-while

Observe que a saída do laço ocorre no final após o corpo do laço ter sido executado. Neste caso a primeira vez o laço é executado sempre. Muito utilizado para casos quando temos que executar o fluxo de comandos no laço pelo menos uma vez podendo não ter a necessidade de executar outras vezes. Um exemplo muito usado é quando temos um programa que executa um menu e tem uma opção de parada tipo digirar um número para parar ou fazer outras operações como inserir, buscar valores, exibir informações etc.

Vamos fazer um exemplo prático que será o exercício 4.7.

Faça um programa que leia uma sequência de números inteiros diferente de zero e apresente a média nos números pares e a média nos números ímpares.

O programa deve ler uma sequência de números inteiros diferentes de zero.

O programa deve apresentar duas linhas, a primeira contendo a mensagem: "MEDIA PAR = mp"e a segunda com a mensagem: "MEDIA IMPAR = mi", onde mp e mi são os valores das médias dos números pares e ímpares respectivamente.

Resultado do programa:

```
01    #include <stdio.h>

02    int main() {
03    int n;
```

```
04      int np, ni;
05      double sp, si;

06      np = ni = 0;
07      sp = si = 0;
08      scanf("%d", &n);
09      do{
10              if( n%2 == 0 ) {
11                      np++;
12                      sp+=n;
13              } else {
14                      ni++;
15                      si+=n;
16              }
17          scanf("%d", &n);
18         }while(n);
19      if( np != 0) sp = sp/np;
20      if( ni != 0) si = si/ni;
21      printf("MEDIA PAR: %f\n", sp);
22      printf("MEDIA IMPAR: %f\n", si);
23      return 0;
24      }
/* Se a entrada for a sequência: 12 12 12 12 14 14 15 151 5 7 7 7 7 7 7 12 4 0
a saída deverá ser:  MEDIA PAR: 11.500000
                     MEDIA IMPAR: 22.000000
*/
```

Observe que o comando ***do-while*** a condição do while ocorre após o fechamento "}" do laço e o comando ***while***(condição); possui um ";" o que não acontece no comando ***while*** apenas. O laço ***do-while*** nesse exemplo ocorre entre as linhas 09 e 18.

Os comandos de repetição deste capítulo foram apresentados e a seguir vamos colocar exercícios para serem praticados pelos estudantes.

7 – Exercícios:

Exercícios de expressões:

E7.1 – Custo final de um carro:

O custo ao consumidor de um carro novo é a soma do custo de fábrica com a porcentagem do distribuidor e dos impostos (aplicados ao custo de fábrica). Supondo que a porcentagem do distribuidor seja de x% do preço de fábrica e os impostos de y% do preço de fábrica, fazer um programa para ler o custo de fábrica de um carro, a percentagem do distribuidor e o percentual de impostos, calcular e imprimir o custo final do carro ao consumidor.

Entrada: O programa deve ler três valores na entrada: o preço de fábrica do carro, o percentual do distribuidor e o percentual de impostos. Cada valor aparece em uma linha de entrada. Todos os valores são do tipo float.

Saída: O programa deve imprimir uma linha, contento a frase O VALOR DO CARRO E = Z, onde Z é o valor do preço final do carro ao consumidor. O valor de Z deve ter duas casas decimais. Após imprimir o valor do preço final, o program deve imprimir o caractere de quebra de linha '\n'.

Caso de teste: Entrada: 25000 12 30

Saída: O VALOR DO CARRO E = 35500.00

E7.2 – Distância entre dois pontos:

Dados dois pontos A e B, cujas coordenadas A(x_1 , y_1) e B(x_2 , y_2) serão informadas via teclado, desenvolver um programa que calcule a distância entre A e B.

Entrada: O programa deve ler os quatro valores reais correspondendo às coordenadas dos dois pontos : x_1 , y_1 , x_2 , y_2 ,nessa ordem, e um valor por linha.

Saída: O programa deve imprimir uma linha contendo a frase: A DISTANCIA ENTRE A e B = X, onde X é o valor da distância entre os dois pontos e deve conter no máximo 2 casas decimais. Após o valor da distância, o programa deve imprimir um caractere de quebra de linha: '\n'.

Observação: A distância entre dois pontos é computada pela fórmula:

$d = \mathbf{sqrt}((x_2 - x_1)^2 + (y_2 - y_1)^2)$ → ***sqrt*** é função da biblioteca <math.h>

Você pode usar a função sqrt() para calcular a raiz quadrada na fórmula da distância. Para computar o quadrado de um valor x você pode usar a função ***pow***(x,2). Para usar essas funções, você precisa colocar ***#include <math.h>*** no início do texto do programa.

Caso de teste: Entrada: 3 4 5 6

Saída: A DISTANCIA ENTRE A e B = 2.83

E7.3 – Número Invertido:

Escreva um programa para ler um número de três dígitos e imprimir o número invertido.

Entrada: A entrada contém apenas um número com três dígitos. Esse número é diferente de zero e não é múltiplo de 10 ou 100.

Saída: A saída deve conter apenas uma linha com o número correspondente ao valor da entrada, com seus dígitos invertidos. Logo após o número, deve ser impresso o caractere de quebra de linha: ’\n’.

Caso de teste: Entrada: 123 Saída: 321

Entrada: 987 Saída: 789

7.4 – Valor em notas e moedas:

Escreva um algoritmo par ler um valor em reais e calcular qual o menor número possível de notas de $R 100, $R 50, $R 10 e moedas de $R 1 em que o valor lido pode ser decomposto. O programa deve escrever a quantidade de cada nota e moeda a ser utilizada.

Entrada: O programa deve ler uma única linha na entrada, contendo um valor em Reais. Considere que somente um número inteiro seja fornecido como entrada.

Saída: O programa deve imprimir quatro frases, uma em cada linha: NOTAS DE 100 = X, NOTAS DE 50 = Y , NOTAS DE 10 = Z, MOEDAS DE 1 = W , onde X, Y , Z e W correspondem às quantidades de cada nota ou moeda necessárias para corresponder ao valor em Reais dado como entrada. Após cada quantidade, o programa deve imprimir um caractere de quebra de linha: ‘\n’.

Caso de teste: Entrada: 46395

Saída: NOTAS DE 100 = 463

NOTAS DE 50 = 1

NOTAS DE 10 = 4

MOEDAS DE 1 = 5

Exercícios de decisão:

7.5 – Notas em conceitos:

Em um curso de mestrado as avaliações dos alunos no histórico escolar aparecem em forma de conceito. O regulamento do mestrado indica que um professor pode avaliar seus alunos com notas convencionais de zero a dez, mas precisa repassar à secretaria do curso a avaliação em termos de conceito. Nesse caso, a seguinte tabela de conversão deve ser usada pelo professor:

Intervalo da Nota	**Conceito**
[9.0 a 10.0]	**A**
[7.5 a 8.9]	**B**
[6.0 a 7.4]	**C**
[0.0 a 5.9]	**D**

Escreva um programa para ler uma nota e converte-la no conceito correspondente.

Entrada: A entrada consiste em uma linha com um valor real entre 0 e 10 e com uma casa decimal.

Saída: O programa deve imprimir a seguinte frase: NOTA = x CONCEITO = y, onde x é o valor da nota lido na entrada, impresso com uma casa decimal y é o conceito correspondente. Após a frase, o programa deve imprimir o caractere de quebra de linha: '\n'.

Caso de teste: Entrada: 3.4 saída: NOTA = 3.4 CONCEITO = D

Entrada: 6.0 saída: NOTA = 6.0 CONCEITO = C

7.6 – Reajuste Salarial:

Fazer um algoritmo que calcule e imprima o salário reajustado de um funcionário de acordo com as seguintes regras:

• Salário de até R$ 300,00, reajuste de 50%;

• Salário maior que R$300,00 reajuste de 30%;

Entrada: A entrada conterá uma linha com o salário do funcionário.

Saída: A saída deve conter, numa linha com a frase: SALARIO COM REAJUSTE = x, onde x é um valor real com duas casas decimais e corresponde ao valor do salário reajustado. Logo em seguida ao valor de x, o programa devem imprimir o caractere de quebra de linha: '\n'.

Caso de Teste: Entrada: 755.00 Saída: SALARIO COM REAJUSTE = 981.50

Entrada: 265.32 Saída: SALARIO COM REAJUSTE = 397.98

7.7 – Soma dos 3 menores valores:

Fazer um programa para quatro valores inteiros e imprimir a soma dos três menores.

Entrada: O programa deve ler quatros valores inteiros na entrada. Cada valor ocupa uma linha na entrada.

Saída: O programa deve imprimir uma linha contendo o valor da soma dos três menores números. Após o valor da soma, o programa deve imprimir um caractere de quebra de linha: '\n'.

Caso de teste: Entrada: 9 4 2 12 Saída: 15

7.8 – Ordem:

Você receberá três valores inteiros e deve descobrir quais deles correspondem às variáveis a, b e c. Os números não serão dados em ordem exata, mas sabemos que o valor

correspondente a a é menor do que o valor correspondente a b, e que o valor correspondente a b é menor do que o correspondente a c . Será informada a você a ordem em que os valores associados a cada variável devem ser impressos. Escreva um programa que imprima os valores na ordem requisitada.

Entrada: A entrada conterá duas linhas. A primeira, com três números inteiros positivos, separados entre si por um espaço. Todos os três números são inferiores ou iguais a 100. A segunda linha conterá três letras maiúsculas A, B e C (sem espaços entre elas) representando a ordem desejada de impressão dos valores das variáveis.

Saída: A saída deve conter, numa linha, os inteiros a, b e c na ordem desejada, separados por espaços simples. Após o último número da saída deve aparecer apenas o caractere de quebra de linha: '\n'.

Observação: Após o último número na primeira linha da entrada, está no buffer de entrada o caractere '\n'. Com isso ao tentar ler o primeiro caractere (A, B, ou C) na segunda linha de entrada com scanf("%d", &x); será lido o caractere '\n' na variável x, ao invés de uma das letras na entrada (A, B, ou C). Para evitar isso, você pode fazer com que a leitura do último número na primeira linha consuma o caractere '\n' da primeira linha, colocando esse caractere na especificação de formato do scanf () . Por exemplo, suponha que você declarou as seguintes variáveis na entrada: int a , b,c; para armazenar os três número da primeira linha e char x, y, z;, para armazenar as três letras que aparecem na segunda linha de entrada. A leitura dessas variáveis de entrada pode ser realizada assim: scanf("%d %d %d\n", &a, &b, &c); scanf(%c%c%c", &x, &y, &z); Repare o '\n' ao final da formatação do primeiro scanf e repare que não há espaços entre os "%c"na formatação do segundo scanf. O '\n'. ao final da formatação do primeiro scanf () faz com que o caractere de quebra de linha seja consumido no buffer. Asssim, no segundo scanf () será armazenada na variável x a primeira letra da segunda linha e não o '\n', resolvendo o problema da leitura.

Casos de teste:

Entrada	Saída
1 5 3 A B C	1 3 5
6 4 2 C A B	6 2 4

Exercícios de Repetição:

7.9 - Somatório simples:

Faça um programa que leia um valor n, inteiro e positivo, calcule e mostre a seguinte soma:

$$S = \sum_{k=1}^{n} 1/k = 1 + 1/2 + 1/3 + 1/4 + \ldots + 1/n$$

Entrada: O programa deve ler um número inteiro positivo e maior que 1.

Saída: O programa deve apresentar uma linha contendo o valor final do somatório com 6 casas decimais. Caso o número lido não atenda as especificações da entrada, o programa deve apresentar a mensagem: "Numero invalido!".

Observação: Use precisão dupla para o cálculo de S.

Caso de Teste: Entrada: 10 Saída: 2.928968

7.10 – Companhia de Teatro:

Uma companhia de teatro deseja dar uma série de espetáculos. A direção calcula que o ingresso sendo vendido ao valor comum de mercado (ValorIngresso), serão vendidos 120 ingressos e que as despesas fixas serão de R$ 200,00 mais R$ 0,05 por cada ingresso. Diminuindo-se R$ 0,50 o preço dos ingressos, espera-se que as vendas aumentem em 25 ingressos. Aumentando-se R$ 0,50 o preço dos ingressos, espera-se que as vendas diminuam 30 ingressos. Para resolver este problema, a companhia de teatro deseja que você faça um programa que escreva uma lista de valores de lucros esperados em função do preço do ingresso, fazendo-se variar esse preço de A a B de R$ 1,00 em R$ 1,00. O programa deve apresentar na tela um resumo contendo o preço do ingresso informado, o lucro máximo calculado e a quantidade de ingressos vendidos para a obtenção desse lucro.

Entrada: O programa deve ler três números reais: ValorIngresso, correspondente ao valor de mercado dos ingressos, ValorInicial e ValorFinal correspondentes ao intervalo de valores que se deseja testar. Caso o ValorInicial informado seja maior ou igual ao ValorFinal, o programa deve encerrar após apresentar a mensagem: "INTERVALO INVALIDO."

Saída: O programa deve apresentar na tela uma linha para cada valor testado com o seguinte formato: "V: xxx.xx, N: xxx, L: xxx.xx", onde V é o valor do ingresso, N é a quantidade de ingressos vendidos e L o lucro obtido. Ao final, o programa deve apresentar um resumo contendo três linhas com o seguinte formato:

"Melhor valor final: xxx.xx"

"Lucro: xxx.xx"

"Numero de ingressos: xx"

Observação: Todos os valores reais devem ser apresentados com 2 casas decimais. Caso o intervalo de valores indicados não produza lucro positivo, os valores que devem aparecer no resumo devem assumir o valor zero.

Casos de testes:

Entradas	Saídas
5 2 8	V: 2.00, N: 270, L: 326.50 V: 3.00, N: 220, L: 449.00 V: 4.00, N: 170, L: 471.50 V: 5.00, N: 120, L: 394.00 V: 6.00, N: 60, L: 157.00 V: 7.00, N: 0, L: -200.00 V: 8.00, N: -60, L: -677.00 Melhor valor final: 4.00 Lucro: 471.50 Numero de ingressos: 170
0.5 0 2	V: 0.00, N: 145, L: -207.25 V: 1.00, N: 90, L: -114.50 V: 2.00, N: 30, L: -141.50 Melhor valor final: 0.00 Lucro: 0.00 Numero de ingressos: 0

7.11 – Mínimo Múltiplo Comum:

Faça um programa que calcule o Mínimo Múltiplo Comum (MMC) de 3 números inteiros. A Figura 7.1 apresenta um exemplo de cálculo de MMC.

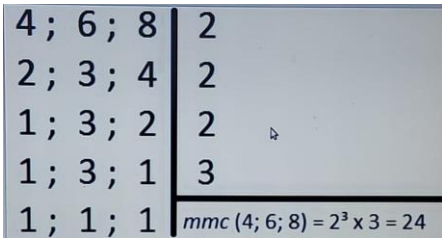


Figura 7.1 – Cálculo de MMC.

Entrada: O programa deve ler 3 números inteiros diferentes de zero.

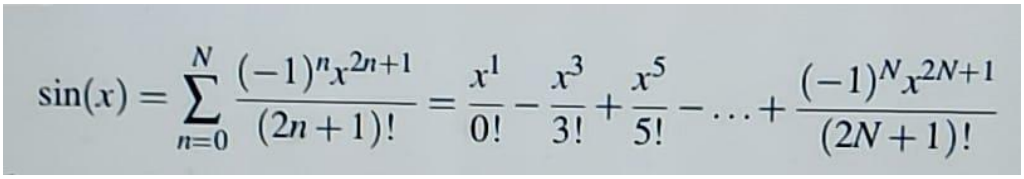
Saída: A saída é composta por várias linhas. O programa deve replicar a saída do procedimento da Figura 7.1 com expresso nos exemplos de entrada e saída. Cada linha deve ser impressa com o seguinte código: "%d %d %d :%d", onde o número 5 indica a quantidade mínima de espaços ou dígitos do número a ser apresentado.

Casos de teste:

Entrada	Saída
10 5 6	10 5 6 :2 5 5 3 :3 5 5 1 :5 MMC: 30
3 5 7	3 5 7 :3 1 5 7 :5 1 1 7 :7 MMC: 105

7.12 – Série de Taylor seno:

Escreva um programa que dado um número real x e a quantidade de termos N, calcule o valor da função sin(x), a partir da série:



$$\sin(x) = \sum_{n=0}^{N} \frac{(-1)^n x^{2n+1}}{(2n+1)!} = \frac{x^1}{0!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + \frac{(-1)^N x^{2N+1}}{(2N+1)!}$$

, onde x é o ângulo em radianos e N a quantidade de termos da série menos 1.

Entrada: O programa deve ler o valor de x e N.

Saída: O programa deve apresentar uma linha contendo o texto "seno(x) = y\n", onde x é o ângulo fornecido pelo usuário e y o seno do ângulo. x deve ser impresso com 2 casas decimais e y com 6 casas decimais.

Observação: Neste tipo de problema, a quantidade de termos pode gerar números muito grandes por conta da operação de fatorial e potenciação de x. Atente-se aos tipos de dados usados nas declarações das variáveis e não use valores de N maiores que 9. Lembre-se que um ângulo qualquer sempre pode ser representado por um valor entre 0 e 2π. Use a constante M_PI da biblioteca <math.h>. Como sugestão de desafio à solução do problema, tente escrever um algoritmo que use apenas um laço de repetição.

Caso de teste:

Entrada	Saída
2 9	seno(2.00) = 0.909297
3.14	seno(3.14) = 0.001614

6	
1 4	seno(1.00) = 0.841471

Encerramos aqui uma lista de exercícios para prática desse conteúdo dado no Capítulo 4. A seguir veremos vários outros comandos dentro das estruturas homogêneas trabalhadas em vetores e matrizes.

A seguir colocaremos as respostas dos exercícios dados nessa sessão.

R7.1 – Custo final do Carro:

```
#include <stdio.h>
int main(){
	float precoFabrica,percDist, percImposto,precoFinal;
	scanf("%f",&precoFabrica);
	scanf("%f",&percDist);
	scanf("%f",&percImposto);
	printf("O VALOR DO CARRO E = %.2f\n",precoFabrica+percDist/100*precoFabrica+percImposto/100*precoFabrica);
}
```

R7.2 - Distância entre dois pontos:

```
#include <stdio.h>
#include <math.h>
int main(){
	float x1,y1,x2,y2,dist;
	scanf("%f",&x1);
	scanf("%f",&y1);
	scanf("%f",&x2);
	scanf("%f",&y2);
	dist=sqrt( (x1-x2)*(x1-x2)+(y1-y2)*(y1-y2));
	printf("A DISTANCIA ENTRE A e B = %.2f\n",dist);
}
```

R7.3 – Numero Invertido:

```
#include <stdio.h>
int main() {
	int n, i, dig1, dig2, dig3;
	//Lendo um numero inteiro:
	scanf("%d",&n);
	//Isolando os digitos do numero:
	dig1=n/100;
	dig2=n%100;
	dig2=dig2/10;
	dig3=n%10;
```

```
        //ImpriminRdo os digitos na ordem inversa
        //As duas formas alternativas abaixo  funcionam
        //printf("%d\n",dig3*100+dig2*10+dig1);
        printf("%d%d%d\n",dig3,dig2,dig1);
        return 0;
}
```

R7.4 – Valores em Notas:

```
#include <stdio.h>
int main()
{
    int valor, n100, n50, n10, m1, resto;
    scanf("%d",&valor);
        n100=valor/100;
        resto=valor%100;
        n50=resto/50;
        resto=resto%50;
        n10=resto/10;
        m1=resto%10;
        printf("NOTAS DE 100 = %d\n", n100);
        printf("NOTAS DE 50 = %d\n", n50);
        printf("NOTAS DE 10 = %d\n", n10);
        printf("MOEDAS DE 1 = %d\n", m1);
    return 0;
}
```

R7.5 – Notas Conceitos:

```
#include <stdio.h>
int main() {
        float nota;
        char conceito;
        scanf("%f", &nota);
        if (nota<6.0){
                conceito='D';
        }
        else{
                if(nota<7.5){
                        conceito='C';
                }
                else{
                        if(nota<9.0){
                                conceito='B';
                        }
                        else{
                                conceito='A';
                        }
                }
```

```
        }
        printf("NOTA = %.1f CONCEITO = %c\n", nota,conceito);
        return 0;
}
```

R7.6 – Reajuste Salarial:

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

int main(){

        float salAtual, salReajustado;
        scanf("%f", &salAtual);
        if (salAtual <=300.0){
                salReajustado=salAtual+ 0.5*salAtual;
        }
        else{
        salReajustado=salAtual+ 0.3*salAtual;
        }
        printf("SALARIO COM REAJUSTE = %.2f\n", salReajustado);
        return 0;
}
```

R7.7 – Soma dos 3 menores valores:

```
#include <stdio.h>
int main(){
        int n1, n2,n3,n4,maior, soma;
        scanf("%d",&n1);
        scanf("%d",&n2);
        scanf("%d",&n3);
        scanf("%d",&n4);
        soma=n1+n2+n3+n4;
        if(n1>=n2 && n1>=n3 && n1>=n4){
                maior=n1;
        }
        if(n2>=n1 && n2>=n3 && n2>=n4){
                maior=n2;
        }
        if(n3>=n1 && n3>=n2 && n3>=n4){
                maior=n3;
        }
        if(n4>=n1 && n4>=n2 && n4>=n3){
                maior=n4;
        }
        printf("%d\n",soma-maior);
```

```
}
```

R7.8 – Ordem:

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

int main(){

	int a,b,c, menor, meio, maior;
	char X, Y,Z;
	scanf("%d %d %d\n", &a, &b, &c);
	//scanf("%d", &b);
	//scanf("%d\n", &c);
	scanf("%c%c%c", &X, &Y, &Z);
	//scanf("%c", &Y);
	//scanf("%c", &Z);
	//Encontrar o menor elemento, o intermediario e o maior elemento
	if (a<=b && a<=c){
		menor=a;
		if(b<=c){
			meio=b;
			maior=c;
		}
		else{
			meio=c;
			maior=b;
		}
	}
	else{
		if (b<=a && b<=c) {
			menor=b;
			if(a<=c){
				meio=a;
				maior=c;
			}
			else{
				meio=c;
				maior=a;
			}
		}
		else{
			menor=c;
			if(a<=b){
				meio=a;
				maior=b;
			}
			else{
```

```
                        meio=b;
                        maior=a;
                    }
                }
            }
        if (X=='A') {
            printf("%d ", menor);
        }
        if (X=='B') {
            printf("%d ", meio);
        }
        if (X=='C') {
            printf("%d ", maior);
        }
        if (Y=='A') {
            printf("%d ", menor);
        }
        if (Y=='B') {
            printf("%d ", meio);
        }
        if (Y=='C') {
            printf("%d\n", maior);
        }
        if (Z=='A') {
            printf("%d\n", menor);
        }
        if (Z=='B') {
            printf("%d\n", meio);
        }
        if (Z=='C') {
            printf("%d\n", maior);
        }
        return 0;
}
```

R7.9 – Somatório Simples:

```
#include <stdio.h>

int main() {

        int n, k;
        double soma;
        scanf("%d",&n);
        if( n <= 1 ) {
            printf("Numero invalido!\n");
            return 0;
        }
        soma = 0.0;
        k = 1;
```

```
	while( k <= n ) {
		soma +=  1.0/k;
		k++;
	}
	printf("%.6f", soma);
	return 0;
}
```

R7.10 – Companhia do Teatro:

```
#include <stdio.h>

int main() {
	double ValorIngresso, ValorInicial, ValorFinal;
	double MelhorValor, MelhorLucro;
	int MelhorNIV;
	double val, lucro;

	int niv;// número de ingressos vendidos

	scanf("%lf %lf %lf",&ValorIngresso, &ValorInicial, &ValorFinal);

	if( ValorInicial >= ValorFinal ) {
		printf("INTERVALO INVALIDO.\n");
		return 0;
	}

	MelhorLucro = 0;
	MelhorValor = 0;
	MelhorNIV = 0;

	val = ValorInicial;
	while( val <= ValorFinal ) {

		if( val < ValorIngresso ) {
			niv = 120 + (ValorIngresso-val)*25/0.5;
		} else {
			niv = 120 - (val-ValorIngresso)*30/0.5;
		}

		lucro = (niv*val) - (200+niv*0.05);
		printf("V: %.2f, N: %i, L: %.2f\n", val, niv, lucro);

		if( lucro > MelhorLucro ) {
			MelhorLucro = lucro;
			MelhorValor = val;
			MelhorNIV = niv;
		}
```

```
            val += 1;
        }
        printf( \
            "Melhor valor final: %.2f\n" \
            "Lucro: %.2f\n" \
            "Numero de ingressos: %i\n", MelhorValor, MelhorLucro, MelhorNIV);
        return 0;
}
```

R7.11 – Minimo Multiplo comum:

```
#include <stdio.h>

int main() {

        int n1, n2, n3;
        int m;
        int mmc;
        mmc = 1;
        m = 1;
        scanf("%d %d %d", &n1, &n2, &n3);
        while( n1 > 1 || n2 > 1 || n3 > 1 ) {
            m = 2;

            while( n1%m!=0 && n2%m!=0 && n3%m!=0 ) {
                m++;
            }
            printf("%d %d %d :", n1, n2, n3);
            printf("%d\n", m);
            mmc *= m;
            if( n1%m == 0) n1 /= m;
            if( n2%m == 0) n2 /= m;
            if( n3%m == 0) n3 /= m;
        }
        printf("MMC: %d\n", mmc);
        return 0;
}
```

R7.12 – Série de Taylor seno:

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

int main() {
        double x, sinx, px;
        int n, N;
        long int f;
```

```
	char s = -1;

	scanf("%lf%d",&x,&N);
	//x = x*180/3.14;
	x = x/(2*M_PI);
	x = x - (int)x;
	x = x*(2*M_PI);

	//printf("sin(%Lf) = \n", x);

	f = 1;
	px = x;
	sinx = x;
	s = 1;
	//printf(" + %Lf/1 = %Lf\n", px, sinx);
	for( n = 1; n <= N; n++ ) {
		px *= x*x;
		f *= (2*n+1)*(2*n);
		s = -s;
		sinx += s*px/f;
		//printf("%s", s<0?" - ":" + ");
		//if(s<0) printf("%.2Lf/%lld = %Lf\n", px, f, s*px/f);
		//else printf("%.2Lf/%lld = %Lf\n", s*px, f, s*px/f);
	}
	printf("seno(%.2lf) = %.6lf\n", x, sinx);
	return 0;
}
```

CAPÍTULO V

DADOS ESTRUTURADOS

1 – Introdução.

Neste capítulo iremos trabalhar com as estruturas homogenias (Vetores e Matrizes) ampliando a forma de armazenamento de nossas informações. Depois será introduzido geração de tipos de dados heterogêneos onde ajudarão a dar exemplos mais reais. Finalizando o capítulo com armazenamentos da informação em arquivos textos ou binários.

Vários exemplos serão mostrados para ilustrar o uso dessas estruturas e como mapear os elementos armazenados nela. O usuário tem uma visão externa de como essas estruturas funcionam, porém a implementação computacional tem outra forma de armazenar, para isso faremos um mapeamento sempre informando na visão do usuário.

Livro Didático para leitura: Waldemar Celes, Renato Cerqueira, José Lucas Rangel, *Introdução a Estruturas de Dados* , Editora Campus (2004). Thomas Cormen, e outros – Algorithmos, teoria e prática, 3ª Edição, 2012.

2 – Estruturas homogêneas unidimensionais (vetor)

Nos capítulos anteriores estudamos as opções disponíveis na linguagem C para representar:

- Números inteiros em diversos intervalos.
- Números fracionários com várias alternativas de precisão e magnitude.
- Letras, dígitos, símbolos e outros caracteres.

Todas estas opções estão definidas previamente na própria linguagem C e, portanto, são chamadas de tipos de dados primitivos da linguagem C. Iremos introduzir mecanismos para definir outros tipos de dados, conforme a necessidade do seu programa. Assim, eles são chamados de tipos de dados do programador. Porém iniciaremos com tipos Homogêneos, ou seja, cria um vetor de um mesmo tipo de dados primitivo (inteiro, real ou letras). Depois avançaremos para outros tipos Heterogêneos.

Um vetor é uma sequência de vários valores do mesmo tipo, armazenados sequencialmente na memória, e fazendo uso de um mesmo nome de variável para acessar esses valores. Um vetor também pode ser entendido logicamente como uma lista de elementos de um mesmo tipo. Cada elemento desta sequência pode ser acessado individualmente através de um índice dado por um número inteiro. Os elementos são indexados de 0 até N-1, onde N é a quantidade de elementos do vetor. O valor de N também é chamado de dimensão ou tamanho do vetor. O vetor tem tamanho fixo durante a execução do programa, definido na declaração, sendo assim é uma estrutura estática. Durante a execução não é possível

aumentar ou diminuir o tamanho do vetor. Note que a numeração começa em zero, e não em um. Essa é uma fonte comum de erros.

A Figura 5.1 ilustra um vetor com 10 elementos, denominados v_0, v_1, … v_9, todos eles do tipo int. É importante saber que os elementos do vetor são armazenados sequencialmente na memória do computador. Assim, na figura, se cada valor de tipo int ocupar 4 bytes de memória, teremos 40 bytes consecutivos reservados na memória do computador para armazenar todos os valores do vetor. No entanto, por ora, não faremos uso explícito dessa informação, uma vez que o compilador se encarregará de endereçar cada elemento do vetor automaticamente, conforme as necessidades do programador, como veremos. Declaração de vetores

int vetor[10]; → o compilador reserva um local sequencial para armazenar 10 int

10	8	11	16	9	14	21	1	18	3	vetor
0	1	2	3	4	5	6	7	8	9	

Figura 5.1 – Demonstração da alocação de um vetor de tamanho 10 de inteiros

A declaração de vetores obedece à mesma sintaxe da declaração de variáveis. A diferença está no valor entre colchetes, que determina quantos elementos ele armazenará, ou seja, em outras palavras, determina o seu tamanho ou dimensão. Observe no exemplo da Figura 5.1 (int vetor[10];). Ao declarar o vetor o compilador reserva o espaço solicitado e põem o nome da variável nesse espaço.

Para declarar um vetor de um determinado tipo basta por tipo nome_vetor[tamanho]; onde o tipo pode ser qualquer tipo primitivo ou um tipo criado pelo usuário. O nome do vetor deve ser sempre algo que represente seu uso no mundo real do problema. O tamanho deve ser pré-definido, não podendo ser alterado em tempo de execução.

O tamanho do vetor deve ser um número inteiro ou constante pré-estabelecido no programa, não pode ser uma expressão do tipo: int vetor[n*2]; onde n será lido anteriormente (esse tipo de declaração é ERRADA).

Conforme já observado anteriormente, uma invocação de nome da variável é possível acessar um dado na memória. O tipo do valor retornado é igual ao tipo utilizado na declaração da variável. O nome da variável também é denominado de uma expressão de referência de memória, ou simplesmente de referência de memória. No caso de vetores, temos uma nova expressão de referência de memória: o operador de índice entre [ ]. Ele utiliza uma referência de memória (normalmente uma variável do tipo vetor) e um número inteiro para representar o

índice. Ele retorna uma referência para o elemento correspondente ao índice. O tipo do valor retornado é o mesmo tipo da declaração do vetor.

Conforme mostrado na Figura 5.1 o valor armazenado em vetor[3] é o valor 16. Porém esse valor seria o elemento 4 para o usuário e não o 3 armazenado. Para atribuir o valor 10 na primeira posição do vetor (armazenado na posição 0) devemos fazer um comando de atribuição da seguinte forma:

```
vetor[0] = 10;
```

Note que o índice zero indica a primeira posição no vetor (computacionalmente). A expressão vetor[0] referência a posição de memória correspondente ao elemento de índice zero no vetor. Para somar os primeiros três elementos do vetor e armazenar o resultado no quarto elemento do vetor, faremos:

```
vetor[3] = vetor[0] + vetor[1] + vetor[2];
```

Em expressões, uma referência indexada a um vetor pode ser usada da mesma forma e nas mesmas posições em que usaríamos variáveis convencionais de mesmo tipo. Tudo se passa como se tivéssemos várias variáveis declaradas simultaneamente, todas de mesmo tipo, e com “nomes” vetor[0], vetor[1], e assim por diante. É muito comum utilizar a estrutura de repetição for para percorrer todos os elementos de um vetor. Por exemplo, para imprimir todos os elementos de um vetor de 100 elementos:

```
int indice;
int vetor[100];
...
for (indice = 0; indice < 100; indice++) {
      printf("%d", vetor[indice]);
}
```

Em geral na programação para ler todos os elementos do vetor conforme mostrado na Figura 5.1 faremos:

```
int vetor[10], i;
for(i=0;i<10;i++)
    scanf("%d",&vetor[i]); //entrada de valores pelo usuário
```

o mesmo para efetuar a escrita do vetor:

```
for(i=0;i<10-1;i++)
     printf("%d - ",vetor[i]);
printf("%d\n",vetor[i]); //exibição dos valores do vetor
```

Lembre-se para um vetor de tamanho N indicado pelo problema o vetor irá de 0 até N-1, por ser uma estrutura estática utilizamos uma constante com uso da diretiva (*#define N 1000*) para usarmos na declaração do vetor no programa principal.

Supomos que um exercício tenha que trabalhar com a entrada de um inteiro positivo maior que 2 e menor que 1000, para entrarmos com n elementos do tipo inteiro variando de 0 a 100. Neste caso, por não sabermos qual o valor inicial, utiliza-se a capacidade máxima solicitada pelo problema:

```
#include <stdio.h>
#define N 1000

int main() {
 int vetor[N], n, i;
 printf("Entre com a quantidade de elementos a ser usado de 3 a 1000\n");
 scanf("%d",&n);
 printf("Entre com oa valores do vetor: \n");
 for(i=0;i<n;i++) {
    printf("\nvetor[%3d] = ",i);
        scanf("%d",&vetor[i]);
 }
 printf("\nExibindo o vetor : \n");
 for(i=0;i<n;i++) {
    printf("\nvetor[%3d] = %3d",i,vetor[i]);
 }
return 0;
}
```

Observe que mesmo que o vetor tenha 1000 espaços de inteiro reservado na memória para armazenamento, só utilizaremos n posições (entrada pelo usuário). Isso acontece quando o vetor pode ter tamanho variável em tempo de execução, o usuário é quem definirá quantos elementos serão usados no vetor.

Quando conhecemos a quantidade de valores exata de um vetor, e quais são seus elementos, podemos declarar e já definir os valores para as posições do vetor. Exemplo:

tipo variável[n] = { $elem_0$, $elem_1$, $elem_2$, ... $elem_{n-1}$ };

ou

tipo variável[ ] = { $elem_0$, $elem_1$, $elem_2$, ... $elem_{n-1}$ };

Neste último ele monta o vetor com tamanho n.

O acesso aos valores de um vetor temos que tomar cuidado com os limites declarados no vetor(tamanho máximo do vetor). Passar pelo indice (n-1) de um vetor significa entrar em área não pertencente mais ao vetor, podendo ocasionar erros graves ou desconhecidos. Para

buscar um valor dentro de um vetor onde seus valores estão em uma ordem qualquer (aleatória), devemos usar um laço de consulta conforme mostraremos abaixo:

```
#include <stdio.h>
#include <stdlib.h>
#define N 10

//  10  40  20  15  18  6   32  44  1  33
int main() {
        int vet[N], i, x;
        //Entrada dos valores do vetor
        for(i=0;i<N;i++)
           scanf("%d",&vet[i]);
        //Entrada de um valor que pretendemos buscar no vetor
        printf("Entre com um valor que deseja buscar no vetor\n");
        scanf("%d",&x);
        i=0;
        while((i<N)&&(x!=vet[i])) //laço para buscar um valor no vetor
           i++;
        if(i==N)
          printf("O valor %d nao esta no vetor!!!!\n",x);
         else
          printf("O valor %d esta na posicao %d do vetor!!!!\n",x,i+1);
return 0;
}
```

Para copiar um vetor A em um vetor B não podemos fazer como variáveis simples tipo B = A; no caso de vetor temos que utilizar um comando de laço (for), para passar elemento por elemento do tipo:

```
for(i=0;i<N;i++)
    B[i] = A[i]; //sendo A e B dois vetores de mesmo tipo de dados
```

As buscas em um vetor são controladas pelo tipo de entrada que o vetor teve, se os valores estiverem em uma ordem crescente, podemos mudar a forma de buscar um valor no vetor da seguinte forma:

```
#include <stdio.h>
#include <stdlib.h>
#define N 10

//  5  10  15  17  18  20   30  44  50  63 (ordenado)
int main() {
        int vet[N], i, x;
        for(i=0;i<N-1;i++)
```

```
            scanf("%d",&vet[i]);

          printf("Entre com um valor que deseja buscar no vetor\n");
          scanf("%d",&x);
          i=0;
          while((i<N)&&(x<vet[i]))
             i++;
          if((i==N)||(x!=vet[i]))
            printf("O valor %d nao esta no vetor!!!!\n",x);
           else
       printf("O valor %d esta na posicao %d do vetor!!!!\n",x,i+1);
return 0;
}
```

Existem outras formas de busca como a busca binária, quando os valores estão ordenados. A busca binária procura o elemento central do vetor e compara com o valor de x, caso o valor seja menor, a busca continua na parte inicial do vetor (1ª Metade), caso seja maior continuará na parte final do vetor (2ª Metade), e assim sucessivamente o algoritmo irá quebrar o vetor em 2 partes até alcançar o valor ou fazer (inicio > fim).

Exemplo de busca binária:

```
#include <stdio.h>
#define MAX 20

int main() {
int vet[MAX], i, x, ini, fim, meio;

for(i=0;i<MAX;i++) {
  scanf("%d",&vet[i]);
}
for(i=0;i<MAX-1;i++) {
  printf("%d - ",vet[i]);
 }
printf("%d\n",vet[i]);

printf("Entre com valor procurado:");
scanf("%d",&x);
ini = 0;
fim = MAX-1;
meio = (ini + fim)/2;
while(ini <= fim)
  { meio = (ini + fim)/2;
          if(x == vet[meio]) {
             printf("valor encontrado de %d esta na posicao %d\n",x,meio+1);
             fim = -1;
   }
```

```
          else if(x<vet[meio])
                 fim = meio - 1;
                else ini = meio + 1;
}
if(fim != -1)
  printf("\nO valor %d nao esta no vetor\n",x);

return 0;
}
```

Desta forma o algoritmo de busca binária se torna mais rápido quando N passar de 1000 elementos. Por exemplo para N = 2^{20} = 1.048.576 elementos, significa que em 20 consultas no pior caso encontraremos o valor de x caso ele esteja no vetor.

Um vetor de char é também denominado de string, ou cadeia de caracteres. Por exemplo um nome pode ser armazenado em um string de tamanho 40:

char nome_cliente[40];

Neste caso se o usuário utilizar um nome inferior a 40 caracteres ele será gravado no veto nome_cliente, colocando depois do último caractere um encerramento que é representado pelo caractere ‘\0’. Se o tamanho do nome for 40 ou mais só será gravado 39 caracteres, sendo que na última posição do vetor será colocado o caractere ‘\0’.

A leitura de uma string não precisa utilizar o endereço (representado pelo caractere &) como fazemos com inteiro, char, float e outros. Neste caso podemos usar:

scanf(“%s”,nome_cliente);

onde nome_cliente representa o endereço da primeira posição do vetor de char. Ou seja, é mesmo que escrever &nome_cliente[0]. Para ler um nome com espaço e completo com scanf devemos utilizar a diretiva scanf(“[\n^]”,nome_cliente); se usarmos só o %s quando ocorrer o espaço tipo Joao da Silva ele só lerá Joao e o restante será jogado para as próximas leituras. Outro comando que podemos usar para ler string é o comando fgets(nome_cliente, 40, stdin); já comentado anteriormente.

Iniciaremos a seguir a estrutura bidimensional (matriz).

3 – Estruturas homogêneas bidimensionais (matriz)

A noção de matriz é a generalização imediata da noção de vetor. Uma matriz é uma estrutura de vários valores do mesmo tipo, armazenados sequencialmente e fazendo uso de um mesmo nome de variável para acessar esses valores. Cada elemento da matriz pode ser acessado individualmente através de dois índices com valores inteiros. Estes índices poderiam

ser interpretados como a linha e a coluna da matriz. A linguagem C define uma matriz como um vetor, cujos elementos são novamente vetores de mesmo tamanho e tipo. Na verdade, o número de linhas corresponde ao número de elementos do vetor externo, e o número de colunas é o tamanho dos vetores internos que constituem cada elemento dos vetores externos. A Figura 5.2 mostra como a matriz bidimensional é representada computacionalmente:

int matriz[4][10];

	0	1	2	3	4	5	6	7	8	9
$linha_0[10]$	V_0	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9
$linha_1[10]$	V_0	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9
$linha_2[10]$	V_0	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9
$linha_3[10]$	V_0	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9

Figura 5.2: representação de uma matriz 4x10

Todos os conceitos e todas as regras válidas para os vetores também são válidas para as matrizes. A matriz tem tamanho fixo, definido na declaração. As linhas são numeradas de 0 até linhas-1 e as colunas de 0 até colunas-1. Para colocar o valor 10 por exemplo no inicio da matriz fazemos:

matriz[0][0] = 10;

A declaração de matrizes obedece a mesma sintaxe que a declaração de vetores, exceto pela adição de uma nova dimensão escrita entre colchetes[].

tipo variável[linhas][colunas];

Por exemplo, para declarar uma matriz de números inteiros com 4 linhas e 10 colunas:

int matriz[4][10];

A atribuição ou leitura de valores em uma matriz utiliza dois índices. O primeiro elemento é armazenado em matriz[0][0] o segundo em matriz[0][1] e assim por diante, até o último elemento, que é armazenado em matriz[linhas-1][colunas-1]. Para atribuir o valor 3 na linha 2, coluna 5, escrevemos:

matriz[1][4] = 3;

Note como a linha 2 é numerada como 1, pois começamos a contar a partir do zero. Idem, a coluna 5 é numerada como 4. Para percorrer os elementos de uma matriz, são necessárias duas estruturas de repetição for, uma dentro da outra. O for externo percorre as linhas da

matriz, o for interno percorre as colunas de uma determinada linha que está fixada pelo for externo.

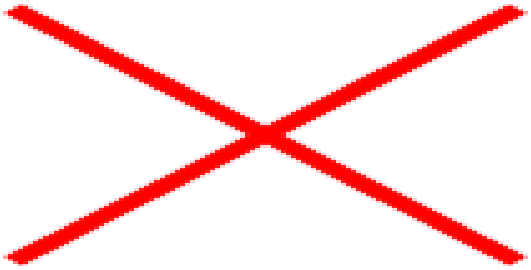
Por exemplo, para imprimir todos os elementos de uma matriz 4x10, linha por linha:

```
int linha, coluna;
int matriz[4][10];
...
for (linha = 0; linha < 4; linha++) {
      for (coluna = 0; coluna < 10; coluna++) {
            printf("%d ", matriz[linha][coluna]);
      }
      printf("\n");
}
```

para cada valor de linha fixado pelo for externo entre 0 e 3, o for interno executa, varrendo as colunas correspondentes àquela linha, de 0 a 9. Assim, quando o for interno executa, o valor de linha está fixado pelo for externo, fazendo com que o printf imprima os valores da linha linha da matriz.

Chamamos de Matriz a todo conjunto de “valores”, dispostos em linhas e colunas. Dada uma matriz A denotaremos cada elemento da matriz A por a_{ij} onde i é o número da linha e j é o número da coluna desse elemento.

A =



Muitas vezes trabalharemos com matriz quadrada, onde o número de linhas é igual ao número de colunas:

$$A=\begin{bmatrix} 1 & 3 & -5 \\ 0 & -2 & 4 \\ 2 & 3 & 6 \end{bmatrix}$$

Alguns tipos de Matriz serão mostradas a seguir. Matriz **transposta**, é um tipo de matriz trocando-se o numero da linha pelo número da coluna, exemplo:

$$A^T = \begin{bmatrix} 1 & 0 & 2 \\ 3 & -2 & 3 \\ -5 & 4 & 6 \end{bmatrix}$$

Matriz **identidade** é a matriz quadrada cujos elementos da diagonal principal são iguais a 1 e os demais elementos iguais a zero.

$$A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Matriz Nula, quando todos elementos são iguais a zero.

$$0 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Matriz triangular inferior: quando todos os elementos da diagonal principal para baixo são diferentes de 0 os demais são nulos (iguais a 0).

$$M_{ti} = \begin{pmatrix} 4 & 0 & 0 \\ 5 & 2 & 0 \\ 3 & 1 & 6 \end{pmatrix}$$

A matriz triangular superior é quando todos os elementos da diagonal principal para cima são diferentes de 0 os abaixo da diagonal são nulos (iguais a 0).

Matriz Diagonal é quando somente os elementos da diagonal principal são diferente de 0 os demais são 0.

$$D = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 3 \end{pmatrix}$$

Uma matriz pode sofrer uma multiplicação por um escalar k pré-definido, sendo que cada elemento da matriz fosse multiplicado pelo escalar k.

Matriz simétrica é quando os elementos opostos à diagonal principal são iguais (o elemento ($a_{ij} = a_{ji}$). A matriz A é simétrica quanto $A = A^T$.

A= 

Matriz Antissimétrica, quando os elementos da diagonal principal forem nulos e $a_{ij} = -a_{ji}$. Ou é o mesmo que dizer que $A = - A^T$.

A= 

A igualdade entre duas Matriz A e B é realizado elemento por elemento, só será igual se todos os elementos de uma matriz NxM ocorrer de $a_{ij} = b_{ij}$ sendo que i varia de 0 até N-1 e

j variar de 0 até M-1. A comparação deve ser realizada com 2 laços um percorrendo as linhas (representado pelo índice i) e outro percorrendo as colunas (representado pelo índice j).

```
int main( ) {
int A[N][M], B[N][M], i, j, igual = 0;
….
for (i=0;i<N;i++)
    for(j=0;j<M;j++)
        if(A[i][j] !=B[i][j])
            igual = 1;
….
return 0;
}
```

A soma ou subtração entre duas matrizes A e B só podem ocorrer se tiverem os mesmos números de linhas e colunas. Ja a multiplicação entre duas matrizes A e B só podem ocorrer se o número de colunas de A for igual ao número de linhas de B e a matriz resultante receberá o número de linhas de A e o número de Colunas de B.

Neste caso se C = AxB sendo que A tem a dimensão de nxp e B tem a dimensão pxm, então C terá a dimensão nxm.

Supondo que as matrizes M1, M2 e M3 se referem a M3 = M1xM2, onde M1, M2 e M3 são matrizes quadradas de dimensão N.

```
int main( ) {
 int M1[N][N],int M2[N][N], int M3[N][N])
 int i, j, k; int L1, L2, C1, C2;// sendo C colunas e L linhas
 C1 = N; C2 = N; L1 = N; L2 = N; // certo quando a matriz for quadrática
 ….
 for(i = 0; i < L1; i++) {
     for(j = 0; j < C2; j++) {
         M3[i][j] = 0;
         for(k = 0; k < C1; k++) {
             M3[i][j] += M1[i][k] * M2[k][j];
         }
     }
 }
 …..
 return 0;
}
```

Como M3 é o resultado de M1 x M2 k toma as ocorrências de C1 ou L2 por isso que tivemos que testar se essas variáveis são iguais, como nossa Matriz é quadrada então isso já acontece.

As buscas em matriz, ocorre da mesma forma que em vetor, utilizaremos laços, e buscar um determinado valor exige percorrer linhas e depois colunas até encontrar ou até atingir o final de todos os elementos da matriz.

Faremos alguns exercícios que solicitam a busca de um ou vários valores em uma determinada Matriz. Para isso utilizaremos **for**(i…) para percorrer as linhas da matriz e **for**(j…) para percorrer as colunas. Se encontrar o valor informaremos a linha e coluna, caso contrário avisaremos que o valor não está na matriz. Para isso utilizaremos uma variável achei = 0 e linha = -1 e coluna = -1. Ao encontrar o valor x na matriz mudaremos achei para 1 e linha será o número da linha e coluna o número da coluna para o usuário, que seria linha = i+1; e coluna = j+1.

4 – Uso de estruturas heterogêneas (registros)

O uso de registros, em C denominado de struct, são formas de criarmos tipos de dados heterogêneos, que representem a mesma informação do mundo real. O comando para criação de um tipo de dado em C é dado por:

```
typedef struct { int codigo;
                 char nome_func[40];
                 char data_nasc[11];
                 char email[60];
                 char cargo[30];
                 char data_admissao[11];
                 float salario;
               }Empregado;
```

Neste caso typedef está definindo um tipo de dados sendo um registro (ou um conjunto de valores) contendo inteiro, strings e float, e foi chamado de Empregado. Neste caso a partir desta criação Empregado pode ser usado no programa como um tipo do usuário. Como devemos armazenar os Empregados de uma empresa em uma estrutura de vetor sendo que a empresa possui menos que 1000 funcionários até o presente momento, mas que tem por exemplo intenção de contratações futuras podendo passar de 1000 funcionários. Neste caso podemos criar um tipo de dados para controlar os empregados da empresa.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define Nfunc 5000

typedef struct { int codigo;
```

```
                char nome_emp[40];
                char data_nasc[11];
                char email[60];
                char cargo[30];
                char data_admissao[11];
                float salario;
              }Empregado; // gastou 164 bytes
typedef struct {Empregado func[Nfunc];
                int quantidade_func;
                char Nome_empresa[50];
                char cnpj[20];
                char tel_empresa[15];
              }Empresa; // 820089 bytes
```

Observe que agora temos um tipo de dados heterogêneo denominado Empresa onde guarda todos os Empregados em um vetor de tamanho 5000. Neste tipo de dados temos as informações básicas da empresa.

Desta forma teremos um vetor com valores heterogêneos contendo todas as informações necessárias de um Empregado. A quantidade de empregados é o controle desse vetor, ou seja, inicialmente essa quantidade será 0, e marcará onde o empregado cadastrado no sistema será guardado no vetor. Quando tivermos 1000 empregados, a quantidade de empregados marcara 1000 e essa será a posição do 1001º empregado cadastrado no vetor que está no tipo de dados Empresa.

Como acessar os valores dentro dos registros (*struct*)? Inicialmente iremos mostrar como isso acontece. Para isso vamos supor que temos que declarar os tipos Empregado e Empresa como fazemos no início da função principal main( ).

```
        Empregado func;
        Empresa Emp;
        Emp.quantidade_func = 0; //iniciando o controle de quantidade de funcionário
        ….
return 0;
}
```

Quando criarmos um programa para controlar esses dados de uma Empresa e seus funcionários, devemos acrescentar funções e procedimentos, que veremos no capítulo seguinte. Apenas para ilustrar neste capítulo vamos criar um procedimento para efetuar a leitura de um empregado. Para isso devemos passar como parâmetro um Empregado e como

iremos alterar seus dados devemos passar como referência (ponteiro). Neste caso o procedimento seria:

```
void ler_empregado(Empregado *E) {
   printf(“Entre com o código do empregado: \n”);
   scanf(“%d”,&Eco->digo);
   getchar();
   printf(“Entre com o nome do empregado: \n”);
   fgets(E->nome_func, 40, stdin);
   printf(“Entre com o data de nascimento do empregado: \n”);
   fgets(E->data_nasc, 15,stdin);
   // e assim sucessivamente, até ler todas as informações do funcionário
}
```

Na função main esse procedimento será chamado da seguinte forma:

```
int main() {
  Empregado func;
  Empresa Emp;
  ….
   ler_empregado(&func);
   ….
 return 0;
}
```

Todo controle dos empregados de uma empresa neste caso é realizado com MENU e opções das principais funcionalidades que o usuário deverá ter para atender suas necessidades pré-estabelecidas. No MENU são colocadas as opções de cadastrar um empregado, consultar um empregado, gerar folha de pagamento, informar o maior salário da empresa, ou o menor salário da empresa, e assim sucessivamente, de acordo com as necessidades da empresa para gerar o controle de todos empregados.

Em capítulos mais a seguir iremos abordar novamente a definição de tipos de dados heterogêneos (***struct***), para ilustrar outros exemplos quando formos abordar funções e procedimentos.

A seguir vamos dar exemplos de como podemos armazenar em arquivos informações que acharmos necessário. Temos dois tipos de arquivos para isso: texto ou binário. Os arquivos textos podem ser lidos fora do programa, serve como geração de relatórios ou outras informações para logísticas de controle do usuário. O outro arquivo binário é para armazenar

grandes volumes de informação de uma só vez e só pode ser lido pelo próprio programa que o gerou.

5 – Uso de arquivos de armazenamentos

Os programas em C utilizam o conceito de fluxo de dados (***streams***) para comunicarem-se com dispositivos do computador, com arquivos em disco, ou com outros programas.

Os fluxos permitem uma padronização das operações de entrada e saída:

- Um conjunto fixo de funções existem para ler e escrever nos fluxos.
- Há um mecanismo para associar fluxos a um dispositivo ou programa.
- Esse modelo de comunicação permite ao programador enviar e ler dados para/de diversos dispositivos usando as mesmas funções de E/S.

O Modelo de um fluxo de dados é dado pelas seguintes características:

- O fluxo corresponde a uma abstração, implementada por uma área de memória e um conjunto de funções, que dão a ideia de existir um fluxo de dados entre um produtor e um consumidor.
- O fluxo funciona como um ***buffer***, uma área de dados na memória principal utilizadas para guardar dados durante a comunicação entre o produtor e consumidor.
- Funciona também como uma fila: o produtor insere dados em uma extremidade do ***buffer*** e o consumidor obtém os dados na outra extremidade.

O modelo de fluxo de dados é dado pela Figura 5.3.

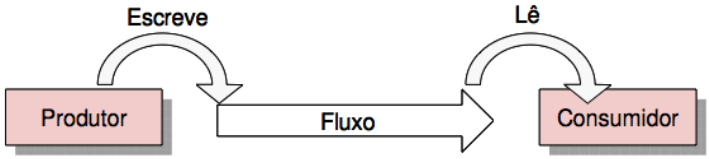


Figura 5.3: Modelo de ***Stream***

O produtor e consumidor podem ser representado pelo programa ou dispositivo de E/S. Podendo ser o próprio programa que Lê e Escreve, ou pode ser lido pelo programa e gerado uma saída para outro dispositivo utilizar.

Os **tipos de Operação** do fluxo podem ser:

- Fluxo de Leitura apenas;
- Fluxo de Escrita apenas;
- Leitura e Escrita;

Quanto ao **tipo de acesso** pode ser sequencial ou aleatório. A seguir veremos exemplos de cada tipo de acesso.

Quanto ao **tipo de dados** pode ser apenas texto ou binário.

O fluxo da linguagem C é criado usando um ponteiro para o tipo FILE (definido em stdio.h). Ex. FILE * arq; FILE é definida como uma ***struct*** cujos campos contêm várias informações sobre o fluxo, sendo:

- Operações
- Sobre Fluxos
- Arquivos
- Binários
- Um ponteiro para o ***buffer*** que armazena os dados do fluxo.
- Tipo de operação permitida no fluxo.
- Códigos de erro.
- Ponteiros para funções.
- Um indicador da posição atual no buffer.
- Etc.

A operação de abertura de um arquivo é dado pelo comando fopen( ); Deve ser a primeira operação a ser executada sobre o fluxo e serve para:

1. Tenta alocar na memória uma **struct FILE** para armazenar dados sobre o fluxo.

2. Alocar o ***buffer*** que armazenará dados do fluxo e armazenar seu endereço na **struct FILE** correspondente ao fluxo;

3. Associar um dispositivo de E/S ao fluxo;

4. Definir as operações de E/S que são permitidas no fluxo;

5. Definir o tipo de acesso no fluxo;

6. Definir o tipo de dados do fluxo (texto ou binário);

A forma geral da operação de abertura do arquivo FILE é dada da seguinte forma:

FILE $*$f = open (nome $-$do $-$arquivo , modo) ;

Onde nome-do-arquivo é uma constante string ou uma variável char* que armazena o caminho no disco onde se encontra o arquivo. E o modo é uma é uma constante string ou uma variável char* que descreve o tipo de operação possível no arquivo, o tipo de acesso e o tipo de dados.

O modo de abertura pelo modo sequencial é dado pelas diretivas:

r – (de *read*) abre o arquivo para leitura sequencial apenas. A função fopen() retorna NULL se o arquivo não existir.

w – (de *write*) abre o arquivo para escrita sequencial apenas. Se o arquivo não existir ele é criado. Se existir seu conteúdo original é apagado.

a – (de *append*) abre o arquivo somente para escrita. Se o arquivo não existir, ele é criado. Se existir, não é apagado. Permite escrever apenas ao final do arquivo.

Já a operação de abertura no modo aleatório é dado por:

r+ - abre o arquivo para leitura e escrita. Se arquivo não existir, retorna NULL. O fluxo é de leitura e escrita e acesso aleatório.

w+ - abre o arquivo para leitura e escrita. Se o arquivo não existe, ele é criado. Se existe, seu conteúdo será apagado. O acesso é aleatório.

a+ - abre o arquivo para leitura e escrita. Se o arquivo existir, mantém o conteúdo existente. A leitura pode ser feita de modo aleatório, mas a escrita só pode ser feita ao final do arquivo.

Os modos de abertura de tipos de dados são dados pelas seguintes diretrizes:

- Acrescentando-se um b na ***string*** de modo indica-se que o fluxo é binário.

- Acrescentando-se um t na ***string*** de modo indica-se que o fluxo é um arquivo de texto. Na ausência de b e t no modo, o arquivo é aberto como arquivo tipo texto.

- Arquivos binários armazenam os dados em formato binário. São utilizados para armazenar em disco os dados armazenados em formato binário na memória do computador: ***int, floats, structs***, etc.

- Arquivos texto armazenam sequências de caracteres.

A operação de leitura de arquivo do tipo texto é dado pela função fgetc() da seguinte forma: - O cabeçalho: **int** fgetc (File* pFile);

- Retorna o próximo caractere de um fluxo tipo texto e avança o ponteiro do arquivo para o próximo caractere.

- Embora retorne valores da tabela ASCII. O tipo retornado é um int , porque retorna EOF (-1) se o fim do arquivo for encontrado (não há um caractere para representar fim de arquivo). Por isso, a função não pode ser do tipo ***char***, pois qualquer um dos 256 valores da tabela ASCII ou -1 podem ser retornados.

- O arquivo deve ter sido aberto em modo texto e com alguma das opções que permite leitura.

A seguir vamos apresentar um exemplo:

```
#include <stdlib.h>
#include <stdio.h>
int main () {
  FILE * pFile;
  int c;
  int n = 0;
//Abrir um arquivo tipo texto apenas para leitura sequencial
  pFile=fopen ("Arquivo.txt","rt");
//Testar se o arquivo existe
  if (pFile==NULL){
    printf("Arquivo nao foi encontrado - Terinando o programa.\n");
    exit(1);
  }
//Leitura caractere por caractere
  c = fgetc (pFile);
  while (c!=EOF) {
    putchar (c);
    c=fgetc(pFile);
  }
  // Fechando o arquivo.
  fclose (pFile);
  return 0;
}
```

As operações de Leitura de Arquivos do tipo texto com entradas formatadas são dadas pela função ***fscanf( );***

O Cabeçalho é dados pelo comando: **int *fscanf*** (**FILE** ∗arq, const **char** ∗format ,...) ;

Tem a mesma funcionalidade da função ***scanf ()*** , porém a entrada é feita a partir do arquivo indicado por ****arq***.

Retorna o número de itens lido se conseguir ler algum dado. Se não conseguir ler nenhum dado por causa de um erro na entrada ou por ter encontrado o fim do arquivo, retorna ***EOF***.

O arquivo deve ter sido aberto em modo texto e com alguma das opções que permite leitura.

As Operações de escrita em arquivos texto é dada pela função: ***fputc***( );

Cabeçalho: **int *fputc*** (int c , FILE ∗ arq) ;

- Imprime o caractere passado pelo parâmetro c no arquivo indicado por arq.

- Retorna o mesmo caractere recebido no parâmetro c, se conseguir realizar a escrita.

- Retorna EOF caso ocorra um erro durante a escrita.

O arquivo deve ter sido aberto em modo texto e com alguma das opções que permita a escrita. Exemplo: Mostrar como copiar um arquivo Arq1 para o arquivo Arq2.

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

int main ()
{
  FILE* pFile1;
  FILE* pFile2;
  char linha[500];
  int n = 0;
  pFile1=fopen ("Arquivo.txt","rt");
  if (pFile1==NULL){
   printf("Arquivo nao foi encontrado - Terminando o programa.\n");
   exit(1);
  }
  pFile2=fopen("Copia.txt", "wt");
        if (pFile2==NULL){
                printf("Arquivo de saida nao foi criado - Terminando o programa.\n");
                exit(1);
  }
        n=fscanf(pFile1,"%[^\n]", linha);
  while (n!=EOF) {
                fgetc(pFile1); // para consumir o \n
       fprintf(pFile2,"%s\n",linha);
                n=fscanf(pFile1,"%[^\n]", linha);
  }
  fclose (pFile1);
  fclose(pFile2);
  return 0;
}
```

A operação de escrita em arquivo Texto é realizada pela função: ***fprintf***( ):

Cabeçalho: **int *fprintf*** (**FILE** ∗arq , **const char** ∗formato , ...) ;

Tem a mesma funcionalidade da função ***printf*** () , porém a saída é feita a partir do arquivo indicado por ***arq**.

Retorna o número de caracteres escritos, caso consiga realizar a escrita com sucesso. Retorna um número negativo em caso contrário.

O arquivo deve ter sido aberto em modo texto e com alguma das opções que permite escrita.

Fluxos especiais do tipo texto são dados pelos fluxos:

- Padrões: - Como dito anteriormente, os fluxos permitem padronizar a entrada e a saída de modo que fiquem independentes do dispositivo usado. Assim, operações semelhantes são feitas tanto para arquivos em disco como para o teclado ou terminal, por exemplo.

- Para todos os programas em C, mesmo os que não abram arquivos explicitamente, são criados alguns fluxos automaticamente. Quatro desses fluxos são os seguintes: ***stdin***, ***stdout***, ***stderr***, ***stdprn***.

Detalhamento do fluxo padrão: - ***stdin*** - fluxo de entrada padrão - geralmente associado ao teclado, é utilizado pelas funções de entrada que não usam explicitamente arquivos: ***scanf*** () , ***getchar*** () .

- ***stdout*** - fluxo de saída padrão - geralmente associado ao terminal. Utilizado por funções que não usam explicitamente arquivos: ***printf*** () , ***putchar***() .

- ***stderr*** - fluxo de mensagens de erro - pode ser utilizado para enviar mensagens de erro para um fluxo diferente do ***stdout***.

- ***stdprn*** - fluxo destinado a impressão ao qual se conecta um impressora.

O USO dos fluxos padrões são dados da seguinte forma: - Os quatro fluxos são automaticamente abertos e automaticamente fechados. São do tipo texto.

- Equivalência:

printf ("Um dois") ; →***fprintf*** (sdtdout , "Um dois") ; escrita

scanf ("%d" , &x) ; –> ***fscanf*** (sdtdin , "%d" , &x) ; leitura

Impressão de mensagem de erro em ***stderr***: = ***printf***(***stderr*** , "Nao ha memoria suficiente") ;

- ***perror*** ("Nao ha memoria suficiente") ; (definida na **stdio.h**).

Saída na impressora padrão: ***fprintf*** (***stdprn*** , "Um dois") ;

O Uso de arquivos binários são dados pelo fluxo de bytes, que são contados pelo tipo criado para armazenamento. Em um arquivo binário, os bytes são transmitidos, em uma operação de escrita, como estão na memória RAM. Assim, se um valor do tipo int for gravado em um arquivo binário, seus bytes são copiados como estão armazenados e não são traduzidos em caracteres como o ocorre quando se imprime no formato "%d" da função ***fprintf***();

De forma análoga, durante uma operação de leitura de um arquivo binário os bytes armazenados no arquivo são copiados como estão no arquivo para a memória RAM do computador.

Dado que é feita uma transmissão de *bytes* quando o arquivo é binário, pode-se transferir grandes blocos de *bytes* de uma vez entre a memória e o dispositivo de E/S.

No exemplo criado de controle de empregado dado no item anterior, se fossemos criar um arquivo binário do tipo Empresa teríamos um fluxo de 5000xnumero de bytes do tipo

empregado (que seria 164 bytes) + a quantidade de bytes criada no tipo Empresa dando um total de 820.089 bytes. Esse será o volume de bytes que esse tipo de arquivo binário ira transitar no ***buffer*** do computador.

A operação de escrita em um arquivo binário é dado por: ***fwrite***( );

Cabeçalho: size_t ***fwrite*** (const void * ptr , size_t tam, size_t cont , FILE * arq) ;

Transfere *tam* **bytes** a partir do endereço ptr da RAM *cont* vezes para o arquivo cujo fluxo está indicado por arq.

O indicador de posição do arquivo é deslocado para frente *cont* * *tam* **bytes**.

A função retorna o número de blocos de tamanho tam que foram gravados. Se o valor de retorno for diferente do parâmetro cont é porque houve um erro de escrita e a operação falhou em transferir a quantidade de blocos especificada em *cont*.

Já a operação de leitura em um arquivo binário é dado pela função: ***fread***( );

Cabeçalho: size_t ***fread*** (const void * ptr , size_t tam, size_t cont , FILE * arq) ;

Transfere *cont* blocos de *tam* **bytes** do arquivo arq e os armazena a partir do endereço **ptr** da *RAM* .

O indicador de posição do arquivo é deslocado para frente *cont* * *tam* bytes.

A função retorna o número de blocos de tamanho *tam* que foram lidos. Se o valor de retorno for diferente do parâmetro *cont* é porque o arquivo terminou, ou houve um erro de leitura e a operação falhou em transferir a quantidade de blocos especificada em *cont*.

Verificando o fim de arquivo em arquivo binário, usa-se a função: ***feof***();

Arquivos binários não podem usar EOF para checar fim de arquivo, pois os **bytes** podem representar qualquer coisa na memória. A escrita de uma variável do tipo **char** contendo o valor -1 (EOF) seria interpretada como fim de arquivo.

Na linguagem C há uma função para detectar fim de arquivo. É a função ***feof*** () .

Cabeçalho: **int *feof*** (FILE * arq) ;

A função retorna um valor diferente de zero se o fim do arquivo foi atingido e zero, em caso contrário.

Apesar de ser necessária para detecção de fim em arquivos binários, ***feof***() pode ser utilizada também com arquivos texto.

Já o acesso Aleatório em arquivos é dado com a mudança do indicador de posição do arquivo. O indicador de posição de arquivo é um valor que até agora foi utilizado indiretamente: (a) após a abertura do arquivo ele é indica o início do arquivo (b) Após cada operação de leitura/escrita ele é avançado de acordo com o número de ***bytes*** lidos/escritos.

É possível mudar diretamente o indicador de posição de arquivo através da função ***fseek*** ().

Após a mudança, é possível ler ou gravar em uma outra posição do arquivo, fazendo com que o processamento da leitura/escrita deixe de ser sequencial. Denominamos esse tipo de acesso a qualquer posição de acesso aleatório.

A escrita/leitura nesse caso só é possível se o arquivo foi aberto no modo acesso aleatório.

Outra função de acesso aleatório a arquivos é a função: ***fseek***();

Cabeçalho: **int** ***fseek*** (**FILE** ∗ arq , ***long*** **int** desloc , **int** origem) ;

O argumento desloc corresponde a número de bytes que o indicador de posição deve ser deslocado em relação a origem.

A origem pode ser uma das seguintes constantes definidas em **stdio.h** :

SEEK_SET	Início do arquivo
SEEK_CUR	Posição atual do indicador de posição
SEEK_END	Final do arquivo

A função ***fseek***() retorna zero se executar corretamente e um outro valor em caso contrário. Existem outras funções para arquivo, como é o caso da função: ***ftell***( );

Cabeçalho: **long int** ***ftell***(**FILE** ∗ arq) ;

Retorna o valor do indicador de posição atual do arquivo cujo ponteiro é indicado por ***arq***.

Para arquivos binários o valor do indicador corresponde ao número de bytes entre o início do arquivo e a posição atual do indicador.

Para arquivos texto, o valor pode não ter significado, mas ainda pode ser usado para voltar à posição atual, por guardar o valor atual em uma variável **long int** e usar futuramente essa variável como parâmetro para ***fseek*** (). Em caso de falha, a função retorna -1.

Outra função é que retorna erro é a função: ***ferror***();

Cabeçalho: **int** ***ferror***(**FILE** ∗ arq) ;

Definição Verifica se um erro foi indicado na ***struct FILE*** para o fluxo *arq*.

Um valor diferente de zero retornado indica que houve algum erro com a última operação executada sobre o arquivo.

O valor de indicação de erro na struct FIlLE é limpado sempre que uma das operações: ***rewind***() e ***freopen***() ocorre sobre o fluxo

A função de fluxo: ***fflush***():

Cabeçalho: **int** ***fflush*** (**FILE** ∗ stream) ;

Se fluxo foi aberto para escrita ou se foi aberto para leitura/escrita e a última operação foi de escrita, a função descarrega o buffer do fluxo no dispositivo de saída a ele associado. Se arq for **NULL**, todos os *buffers* de todos os fluxos na situação anterior são descarregados.

A função não fecha os fluxos.

Retorna zero se obteve sucesso e **EOF** em caso contrário.

Já a função: ***rewind***(); //reinicia o arquivo

Cabeçalho: **void** ***rewind*** (**FILE** ∗ arq) ;

Faz com que o indicador de posição do arquivo aponte para o início do fluxo *arq*.

Função ***freeopen***();

Cabeçalho:**FILE** ∗ ***freopen*** (*const* **char** ∗ nomearq, *const* **char** ∗ modo, FILE ∗arq);

Tenta fechar o fluxo atual indicado por *arq*. Reutiliza o fluxo, abrindo-o com associação ao arquivo especificado em *nomearq* e no modo especificado na **string** modo.

Essa função é especificamente útil para redirecionar a saída padrão dentro do programa para um outro dispositivo ou arquivo em disco.

Se executar com sucesso, a função retorna o ponteiro para arq, caso contrário, retorna **NULL**.

O fechamento de fluxo é dado pela função: ***fclose***();

O fechamento de um fluxo consiste em:

- Realizar descarga do conteúdo do ***buffer*** no dispositivo de saída.
- Fazer a desvinculação do fluxo com o dispositivo através de chamadas ao sistema operacional;
- Liberar a ***struct*** **FILE** alocada para representar o fluxo.

O Cabeçalho da função: **int** ***fclose***(**FILE** ∗ stream);

O Retorno da função de fechamento será:

- Zero, se o fluxo foi fechado com sucesso;
- EOF em caso de falha.

A seguir vamos usar um exemplo para ler e escrever um arquivo binário do tipo Empresa dada no item 5.4.

```
#include <stdio.h>

#include <stdlib.h>
#include <string.h>
#define Nfunc 5000

typedef struct { int codigo;
                                char nome_emp[40];
                                char data_nasc[11];
                                char email[60];
                                char cargo[30];
                                char data_admissao[11];
                                float salario;
                              }Empregado;
typedef struct {Empregado func[Nfunc];
                                      int quantidade_emp;
                                      char Nome_empresa[50];
                                      char cnpj[20];
                                      char tel_empresa[15];
                                      }Empresa;

void Gravar_empresa (Empresa E) {   // grava todo o tipo Empresa
   FILE *fp;

   fp = fopen ("Empresa.dat", "wb"); //arquivo para escrita binária, apaga e grava tudo denovo
   if (fp == NULL) {
      printf ("Erro ao abrir o arquivo.\n");  //aqui pode por a função que inicializa Empresa.
      return 0;
   }
   else {
          printf ("Arquivo Binario Empresa.dat foi criado com sucesso.\n");
         fwrite(&E, sizeof(Empresa), 1, fp);
         }
     fclose (fp);
  }/*Função responsável por exibir o conteúdo do arquivo Empresa.dat*/

void Carregar_empresa(Empresa *E){
     FILE *fp;

     fp = fopen ("Empresa.dat", "rb");
      if (fp == NULL) {
                  return;
}  else {
         fread(&(*E),sizeof(Empresa),1,fp);
        //faz a leitura no arquivo Empresa.dat e passa para a variável tipo empresa
         fclose(fp); //fecha o arquivo e copia no buffer da memória RAM
        }
}
```

// como esses procedimentos são usados no programa main()?

```
int main( ) {
        Empresa emp;
        Empregado func;
        .....
        Carregar_empresa(&emp);
        .....

        // la no fechamento do programa

        Gravar_empresa(emp);
  return 0;
 }
```

7 – Exercícios:

Iremos colocar 3 exercícios de cada modalidade vista neste capítulo.

7.1 - Exercícios de Vetor:

E7.1.1 - Maior Elemento de um vetor

Faça um programa que receba vários vetores e informe para cada um deles qual o maior elemento e o indice (da primeira ocorrência) em que encontra-se tal elemento.

Entrada

O programa possui vários casos de testes. A primeira de cada caso contem um inteiro N, $1 < N \leq 10000$, representando o tamanho do vetor. A segunda linha conterá N inteiros entre 0 e 1000, representando os N elementos do vetor. A entrada termina quando N=0.

Saída

O programa gera N linhas de saída, com dois inteiros separados por um espaço em branco. O primeiro inteiro é o índice da primeira ocorrência do maior elemento do vetor e o segundo inteiro é o maior valor do vetor. Após a impressão de cada saída, inclusive a última, quebre uma linha.

Casos de Testes:

Entrada
10 6 54 7 3 73 6 67 23 6 9 5 9 8 7 6 5 8 0 1 2 3 4 5 6 7

0
Saída
4 73 0 9 7 7

E7.1.2 – Maior frequência de uma Nota:

Dada uma sequência de N notas entre 0 e 10, escreva um programa que exiba o valor da última nota informada e quantas vezes ela apareceu no conjunto. O programa deve exibir ainda a maior nota informada e a posição (índice do vetor) da sua primeira ocorrência.

Entrada

Na primeira linha há um inteiro N, sendo 1 ≤ N ≤ 10000 representando a quantidade de notas da sequência. Não é necessário validar o valor de N na entrada. Nas N linhas seguintes haverá um número inteiro entre 0 e 10, inclusive, em cada linha.

Saída

O programa gera 2 linhas de saída. A primeira linha exibirá a frequência da última nota informada e a segunda linha exibirá a maior nota e a posição (índice do vetor) da sua primeira ocorrência, seguindo o formato da saída apresentado a seguir. Não se esqueça de quebrar uma linha após a última impressão.

Caso de Teste:

Entrada
11 5 6 3 4 3 8 7 4 8 6 4
Saída
Nota 4, 3 vezes Nota 8, indice 5

E71.1.3: Contagem de Elementos únicos em um vetor:

Elabore um programa que conte o número total de elementos únicos em um vetor de números inteiros.

Entrada

A entrada contém duas linhas. A primeira, contém um valor inteiro *n < 5000* que corresponde ao número de elementos que aparecem na segunda linha. A segunda linha contém n valores inteiros, separados entre si por um espaço.

Saída

A saída é formada por uma linha contendo um valor inteiro que corresponde ao número de elementos que aparecem apenas uma vez no vetor. Após o valor, o programa deve imprimir o caractere de quebra de linha.

Caso de teste:

Entrada
7 3 6 2 9 2 7 9
Saída
3

7.2 - Exercícios de Matriz:

7.2.1 – Ampulheta:

O objetivo desse exercício é identificar um conjunto de elementos cuja somatória seja o maior valor entre todos os conjuntos definidos por um padrão na forma de ampulheta, em uma matriz de inteiros, quadrada, e de ordem 6. Cada elemento da matriz está no intervalo [-9, 9]. Uma “ampulheta” é formada pelos valores que estão posicionados de acordo com a seguinte configuração:

a	*B*	*c*
	D	
e	*F*	*g*

uma ampulheta é a soma de todos os valores presentes nas respectivas posições. Seu programa deve retornar o maior valor entre todos os valores de ampulheta possíveis na matriz.

Entrada

Uma matriz quadrada de ordem 6.

Saída

Um único inteiro que corresponde à maior soma de todos os valores de ampulheta. Após imprimir o valor quebre a linha.

Caso de Teste:

Entrada:	Saída:
1 1 1 0 0 0 0 1 0 0 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0	7

Entrada:	Saída:
1 1 1 0 0 0 0 1 0 0 0 0 1 1 1 0 0 0 0 0 2 4 4 0 0 0 0 2 0 0 0 0 1 2 4 0	19

E7.2.2: Cidade Segura:

Com o aumento da violência na cidade o prefeito decidiu instalar câmeras de vigilância em todas as esquinas. A cada mês, um mapa atualizado com as câmeras em funcionamento é disponibilizado no site da prefeitura. A secretaria de segurança considera que uma esquina é segura se existem câmeras em funcionamento, pelo menos, duas de suas quatro esquinas. Nesta cidade todas as quadras são quadrados de mesmo tamanho. Sua tarefa é, dado o mapa das câmeras em funcionamento nas esquinas, indicar o status de todas as quadras da cidade.

Entrada

A primeira linha da entrada tem um inteiro positivo N (1 ≤ N ≤ 100). Nas próximas N+1 linhas, existem N+1 números, que indicam, para cada esquina, a presença de uma câmera em funcionamento ou de uma câmera defeituosa. O número 1 indica que existe uma câmera funcionando na esquina, enquanto o número zero indica que não há câmera funcionando.

Saída

A saída é dada em N linhas. Cada linha tem N caracteres, indicando se a quadra correspondente é segura ou insegura. Se uma quadra é segura, mostre o caractere S, caso contrário mostre o caractere U. Após a última linha não se esqueça de saltar uma linha.

Caso de Teste:

Entrada:	Saída:
1 1 0 0 0	U

Entrada:	Saída:
2 1 0 0 1 1 0 0 0 1	SU SS

Entrada:	Saída:
3 1 1 0 1 1 0 1 0 1 0 0 1 0 1 1 0	SSS SUS SSS

E7.2.3: Cadê Wally?

Wally costuma morar em um ambiente representado por uma matriz bidimensional de números inteiros de ordem n x m (n linhas e m colunas). Ele é único no ambiente e é representado na matriz por quatro números, distribuídos da seguinte forma:

	4	
0	1111	0
	8	

O número 1111 representa a camisa listrada em vermelho e branco, e o número 4 representa seu gorro das mesmas cores. Os números zero e oito representam suas extremidades. A matriz representa um ambiente bidimensional circular:

• Para o índice i=0, a celula à esquerda está no índice m-1.

• Para o índice i=m-1, a celula à direita está no índice 0.

• Para o índice j=0, a celula superior está no índice n-1.

• Para o índice j=n-1, a celula inferior está no índice 0.

Crie um programa que permita imprimir os índices i,j (da matriz) onde está a camisa do Wally. Caso o Wally não estiver na matriz mostre a seguinte mensagem: “WALLY NAO ESTA

NA MATRIZ”. Na seguinte matriz de 7 linhas e 6 colunas, a camisa do Wally está nos índices i=3, j=0.

0	1111	0	1	0	0
0	0	0	0	1111	0
4	0	1	3	45	53
1111	0	89	211	87	0
8	4	56	4	55	98
0	2222	0	11	0	5
0	8	23	8	66	3

Entrada

Dois valores inteiros, n e m, seguidos dos elementos inteiros da matriz de ordem n x m, com n>=3 e m>=3.

Saída

Se o Wally estiver na matriz: o índice i,j da localização do Wally. Caso contrário “WALLY NAO ESTA NA MATRIZ” (sem acentos)

Caso de Teste:

Entrada:	**Saída:**
3 7 4 7 7 7 7 7 7 1111 0 7 7 7 7 0 8 7 7 7 7 7 7	1 0

Entrada:	**Saída:**
5 4 5 5 8 5 5 5 5 5 5 5 5 5 5 5 4 5 5 0 1111 0	4 2

Entrada:	**Saída:**
3 3 1 4 1 0 1111 0 1 8 1	1 1

7.3 - Exercícios de Tipos de Dados heterogêneos:

E7.3.1: Tradutor do Papai Noel: tirada de:

https://www.urionlinejudge.com.br/judge/pt/problems/view/1763

Nicolau já está bastante cansado e sua memória não é mais a mesma. Você, como navegador, deverá auxiliar o Papai Noel a gritar a frase “Feliz Natal” no idioma correto de cada país de que o trenó está sobrevoando.

Como você é um elfo muito esperto, você já criou um pequeno app no seu celular (sim, elfos tem celular) que irá lhe informar a frase no idioma correto dado o nome do país. Como o trenó é moderno (foi atualizado no ano 2000) ele exibe no painel de navegação o nome do país atual.

Os dados inseridos no seu app foram:

País	Frase
brasil	Feliz Natal!
alemanha	Frohliche Weihnachten!
austria	Frohe Weihnacht!
coreia	Chuk Sung Tan!
espanha	Feliz Navidad!
grecia	Kala Christougena!
estados-unidos	Merry Christmas!
inglaterra	Merry Christmas!
australia	Merry Christmas!
portugal	Feliz Natal!
suecia	God Jul!
turquia	Mutlu Noeller
argentina	Feliz Navidad!
chile	Feliz Navidad!
mexico	Feliz Navidad!
antardida	Merry Christmas!
canada	Merry Christmas!
irlanda	Nollaig Shona Dhuit!

belgica	Zalig Kerstfeest!
italia	Buon Natale!
libia	Buon Natale!
siria	Milad Mubarak!
marrocos	Milad Mubarak!
japao	Merii Kurisumasu!

Para não correr o risco de informar o nome errado você decidiu testar o aplicativo mais algumas vezes.

Entrada

Você irá testar o seu aplicativo com diversos nomes de países, simulando os dados informados pelo painel de navegação do trenó. A entrada termina por fim de arquivo.

Saída

O seu aplicativo deverá mostrar na tela a frase no idioma correto. Caso ela não esteja cadastrada, você deverá exibir a mensagem -– NOT FOUND —"para que depois dos testes você possa completar o banco de dados.

Caso de Teste:

Entrada
uri-online-judge alemanha brasil austria
Saída
-- NOT FOUND -- Frohliche Weihnachten! Feliz Natal! Frohe Weihnacht!

E7.3.2: Ordenação de Datas:

Uma determinada professora quer ordenar seus alunos em ordem crescente de idade. Escreva um programa em C que leia os dados dos alunos, entre eles a data de nascimento, ordene os alunos em ordem crescente de idade. Para isso seu programa deve ter uma função ComparaDataNasc() que recebe dois parâmetros. O primeiro corresponde a uma struct (ou um ponteiro para uma struct) que contém o dia, o mês e o ano de nascimento de uma aluno. O

segundo parâmetro tem o memo tipo de dado do primeiro e contém a data de nascimento do segundo aluno. Essa função retorna 1 se o primeiro aluno é mais novo ou tem a mesma idade do segundo aluno e retorna zero em caso contrário. Essa função deve ser chamada pela função que ordena os alunos em ordem crescente de idade.

Entrada

A entrada contém apenas um caso de teste. A primeira linha da entrada contém um número inteiro n, (1 ≤ n ≤ 30) que corresponde ao número de alunos da turma. Em seguida há n linhas, contendo cada uma:

- a matrícula de um aluno (int);
- o dia de nascimento de um aluno (int);
- o mês de nascimento de um aluno (int);
- o ano de nascimento de um aluno (int);
- o nome de um aluno (no máximo 200 caracteres);

Saída

A saída é formada por n linhas, cada uma correspondendo ao um aluno, ordenadas em ordem crescente de idade dos alunos. Cada linha deve ter o seguinte formato: Matric.: m Nome: n Data Nasc.: dd/mm/aa, onde m é a matricula de um aluno, n, o seu nome, e dd, mm e aa, são respectivamente o dia, o mês e o ano do seu nascimento.

Caso de Teste:

Entrada	Saída
5 12345 12 07 1978 Felizbina Freitas 23489 11 03 2009 Joao Feliz da Tristeza 98762 05 12 1976 Maria Batista de Souza 34561 11 07 1978 Roberto de Assis 34599 07 05 1976 Luiz Alberto Ferreira	Matric.: 23489 Nome: Joao Feliz da Tristeza Data Nasc: 11/3/2009 Matric.: 12345 Nome: Felizbina Freitas Data Nasc: 12/7/1978 Matric.: 34561 Nome: Roberto de Assis Data Nasc: 11/7/1978 Matric.: 98762 Nome: Maria Batista de Souza Data Nasc: 5/12/1976 Matric.: 34599 Nome: Luiz Alberto Ferreira Data Nasc: 7/5/1976

Entrada	Saída
10 12345 12 07 1978 Felizbina Freitas 23489 11 03 2009 Joao Feliz da Tristeza 98762 05 12 1976 Maria Batista de Souza 34561 11 07 1978 Roberto de Assis 34599 07 05 1976 Luiz Alberto Ferreira 43125 1 07 1978 Flavio Antonio 77889 11 07 2009 Luiz Antonio Silva 55443 12 07 1977 Laura Oliveira 98765 04 05 1997 Fernando Antonio 90891 08 06 2000 Luiza Maria	Matric.: 77889 Nome: Luiz Antonio Silva Data Nasc: 11/7/2009 Matric.: 23489 Nome: Joao Feliz da Tristeza Data Nasc: 11/3/2009 Matric.: 90891 Nome: Luiza Maria Data Nasc: 8/6/2000 Matric.: 98765 Nome: Fernando Antonio Data Nasc: 4/5/1997 Matric.: 12345 Nome: Felizbina Freitas Data Nasc: 12/7/1978 Matric.: 34561 Nome: Roberto de Assis Data Nasc: 11/7/1978 Matric.: 43125 Nome: Flavio Antonio Data Nasc: 1/7/1978 Matric.: 55443 Nome: Laura Oliveira Data Nasc: 12/7/1977 Matric.: 98762 Nome: Maria Batista de Souza Data Nasc: 5/12/1976 Matric.: 34599 Nome: Luiz Alberto Ferreira Data Nasc: 7/5/1976

E7.3.3: Mercado: tirado de:

https://www.urionlinejudge.com.br/judge/pt/problems/view/1281

Dona Parcinova costuma ir regularmente à feira para comprar frutas e legumes. Ela pediu então à sua filha, Mangojata, que a ajudasse com as contas e que fizesse um programa que calculasse o valor que precisa levar para poder comprar tudo que está em sua lista de compras, considerando a quantidade de cada tipo de fruta ou legume e os preços destes itens.

Entrada

A primeira linha de entrada contém um inteiro N que indica a quantidade de idas à feira de dona Parcinova (que nada mais é do que o número de casos de teste que vem a seguir). Cada caso de teste inicia com um inteiro M que indica a quantidade de produtos que estão disponíveis para venda na feira. Seguem os M produtos com seus preços respectivos por unidade ou Kg. A próxima linha de entrada contém um inteiro P($1 \leq P \leq M$) que indica a quantidade de diferentes produtos que dona Parcinova deseja comprar. Seguem P linhas contendo cada uma delas um texto (com até 50 caracteres) e um valor inteiro, que indicam respectivamente o nome de cada produto e a quantidade deste produto.

Saída

Para cada caso de teste, imprima o valor que será gasto por dona Parcinova no seguinte formato: R$ seguido de um espaço e seguido do valor, com 2 casas decimais, conforme o exemplo abaixo.

Caso de Teste:

Entrada
2 4 mamao 2.19 cebola 3.10 tomate 2.80 uva 2.73 3 mamao 2 tomate 1 uva 3 5 morango 6.70 repolho 1.12 brocolis 1.71 tomate 2.80 cebola 2.81 4 brocolis 2 tomate 1 cebola 1

morango 1
Saída
R$ 15.37 R$ 15.73

========== para o professor acompanhar ==========

Resolução dos problemas:

R7.1.1 – Maior Elemento:

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

int maior(int *v, int n)
{
    int i, iMaior;
    for(iMaior=0, i=1; i<n; i++)
    {
        if(v[i] > v[iMaior])
        {
            iMaior = i;
        }
    }
    return iMaior;
}

int main()
{
    int *v, n, iMaior, i;
    scanf("%d", &n);
    while(n!=0)
    {
        v = (int*) malloc(n*sizeof(int));
        for(i=0; i<n; i++)
        {
            scanf("%d", &v[i]);
        }
        iMaior = maior(v, n);
        printf("%d %d\n", iMaior, v[iMaior]);
        free(v);
        scanf("%d", &n);
    }
    return 0;
}
```

R7.1.2: Maior frequência de uma nota em um vetor:

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

```
int main()
{
	int n, iMaior, maior, i, v=0;
	int vetFreq[11] = {0};
	int iApareceu[11] = {0};
	int apareceu[11] = {0};

	scanf("%d", &n);

	for(i=0; i<n; i++)
	{
		scanf("%d", &v);
		if(!apareceu[v])
		{
			iApareceu[v] = i;
			apareceu[v] = 1;
		}
		vetFreq[v]++;
	}

	for(i=10; i>=0 && !apareceu[i]; i--);

	printf("Nota %d, %d vezes\nNota %d, indice %d\n", v, vetFreq[v], i, iApareceu[i]);
	return 0;
}
```

R.7.1.3: Contagem de Elementos únicos em um vetor:

```
#include <stdio.h>
#define MAX 1000

int main() {
    int n, i, j;
    int flag, counter = 0;
    int vec[MAX];

    scanf("%d", &n);

    // lendo os elementos
    for (i = 0; i < n; i++) {
        scanf("%d", &vec[i]);
    }

    // iterando cada elemento do vetor, pelo mesmo vetor, de um elemento a frente até o ultimo
    for (i = 0; i < n; i++) {
        flag = 1;

        for (j = 0; j < n; j++) {
         // agora verificaremos se existe algum elemento igual a vec[i], se sim, flag será igual a 0
            if (i != j && vec[i] == vec[j]) {
```

```
                flag = 0;
                break;
            }
        }
        if (flag) counter++;
    }
    printf("%d\n", counter);
    return 0;
}
```

R7.2.1: Ampulheta

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

int maiorAmpulheta (int M[][6]) {
        int i, j, r, n=6;
        int max = -9*7; // a ampulheta eh formada por 7 elementos. O menor valor para cada elemento eh -9

        for(i=0; i<n-2; i=i+1) {
                for(j=0; j<n-2; j=j+1) {
                        r = (M[i][j] + M[i][j+1] + M[i][j+2]) + M[i+1][j+1] + (M[i+2][j] + M[i+2][j+1] + M[i+2][j+2]);
                        if(max<r) {
                                max=r;
                        }
                }
        }

        return max;
}

int main() {
        // Leitura dos elementos da matriz
        int i, j, n=6, max;
        int M[6][6];

        for(i=0; i<n; i=i+1) {
                for(j=0; j<n; j=j+1) {
                        scanf("%d", &M[i][j]);
                }
        }

        // Processamento principal
        max = maiorAmpulheta(M);
        printf("%d\n", max);
        return 0;
}
```

R7.2.2: Cidade Segura:

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int n, i, j;
    int ** map;

    scanf("%d", &n);
    map = (int **) malloc((n+1) * sizeof(int *));
    for(i = 0; i < n+1; i++) {
        map[i] = (int *) malloc((n+1) * sizeof(int));
    }

    for(i=0; i<n+1; i++)
    {
        for(j=0; j<n+1; j++)
        {
            scanf("%d", &map[i][j]);
        }
    }

    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
        {
            if(map[i][j]+map[i][j+1]+map[i+1][j]+map[i+1][j+1]>=2)
            {
                printf("S");
            }
            else
            {
                printf("U");
            }
        }
        printf("\n");
    }

    return 0;
}
```

R7.2.3: Cadê Wally?

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

int limitar (int indice, int limite) {
        if (indice<=-1) {
                return limite-1;
        }
        if (indice>=limite) {
```

```
                return 0;
        }
        return indice;
}

void acharWally (int ** M, int n, int m) {
        int i, j, wi=-1, wj=-1;

        for (i=0; i<n && wi==-1; i=i+1) {
                for (j=0; j<m && wj==-1; j=j+1) {
                        if (M[i][j]==1111 && M[i][limitar(j-1, m)]==0 && M[i][limitar(j+1, m)]==0 &&
M[limitar(i-1, n)][j]==4 && M[limitar(i+1, n)][j]==8 ) {
                                wi = i;
                                wj = j;
                        }
                }
        }

        if (wi!=-1){
                printf("%d %d",wi,wj);
        }
        else {
                printf("WALLY NAO ESTA NA MATRIZ\n");
        }
}


int main()
{
        // Leitura dos elementos da matriz
        int i, j, ** M, n, m;
        scanf("%d %d", &n, &m);
        M = (int **) malloc(n * sizeof(int *));
        for(i = 0; i < n; i++) {
                M[i] = (int *) malloc(m * sizeof(int));
        }

        for(i=0; i<n; i=i+1) {
                for(j=0; j<m; j=j+1) {
                        scanf("%d", &M[i][j]);
                }
        }

        // Processamento principal
        acharWally(M, n, m);

        return 0;
}
```

R7.3.1: Tradutor de Papai Noel:

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

typedef struct{
    char nome[20];
    char traducao[50];
}Pais;

int search(Pais *v,int tam,char *key){              // retorna -1 se n achar a chave/nome do pais, e o índice do pais se achar
        int i;
        for(i=0;i<24;i++){
                if(strcmp(v[i].nome,key)==0)
                        break;
        }

        if(i==24)
                return -1;

        return i;
}


int main(){
        Pais paises[24]={"brasil","Feliz Natal!",
        "alemanha","Frohliche Weihnachten!",
        "austria","Frohe Weihnacht!",
        "coreia","Chuk Sung Tan!",
        "espanha","Feliz Navidad!",
        "grecia","Kala Christougena!",
        "estados-unidos","Merry Christmas!",
        "inglaterra","Merry Christmas!",
        "australia","Merry Christmas!",
        "portugal","Feliz Natal!",
        "suecia","God Jul!",
        "turquia","Mutlu Noeller",
        "argentina","Feliz Navidad!",
        "chile","Feliz Navidad!",
        "mexico","Feliz Navidad!",
        "antardida","Merry Christmas!",
        "canada","Merry Christmas!",
        "irlanda","Nollaig Shona Dhuit!",
        "belgica","Zalig Kerstfeest!",
        "italia","Buon Natale!",
        "libia","Buon Natale!",
        "siria","Milad Mubarak!",
        "marrocos","Milad Mubarak!",
        "japao","Merii Kurisumasu!"};
```

```
	int i;

	char chave[20],saida[50];
	short verif;


	while (scanf("%s",chave)!=EOF){
		verif=search(paises,24,chave);
		if(verif<0)
			printf("-- NOT FOUND --\n");
		else
			printf("%s\n",paises[verif].traducao );
	}

}
```

R7.3.2: Ordenação de Datas:

```
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>

#define NUM_ALUNOS 30
typedef struct data{
	int dia;
	int mes;
	int ano;
} TData;

typedef struct aluno{
	int matricula;
	char* nome;
	TData nascimento;
} TAluno;

int comparaSeMaisNovo(TAluno* a1, TAluno* a2);
void ordena(TAluno vet[], int n);
int copiaNome(char* buf, TAluno* aluno);
void imprime(TAluno vet[], int N);
int main() {
char buffer[1000];
	int N, i=0,j;

	TAluno vet[NUM_ALUNOS];

	scanf("%d", &N);
	for (i=0;i<N;i++){
		scanf("%d", &vet[i].matricula);
		scanf("%d", &vet[i].nascimento.dia);
```

```
                scanf("%d", &vet[i].nascimento.mes);
                scanf("%d", &vet[i].nascimento.ano);
                scanf("%[^\n]",buffer);getchar();
                //Chama funcao para alocar espaco suficiente para armazenar o nome e copia-
lo do buffer para o campo nome da struct de um aluno
                if ( !copiaNome(buffer,&vet[i])){
                        printf("O programa nao pode ser executado por falta de memoria\n");
                        exit(1);
                }
        }
        ordena(vet, N);
        imprime(vet, N);
        return(0);
}

int comparaSeMaisNovo(TAluno* a1, TAluno* a2){
        if (a1->nascimento.ano>a2->nascimento.ano) {//se ano do a1> ano do a2, a1 mais novo
que a2
   return(1);
        }
        if(a2->nascimento.ano > a1->nascimento.ano){//a1 nao e mais novo aque a2
                return(0);
        }
        //entao a1==a2
        if (a1->nascimento.mes>a2->nascimento.mes) {
                return (1); //apesar de nascerem no mesmo ano a1 nasceu alguns meses
                                                        //depois de a2
        }
        if (a2->nascimento.mes>a1->nascimento.mes) {
                return (0); //apesar de nascerem no mesmo ano a2 nasceu alguns meses
                //depois de a1, entao a1 nao e o mais novo
        }
        //Nesse ponto do codigo, a1 e a2 nasceram no mesmo ano e no mesmo mes
        // o dia de nascimento vai desempatar
        if(a1->nascimento.dia>= a2->nascimento.dia){ //a1 é mais novo ou tem a mesma idade
de a2
                return(1);
        }
        //a2 e definitivamente mais novo
        return(0);
}
void ordena(TAluno vet[], int n){
        int i,j, indMenor;
        TAluno aux;
        for (i=0; i<n; i++) {
        indMenor=i;
                for (j=i+1; j<n; j++) {
                        if(comparaSeMaisNovo(&vet[j],&vet[indMenor])){
                                indMenor=j;
```

```
                        }
                }
                aux=vet[i];
                vet[i]=vet[indMenor];
                vet[indMenor]=aux;
        }
}
void imprime(TAluno vet[], int N){
        int i;
        for (i=0; i<N;i++) {
   printf("Matric.: %d Nome: %s Data Nasc: %d/%d/%d\n", vet[i].matricula, vet[i].nome,
vet[i].nascimento.dia, vet[i].nascimento.mes, vet[i].nascimento.ano);
        }
}
int copiaNome(char* buf, TAluno* aluno){
        aluno->nome=(char*)malloc(sizeof(char)*(strlen(buf)+1));
        if (!aluno->nome) {
   return(0);
        }
        strcpy(aluno->nome,buf);
        return(1);
}

        R7.3.3: Mercado:
#include<stdio.h>
#include<string.h>
#define MAX 2000
struct prodFeira{
        char prod[51];
        float preco;
};

struct prodCompra{
        char prod[51];
        int quant;
};

float pesquisaProd(struct prodFeira f[], int tam, char * s){
        int i;
        for (i=0; i<tam; i++) {
   if(strcmp(s,f[i].prod)==0){
                        return (f[i].preco);
                }
        }
        return(0.0);
}

int main(){
        struct prodFeira feira[MAX];
```

```
        struct prodCompra compra;
        int i,casos, quantOferta, quantCompra;
        float valorTot;
        scanf("%d", &casos); getchar();
        //printf("Casos: %d\n", casos);
        while (casos--) {

    scanf("%d", &quantOferta);getchar();
                //Le produtos a venda
                for (i=0; i<quantOferta; i++) {
                        scanf("%s", feira[i].prod);
                        scanf("%f", &feira[i].preco); getchar();
                //printf("%s %f\n", feira[i].prod, feira[i].preco);
                }
                scanf("%d", &quantCompra); getchar();
                valorTot=0.0;
                //Le produtos a serem comprados
                //printf("quantOferta: %d\n", quantOferta);
                //printf("quantCompra: %d\n", quantCompra);
                for (i=0; i<quantCompra; i++) {
                        scanf("%s", compra.prod);
                        scanf("%d", &compra.quant); getchar();
                        //printf("Compras %s %d\n", compra.prod, compra.quant );
                        valorTot+= pesquisaProd(feira, quantOferta, compra.prod)*compra.quant;
                }
                printf("R$ %.2f\n", valorTot);
        }
}
```

========== para o professor acompanhar ==========

CAPÍTULO VI:

FUNÇÕES

1 – Introdução

Neste Capítulo estaremos introduzindo o uso de Funções e Procedimentos de forma a ensinar a modularização de um programa. Bem como trabalha com as passagens de parâmetros e suas práticas reais. Toda função ou procedimento são distintos pelo tipo de returno, no caso de procedimento não retorna nada (inicia com void nome_procedimento(parâmetros). Já a função retorna um tipo de dados pré existente, podendo ser um tipo primitivo ou um tipo criado pelo usuário.

Também iremos mostrar a funcionalidade de um programa, com suas principais funções e procedimentos que servem para atender a demanda do usuário. Uma funcionalidade é uma operação que será necessário no programa para suprir os requisitos estipulados pelo usuário como: cadastrar, consultar, exibir, etc.

Leitura para entender este capítulo: Thomas Cornem, e outros – Algoritmos – Teoria e Prática, 3ª Edição, Editora LTC, 2012.

2 – Funções

CAPÍTULO VII: DESENVOLVIMENTO DE ALGORITMOS POR REFINAMENTOS SUCESSIVOS

CAPÍTULO VIII: ASPECTOS DE IMPLEMENTAÇÃO DE ALGORITMOS