Universidade Federal do Espírito Santo
Centro Universitário Norte do Espírito Santo
Departamento de Ciências Exatas
Curso de Engenharia da Computação

Processamento de Dados I

Professor: M.Sc. Flávio Duarte Couto Oliveira
flaviodco@yahoo.com.br

Sumario

1. Introdução a Computação

Apesar dos computadores eletrônicos terem efetivamente aparecido somente na década de 40, os fundamentos em que se baseiam remontam a centenas ou até mesmo milhares de anos.

Se levarmos em conta que o termo COMPUTAR, significa fazer cálculos, contar, efetuar operações aritméticas, COMPUTADOR seria então o mecanismo ou máquina que auxilia essa tarefa, com vantagens no tempo gasto e na precisão. Inicialmente o homem utilizou seus próprios dedos para essa tarefa, dando origem ao sistema DECIMAL e aos termos DIGITAL e DIGITO . Para auxílio deste método, eram usados gravetos, contas ou marcas na parede.

Apartir do momento que o homem pré-histórico trocou seus hábitos nômades por aldeias e tribos fixas, desenvolvendo a lavoura, tornou-se necessário um método para a contagem do tempo, delimitando as épocas de plantio e colheita.

Tábuas de argila foram desenterradas por arqueólogos no Oriente Médio, próximo à Babilônia, contendo tabuadas de multiplicação e recíprocos, acredita-se que tenham sido escritas por volta de 1700 a.C. e usavam o sistema sexagesimal (base 60), dando origem às nossas atuais unidades de tempo.

1.1. A evolução da Informática

500 AC Ábaco:

Na medida em que os cálculos foram se complicando e aumentando de tamanho, sentiu-se a necessidade de um instrumento que viesse em auxílio, surgindo assim há cerca de 2.500 anos atrás o ÁBACO. Este era formado por fios paralelos e contas ou arruelas deslizantes, que de acordo com a sua posição, representava a quantidade a ser trabalhada.

O ábaco russo era o mais simples: continham 10 contas, bastando contá-las para obtermos suas quantidades numéricas. O ábaco chinês possuía 2 conjuntos por fio, contendo 5 contas no conjunto das unidades e 2 contas que representavam 5 unidades. A variante do ábaco mais conhecida é o SOROBAN, ábaco japones simplificado (com 5 contas por fio, agrupadas 4x1), ainda hoje é utilizado, sendo que em uso de mãos treinadas continuam eficientes e rápidos para trabalhos mais simples.

Esse sistema de contas e fios recebeu o nome de calculi pelos romanos, dando origem à palavra cálculo.

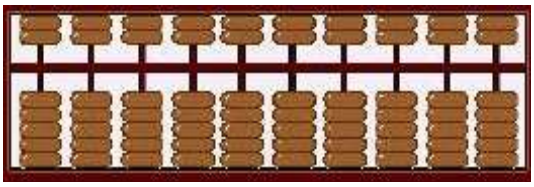


Figura 1.1 – Ábaco

1614 Logaritmos e régua de cálculo:
Os Bastões de Napier foram criados como auxílio à multiplicação, pelo nobre escocês de Edinburgo, o matemático John Napier (1550-1617), inventor dos logaritmos. Dispositivos semelhantes já vinham sendo usados desde o século XVI, mas somente em 1614 foram documentados. Os bastões de Napier eram um conjunto de 9 bastões, um para cada dígito, que transformavam a multiplicação de dois números numa soma das tabuadas de cada dígito.

Em 1633, um sacerdote inglês chamado William Oughtred, teve a idéia de representar esses logaritmos de Napier em escalas de madeira, marfim ou outro material, chamando-o de Círculos de Proporção. Este dispositivo originou a conhecida Régua de Cálculos. Como os logaritmos são representados por traços na régua e sua divisão e produto são obtidos pela adição e subtração de comprimentos, considera-se como o primeiro computador analógico da história.

1642 Calculadora de Pascal:
O matemático francês Blaise Pascal começa a construir sua máquina de calcular. Ela é composta por rodas dentadas. O usuário disca os números nas rodas dentadas para realizar os cálculos.

Figura 1.2 - Calculadora de Pascal

1672 Calculadora de Leibniz:
O matemático alemão Gottfried Leibniz aperfeiçoa a calculadora de Pascal facilitando as operações de multiplicação e divisão.

Figura 1.3 - Calculadora de Leibniz

1822 Máquinas de Babbage:
O matemático britânico Charles Babbage começa a trabalhar no projeto de uma máquina diferencial e de uma máquina analítica. Por razões diversas suas máquinas não chegam a ser construídas. Os projetos, todavia, servem de base a pesquisadores que vem depois para o desenvolvimento dos computadores modernos.

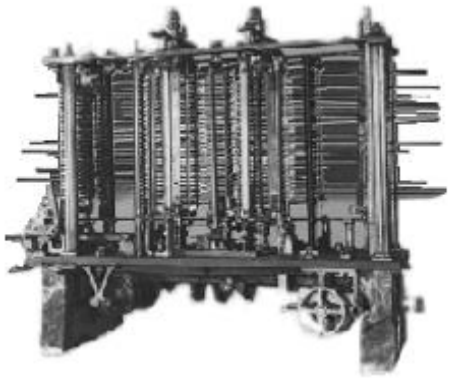


Figura 1.4 - Máquinas de Babbage

1848 Álgebra booleana:
Uma das maiores contribuições para a História da Informática não é uma máquina, mas uma teoria matemática. O matemático inglês George Boole desenvolve a chamada álgebra booleana que cria a base teórica para todo o desenvolvimento posterior da Informática.

1890 Computador mecânico de cartões:
Hermann Hollerith desenvolve o primeiro computador mecânico para acelerar os trabalhos do censo americano de 1890. A máquina de Hollerith lê os cartões perfurados usados no recenseamento. A empresa de Hollerith em 1924 se torna a IBM (Internacional Business Machines).

1938 Teoria da Informação:
O matemático americano Shannon publica uma tese que mais tarde será conhecida como Teoria da Informação. A partir da Teoria da Informação fica demonstrado que a melhor maneira de processar dados é utilizando o sistema binário de contagem.

1943 Mark I:
O Mark I é desenvolvido num projeto conjunto da Marinha Americana com a IBM e chefiado pelo americano Howard Aiken. O Mark I é considerado o primeiro computador moderno. Trabalha com cartões perfurados e relês elétricos. É usado para fazer cálculos complexos. Em um dia faz cálculos que antes levavam seis meses.

1945 ENIAC:
O exército americano quer uma máquina que faça complexos cálculos balísticos. John Mauchly e J. Presper Eckert apresentam o projeto de uma máquina com válvulas eletrônicas. Em 1945 começa a funcionar o ENIAC (Eletronical Numerical Integrator and Computer). O ENIAC mede 5,5m de altura por 25m de comprimento e pesa 30 toneladas.

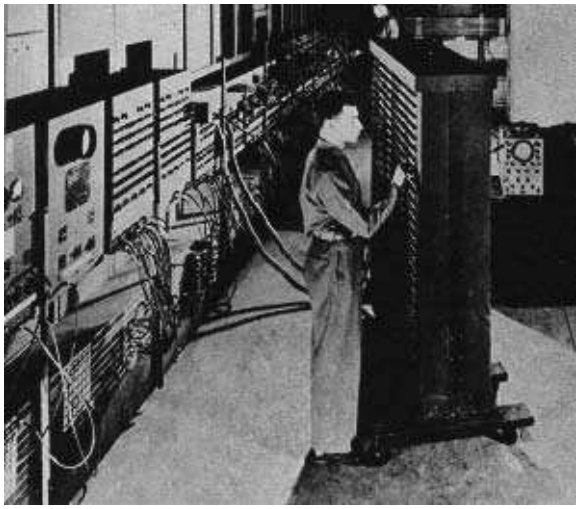


Figura 1.5 – ENI AC

1947 Transistor:
A equipe do laboratório Bell, chefiada pelo americano Willian Shockley desenvolve o primeiro transistor. A invenção do transistor substitui as válvulas, servindo de base para a criação dos circuitos integrados e mais tarde dos modernos processadores.

1951 UNIVAC:
Os desenvolvedores do Mark I, Mauchly e Eckert, lançam o primeiro computador comercial. O UNIVAC (Universal Automatic Computer) era eletrônico e armazenava dados em fitas magnéticas. Este computador foi produzido pela mpresa Remington Rand. O primeiro comprador foi o Departamento Americano do Censo de 1951.

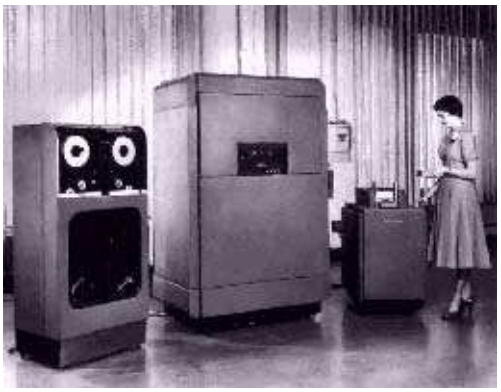


Figura 1.6 - UNIVAC

1957 FORTRAN:
O americano John Backus apresenta a primeira linguagem de alto nível para computadores, o FORTRAN. Depois dela surgem o COBOL, ALGOL, LISP e Pascal.

1959 Circuito integrado:
É proposto originalmente pelo inglês G.W. Dummer em 1952. É patenteado em 1959 por Jack St Clair Kilby da Texas Intruments. O circuito integrado utiliza transistores alojados em pequenas cápsulas de material semi condutor. Circuitos eletrônicos imensos passam a ser compactados em pequenos chips.

1964 IBM 360:
A IBM, líder na fabricação de computadores comerciais, lança a família de computadores 360. A família 360 é chamada de terceira geração e torna−se um marco da indústria. Utiliza o conceito de multi tarefa, emulação de outros computadores e de compatibilidade.

Figura 1.7 - IBM 360

1969 ARPANET:
Quatro universidades americanas interligam seus computadores em rede A partir desta rede, chamada ARPANET, nasce a Internet.

1971 Chip programável:
O americano Ted Hoff, da empresa Intel, desenvolve o primeiro chip programável, o 4004. Este chip abre o caminho para os processadores atuais.

1975 Altair:
O americano Edward Roberts lança o primeiro computador popular, o Altair. O kit para montagem do Altair custa cerca de 500 dólares e utiliza o processador 80080 da Intel.

Figura 1.8 - Altair 8800

1976 Basic:
Os americanos Paul Allen e Bill Gates desenvolvem a linguagem Basic, para facilitar a utilização do Altair. Esta linguagem existia desde 1965.
Apple: Os americanos Steven Jobs e Stephen Wozniac criam a empresa Apple, que nos anos seguintes populariza os micro computadores.

Figura 1.9 - Apple II

1980 IBM PC:
A IBM lança a sua versão de computador pessoal. O PC (personal computer) da IBM estabelece o padrão para os atuais computadores pessoais. O chip utilizado é o 80086 da Intel e o sistema operacional usado é o MS−DOS, desenvolvido pela empresa Microsoft, de Bill Gates.

1985 Windows:
A Microsoft lança um sistema operacional com interface gráfica que se torna o software mais popular da História da Informática.

1989 WWW:
Tim Bernes Lee apresenta um padrão de comunicação que deixa a Internet mais atraente e intuitiva. Esta forma de divulgação, conhecida como Word Wide Web, impulsiona a popularização da Internet.

1993 Mosaic:
No laboratório europeu CERN é desenvolvido o software gráfico Mosaic, para navegação na Internet. O Mosaic serve de base para a criação dos navegadores Netscape Navigator e Microsoft Internet Explorer.

2000 AOL Time Warner:
O maior provedor de acesso do mundo, a AOL, assume o controle da maior empresa de comunicação, a TimeWarner. É a maior operação comercial da História. Esta fusão anuncia a integração da Internet com os grandes meios de comunicação para criar o meio de comunicação do século XXI.

1.2. Estrutura de um computador

1.2.1 Hardware e Software:

Hardware: são as unidades físicas, componentes, circuitos integrados, discos e mecanismos que compõem um computador ou seus periféricos.

Software: qualquer programa ou grupo de programas que instrui o hardware sobre a maneira como ele deve executar uma tarefa, inclusive sistemas operacionais, processadores de texto e programas de aplicação.

Aquelas partes do computador que você consegue ver e tocar, como o teclado, o mouse, o monitor, são chamadas de hardware. A palavra hardware é inglesa e nos dá a idéia de produto sólido, palpável. Hardware é equipamento de Informática.

O computador é um conjunto de peças de hardware. Se existisse apenas hardware, os computadores não teriam utilidade, pois, o hardware sozinho não sabe trabalhar.O computador é uma máquina programável, ou seja, o homem deve dar−lhe instruções para que realize tarefas. Estas instruções formam os programas. São os programas que põe o hardware para trabalhar, esses

programas são chamados de Software. Ela nos dá a idéia de um produto impalpável, ou seja, que não podemos tocar. O software é um produto intelectual. Software é programa de computador.

1.2.2 CPU, memória e periféricos:

Unidade Central de Processamento (CPU): grupo de circuitos que executam as funções básicas do computador é composto pela unidade de controle e a unidade lógica e aritmética.

A CPU é o cérebro do computador. É ela que comanda todas as funções do computador. A informática como a conhecemos hoje só se tornou possível graças ao grande desenvolvimento alcançado pelas CPUs. A CPU é um circuito eletrônico muito poderoso que recebem dados, processa e devolve dados processados.

Unidade de Controle: É a unidade responsável pelo tráfico dos dados. Ela obtém dados armazenados na memória e interpreta os mesmos. Controla a transferência de dados da memória para a unidade lógica aritmética, da entrada para a memória e memória para saída.

Unidade Lógica Aritmética: Nesta unidade são feitos todos os cálculos aritméticos e qualquer manipulação de dados, sejam eles numéricos ou não.

Memória: A CPU processa dados. Dados brutos entram na CPU e dados processados saem dela. Esses dados precisam ser armazenados. O computador tem dispositivos capazes de reter informações. São as memórias. Normalmente num computador existem vários tipos de memória para armazenarem dados.

Periféricos: Considera–se que as partes principais do computador são a CPU e as memórias. As demais partes são chamadas de periféricos. Exemplos de periféricos: mouse, teclado, monitor e impressora.

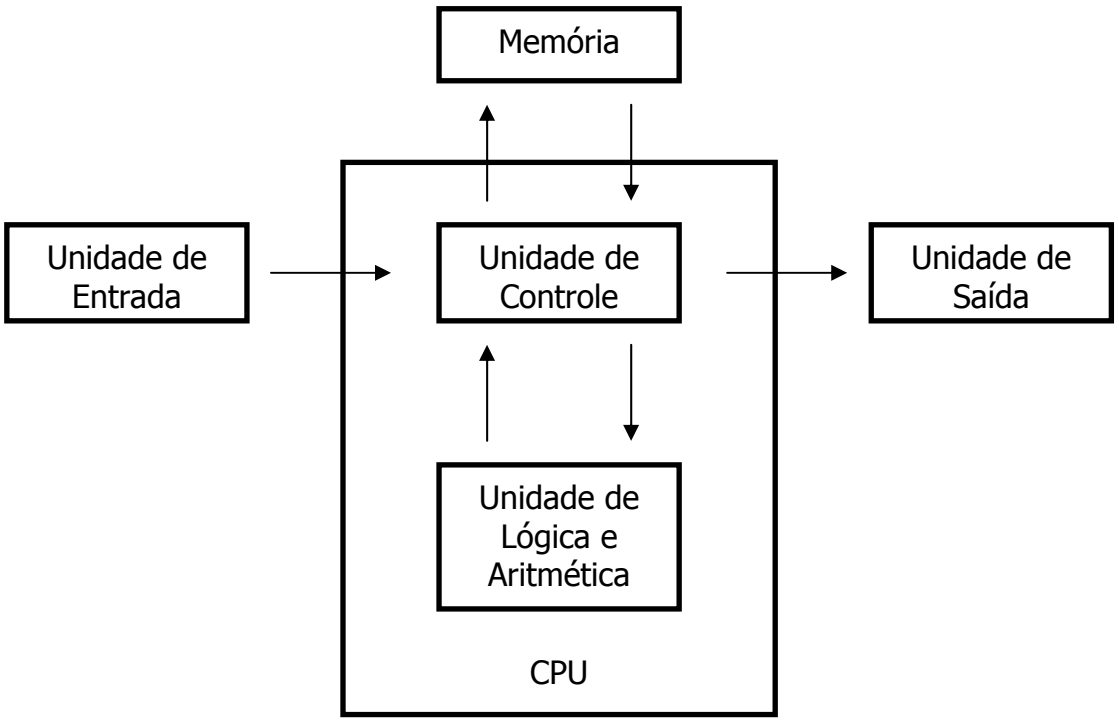


Figura 1.10 – Computador

1.3. Conceitos de Linguagem Programação

Para armazenar um algoritmo na memória de um computador e para que ele possa, em seguida, comandar as operações a serem executadas, é necessário que ele seja programado, isto é, que seja transcrito para uma linguagem que o computador possa "entender", direta ou indiretamente.

Os computadores só podem executar diretamente os algoritmos expressos em linguagem de máquina, que é um conjunto de instruções capazes de ativar diretamente os dispositivos eletrônicos do computador. Esta linguagem tem vários inconvenientes para os humanos. É diferente para cada tipo de computador, pois depende da sua arquitetura. Além disto, é extremamente rudimentar e exige que, mesmo as operações mais simples ainda sejam refinadas, para expressa-las em termos de registros, acumuladores e outros dispositivos das máquinas. É totalmente expressa em forma numérica (binário ou hexadecimal), que a torna pouco expressiva para os humanos. Exige um cuidado extremo para se estabelecer o posicionamento dos dados e das instruções na memória.

Apesar de tudo isto, nos primeiros computadores, a linguagem de máquina era a única em que se podia fazer a programação. Mas logo, surgiu a idéia de escreverem programas em **linguagem simbólica**, mais conhecida por **Linguagem Assembler** ou **Linguagem Montadora**, em que a linguagem de máquina é expressa não apenas por números, mas também por letras e símbolos mais significativos para os humanos. O posicionamento dos dados e instruções na memória é também feito de forma simbólica. Um programa em linguagem Assembler, para controlar o computador, deve primeiro ser transformado em linguagem de máquina. Como cada comando da linguagem Assembler corresponde a um comando em linguagem de máquina, esta transformação pode ser feita facilmente pelo próprio computador, através de um programa chamado montador ou Assembler, escrito em linguagem de máquina e geralmente distribuído pelo próprio fabricante do computador. Posteriormente, quase todas as linguagens Assembler passaram também a ser dotados de alguns comandos que correspondem a várias instruções em linguagem de máquina.

O sucesso da linguagem Assembler logo animou os primeiros pesquisadores a criar linguagens em que a programação era feita através de uma anotação matemática e algumas palavras da língua inglesa, em que o computador fazia a tradução através de compiladores.

A primeiro linguagem que surgiu foi o FORTRAN (Formula Translation) em 1957.

1957 - Linguagem FORTRAN (Formula Translation);
1959 - Linguagem COBOL (Common Business Oriented Language);
1964 - Linguagem BASIC (Begineer´s All-Purpose Symbolic Instruction Code);
1960 - Linguagem ALGOL (Algorithmic Language);
1968 - Linguagem PASCAL;
1972 - Linguagem C.

A escolha da linguagem de programação para se usar um computador depende antes de tudo da existência de um programa compilador ou de um programa interpretador. Existindo compiladores ou interpretadores para diversas linguagens, a escolha pode ser pela linguagem preferida ou mais familiar para o programador, por aquela melhor implementada no computador, por aquela mais adequada para o tipo de aplicação que se deseja fazer etc.

1.4. Algoritmos

Nesta seção serão abordados os conceitos associados à utilização do computador. O conjunto destes conceitos, técnicas, metodologias e ferramentas, que viabilizam a programação dos computadores, denomina-se software.

O conceito central da programação e da ciência da computação em geral é o de algoritmo. A programação é a arte ou técnica de construir e formular algoritmos de uma forma ordenada. Programas de computadores são formulações concretas de algoritmos abstratos, baseados em representações e estrutura especificas de dados.

Para entender o conceito de algoritmo e necessário saber o conceito de ação.

Ação é um acontecimento que, a partir de um estado inicial, após um período de tempo finito, produz um estado final previsível e bem definido.

Exemplo: foi pedido a duas pessoas para escolher um valor aleatório, e escrevessem os termos de seqüência de Fibonacci inferiores a este valor.

Rubens escolheu 50: 1 1 2 3 5 8 13 21 34

Sandrinho escolheu 13: 1 1 2 3 5 8

Parte do que Rubens e sandrinho escolheram são ações distintas. A escolha dos valores é totalmente imprevisível então não é uma ação. Porém, ao escrever os termos da seqüência de Fibonacci inferiores ao valor, cada uma destas pessoas realizou uma ação especifica: cada pessoa partiu de um estado inicial (1) e após um tempo limitado chegou a um estado final, previsível e bem definido. Apesar disto, mesmo que as ações foram distintas foi visto um mesmo padrão de comportamento, a subordinação a uma mesma norma de execução.

Definição de Algoritmo: é a descrição de um conjunto de comandos que, obedecidos, resultam numa sucessão de ações finitas.

Um algoritmo se destina a resolver um problema: fixa um padrão de comportamento a ser seguido, uma norma de execução a ser trilhada, para se atingir, como resultado final, a solução de um problema.

1.4.1 Refinamento sucessivo

Na vida quotidiana, os algoritmos são encontrados freqüentemente: instruções para se utilizar um aparelho eletrodoméstico, uma receita para preparo de algum prato, o guia de preenchimento da declaração do imposto de renda, a regra para determinação de máximos e mínimos de funções por derivadas sucessivas, a maneira como as contas de água, luz e telefone são calculadas mensalmente.

Um algoritmo é considerado completo se os seus comandos forem do entendimento do seu destinatário.

Num algoritmo, um comando que não for do entendido pelo seu destinatário terá de ser desdobrado em novos comandos, que constituirá em um **refinamento** do comando inicial.

1.4.2 Algoritmo estruturados

Os computadores são maquinas destinadas a resolver problemas com grande rapidez: uma operação aritmética pode ser efetuada num tempo da ordem de nano segundos. O aproveitamento desta grande rapidez exige que as operações sejam efetuadas automaticamente, sem a interferência humana (os humanos são lentos).

Por outro lado, máquina dotada desta rapidez e deste automatismo necessita de um investimento muito alto, só se justificando, caso fosse possível resolver problemas da mais diversa natureza. Estas máquinas são consideradas flexíveis.

A flexibilidade exige que, para cada problema a ser levado ao computador, sejam planejadas as operações correspondentes. O automatismo, por outro lado, exige que o planejamento destas operações seja feito previamente, antes de se utilizar o computador.

Então, a utilização de um computador para resolver problemas exige que se desenvolva um algoritmo. Este algoritmo deve prever antecipadamente todas as situações que possam ocorrer quando for posto em execução.

Por esta razão, nas ultimas décadas surgiram técnicas que permitem sistematizar e ajudar o desenvolvimento de algoritmos para a resolução de grandes e complexos problemas nos computadores: são as técnicas de **desenvolvimento estruturado** de algoritmos.

Os objetivos desta técnicas são:

- Facilitar o desenvolvimento dos algoritmos;
- Facilitar o entendimento humano;
- Antecipar a comprovação da sua correção;
- Facilitar a sua manutenção e sua modificação;
- Permitir que o seu desenvolvimento possa a ser empreendimento simultaneamente por uma equipe de pessoas.

Para atingir estes objetivos, o desenvolvimento estruturado preconiza que:

a) Os algoritmos sejam desenvolvidos por refinamento sucessivo partindo de uma descrição geral e, gradativa e sucessivamente, atacando as minúcias e particularidades. Este desenvolvimento também se denomina "construção hierárquica de algoritmos" e "desenvolvimento de cima para baixo" (em inglês, top-down);
b) Os sucessivos refinamentos são módulos, que delimitam poucas funções e são o mais independente possível, isto é, conservam poucos vínculos com outros módulos;
c) Nos módulos deve ser usado um numero limitado de diferentes comandos e de diferentes **estruturas de controle**.

O **refinamento sucessivo** dos algoritmos permite uma abordagem mais segura e objetiva do problema. A integração dos módulos é atacada quando já se estudou claramente o ambiente em que ele deve atuar. A

alteração dos módulos já desenvolvidos e teoricamente desnecessário. Na prática, consegue-se diminuir muito estas alterações pela atenção e disciplina. Os módulos de mais alto nível podem ser testados antes de se desenvolverem os de níveis mais baixos, permitindo o desenvolvimento mais seguro dos algoritmos.

A **divisão em módulos funcionais** permite contornar a limitação humana para compreender a complexidade. Cada modulo pode ser desenvolvido ou analisado de forma quase independente, dados os poucos vínculos que deve manter com os outros módulos do algoritmo. Cada módulo pode, portanto, ser desenvolvido por uma pessoa, após acertar poucos acordos com o resto da equipe.

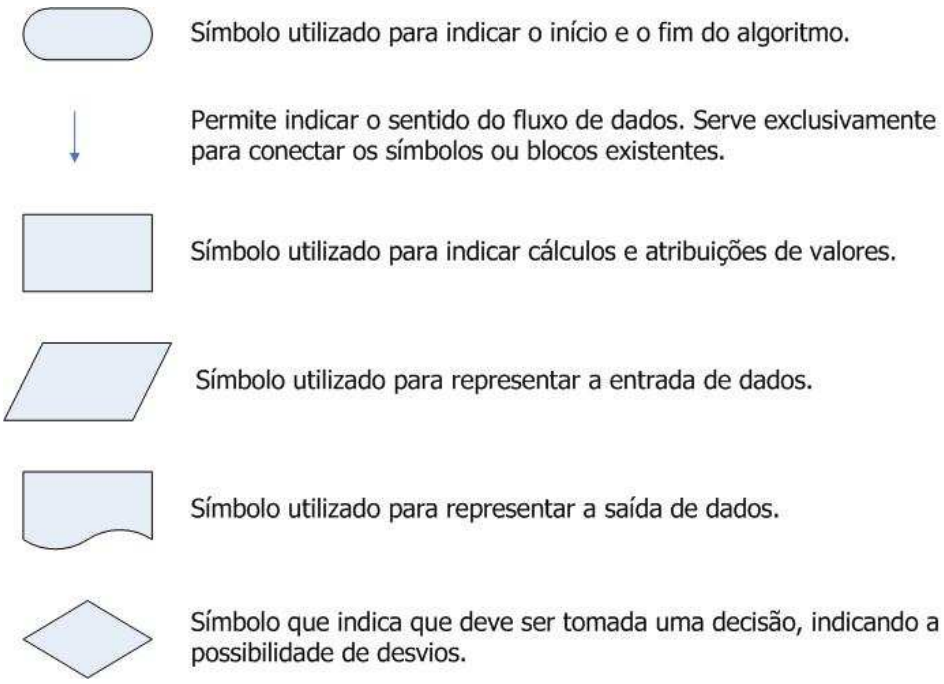
O **uso limitado de diferentes comandos** e diferentes estruturas de controle pode restringir bastante o raciocínio, que fica obrigado a se enquadrar em poucas formas. Mas o benefício em termos de simplicidade e clareza do produto final são imensos.

Clareza e simplicidade de um algoritmo são atributos inestimáveis quando se faz a sua manutenção e a sua modificação. O desenvolvimento estruturado dos algoritmos, aliado à busca da clareza e da simplicidade, facilita a sua manutenção e a sua modificação.

1.4.3 Diagramas de Blocos

Para se obter uma melhor clareza dos algoritmos, costuma-se desenvolvê-los e ilustrá-los com o auxílio de um diagrama de blocos: algumas figuras geométricas e diretrizes são usadas para representar as diversas operações do computador, sendo ligados por setas, para indicar a seqüência da sua execução.

Na Figura 1.11, é um exemplo de diagrama de bloco de um algoritmo de uma seqüência de Fibonacci.



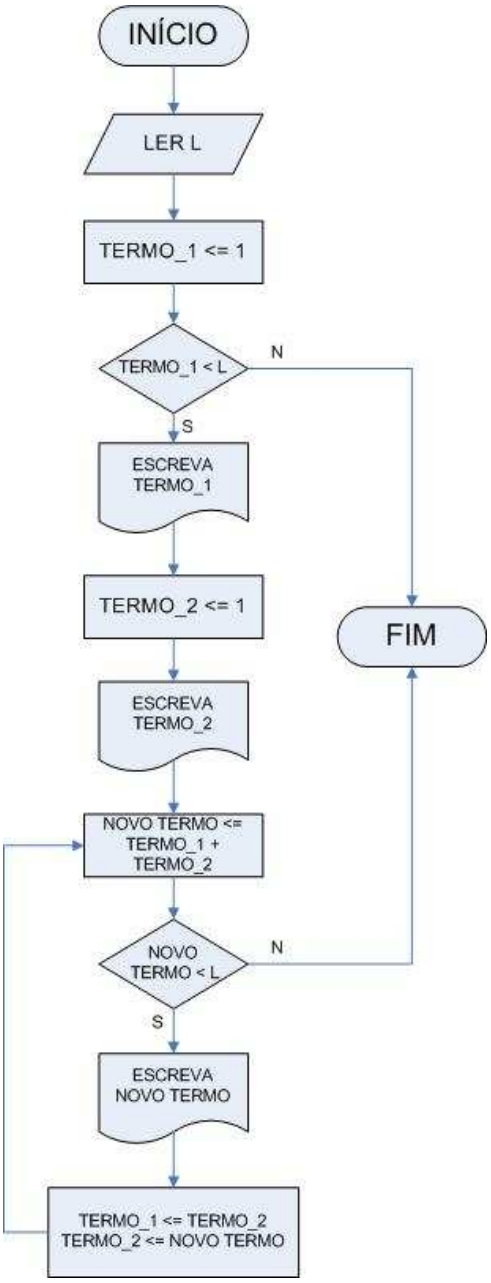


Figura 1.11 – Diagrama de Blocos

1.5. Exercício

Realize uma série que respeite o seguinte padrão:

$$x_0\ x_1\ x_2\ x_3 \ldots x_{i-3}\ x_{i-2}\ x_{i-1}\ x_i \ldots x_k$$

$$x_0 = x_1 = x_2 = 1$$

$$x_i = x_{i-1} + x_{i-3}$$

$$x_i < L$$

2. Elementos de Programação

Uma vez conhecida a definição de algoritmo, a partir deste capítulo será introduzido um conjunto particular de regras e convenções para o seu desenvolvimento.

2.1. Estrutura de um Programa C

Uma particularidade interessante no programa C é seu aspecto modular e funcional, em que, o próprio programa principal é uma função. Esta forma de apresentação da linguagem facilita o desenvolvimento de programas, pois permite o emprego de formas estruturadas e modulares.

```
< definições de pré-processamento>
< protótipos de funções>
< declarações de variáveis globais>

void main(void)
{
    /* corpo da função principal, com declarações
    de suas variáveis, seus comandos e funções */
}

<tipo> func( [lista de parâmetros] )
<declaração dos parâmetros>
{
    /* corpo da função func( ) com suas declarações
    de variáveis, comandos e funções */
}
```

Ex. do primeiro programa:

```
#include <stdio.h> /* inclui informação sobre a biblioteca de
                      comandos*/
main()             /* define uma função chamada main que não
                      recebe argumentos*/
{                  /* comando de main são delimitados por chaves*/
   printf("primeiro programa\n");   /* main chama a função de
                                       biblioteca printf para imprimir esta
                                       seqüência de caracteres; \n
                                       representa o caractere de nova
                                       linha. */
}
```

2.2. Constante

Uma constante é um determinado valor fixo que não se modifica ao longo do tempo, durante a execução de um programa.

Uma constante pode ser um número, um valor lógico ou uma seqüência de caracteres quaisquer com algum significado para o problema em estudo. Conforme o seu tipo, a constante é classificada com sendo **numérica**, **lógica** ou **literal**.

2.2.1 Constante numéricas

A representação de uma constante numérica nos algoritmos é feita no sistema decimal, podendo ser um número com ou sem parte fracionária.

A constante numérica pode ser positiva ou negativa, de acordo com o sinal que precede os algarismos formadores do número. Caso não exista um sinal, a constante é considerada positiva.

Tipo	Numero de bits	intervalo	
char	8	-128	127
unsigned char	8	0	255
signed char	8	-128	127
int	16	-32.768	32.767
unsigned int	16	0	65.535
signed int	16	-32.768	32.767
short int	16	-32.768	32.767
unsigned short int	16	0	65.535
signed short int	16	-32.768	32.767
long int	32	-2.147.483.648	2.147.483.647
signed long int	32	-2.147.483.648	2.147.483.647
unsigned long int	32	0	4.294.967.295
float	32	3,4E-38	3.4E+38
double	64	1,7E-308	1,7E+308
long double	80	3,4E-4932	3,4E+4932

Ex.: 3.14
´a´
10000.0201
1.0567e-4

2.2.2 Constante Lógica

É um valor lógico, isto é, que só pode ser FALSO ou VERDADEIRO, usado em proposições lógicas, conforme será visto mais adiante.

Essas constantes lógicas são representadas em C por números inteiros em que o Falso é o valor 0 e o Verdadeiro por números diferentes de 0 (FALSO = 0 e VERDADEIRO ≠ 0).

Obs.: todo número inteiro pode ser interpretado como uma constante lógica na linguagem de programação em C.

2.2.3 Constante Literal

Uma constante deste tipo pode ser qualquer seqüência de caracteres (letras, dígitos ou símbolos especiais) que forme um literal com algum significado para o problema em estudo.

Em C uma constante literal e chamada de tipo de dado string, só que em C não existe realmente uma String no seu lugar existe uma matriz de caracteres. Uma string é uma matriz tipo char que termina com '\0'. Por essa razão uma string deve conter uma posição a mais do que o número de caracteres que se deseja. Constantes string´s são uma lista de caracteres que aparecem entre aspas, não sendo necessário colocar o '\0', que é colocado pelo compilador.

Ex.: "Oi!"
"Oi tudo bem!"

2.3. Variáveis

Sabe-se da Matemática que uma variável é a representação simbólica dos elementos de um certo conjunto.

Nos algoritmos, destinados a resolver um problema no computador, a cada variável correspondente uma posição de memória, cujo conteúdo pode variar ao longo do tempo durante a execução de um programa. Embora uma variável possa assumir diferentes valores, ela só pode armazenar um valor a cada instante.

Toda variável é identificada por um nome ou identificador. Os identificadores representam a posição de memória que armazenam os valores.

2.3.1 Formação dos identificadores

Um identificador é a formado por um ou mais caracteres, sendo que o primeiro caractere deve, obrigatoriamente, ser uma letra e os caracteres seguintes, letras dígitos, não sendo permitido o uso de símbolos especiais.

É recomendável que os nomes das variáveis sejam os mais significativos possíveis, isto é, que reflitam, da melhor maneira, a natureza dos valores que nelas estão sendo armazenados.

A linguagem C é case sensitive, ou seja, as letras maiúsculas diferem das minúsculas.

2.3.2 Declaração de Variáveis

As variáveis só podem armazenar valores do mesmo tipo, de maneira que também podem ser classificadas como sendo numéricas, lógicas e literais.

Na linguagem C, isso é feito da seguinte forma:

Nome de uma variável:

- tamanho depende do ambiente (em média, 14 caracteres)
- letras maiúsculas, minúsculas, números e "_"
- na prática:
  - ✓ letras minúsculas: variáveis, funções e comandos;
  - ✓ letras maiúsculas: constantes.

Ex.:

```
int salario;
char letra;
const int idade = 23;
double pi = 1.34e-9;
char nome[9] = "Flávio";
char cidade[20] = "Linhares";
```

2.4. Comentários

A esta altura o leitor já percebeu a preocupação existente com a clareza do algoritmo, ou seja, o grau de facilidade que as pessoas terão em compreender o que nele está descrito. Com esse objetivo, usa-se um meio muito importante para isso, comentário.

Na linguagem C, o comentário é delimitado por ***/ /*** (não é padrão ansi C) ou ***/ **** (abre comentário) *****/*** (fecha comentário).

Ex.:

```
int salário = -1000;      // Armazena o salário de um Professor
const int Idade = 45;     /* Armazena a idade do professor Roney* /
```

2.5. Expressões Aritméticas, Lógicas e Literais

2.5.1 Expressões Aritméticas

Denomina-se expressão aritmética aquela cujos operadores são aritméticos e cujos os operandos são constantes e/ou variáveis do tipo numérico.

O conjunto de operações básicas adotado e o que se conhece da Matemática, a saber:

- Adição (+)
- Multiplicação (*)
- Subtração (-)
- Divisão (/)
- Resto (%)

Para se obter uma seqüência de calculo diferente, vários níveis de parênteses podem ser usados para quebrar as prioridades definidas. Não é permitido o uso de colchetes e chaves, uma vez que estes símbolos são utilizados nos algoritmos para outras finalidades.

A seguir, são representadas algumas expressões aritméticas, onde aparecem as operações mencionadas acima.

```
#include <stdio.h>

main()
{
        int x,y;
        double c,z;

        x = 10;
        y=3;
        c = 10.03;

        x += 2;                 // x = x + 2;
        y*= x+1;                // y = y*(x+1);
        y-= x;                  // y = y-x;
        x = 9 % 2;
        z = c * 3;

        printf("\n%d %d %f %f",x,y,c,z);
        getch();
        return 0;
}
```

2.5.2 Expressões Lógicas

É comum nos algoritmos surgirem situações em que a execução de uma ação, ou seqüência de subações, esta sujeita a uma certa condição. Esta condição é representada no texto do algoritmo por meio de uma expressão lógica.

Denomina-se expressão lógica a expressão cujos operadores são lógicos e cujos operandos são relações, constantes e/ou variáveis do tipo lógico.

Relações:

Uma expressão relacional, ou simplesmente relação, é uma comparação realizada entre dois valores de mesmo tipo básico. Estes valores são representados na relação através de constantes, variáveis ou expressões aritméticas, estas últimas para o caso de valores numéricos.

Tanto os operadores de relação como os lógicos tem precedência menor que os operadores aritméticos. As operações de avaliação produzem um resultado 0 ou 1.

>	maior que
> =	maior ou igual
<	menor
< =	menor ou igual
= =	igual
!=	não igual

O resultado obtido de uma relação é sempre um **valor lógico.**

Lógicos:

A Álgebra das Proposições define três conectivos usados na formação de novas proposições a partir de outras já conhecidas. Estes conectivos são os operadores nas expressões lógicas.

&&	and
||	ou
!	not

2.5.3 Expressões Literais

Uma expressão literal é aquela formada por operadores literais e operandos que são constantes e/ou variáveis do tipo literal.

Embora estas expressões tenham grande importância no estudo de programação, o objetivo desta seção é apenas o de introduzir o assunto. Isto porque as operações entre valores literais são bastante diversificadas e dependem das características de cada língua de programação.

2.6. Comandos de Atribuição

Este comando permite que se forneça um valor a uma certa variável, onde a natureza deste valor tem de ser compatível com o tipo da variável na qual está sendo armazenado.

O comando de atribuição tem a forma geral apresentada a seguir:

Identificador = expressão;

Ex.:

valor = 10.02;

teste = "sou eu e não é eu";

cabo = 20 * (13.1 + largura);

2.7. Comandos de I/O

As unidades de entrada e saída são dispositivos que possibilitam a comunicação entre o usuário e o computador.

Por exemplo, através de um teclado, o usuário consegue dar entrada ao programa e aos dados na memória do computador. Por sua vez, o computador pode emitir os resultados e outras mensagens para o usuário através de uma impressora de linhas.

Os funções de I/O mais conhecidas na linguagem C é o printf() e o scanf().

2.7.1 Função printf()

Sintaxe:

printf("expressão de controle",argumentos);

É uma função de I/O, que permite escrever no dispositivo padrão (tela).

A expressão de controle pode conter caracteres que serão exibidos na tela e os códigos de formatação que indicam o formato em que os argumentos devem ser impressos. Cada argumento deve ser separado por vírgula.

\n	Nova linha
\t	Tab
\b	retrocesso
\”	Aspas
\\	Barra
\f	Salta formulário
\0	Nulo

%c	Caractere simples
%d	Notação decimal
%e	Notação científica
%f	Ponto flutuante
%o	Octal
%s	Cadeia de caracteres
%u	Decimal sem sinal
%x	hexadecimal

Ex:

```
#include <stdio.h>
main()
{
    printf("Este é o numero dois: %d",2);
    printf("%s está a %d milhões de milhas\ndo sol","Vênus",67);
}
```

Tamanho de campos na impressão:

Ex:

```
#include <stdio.h>
main()
{
    printf("\n%2d",350);
    printf("\n%4d",350);
    printf("\n%6d",350)
}
```

Para arredondamento:

Ex:

```
#include <stdio.h>
main()
{
    printf("\n%4.2f",3456.78);
    printf("\n%3.2f",3456.78);
    printf("\n%3.1f",3456.78);
    printf("\n%10.3f",3456.78);
}
```

Para alinhamento:
Ex:

```
#include <stdio.h>
main()
{
    printf("\n%10.2f %10.2f %10.2f",8.0,15.3,584.13);
    printf("\n%10.2f %10.2f %10.2f",834.0,1500.55,4890.21);
}
```

Complementando com zeros à esquerda:
Ex:

```
#include <stdio.h>
main()
{
    printf("\n%04d",21);
    printf("\n%06d",21);
    printf("\n%6.4d",21);
    printf("\n%6.0d",21);
}
```

Imprimindo caracteres:
Ex:

```
#include <stdio.h>
main()
{
    printf("%d %c %x %o\n",'A','A','A','A');
    printf("%c %c %c %c\n",'A',65,0x41,0101);
}
```

A tabela ASCII possui 256 códigos de 0 a 255, se imprimirmos em formato caractere um número maior que 255, será impresso o resto da divisão do número por 256; se o número for 3393 será impresso A pois o resto de 3393 por 256 é 65.

2.7.2 Função scanf()

Também é uma função de I/O implementada em todos compiladores C. Ela é o complemento de printf() e nos permite ler dados formatados da entrada padrão(teclado). Sua sintaxe é similar a printf().

scanf("expressão de controle", argumentos);

A lista de argumentos deve consistir nos endereços das variáveis. C oferece um operador para tipos básicos chamado operador de endereço e referenciado pelo símbolo "&" que retorna o endereço do operando.

Operador de endereço &:

A memória do computador é dividida em bytes, e são numerados de 0 até o limite da memória. Estas posições são chamadas de endereços. Toda variável ocupa uma certa localização na memória, e seu endereço é o primeiro byte ocupado por ela.

Ex:

```
#include <stdio.h>
main()
{
    int num;
    printf("Digite um numero: ");
    scanf("%d",&num);
    printf("\no numero é %d",num);
    printf("\no endereco e %u",&num);
}
```

A melhor forma para se ler uma string é usar a função fgets(), a funções scanf() e gets(), elas não sabem o tamanho do vetor de caracteres, não podendo verificar o tamanho de linha sendo lida é maior que a área reservada. Apesar disso é muito usada, e tem servido para os hackers como porta de entrada de inúmeros esquemas de quebra de segurança.

```
char *fgets(char *s,int n,FILE *stream)
```

EX.:

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

int main()
{
    char area[100];
    int rc;
    int d,m,a;

    printf("Digite uma DATA(dia/mes/ano): ");
    fgets(area,99,stdin);
    rc = sscanf(area,"%d/%d/%d",&d,&m,&a);

    if(rc!= 3)
    {
        printf("\n\tDATA COM FORMATO INVALIDO\n");
        system("PAUSE");
        return -1;
    }

    printf("\nDIA = %d\n",d);
    printf("\nMES = %d\n",m);
    printf("\nANO = %d\n",a);

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

Obs.: para limpar o fluxo e poder ler um outro dado do buffer, é necessário usar o fflush(stdin).

2.8. Estruturas Seqüencial, Condicional e de Repetição

2.8.1 Estrutura Seqüencial

Num algoritmo aparecem em primeiro lugar as declarações seguidas por comandos que, se não houver indicações em contrário, deverão ser executados numa seqüência linear, seguindo-se o texto em que estão escritos, de cima para baixo.

Ex.:

```
#include <stdio.h>
main()
{
    int num;
    printf("Digite um numero: ");
    scanf("%d",&num);
    printf("\no numero é %d",num);
    printf("\no endereco e %u",&num);
}
```

2.8.2 Estrutura Condicional

sintaxe:

```
if (condição)
        comando;
else
        comando;
```

Se a condição for verdadeira (qualquer coisa diferente de 0), o computador executará o comando ou o bloco, de outro modo, se a cláusula for falsa (for igual a 0), o computador executará o comando ou o bloco que se encontra após a palavra else.

Ex:

```
#include <stdio.h>
main()
{
    int a,b;
    printf("digite dois números:");
    scanf("%d%d",&a,&b);
    if (b)
        printf("%d\n",a/b);
    else
        printf("divisão por zero\n");
}
```

Ex:

```
#include <stdio.h>
main()
{
    int num,segredo;
    segredo=143;
    printf("Qual e o numero: ");
    scanf("%d",&num);
    if (segredo==num)
    {
        printf("Acertou!");
        printf("\nO numero e %d\n",segredo);
    }
    else if (segredo<num)
        printf("Errado, muito alto!\n");
    else
        printf("Errado, muito baixo!\n");
}
```

EXERCÍCIO

Faça um programa que receba uma frase com 10 letras elimine as vogais e que imprime somente as consoantes. Para isto, use um 'if' para testar.

O Comando switch

O comando if-else e o comando switch são os dois comandos de tomada de decisão. Sem dúvida alguma o mais importante dos dois é o if, mas o comando switch tem aplicações valiosas. Mais uma vez vale lembrar que devemos usar o comando certo no local certo. Isto assegura um código limpo e de fácil entendimento. O comando switch é próprio para se testar uma variável em relação a diversos valores pré-estabelecidos. Sua forma geral é:

```
switch (variável)
{
case constante_1:
comandos_1;
break;
case constante_2:
comandos _2;
break;
.
.
.
case constante_n:
comandos _n;
break;
default
comandos _default;
```

}

Ex.:

```
#include <stdio.h>

int main()
{
int z;

    printf("Digite um valor: ");
    scanf("%d",&z);
    printf("\n");
    switch (z)
    {
        case 1:
                printf("declaracao 1");
                break;
        case 2:
                printf("declaracao 2");
                break;
        case 3:
                printf("declaracao 3");
                break;
        default:
                printf("declaracao default");
                break;
    }
    printf("\n");
}
```

2.8.3 Estrutura de Repetição

A estrutura de repetição permite que uma seqüência de comandos seja executada repetidamente até que uma determinada condição de interrupção seja satisfeita.

O Comando while

O comando while tem a seguinte forma geral:

```
while (condição)
{
      comandos;
}
```

Vamos tentar mostrar como o while funciona fazendo uma analogia. Então o while seria equivalente a:

```
if (condição)
{
    comandos;
    "Volte para o comando if"
}
```

Podemos ver que a estrutura while testa uma condição. Se esta for verdadeira a declaração é executada e faz-se o teste novamente, e assim por diante. Assim como no caso do for, podemos fazer um loop infinito. Para tanto basta colocar uma expressão eternamente verdadeira na condição. Pode-se também omitir a declaração e fazer um loop sem conteúdo. Vamos ver um exemplo do uso do while. O programa abaixo é executado enquanto i for menor que 100. Veja que ele seria implementado mais naturalmente com um for ...

Ex.:

```
#include <stdio.h>
int main ()
{
    int i = 0;
    while ( i < 100)
    {
        printf(" %d", i);
        i++;
    }
    return(0);
}
```

O programa abaixo espera o usuário digitar a tecla 'q' e só depois finaliza:

```
#include <stdio.h>
int main ()
{
    char Ch;
    Ch='\0';
    while (Ch!='q')
    {
        scanf("%c", &Ch);
    }
    return(0);
}
```

EXERCÍCIO

Refaça o programa da página anterior. Use o comando while para fechar o loop.

O Comando do-while

A terceira estrutura de repetição que veremos é o do-while de forma geral:

```
do
{
        comandos;
} while (condição);
```

Mesmo que a declaração seja apenas um comando é uma boa prática deixar as chaves. O ponto-e-vírgula final é obrigatório. Vamos, como anteriormente, ver o funcionamento da estrutura do-while "por dentro":

```
comandos;
if (condição) "Volta para a declaração"
```

Vemos pela análise do bloco acima que a estrutura do-while executa a declaração, testa a condição e, se esta for verdadeira, volta para a declaração. A grande novidade no comando do-while é que ele, ao contrário do for e do while, garante que a declaração será executada pelo menos uma vez.

Um dos usos da extrutura do-while é em menus, nos quais você quer garantir que o valor digitado pelo usuário seja válido, conforme apresentado abaixo:

```
#include <stdio.h>
int main ()
{
    int i;
    do
    {
        printf ("\n\nEscolha a fruta pelo numero:\n\n");
        printf ("\t(1)...Mamao\n");
        printf ("\t(2)...Abacaxi\n");
        printf ("\t(3)...Laranja\n\n");
        scanf("%d", &i);
    } while ((i<1)||(i>3));
    switch (i)
    {
        case 1:
            printf ("\t\tVoce escolheu Mamao.\n");
            break;
        case 2:
            printf ("\t\tVoce escolheu Abacaxi.\n");
            break;
        case 3:
            printf ("\t\tVoce escolheu Laranja.\n");
            break;
    }
    return(0);
}
```

EXERCÍCIO

Refaça o auto avaliação do comando FOR utilizando o laço do-while para controlar o fluxo.

O Comando for

for é a primeira de uma série de três estruturas para se trabalhar com loops de repetição. As outras são **while** e **do**. As três compõem a segunda família de comandos de controle de fluxo. Podemos pensar nesta família como sendo a das estruturas de repetição controlada.

O loop **for** é usado para repetir um comando, ou bloco de comandos, diversas vezes, de maneira que se possa ter um bom controle sobre o loop. Sua forma geral é:

```
for (inicialização;condição;incremento)
{
        comandos;
}
```

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

```
inicialização;
if (condição)
{
        comandos;
        incremento;
        "Volte para o comando if"
}
```

Podemos ver, então, que o **for** executa a inicialização incondicionalmente e testa a condição. Se a condição for falsa ele não faz mais nada. Se a condição for verdadeira ele executa a declaração, faz o incremento e volta a testar a condição. Ele fica repetindo estas operações até que a condição seja falsa. Um ponto importante é que podemos omitir qualquer um dos elementos do **for**, isto é, se não quisermos uma inicialização poderemos omiti-la. Abaixo vemos um programa que coloca os primeiros 100 números inteiros na tela:

Ex.:

```
#include <stdio.h>
int main ()
{
    int count;
    for (count=1; count<=100; count++)
          printf ("%d ",count);
    return(0);
}
```

Note que, no exemplo acima, há uma diferença em relação ao exemplo anterior. O incremento da variável **count** é feito usando o operador de

incremento que nós conhecemos. Esta é a forma usual de se fazer o incremento (ou decremento) em um loop **for**.

O **for** na linguagem C é bastante flexível. Temos acesso à inicialização, à condição e ao incremento. Qualquer uma destas partes do for pode ser uma expressão qualquer do C, desde que ela seja válida. Isto nos permite fazer o que quisermos com o comando. As três formas do for abaixo são válidas:

```
for ( count = 1; count < 100 ; count++)
{
    ...
}

for (count = 1; count < NUMERO_DE_ELEMENTOS ; count++)
{
    ...
}
for (count = 1; count < BusqueNumeroDeElementos() ;
count+=2)
{
    ...
}
```

etc ...

Preste atenção ao último exemplo: o incremento está sendo feito de dois em dois. Além disto, no teste está sendo utilizada uma função (BusqueNumeroDeElementos()) que retorna um valor que está sendo comparado com count.

O loop infinito

O loop infinito tem a forma

```
for (inicialização; ;incremento)
{
    comandos;
}
```

Este loop chama-se loop infinito porque será executado para sempre (não existindo a condição, ela será sempre considerada verdadeira), a não ser que ele seja interrompido. Para interromper um loop como este usamos o comando **break**. O comando **break** vai quebrar o loop infinito e o programa continuará sua execução normalmente.

Como exemplo vamos ver um programa que faz a leitura de uma tecla e sua impressão na tela, até que o usuario aperte uma tecla sinalizadora de final (um FLAG). O nosso FLAG será a letra 'X'. Repare que tivemos que usar dois scanf() dentro do for. Um busca o caractere que foi digitado e o outro busca o outro caracter digitado na seqüência, que é o caractere correspondente ao <ENTER>.

```
#include <stdio.h>
int main ()
{
    int Count;
    char ch;
    printf(" Digite uma letra - <X para sair> ");
    for (Count=1;;Count++)
    {
        scanf("%c", &ch);
        if (ch == 'X') break;
        printf("\nLetra: %c \n",ch);
        scanf("%c", &ch);
    }
    return(0);
}
```

O loop sem conteúdo

Loop sem conteúdo é aquele no qual se omite a declaração. Sua forma geral é (atenção ao ponto e vírgula!):

for (inicialização;condição;incremento);

Uma das aplicações desta estrutura é gerar tempos de espera.
O programa

```
#include <stdio.h>
int main ()
{
    long int i;
    printf("\a");             /* Imprime o caracter de alerta (um beep) */
    for (i=0; i<10000000; i++);  /* Espera 10.000.000 de iteracoes */
    printf("\a");                    /* Imprime outro caracter de alerta */
    return(0);
}
```

EXERCÍCIO

Faça um programa que inverta uma string: leia uma frase com a função fgets() e armazene-a invertida em outra string. Use o comando for para varrer a string até o seu final.

2.9. Exercícios:

1) Elabore programa que apresente mensagem "alô mundo!".
2) Elabore programa que imprima o seu nome.
3) Leia o nome e as notas de um aluno. Apresente seu nome e sua média.
4) Elaborar programa que imprima a tabuada de um número dado.
5) Dados 3 números, imprima a média destes números. Verifique se todos são positivos. Caso algum número seja negativo, indique ao usuário que a média não será calculada.
6) Elabore programa que leia "n" números digitados e apresente sua média.
7) Elabore programa que imprima a média de "n" números, excluindo o menor deles.
8) Dados "n" números apresente a média entre o maior e o menor número da seqüência fornecida.
9) Leia uma letra, um número inteiro, um número com casas decimais e uma string, depois os apresente.
10) Fazer um algoritmo em C que calcule e escreva o valor de S:

a) $S = \frac{1}{1} + \frac{3}{2} + \frac{5}{3} + \frac{7}{4} + \ldots + \frac{99}{50}$

b) $S = \frac{2^1}{50} + \frac{2^2}{49} + \frac{2^3}{48} + \ldots + \frac{2^{50}}{1}$

Obs.: **pow(x,y)** = x^y, para utilizar declarar nas definições de pré-processamento a biblioteca math.h, ainda x e y devem ser declaradas como double.

11) Escreva uma algoritmo em C para gerar e escrever uma tabela com os valores do seno de um ângulo A em radianos, utilizando a série de Mac-Laurin truncada, apresentada a seguir:

$$sen(A) = A - \frac{A^3}{3!} + \frac{A^5}{5!} - \frac{A^7}{7!}$$

12) Fazer um algoritmo em C que calcule o valor de e^x através da série:

$$e^x = x^0 - \frac{x^1}{1!} + \frac{x^2}{2!} - \frac{x^3}{3!} \cdots$$

13) Fazer um algoritmo em C:
   a) calcule o valor do cosseno de x em rad, através de 20 termos da série seguinte:

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \frac{x^8}{8!} - \cdots$$

   b) Calcule a diferença entre o valor calculado no item a e o valor fornecido pela função cos(x) da biblioteca math.h no qual x deve ser declarado como um valor do tipo doublé;
   c) Imprima o que foi calculado nos itens a e b.

14) Fazer um algoritmo em C que imprima uma tabela com os valores de x,y e f(x,y), para x = 1,4,9,16,...,100 e y = 0,1,2,3,...,5.

$$f(x,y) = \frac{x^2 + 3.x + y^2}{x.y - 5.y - 3.x + 15}$$

15) Dados 3 números, os imprima em ordem crescente usando apenas 1 comando printf.
16) Dados 2 números inteiros, imprima "ambos positivos ou negativos" se ambos forem positivos ou negativos, "sinais opostos" se tiverem sinais opostos ou "um deles e zero" se um deles for zero.
17) Elabore um menu de opções com 4 situações diversas, utilizando switch.
18) Elabore programa que permita a 5 pessoas escolherem sua cor favorita entre Verde, Vermelho, Amarelo, Azul, Laranja ou Roxo e exiba os resultados.
19) Elabore programa que permita a escolha entre 1, 2 e 3 ou indique erro de escolha.

3. Estruturas de Dados Fundamentais

3.1. Introdução

Como já foi visto variáveis correspondem a posições de memória, às quais o programador tem acesso, através de um algoritmo, visando atingir resultados propostos. Uma variável passa a existir a partir de sua declaração, quando, então, lhe são associados um nome ou identificador e a respectiva posição de memória por ela representada. Qualquer referência ao seu identificador significa o acesso ao conteúdo de uma única posição de memória.

3.2. Vetores

Um vetor é uma variável composta homogênea unidimensional formada por uma seqüência de variáveis, todas do mesmo tipo, com o mesmo identificador(mesmo nome) e alocadas sequencialmente na memória. Uma vez que as variáveis têm o mesmo nome, o que as distingues é um índice, que referencia sua localização dentro da estrutura.

Vetores é uma estrutura de dados muito utilizada. É importante notar que vetores, matrizes bidimensionais e matrizes de qualquer dimensão são caracterizados por terem todos os elementos pertencentes ao mesmo tipo de dado. Para se declarar um vetor pode utilizar a seguinte forma geral:

tipo_da_variável nome_da_variável [tamanho];

Os índices utilizados na linguagem C para identificar as posições de um vetor começam sempre em 0 (zero) e vão até o tamanho do vetor menos uma unidade.

Ex.:

```
#include <stdio.h>

main()
{
    int vet[10];

    vet[0]  = 5;
    vet[1]  = 8;
    vet[2]  = 9;
    vet[3]  = 10;
    vet[4]  = 78;
    vet[5]  = 98;
    vet[6]  = 24;
    vet[7]  = 7;
    vet[8]  = 6;
    vet[9]  = 2;
}
```

vet	5	8	9	10	78	98	24	7	6	2
índice	0	1	2	3	4	5	6	7	8	9

```
#include <stdio.h>

main()
{
    char x[5];

    x[0]  = 'A';
    x[1]  = '*';
    x[2]  = '2';
    x[3]  = '@';
    x[4]  = '\0';

}
```

x	'A'	'*'	'2'	'@'	'\0'
índice	0	1	2	3	4

No exemplo acima, o vetor vet possui dez posições, começando pela posição 0 e indo até 9(tamanho do vetor – 1). Em cada posição poderão ser armazenados números inteiros, conforme especificado pelo tipo int na declaração.

Da mesma forma, o vetor chamado x possui cinco posições, começando pela posição 0 e indo até a posição 4 (tamanho do vetor - 1). Em cada posição poderão ser armazenados caracteres, conforme especificado pelo tipo char na declaração.

Strings

É importante ressaltar que na linguagem C não existe o tipo de dado string, como ocorre no PASCAL e outras linguagens. Dessa maneira, para armazenar uma cadeia de caracteres, como o nome completo de uma pessoa, deve-se declarar um vetor de char, onde cada posição equivale a um caractere ou uma letra do nome. Deve-se lembrar, entretanto, que toda vez que se fizer uso de um vetor para armazenar uma cadeia de caractere, deve-se definir uma posição a mais que a necessária para armazenar a marca de finalização de cadeia(´\0´).

```
char nome_da_string [tamanho];
```

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

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

Fazer isto é um desastre. Quando você terminar de ler a seção que trata de ponteiros você entenderá porquê. As strings devem ser igualadas elemento a elemento.

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

```
#include <stdio.h>
int main ()
{
    int count;
    char str1[100],str2[100];
    .... /* Aqui o programa le str1 que sera
    copiada para str2 */

    for (count= 0;str1[count];count+ + )
    {
        str2[count]= str1[count];
    }
    str2[count]= '\ 0';

    .... /* Aqui o programa continua */
}
```

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

Vamos ver agora algumas funções básicas para manipulação de strings.

- **gets()**

A função gets() lê uma string do teclado. Sua forma geral é:
gets (nome_da_string);
O programa abaixo demonstra o funcionamento da função gets():

```
#include <stdio.h>
int main()
{
    char string[100];
    printf ("Digite o seu nome: ");
    gets(string);
    printf("\n\n Ola %s",string);
    return(0);
}
```

Repare que é válido passar para a função printf() o nome da string. Você verá mais adiante porque isto é válido. Como o primeiro argumento da função printf() é uma string também é válido fazer: printf (string);
isto simplesmente imprimirá a string.

- **fgets()**

Como já foi falado, a função mais eficiente para ler uma string é o fgets(). A função fgets() é usada para ler uma string de um fluxo de caracteres.

fgets(nome_da_string,tamanho_da_string,entrada_padrão);

obs.: fgets() pode ser usada para ler uma string de teclado e só usar como entrada_padrão stdin.

- **strcpy()**

Sua forma geral é:
strcpy (string_destino,string_origem);
A função strcpy() copia a string-origem para a string-destino. Seu funcionamento é semelhante ao da rotina apresentada na seção anterior. As funções apresentadas nestas seções estão no arquivo cabeçalho string.h. A seguir apresentamos um exemplo de uso da função strcpy():

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char str1[100],str2[100],str3[100];
    printf ("Entre com uma string: ");
    gets (str1);
    strcpy (str2,str1);                          /* Copia str1 em str2 */
    strcpy (str3,"Voce digitou a string ");      /* Copia "Voce digitou a
                                                    string" em str3 */
    printf ("\n\n%s%s",str3,str2);
    return(0);
}
```

- **strcat**

A função strcat() tem a seguinte forma geral:
strcat (string_destino,string_origem);
A string de origem permanecerá inalterada e será anexada ao fim da string de destino. Um exemplo:

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char str1[100],str2[100];
    printf ("Entre com uma string: ");
    gets (str1);
    strcpy (str2,"Voce digitou a string: ");
    strcat (str2,str1);    /* str2 armazenara' Voce digitou a string + o
                              conteudo de str1 */
    printf ("\n\n%s",str2);
    return(0);
}
```

- **strlen**

Sua forma geral é:

strlen (string);

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

Um exemplo do seu uso:

```
#include <stdio.h>
#include <string.h>
int main ()
{
    int size;
    char str[100];
    printf ("Entre com uma string: ");
    gets (str);
    size=strlen (str);
    printf ("\n\nA string que voce digitou tem tamanho %d",size);
    return(0);
}
```

- **strcmp**

Sua forma geral é:

strcmp (string1,string2);

A função strcmp() compara a string 1 com a string 2. Se as duas forem idênticas a função retorna zero. Se elas forem diferentes a função retorna não zero.

Um exemplo da sua utilização:

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char str1[100],str2[100];
    printf ("Entre com uma string: ");
    gets (str1);
    printf ("\n\nEntre com outra string: ");
    gets (str2);
    if (strcmp(str1,str2))
        printf ("\n\nAs duas strings são diferentes.");
    else
        printf ("\n\nAs duas strings são iguais.");
    return(0);
}
```

Exercício

Faça um programa que leia quatro palavras pelo teclado, e armazene cada palavra em uma string. Depois, concatene todas as strings lidas numa única string. Por fim apresente este como resultado ao final do programa.

3.3. Matrizes

3.3.1 Matrizes bidimensionais

Já vimos como declarar matrizes unidimensionais (vetores). Vamos tratar agora de matrizes bidimensionais. A forma geral da declaração de uma matriz bidimensional é muito parecida com a declaração de um vetor:

tipo_da_variável nome_da_variável [altura][largura];

É muito importante ressaltar que, nesta estrutura, o índice da esquerda indexa as linhas e o da direita indexa as colunas. Quando vamos preencher ou ler uma matriz no C o índice mais à direita varia mais rapidamente que o índice à esquerda. Mais uma vez é bom lembrar que, na linguagem C, os índices variam de zero ao valor declarado, menos um; mas o C não vai verificar isto para o usuário. Manter os índices na faixa permitida é tarefa do programador. Abaixo damos um exemplo do uso de uma matriz:

```
#include <stdio.h>
int main ()
{
    int mtrx [20][10];
    int i,j,count;
    count=1;
    for (i=0;i<20;i++)

    for (j=0;j<10;j++)
    {
        mtrx[i][j]=count;
        count++;
```

```
    }
    return(0);
}
```

No exemplo acima, a matriz **mtrx** é preenchida, sequencialmente por linhas, com os números de 1 a 200. Você deve entender o funcionamento do programa acima antes de prosseguir.

3.3.2 Matrizes de strings

Matrizes de strings são matrizes bidimensionais. Imagine uma string. Ela é um vetor. Se fizermos um vetor de strings estaremos fazendo uma lista de vetores. Esta estrutura é uma matriz bidimensional de chars. Podemos ver a forma geral de uma matriz de strings como sendo:

char nome_da_variável [num_de_strings][compr_das_strings];

Aí surge a pergunta: como acessar uma string individual? Fácil. É só usar apenas o primeiro índice. Então, para acessar uma determinada string faça:

nome_da_variável [índice]

Aqui está um exemplo de um programa que lê 5 strings e as exibe na tela:

```
#include <stdio.h>
int main ()
{
    char strings [5][100];
    int count;
    for(count=0;count<5;count++)
    {
        printf("\n\nDigite uma string: ");
        fgets(strings[count],99,stdin);
    }
    printf("\n\n\nAs strings que voce digitou
            foram:\n\n");
    for(count=0;count<5;count++)
    printf("%s\n",strings[count]);
    return(0);
}
```

3.3.3 Matrizes multidimensionais

O uso de matrizes multidimensionais na linguagem C é simples. Sua forma geral é:

tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN];

Uma matriz N-dimensional funciona basicamente como outros tipos de matrizes. Basta lembrar que o índice que varia mais rapidamente é o índice mais à direita.

Podemos inicializar as matrizes, assim como podemos inicializar variáveis. A forma geral de uma matriz como inicialização é:

tipo_da_variável nome_da_variável [tam1][tam2] ... [tamN] = { lista_de_valores_1,lista_de_valors_2,...,lista_de_valores_N} ;

A lista de valores é composta por valores (do mesmo tipo da variável) separados por vírgula. Os valores devem ser dados na ordem em que serão colocados na matriz. Abaixo vemos alguns exemplos de inicializações de matrizes:

```
float vect [6] = { 1.3, 4.5, 2.7, 4.1, 0.0, 100.1 };
int matrx [3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
char str [10] = { 'J', 'o', 'a', 'o', '\0' };
char str [10] = "Joao";
char str_vect [3][10] = { "Joao", "Maria", "Jose" };
```

O primeiro demonstra inicialização de vetores. O segundo exemplo demonstra a inicialização de matrizes multidimensionais, onde matrx está sendo inicializada com 1, 2, 3 e 4 em sua primeira linha, 5, 6, 7 e 8 na segunda linha e 9, 10, 11 e 12 na última linha. No terceiro exemplo vemos como inicializar uma string e, no quarto exemplo, um modo mais compacto de inicializar uma string. O quinto exemplo combina as duas técnicas para inicializar um vetor de strings. Repare que devemos incluir o ; no final da inicialização.

Inicialização sem especificação de tamanho

Podemos, em alguns casos, inicializar matrizes das quais não sabemos o tamanho a priori. O compilador C vai, neste caso verificar o tamanho do que você declarou e considerar como sendo o tamanho da matriz. Isto ocorre na hora da compilação e não poderá mais ser mudado durante o programa, sendo muito útil, por exemplo, quando vamos inicializar uma string e não queremos contar quantos caracteres serão necessários. Alguns exemplos:

```
char mess [] = "Linguagem C: flexibilidade e poder.";
int matrx [][2] = { 1,2,2,4,3,6,4,8,5,10 };
```

No primeiro exemplo, a string mess terá tamanho 36. Repare que o artifício para realizar a inicialização sem especificação de tamanho é não especificar o tamanho! No segundo exemplo o valor não especificado será 5.

Exercício

O que imprime o programa a seguir? Tente entendê-lo e responder. A seguir, execute-o e comprove o resultado.

```
# include <stdio.h>
int main()
{
    int t, i, M[3][4];
    for (t=0; t<3; ++t)
        for (i=0; i<4; ++i)
            M[t][i] = (t*4)+i+1;
    for (t=0; t<3; ++t)
    {
        for (i=0; i<4; ++i)
            printf ("%3d ", M[t][i]);
        printf ("\n");
    }
    return(0);
}
```

3.4. Registros

São conjuntos de dados logicamente relacionados, mas de tipos diferentes (numérico, literal e lógico).

O conceito de registro visa facilitar o agrupamento de variáveis que não são do mesmo tipo, mas que aguardam estreita relação lógica.

Registros correspondem a conjuntos de posições de memória conhecidas por um mesmo nome e individualizados por identificadores associados a cada conjunto de posições.

O registro é um caso mais geral de variáveis composta na qual os elementos do conjunto não precisam ser, necessariamente, homogêneas ou do mesmo tipo.

Na linguagem C os Registros são chamados de Estruturas (struct).

3.4.1 Estruturas

Uma estrutura agrupa várias variáveis numa só. Funciona como uma ficha pessoal que tenha nome, telefone e endereço. A ficha seria uma estrutura. A estrutura, então, serve para agrupar um conjunto de dados não similares, formando um novo tipo de dados.

Para se criar uma estrutura usa-se o comando struct. Sua forma geral é:

```
struct nome_do_tipo_da_estrutura
{
    tipo_1 nome_1;
    tipo_2 nome_2;
    ...
    tipo_n nome_n;
} variáveis_estrutura;
```

O nome_do_tipo_da_estrutura é o nome para a estrutura. As variáveis_estrutura são opcionais e seriam nomes de variáveis que o usuário já estaria declarando e que seriam do tipo nome_do_tipo_da_estrutura. Um primeiro exemplo:

```
struct est{
int i;
float f;
} a, b;
```

Neste caso, est é uma estrutura com dois campos, i e f. Foram também declaradas duas variáveis, a e b que são do tipo da estrutura, isto é, a possui os campos i e f, o mesmo acontecendo com b.

Vamos criar uma estrutura de endereço:

```
struct tipo_endereco
{
    char rua [50];
    int numero;
    char bairro [20];
    char cidade [30];
    char sigla_estado [3];
    long int CEP;
```

```
};
```

Vamos agora criar uma estrutura chamada ficha_pessoal com os dados pessoais de uma pessoa:

```
struct ficha_pessoal
{
    char nome [50];
    long int telefone;
    struct tipo_endereco endereco;
};
```

Vemos, pelos exemplos acima, que uma estrutura pode fazer parte de outra (a struct tipo_endereco é usada pela struct ficha_pessoal).

Vamos agora utilizar as estruturas declaradas na seção anterior para escrever um programa que preencha uma ficha.

```
#include <stdio.h>
#include <string.h>
struct tipo_endereco
{
    char rua [50];
    int numero;
    char bairro [20];
    char cidade [30];
    char sigla_estado [3];
    long int CEP;
};
struct ficha_pessoal
{
    char nome [50];
    long int telefone;
    struct tipo_endereco endereco;
};
main (void)
{
    struct ficha_pessoal ficha;
    strcpy (ficha.nome,"Luiz Osvaldo Silva");
    ficha.telefone=4921234;

    strcpy (ficha.endereco.rua,"Rua das Flores");
    ficha.endereco.numero=10;
    strcpy (ficha.endereco.bairro,"Cidade Velha");
    strcpy (ficha.endereco.cidade,"Belo Horizonte");
    strcpy (ficha.endereco.sigla_estado,"MG");
    ficha.endereco.CEP=31340230;

    printf(ficha.nome);
    printf("tel.: %d \n",ficha.telefone);
    printf("N.: %d \n",ficha.endereco.numero);
    printf(ficha.endereco.rua);
    printf(ficha.endereco.bairro);
```

```
    printf(ficha.endereco.cidade);
    printf(ficha.endereco.sigla_estado);

    return 0;
}
```

O programa declara uma variável ficha do tipo **ficha_pessoal** e preenche os seus dados. O exemplo mostra como podemos acessar um elemento de uma estrutura: basta usar o ponto (.). Assim, para acessar o campo telefone de ficha, escrevemos:

```
ficha.telefone = 4921234;
```

Como a struct ficha pessoal possui um campo, endereco, que também é uma struct, podemos fazer acesso aos campos desta struct interna da seguinte maneira:

```
ficha.endereco.numero = 10;
ficha.endereco.CEP= 31340230;
```

Desta forma, estamos acessando, primeiramente, o campo endereco da struct ficha e, dentro deste campo, estamos acessando o campo numero e o campo CEP.

Matrizes de estruturas

Um estrutura é como qualquer outro tipo de dado no C. Podemos, portanto, criar matrizes de estruturas. Vamos ver como ficaria a declaração de um vetor de 100 fichas pessoais:

```
struct ficha_pessoal fichas [100];
```

Poderíamos então acessar a segunda letra da sigla de estado da décima terceira ficha fazendo:

```
fichas[12].endereco.sigla_estado[1];
```

Atribuindo

Podemos atribuir duas estruturas que sejam do mesmo tipo. O C irá, neste caso, copiar uma estrutura, campo por campo, na outra. Veja o programa abaixo:

```
struct est1 {
    int i;
    float f;
};
void main()
{
    struct est1 primeira, segunda; /* Declara primeira e segunda como
                                      structs do tipo est1 */
    primeira.i = 10;
    primeira.f = 3.1415;
    segunda = primeira; /* A segunda struct é agora igual a primeira */
    printf(" Os valores armazenasdos na segunda struct sao : %d e %f ",
            segunda.i , segunda.f);
}
```

São declaradas duas estruturas do tipo est1, uma chamada primeira e outra chamada segunda. Atribuem-se valores aos dois campos da struct primeira.

Os valores de primeira são copiados em segunda apenas com a expressão de atribuição:

segunda = primeira;

Todos os campos de primeira serão copiados na segunda. Note que isto é diferente do que acontecia em vetores, onde, para fazer a cópia dos elementos de um vetor em outro, tínhamos que copiar elemento por elemento do vetor. Nas structs é muito mais fácil!

Passando para funções

No exemplo apresentado no ítem usando, vimos o seguinte comando:

strcpy (ficha.nome,"Luiz Osvaldo Silva");

Neste comando um elemento de uma estrutura é passado para uma função. Este tipo de operação pode ser feita sem maiores considerações.

Podemos também passar para uma função uma estrutura inteira. Veja a seguinte função:

```
void PreencheFicha (struct ficha_pessoal ficha)
{
...
}
```

Como vemos acima é fácil passar a estrutura como um todo para a função. Devemos observar que, como em qualquer outra função no C, a passagem da estrutura é feita por valor. A estrutura que está sendo passada, vai ser copiada, campo por campo, em uma variável local da função PreencheFicha. Isto significa que alterações na estrutura dentro da função não terão efeito na variável fora da função. Mais uma vez podemos contornar este pormenor usando ponteiros e passando para a função um ponteiro para a estrutura.

Exercício

1) Escreva um programa fazendo o uso de struct's. Você deverá criar uma struct chamada Ponto, contendo apenas a posição x e y (inteiros) do ponto. Declare 2 pontos, leia a posição (coordenadas x e y) de cada um e calcule a distância entre eles. Apresente no final a distância entre os dois pontos.

2) Seja a seguinte struct que é utilizada para descrever os produtos que estão no estoque de uma loja :

```
struct Produto {
    char nome[30]; /* Nome do produto */
    int codigo; /* Codigo do produto */
    double preco; /* Preco do produto */
};
```

a) Escreva uma instrução que declare duas variáveis(v1 e v2) de Produto;

b) Atribua os valores de v1, "Pe de Moleque", 13205 e R$0,20 e v2 "Cocada Baiana", 15202 e R$0,50;

c) Escreva as instruções para imprimir os campos que foram atribuídos na letra b;

3.5. Arquivos

Para ler ou gravar um arquivo, você deve informar ao sistema a sua intenção de fazer isto, um processo chamado abrir um arquivo. Se você vai gravar em um arquivo, pode ser necessário criá-lo ou desconsiderar seu conteúdo anterior. O sistema verifica seu direito de fazer isto (O arquivo existe? Você tem permissão para acessá-lo?), e se tudo estiver bem, ele retorna ao programa um pequeno inteiro não negativo chamado descritor de arquivo. Sempre que uma entrada ou saída for feita no arquivo, o descritor de arquivo é usado ao invés do nome para identificá-lo. Toda informação sobre um arquivo aberto é mantida pelo sistema; o programa usuário só faz referência ao arquivo por meio do seu descritor.

Existem regras simples para manuseio de um arquivo. Antes que possa ser lido ou gravado, um arquivo deve ser aberto pela função de biblioteca fopen(). fopen() usa um nome externo, faz alguma tarefa inicial e negociação como o sistema operacional(sendo que nos detalhes não estamos interessados), e retorna a ser usado em leituras ou escritas subseqüentes no arquivo.

Este apontador, chamado apontador de arquivo, aponta para uma estrutura que contém informações sobre o arquivo, como o local de um buffer, a posição do caractere corrente no buffer, se o arquivo está sendo lido ou gravado, e se foram encontrados erros ou o fim de arquivo. Os usuários não precisam saber os detalhes, pois as definições obtidas de <stdio.h> incluem uma declaração de estrutura chamada FILE. A única declaração necessária para um apontador de arquivo é exemplificada por

FILE *fp;

FILE *fopen(char *nome,char *modo);

Então, fp é um apontador para um FILE(arquivo), e fopen retorna um apontador para um FILE, apontador será visto em PD II. O bserve que FILE é um nome de tipo, como int.

fp = fopen(nome,modo);

O primeiro argumento de fopen() é uma cadeia de caracteres contendo o nome do arquivo. O segundo argumento é o modo, também uma cadeia de caracteres, que indica como se pretende usar o arquivo. Os modos permitidos incluem leitura (“r”), gravação(“w”) e anexação(“a”).

modo	descrição
"r"	Abre uma arquivo somente para leitura
"w"	Cria um arquivo para gravação
"a"	Anexar; Cria uma arquivo para escrever no final do arquivo ou cria um arquivo para gravação

modo	descrição
"r+"	Abre uma arquivo existente para leitura e escrita
"w+"	Cria um novo arquivo para leitura e escrita. Se o arquivo existir escreve sobre o arquivo existente.
"a+"	Abre para anexar; abre para escrever ou ler no final do arquivo, ou cria se ele não existir

Se um arquivo que não existe for aberto para escrita ou anexação, ele é criado, se for possível. Abrir um arquivo existente para gravação faz com que o conteúdo antigo seja desconsiderado, enquanto abri-lo para anexação o preserva. Tentar ler um arquivo que não existe é um erro, e também pode haver outras causas de erro, como tentar ler um arquivo quando não existe permissão. Se houver um erro, fopen() retornará um NULL.

O próximo passo necessário é uma forma de ler ou gravar no arquivo um vez aberto. Há diversas possibilidades, das quais getc() putc() são as mais simples. getc() retorna o próximo caractere de um arquivo; ele precisa do apontador de arquivo para que saiba qual o arquivo usar.

int getc(FILE *fp)

getc() retorna o próximo caractere do fluxo referenciado por fp; ele retorna EOF para fim de arquivo ou erro.

putc() é uma função de saída:

int putc(int c, FILE *fp)

putc() escreve o caractere c no arquivo fp e retorna o caractere escrito, ou EOF se houver erro.

Para a entrada ou saída formatada de arquivos, as funções fscanf() e fprintf() podem ser usadas. Elas são idênticas a scanf() e printf(), exceto que o primeiro argumento é um apontador de arquivo que especifica o arquivo a ser lido ou gravado; a cadeia de formato é o segundo.

int fscanf(FILE *fp, char *formato)

int fprintf(FILE *fp, char *formato)

obs.: Como foi visto, a melhor forma para se ler uma string é usar a função fgets().

char *fgets(char *linha, int maxlin,FILE *fp)

fgets() lê a próxima linha de entrada(incluindo o caractere de nova linha) do arquivo fp no vetor de caracteres linha; no máximo maxlin-1 caracteres serão lidos. A linha resultante é terminada com '\0'. Caso erro, fgets() retorna NULL.

Também, para saída pode ser usado a função fputs() que grava uma cadeia(que não precisa conter caractere de nova-linha) em um arquivo:

int fputs(char *line,FILE *fp)

Ela retorna EOF se houver um erro, e zero em caso contrário.

Os apontadores de arquivo stdin e stdou são objetos do tipo FILE*. Eles são constantes, entretanto, e não variáveis; não tente atribuir nada a eles.

A função fclose() é o inverso de fopen(), ela encerra a conexão entre o apontador de arquivo e o nome externo que foi estabelecido pro fopen(), liberando o apontador de arquivo para outro arquivo. Como a maioria dos sistemas operacionais tem um limite para o número de arquivos que um programa pode abrir simultaneamente, é uma boa idéia liberar os apontadores quando não são mais necessários. Há ainda outro motivo para usar fclose() em um arquivo de saída – eles esvazia o buffer em que putc() esta coletando a saída. fclose() é chamado automaticamente para cada arquivo aberto quando um programa termina normalmente.

4. Modularização

A ausência de uma metodologia para a construção de programas conduz a um software geralmente cheio de erros e com alto custo de desenvolvimento que, conseqüentemente exige um custo elevado para sua correção e manutenção futuras.

A programação estruturada é hoje o resultado de uma série de estudos e propostas de disciplinas e metodologias para o desenvolvimento de software. Conceitos associados como técnicas de refinamento sucessivo e modularização de programas integram o ferramental para a elaboração de programas visando, principalmente, os aspectos de confiabilidade, legibilidade, manutenibilidade e flexibilidade.

Pode-se reunir as idéias da programação estruturada em três grupos:

- Desenvolvimento de algoritmo por fases ou refinamento sucessivo;
- Uso de um número muito limitado de estruturas de controle;
- Transformação de certos refinamentos sucessivos em módulos.

O desenvolvimento de algoritmos por refinamento sucessivos e o uso de número limitado de estruturas de controle são técnicas largamente usadas por esta apostila.

Quando se desenvolve um algoritmo através de refinamento sucessivo, faz-se uma opção pela divisão do algoritmo; este procedimento conduz à modularização da solução do problema.

Um módulo é, então, um grupo de comandos, constituindo um trecho de algoritmo, com uma função bem definida e o mais independente possível em relação ao resto do algoritmo.

4.1. Função

Em C, uma função é equivalente a uma sub-rotina ou função em Fortran, ou uma procedure ou função em Pascal. Uma função fornece um meio conveniente de modularizar alguma computação, que pode ser depois utilizada sem a preocupação com detalhes de implementação. Com funções projetadas adequadamente, é possível ignorar como uma tarefa e feita; saber o que é feito é suficiente. O C torna o uso de funções fácil, conveniente e eficiente; você vai ver, freqüentemente, uma função com poucas linhas ativada uma vez, só porque ela esclarece alguma parte do código.

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

Abaixo o tipo mais simples de função:

```
#include <stdio.h>
int mensagem () /* Funcao simples: so imprime Ola! */
{
    printf ("Ola! ");
    return(0);
}
```

```
int main ()
{
    mensagem();
    printf ("Eu estou vivo!\n");
    return(0);
}
```

Este programa terá o mesmo resultado que o primeiro exemplo da seção anterior. O que ele faz é definir uma função **mensagem()** que coloca uma string na tela e retorna 0. Esta função é chamada a partir de **main()** , que, como já vimos, também é uma função. A diferença fundamental entre main e as demais funções do problema é que main é uma função especial, cujo diferencial é o fato de ser a primeira função a ser executada em um programa.

4.1.1 Argumentos

Argumentos são as entradas que a função recebe. É através dos argumentos que passamos *parâmetros* para a função. Já vimos funções com argumentos. As funções printf() e scanf() são funções que recebem argumentos.

Vamos ver um outro exemplo simples de função com argumentos:

```
#include <stdio.h>
int square (int x) /* Calcula o quadrado de x */
{
    printf ("O quadrado e %d",(x*x));
    return(0);
}
int main ()
{
    int num;
    printf ("Entre com um numero: ");
    scanf ("%d",&num);
    printf ("\n\n");
    square(num);
    return(0);
}
```

Na definição de **square()** dizemos que a função receberá um argumento inteiro **x**. Quando fazemos a chamada à função, o inteiro **num** é passado como argumento. Há alguns pontos a observar. Em primeiro lugar temos de satisfazer aos requisitos da função quanto ao tipo e à quantidade de argumentos quando a chamamos. Apesar de existirem algumas conversões de tipo, que o C faz automaticamente, é importante ficar atento. Em segundo lugar, não é importante o nome da variável que se passa como argumento, ou seja, a variável **num**, ao ser passada como argumento para **square()** é copiada para a variável **x**. Dentro de **square()** trabalha-se apenas com **x**. Se mudarmos o valor de **x** dentro de **square()** o valor de **num** na função **main()** permanece inalterado. Vamos dar um exemplo de função de mais de uma variável. Repare

que, neste caso, os argumentos são separados por vírgula e que deve-se explicitar o tipo de cada um dos argumentos, um a um. Note, também, que os argumentos passados para a função não necessitam ser todos variáveis porque mesmo sendo constantes serão copiados para a variável de entrada da função.

```
#include <stdio.h>
int mult (float a, float b,float c)  /* Multiplica 3 numeros */
{
    printf ("%f",a*b*c);
    return(0);
}
int main ()
{
    float x,y;
    x=23.5;
    y=12.9;
    mult (x,y,3.87);
    return(0);
}
```

4.1.2 Retornando valores

Muitas vezes é necessário fazer com que uma função retorne um valor. As funções que vimos até aqui estavam retornando o número 0. Podemos especificar um tipo de retorno indicando-o antes do nome da função. Mas para dizer ao C o que vamos retornar precisamos da palavra reservada return. Sabendo disto fica fácil fazer uma função para multiplicar dois inteiros e que retorna o resultado da multiplicação. Veja:

```
#include <stdio.h>

int prod (int x,int y)
{
    return (x*y);
}
int main ()
{
    int saida;
    saida=prod (12,7);
    printf ("A saida e: %d\n",saida);
    return(0);
}
```

Veja que, como prod retorna o valor de 12 multiplicado por 7, este valor pode ser usado em uma expressão qualquer. No programa fizemos a atribuição deste resultado à variável saida, que posteriormente foi impressa usando o printf.

Uma observação adicional: se não especificarmos o tipo de retorno de uma função, o compilador C automaticamente suporá que este tipo é inteiro. Porém, não é uma boa prática não se especificar o valor de retorno e, neste curso, este valor será sempre especificado.

Com relação à função main, o retorno sempre será inteiro. Normalmente faremos a função main retornar um zero quando ela é executada sem qualquer tipo de erro.

Mais um exemplo de função, que agora recebe dois floats e também retorna um float::

```
#include <stdio.h>
float prod (float x,float y)
{
    return (x*y);
}
int main ()
{
    float saida;
    saida=prod (45.2,0.0067);
    printf ("A saida e: %f\n",saida);
    return(0);
}
```

4.1.3 Forma geral

Apresentamos aqui a forma geral de uma função:

```
tipo_de_retorno nome_da_função (lista_de_argumentos)
{
    código_da_função
}
```

Exercício:
Escreva uma função que some dois inteiros e retorne o valor da soma.

4.2. Procedimentos

Procedimentos são blocos de instruções que realizam tarefas específicas. O código de um procedimento é carregado uma vez e pode ser executado quantas vezes for necessário. Dessa maneira, os programas tendem a ficar menores e mais organizados, uma vez que o problema pode ser subdividido em pequenas tarefas.

Em C, não existe procedimentos só existem funções, para implementar um procedimento em C, você deve criar uma função que não retorna nenhum valor. Para não retornar nenhum valor você deve usar o tipo void no retorno d função.

```
void nome_da_função (lista_de_argumentos)
{
    código_da_função
}
```

Obs.: void significa que a função não retorna nenhum tipo de dados.

4.3. Exemplo Prático

Neste item iremos falar sobre um exemplo de um algoritmo para calcular o salário líquido de um Empregado, que possui um salário base de R$ 1000,00.

Para se elaborar um algoritmo para calcular o salário líquido de um empregado é preciso seguir as etapas:

1. Leitura dos dados do Funcionário;
2. Determinar o Salário;
3. Imprimir Salário.

Abaixo é escrito um programa em C, que realiza estas etapas:

```
/* Este programa calcula o salário de um funcionário */

#include<stdio.h>

/* ==== Declaração de variáveis globais ==== */
typedef struct
{
        char nome[50];
        int matricula;
        char funcao[30];
}funcionario;


/* ==== Declaração de funções ==== */

funcionario leitura_dados() /* Leitura de dados */
{
   funcionario n;
        printf("\n\n\tEntre com o nome do funcionario: ");
        fgets(n.nome,49,stdin);
        fflush(stdin);
        printf("\tEntre com a Matricula: ");
        scanf("%d",&n.matricula);
        fflush(stdin);
        printf("\tEntre com a funcao do funcionario: ");
        fgets(n.funcao,29,stdin);
        fflush(stdin);
        return n;
};
```

```
void imprimir_dados(funcionario n,float sal) /* Imprimir Dados */
{
   printf("\n\tNome: %s",n.nome);
        printf("\tMat.: %d\n",n.matricula);
        printf("\tCargo: %s\n",n.funcao);
        printf("\tSalario Liquido: R$ %6.2f\n",sal);
};

float sal_brut(float base) /* Determinar Salário Bruto */
{
   float aux_trans,aux_alim,g1,g2;
   aux_trans = base*0.06;
   aux_alim = base*0.1;
   g1 = base*0.2;
   g2 = base * 0.1;
   return (base + aux_trans + aux_alim + g1 + g2);
};

float deducoes(float bruto) /* Determinar Deduções */
{
   float ir,t_ir,inss;
   if(bruto < 1500)
           t_ir = 0;
   else
           if (bruto < 2500)
               t_ir = 0.1;
           else
                       t_ir = 0.15;
   inss = bruto * 0.11;
   ir = bruto * t_ir;
   return (inss + ir);
}

float sal_liq(float base)  /* Determinar Salário Líquido */
{
   float sal_b, ded;
       sal_b = sal_brut(base);
   ded = deducoes(sal_b);
   return (sal_b - ded);
};
```

```
main()
{

    /* ==== Declaração de Variáveis ==== */
    funcionario func;
    float sal_liquido;
    float sal_base = 1000.0;

    /* ==== Corpo do Programa ==== */
    func = leitura_dados();              /* leitura de dados* /
    sal_liquido = sal_liq(sal_base);     /* calculo do salário líquido* /
    imprimir_dados(func,sal_liquido);    /* imprimir dados e salário
                                            líquido* /
    system("PAUSE");
    return 0;
}
```

Nessa operação esta sendo realizado um refinamento, nesse refinamento não houve preocupação de como se processa o cálculo do Salário Bruto e das deduções, que serão executadas pela função sal_liq(). Essas constituem funções bem definidas no algoritmo e que serão executadas por funções específicas. Nesta fase do projeto do algoritmo, pode-se, então, optar pela elaboração de funções para o cálculo do salário bruto e cálculo das deduções.

A descrição estrutural da modularização pode ser feita através do diagrama hierárquico, como a seguir:

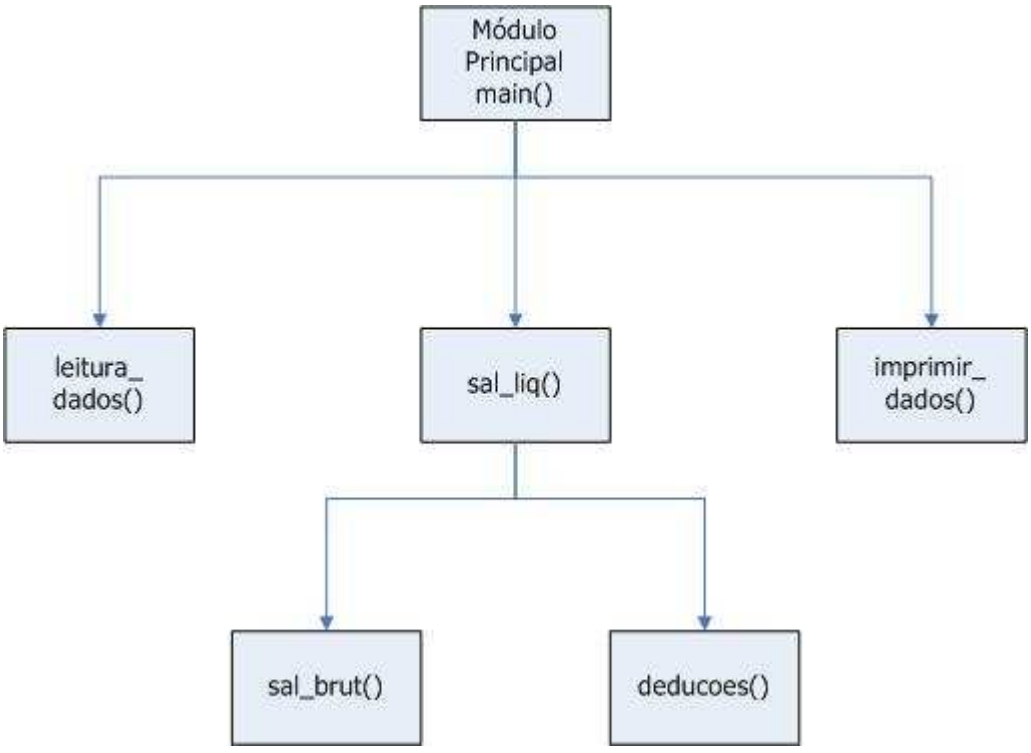


Figura 4.1 - Diagrama hierárquico

A maneira mais intuitiva de proceder à modularização de problemas é feita definindo-se um módulo principal de controle e módulos específicos para funções do algoritmo. No diagrama anterior, o módulo principal tem a função de receber os dados, escrever os resultados e exercer o controle na execução das funções do algoritmo. A determinação das vantagens e deduções é delegada a módulos específicos.

A experiência recomenda que os módulos de um programa devem ter um tamanho limitado. Módulos muito grandes são difíceis de compreendidos e, em geral, são multifuncionais.

Um outro aspecto importante é a possibilidade de cada módulo poder definir as próprias estruturas de dados, suficientes e necessários apenas para atingir objetivo final do módulo.

Todo módulo é constituído por uma seqüência de comandos que operam sobre um conjunto de objetos, que podem ser globais ou locais.

Objetos globais são entidades que podem ser usadas em módulos internos a outro módulo do algoritmo onde foram declaradas.

Objetos locais são entidades que só podem ser usadas no módulo do algoritmo onde foram declaradas. Estes objetos não possuem qualquer significado fora deste módulo. São exemplos de objetos globais ou locais: variáveis, arquivos, outros módulos etc.

Um módulo pode usar objetos globais ou locais em relação a ele. Porém, não pode usar objetos declarados em módulos que não o abrangem. Isto significa que objetos globais, declarados em módulos mais externos ou mesmo nível do módulo principal, podem ser também utilizados em módulos mais internos.

A comunicação entre módulos deverá ser feita através de vínculos, utilizando-se objetos globais ou transferência de parâmetros.

A decisão pela divisão do algoritmo em módulos traz benefícios tais como:

a) A independência do módulo permite uma manutenção mais simples e evita efeitos colaterais em outros pontos do algoritmo;
b) A elaboração do módulo pode ser feita independentemente e em época diferente do restante do algoritmo.
c) Testes e correções dos módulos podem ser feitos em separado;
d) Um módulo independente pode ser utilizado em outros algoritmos que requeiram o mesmo processamento por ele executado.

4.4. Considerações sobre a modularização de programas

Existem várias vantagens e, naturalmente, algumas desvantagens na modularização de programas. Algumas destas desvantagens são especificas de cada linguagem de programação que se escolha para implementação do algoritmo.

Dentre as vantagens da modularização de programas podem-se citar:

1. Partes comuns a vários programas ou que se repetem dentro de um mesmo programa, quando modularizadas em uma função, são programadas e testadas uma só vez, mesmo que tenham que ser executadas com variáveis diferentes;

2. Podem-se constituir bibliotecas de programas (arquivos.h), isto é, uma coleção de módulos que podem ser usados em diferentes programas sem alteração e mesmo por outros programadores. Por exemplo, as funções pré-definidas das linguagens de programação, como citadas no item anterior, constituem uma biblioteca de módulos;
3. A modularização dos programas permite preservar na sua implementação os refinamentos obtidos durante o desenvolvimento dos algoritmos;
4. Economia de memória do computador, uma vez que o módulo é armazenado uma única vez, mesmo que utilizado em diferentes partes do programa. Permite, também, que, em um determinado instante da execução do programa, estejam na memória principal apenas os módulos necessários à execução desse trecho de programa.

Dentre as desvantagens pode-se citar que existe um acréscimo de tempo na execução de programas constituídos de módulos, devido ao tratamento adicional de ativação do módulo etc.

Conclui-se que, sem a modularização dos programas é uma técnica altamente recomendável, tanto pela eficiência no projeto e desenvolvimento dos mesmos quanto pela quantidade, confiabilidade, confiabilidade e manutenção do produto elaborado.

5. Linguagem de Programação C

5.1. Resolução de Problemas Diversos em Linguagem de Programação C

1) Uma pesquisa sobre algumas características físicas da população de uma determinada região coletou os seguintes dados, referentes a cada habitante, para serem analisados:
    - Nome:
    - Sexo (masculino, feminino):
    - Cor dos olhos (azuis,verdes,castanhos):
    - Cor dos cabelos (louros,castanhos,pretos);
    - Idade em anos;

    Fazer um programa que armazena até 100 desses dados em um arquivo, e tenha alternativas para que possa ler este arquivo, e fazer buscar através do primeiro nome das pessoas.