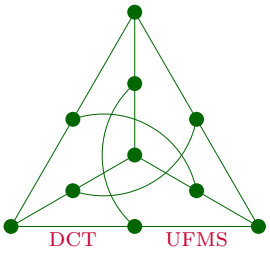
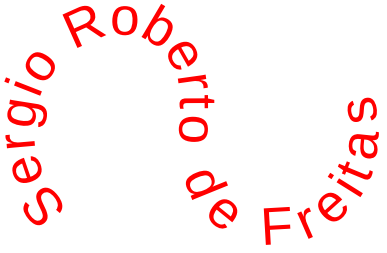
Algoritmos e Estrutura de Dados I

PROF. SERGIO ROBERTO DE FREITAS

ROTEIRO DAS AULAS
DE
ALGORITMOS E ESTRUTURA DE DADOS I

– 2002 –





Departamento de Computação e Estatística
Centro de Ciências Exatas e Tecnologia
Universidade Federal de Mato Grosso do Sul

24 de abril de 2002

Conteúdo

Capítulo 1

História e Evolução dos Computadores e Linguagens

Com o objetivo de entender a história e a evolução dos computadores bem como seu estágio atual devemos de alguma maneira sintetizar o que vem a ser um *Computador*.
Um computador pode ser *definido* sucintamente como:
Equipamento eletrônico capaz de efetuar cálculos e decisões lógicas bilhões de vezes mais rápido que um ser humano.
Isso posto fica evidente que seu principal propósito é auxiliar nas tarefas que são repetitivas e que envolvem grande quantidade de cálculos.
As aplicações mais relevantes do computador utilizam, de algum modo, sua capacidade de *armazenar, acessar e manipular* quantidades de informações que seriam impraticáveis usando uma outra maneira.

1.1 Primórdios da História dos Computadores

É sempre difícil ou porque não dizer quase impossível determinar o ponto inicial de uma história. Esse ponto seria o *marco zero* da história, ou seja, antes dele nada de relevante teria acontecido.
No nosso caso iremos considerar como *marco zero* a criação dos primeiros dispositivos mecânicos para cálculo automático.

1.1.1 As Calculadoras Mecânicas

A primeira calculadora mecânica foi concebida e construída por Wilhem Schickard (1592-1635) que foi professor de astronomia, matemática e línguas

em Tübingen. Muito tempo após sua morte foram encontradas correspondências suas enviadas a Kepler contendo esquemas e descrições da "máquina de Schickard"que podia fazer as operações de adição e subtração de um modo totalmente automático e as de multiplicação e divisão de maneira parcialmente automática.
Numa carta de 1624 Schickard informa a Kepler que havia solicitado a construção de uma dessas máquinas para presente-a-lo mas que infelizmente um incêndio nas oficinas de seu mecânico havia destruído o equipamento já em fase adiantada de execução. Ele lamentava profundamente o ocorrido pois não seria possível, tão cedo, produzir outro.
Esse engenhoso equipamento não pôde ser conhecido na época pois infelizmente Schickard e toda sua família morreram com as epidemias que devastaram a Europa e que foram trazidas pela Guerra dos Trinta Anos.
Esse equipamento foi posteriormente reconstruído, com o auxílio de mestres em mecânica, pelo Baron von Freytag-Löringhoff a partir das informações dadas à Kepler e funcionou como o previsto.

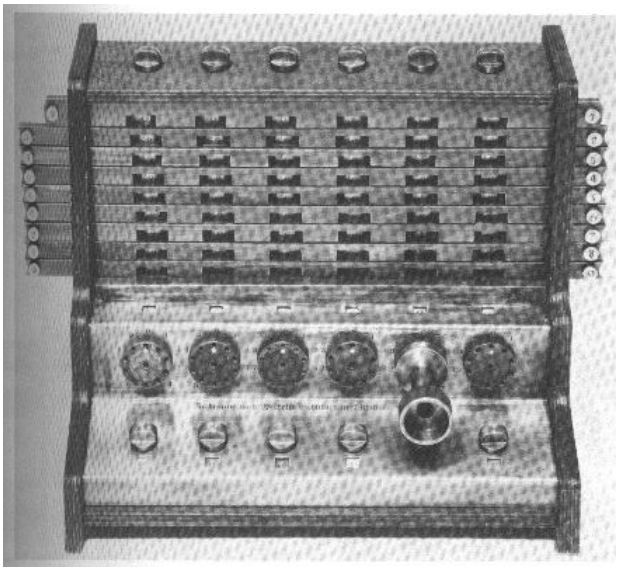


Figura 1.1: Máquina de Schickard

A próxima grande figura nessa direção foi Blaise Pascal (1623-1662) que com a idade de 20 anos construiu um pequeno e simples equipamento que tornou-se o protótipo de uma série de equipamentos desse tipo construídos na França a partir de então.

Essa máquina, apesar de ser considerada por ele e seus contemporâneos como um feito extraordinário, não deve ser considerada mais avançada que a de Schickard pois podia efetuar apenas as operações de adição e subtração.

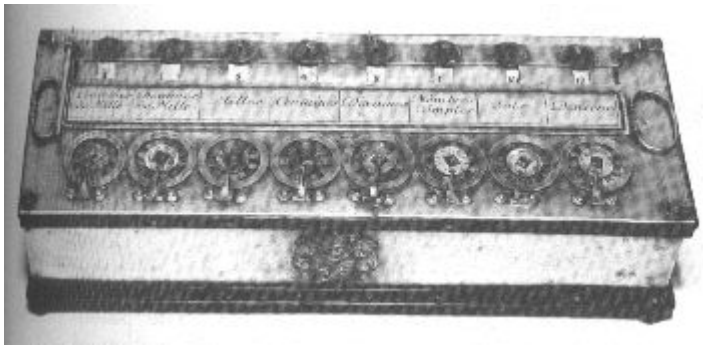


Figura 1.2: Máquina de Pascal

Trinta anos mais tarde Gottfried Wilhem Leibniz (1646-1716) inventou um dispositivo que ficou conhecido como *Leibniz Wheel* que tornou-se até há algum tempo atrás (1970) a peça fundamental das calculadoras mecânicas. Esse dispositivo permitiu-lhe construir uma máquina que suplantou a de Pascal pois não só podia fazer as operações de adição e subtração como também as de multiplicação e divisão.
Leibniz dizia, a bem da verdade, que seu equipamento devia ser entendido como constituído de duas partes distintas trabalhando conjuntamente. A parte que fazia as adições e subtrações nada mais era que a máquina de Pascal, a outra é que efetuava as multiplicações e divisões.
Leibniz tinha a convicção que através de máquinas seria possível liberar o homem das tarefas repetitivas e de simples execução.

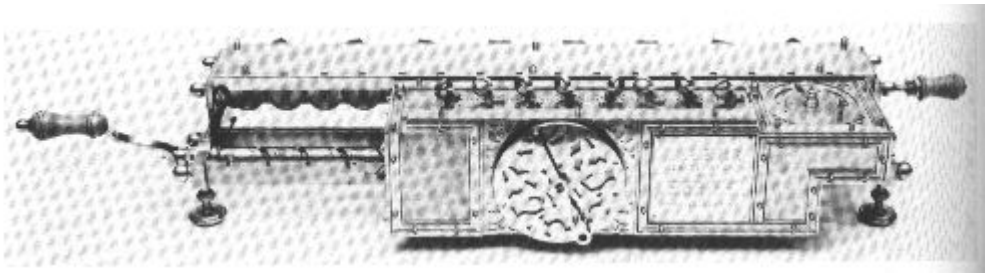


Figura 1.3: Máquina de Leibnitz

1.2 A Máquina de Diferenças

No final do século XVIII houve uma proliferação de *tabelas de dados* de vários tipos.
Essas tabelas tanto podiam ser produzidas por meios matemáticos como: seno, coseno, logaritmos, etc. como por registros de medições físicas: precipitações pluviométricas, densidade em função da altitude, constante gravitacional em diferentes pontos da terra, etc.
Um dos objetivos das tabelas era reduzir o trabalho de cálculo, mas as tabelas produzidas por especialistas continham muitos erros e sua utilização em situações reais, como por exemplo na determinação da longitude com exatidão (fundamental para a navegação intercontinental) era bastante precária e não confiável.
Nestas circunstâncias aparece em cena Charles Babbage (1791-1871) que, concordando inteiramente com as convicções de Leibniz, considerava um trabalho intolerável e de fatigante monotonia a confecção e mais ainda a correção dessas *tabelas de dados e informações*.
Foi dele, indubitávelmente, a idéia de substituir pessoas falíveis (nos cálculos) por máquinas infalíveis.
Seu trabalho *Observations on the Application of Machinery to the Computation of Mathematical Tables* proporcionou-lhe a *Gold Medal* da Royal Astronomical Society - da qual ele foi um dos fundadores - bem como apoio e fundos para o projeto de construção da chamada *Difference Engine*.
Essa máquina era considerada pelo governo de grande valor - particularmente pela Marinha - para a confecção das tabelas náuticas. Dadas as "dificuldades mecânicas"da época e a idéia que ele já concebia para uma máquina mais avançada a *Difference Engine* nunca foi, nesse projeto, totalmente construída.
Considerando que a *Difference Engine* sem dúvida introduziu uma nova idéia básica no campo das computações talvez seja interessante discorrer um pouco mais sobre ela.
A *Difference Engine*, conforme proposta por Babbage, foi concebida para calcular e produzir tabelas de valores de polinômios de grau 6 ou seja funções do tipo $a + bx + cx^2 + dx^3 + ex^4 + fx^5 + gx^6$.
Devemos aqui chamar a atenção para o fato de que isso não representava uma restrição para os propósitos de Babbage pois nessa época já era conhecido o famoso teorema de Karl W. Wierstrass (1815-1897) que afirma que:
Toda função contínua num intervalo pode ser aproximada, com a precisão que se desejar, por um polinômio.

A tabela 1.1 abaixo mostra um exemplo de como era o princípio utilizado pela *Difference Engine* para calcular valores de polinômios somente usando adições e subtrações.

n	n^2+n+41	D_1	D_2
0	41	–	–
1	43	2	–
2	47	4	2
3	53	6	2
4	61	8	2
5	71	10	2

Tabela 1.1: Tabela de Dados da *Difference Engine*

Vamos determinar o valor da expressão n^2+n+41 para $n=6$.
A tabela é construída da seguinte maneira:
A segunda coluna é composta pelo calculo da expressão para as os valores da primeira coluna .
A coluna D_1 é composta pela diferença de valores consecutivos da expressão.
A coluna D_2 é composta pela diferença de valores consecutivos de D_1.
Assim na linha $n=6$ teremos $D_2=2$ (pois D_2 é constante=2), $D_1=10+2=12$ e o valor da expressão $71+12=83$.
Baseados nas idéias de Babage, o sueco Pehr Georg Scheutz (1785-1873) e seu filho Edvard (1821-1881) construíram a máquina de tabular que se tornou um grande sucesso comercial.

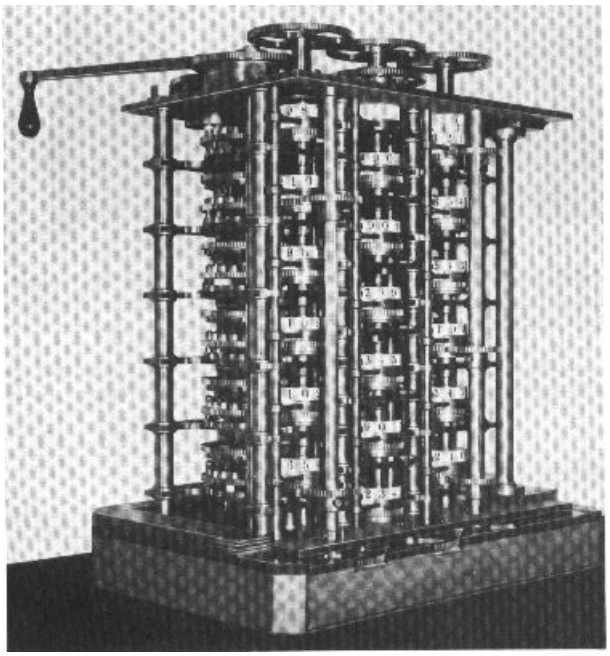


Figura 1.4: Difference Engine

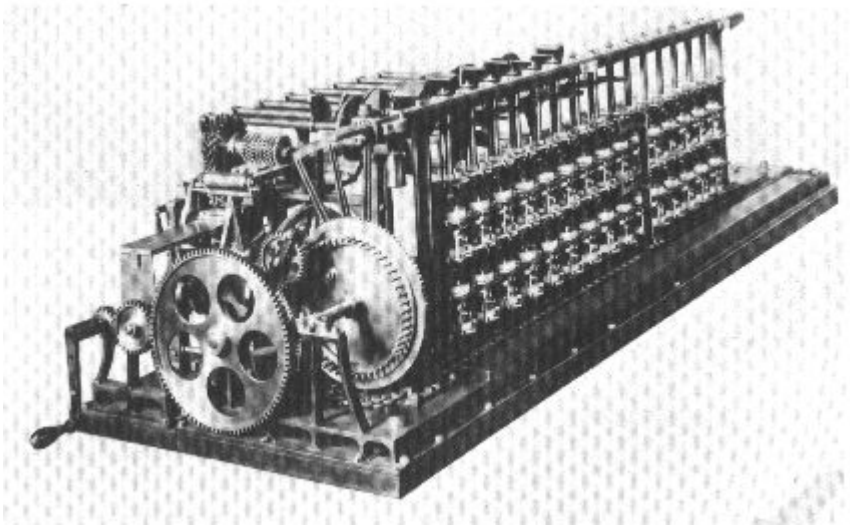


Figura 1.5: Difference Engine Construída por Scheutz

1.3 A Máquina Analítica

Em 1833 Charles Babbage abandonou o projeto da *Difference Engine* pois já vinha desenvolvendo novas idéias e achava que os esforços para o seu término equivaleriam ao desenvolvimento da sua nova máquina a *Analitical Engine*.
O questionamento de Babbage era simples: se era possível construir uma máquina para executar um determinado tipo de cálculo, por que não seria possível construir outra capaz de qualquer tipo de cálculo?
Era a idéia de uma *máquina de cálculo universal* que viria a ser retomada em 1930 por Alan Turing.
Ressalte-se que a *Analitical Engine* estava muito próxima conceitualmente daquilo que hoje é chamado de computador.
A inspiração de Babbage para a concepção da *Analitical Engine* foi o *Tear de Jacquard*.
Joseph-Mariae Jacquard (1752-1834) inventou e construiu um equipamento que automatizou o processo de tecelagem. O trabalho de entrelaçar as fibras para produzir um determinado *padrão* ou *estampa* seguiam um *plano* ou *programa* que podia repetir ou iterar um padrão básico.
Esse equipamento usava uma série de cartões numa ordem e com perfurações em determinadas posições que representavam um determinado padrão.
Em 1812 existiam na França 11000 desses equipamentos.
Na sua concepção a Máquina Analítica poderia seguir conjuntos mutáveis de instruções e, portanto, servir a diferentes funções - mais tarde isso seria denominado software.
Babbage percebeu que para criar estas instruções precisaria de um tipo inteiramente novo de *linguagem* e a imaginou como números, flechas e outros símbolos. Essa linguagem serviria para *programar* a Máquina Analítica, com uma longa série de instruções condicionais, que lhe permitiriam modificar suas ações em resposta a diferentes situações.
Reconhecendo a importância de se terem resultados impressos, Charles procurou que os resultados finais e os intermediários fossem impressos para evitar erros.
Dispositivos de entrada e saída eram assim necessários.
A entrada de dados para a máquina seria feita através de três tipos de cartões: *cartões de números*, com os números das constantes de um problema; *cartões diretivos* para o controle do movimento dos números na máquina; e *cartões de operação* para dirigir a execução das operações tais como adições, subtrações, etc.

Mas o mais genial estava por vir: duas inovações simples mas que produziram um grande impacto.
A primeira era o conceito de "transferência de controle"que permitia à máquina comparar quantidades e, dependendo dos resultados da comparação, desviar para outra instrução ou seqüência de instruções.
A segunda característica era possibilitar que os resultados dos cálculos pudessem alterar outros números e instruções colocadas na máquina, permitindo que o *computador* modificasse seu próprio programa. Nestes temas teve importante participação, Ada Augusta Byron, Lady Lovelace.
Ada Augusta Byron (1851 - 1852) era filha do famoso poeta Lord Byron e foi educada pelo matemático logicista inglês Augustus De Morgan.
Apresentada a Babbage durante a primeira demonstração da Máquina de Diferenças, tornou-se uma importante auxiliar em seu trabalho, sendo sobretudo alguém que compreendeu o alcance das novas invenções e foi uma das suas maiores divulgadoras. Ela é reconhecida como a primeira *programadora*.
Ela dizia que Máquina Analítica *tecia padrões algébricos* do mesmo modo que o *tear de Jacquard* tecia flores e folhas.
De acordo com B.H. Newman, os trabalhos de Ada Byron demonstram que ela teve uma total compreensão dos princípios de um computador programado, com um século de antecedência. Mesmo não estando a máquina construída, Ada escreveu seqüências de instruções tendo descoberto conceitos que seriam largamente utilizados na programação de computadores como subrotinas, loops e saltos.

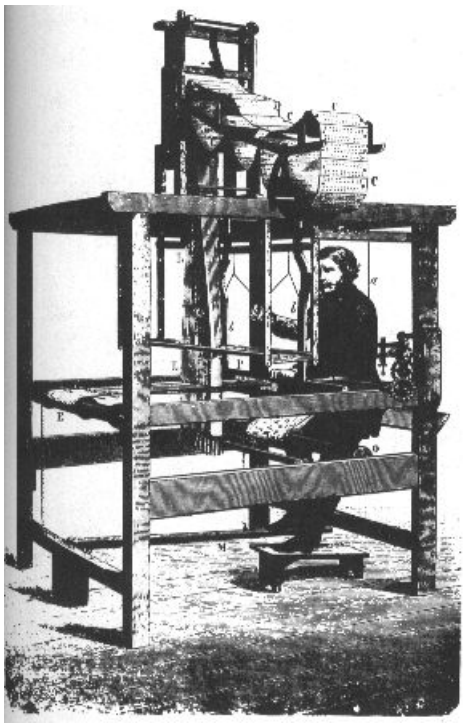


Figura 1.6: Tear de Jacquard

1.4 As Tabuladoras de Billings/Hollerith

O próximo passo importante na História da Computação não está relacionado com tabelas de cálculo de logaritmos ou de novas concepções teóricas de equipamentos.
Os próximos passos no *estado da arte* foram dados por John Shaw Billings(1839-1913) e Herman Hollerith(1860-1929) e as necessidades dos Censos Americanos.
Em 1870 o então Superintendente do Censo estava interessado nas questões relacionadas com a *bioestatística* do censo de 1870.
Esse trabalho interessou intensamente Billings tornando-o proeminente figura do United States Census Office embora nunca tenha sido seu funcionário remunerado.
Billings estava no comando dos trabalhos de *estatísticas vitais* dos censos de 1880 e 1890. Era sua responsabilidade a tabulação e análise dos dados coletados nos censos. Aqui devemos reconhecer a figura do verdadeiro líder

intelectual e pioneiro além de brilhante organizador e administrador. Esse homem, que foi vital nessa etapa, *teve ainda a felicidade de contar em sua equipe com o jovem Hollerith* conforme seu reconhecimento num report de 1880 sobre *Mortality and Vital Statistics*.

Conforme descrito por Dr. Walter F. Wilcox, Billings estava com um companheiro observando centenas de funcionários que estavam engajados na laboriosa tarefa de transferir manualmente items das anotações coletadas através do censo de 1880 para folhas de contabilidade e registros. Penalizado ele disse ao seu companheiro:

Eu penso que existe alguma maneira mecânica de fazer esse trabalho utilizando o princípio dos teares de Jacquard, onde perfurações em determinadas posições de num cartão representariam algum tipo de informação.

A semente havia sido lançada em terra fértil. Seu companheiro era um jovem e talentoso engenheiro chamado Herman Hollerith que depois de convencer-se de que a idéia era praticável e de certificar-se que Billings não tinha interesse comercial nem na idéia nem no seu desenvolvimento passou a dedicar-se à sua implementação.

Seguindo as sugestões de Billings ele usava um sistema de cartões perfurados onde os *buracos* em determinadas posições representavam varias características como: homem ou mulher; branco ou negro; nativo ou estrangeiro, idade etc. A grande vantagem desses cartões é que eles podiam ser preparados em diferentes lugares por diferentes pessoas em diferentes tempos e depois colocadas num *grande pacote* para a tabulação.

Em 1890 Hollerith ganhou a concorrência para o desenvolvimento de um equipamento de processamento de dados para auxiliar o censo americano daquele ano. Para isso foi fundada a *Tabulating Machine Company* que em 1911 tornou-se a *Computer-Tabulating-Recording* e em 1924 torna-se a International Business Machine - IBM sob o comando de Thomas J.Watson.

Na coleta e tabulação das informações do Censo de 1890 foram utilizadas com grande sucesso as máquinas de Hollerith que eram capazes de com grande rapidez *extrair*, *agrupar ordenar* as informações por determinadas características.

Por esse motivo em algumas línguas o *COMPUTADOR* é denominado *ORDENADOR*.

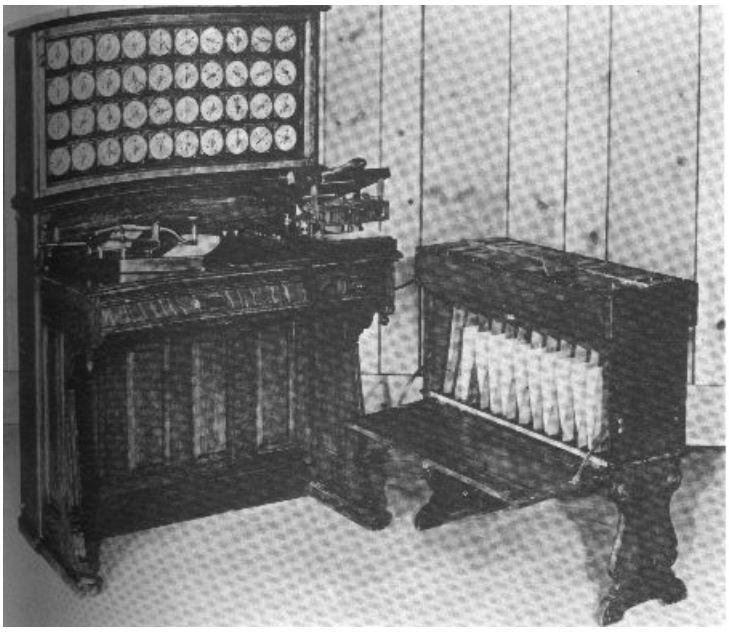


Figura 1.7: Máquina Tabuladora de Hollerith

1.5 A utilização Científica das Tabuladoras

Até 1928 a utilização das tabuladoras e cartões perfurados de Hollerith ficou restrita aos propósitos originais de estatísticas e negócios, mas a partir dai podemos encontrar Comrie usando as tabuladores para fazer tabelas de posições da lua. Este trabalho do neo-zelandez Leslie John Comrie (1893-1950) é da maior importância pois marca a transição da utilização das tabuladoras para *avançados fins científicos*.
Em 1937 Howard H. Aiken delineou quatro pontos que ele considerou como essenciais para que os *computadores digitais eletromecânicos* pudessem servir à ciência de modo mais abrangente:

- Abilidade para manusear números positivos e negativos;
- Possibilidade de utilização das funções matemáticas; (seno, logaritmo, exponencial etc.)
- Automatização total do processo;(não necessidade da intervenção humana)

- Efetuar os procedimentos na seqüência natural dos eventos matemáticos.

1.6 Os Problemas de Balística e a Evolução dos Computadores Eletrônicos

O problema central da balística consiste em determinar a trajetória de um projétil em função da velocidade considerando-se a chamada *função de arrasto* que é determinada pela resistência do ar. Vários físicos e matemáticos se ocuparam desse problema desde Newton. Na metade do século dezenove Francis Bashforth descreveu um modelo preciso para esse fim. Usando suas idéias vários *balicistas* determinaram dados realisticos na determinação da função de arrasto e uma comissão trabalhando de 1880 até 1900 em Gâvre na França sistematizou todos os resultados e produziu a conhecida *Função de Gâvre*. No modelo proposto por Bashforth havia necessidade de serem feitas várias simplificações para facilitar os cálculos. Isso tornava as aproximações de validade duvidosa para diversos tipos de projéteis e esse fato ficou comprovado na I Grande Guerra. Uma simplificação era, por exemplo, considerar a densidade do ar constante para qualquer altitude. Quando a Marinha Alemã construiu seus poderosos canhões eles perceberam que seus disparos alcançavam o dobro da distância prevista pelos cálculos. Essa descoberta permitiu aos balicistas alemães produzir o famoso *Big Bertha* que podia atingir distâncias incríveis. Esse canhão de 23cm de calibre usando um propelente preparado pelo famoso químico Fritz Haber(1861-1924) podia atingir a distância de 145 Km, tendo sido utilizado para bombardear Paris. O advento da Segunda Guerra Mundial provocou um impulso sem precedentes no desenvolvimento da tecnologia dos computadores.
A IBM, em associação com a Marinha Americana, investiu no plano audacioso de um jovem matemático da Universidade de Harvard, chamado Howard Aiken, cujo projeto visava construir um computador programável para todos os fins, ao estilo da *Analitical Engine* de Babbage. A máquina, que recebeu o nome de MARK-I, foi concluída em 1944. Era baseada em um sistema decimal, contrariando as tendências da época, e possuía entrada de dados baseada em cartões perfurados, memória principal e unidade aritmética de controle e saída. Manipulava números de até 23 dígitos. Mesmo antes de ser construído, o MARK-I era obsoleto, por ser um computador ainda baseado em relés (computador eletro-mecânico), entretanto ele abriu caminho para uma nova era de parcerias, principalmente com a área militar. Aiken chegou a construir o MARK-II em 1947, também baseado em relés e em 1949

o MARK-III, já com sistema de programa armazenado.
Em 1941, Konrad Zuse completou um computador operacional, o Z3: um dispositivo controlado por programa e baseado no sistema binário. O Z3 era muito menor que o MARK-I e de construção muito mais barata. Tanto o Z3 como o Z4, seu sucessor, eram usados para resolver problemas de engenharia de aeronaves e de projetos de mísseis. Foi destruído por um bombardeio à Berlim durante a II Grande Guerra.
No final de 1943, na Inglaterra, foram colocadas em operação uma série de máquinas, o Colossus. Em vez de relés, cada uma das novas máquinas usava 2000 válvulas eletrônicas. A máquina foi utilizada pelos ingleses para decifrar códigos utilizados pelos alemães na guerra. O Colossus foi o primeiro computador eletrônico.
Na Escola Moore de Engenharia Elétrica (EUA), foi construido o ENIAC (Eletronic Numerical Integrator and Calculator), que entrou em funcionamento em 1946. Utilizava válvulas eletrônicas e os números eram manipulados na forma decimal. O ENIAC nasceu da necessidade de resolver problemas balísticos, porém quando concluído, a guerra havia acabado e ele mostrou-se capaz de executar diversas tarefas.
A principal desvantagem do ENIAC era a dificuldade de mudar suas instruções ou programas. A máquina só continha memória interna suficiente para manipular os números envolvidos na computação que estava executando. Isso significava que os programas tinham de ser instalados com fios dentro do complexo conjunto de circuitos.
Em 1944, John Von Newmann, matemático húngaro naturalizado americano, desenvolveu a idéia de programa interno e descreveu o fundamento teórico da construção de um computador eletrônico denominado *Modelo de Von Newmann*. A idéia de Newmann era o existência simultânea de dados e instruções no computador e a possibilidade do computador ser programado. A partir dessa idéia de Newmann, construiu-se o EDVAC, sucessor do ENIAC, em 1952.
O EDVAC (Eletronic Discrete Variable Automatic Computer) foi planejado para acelerar o trabalho armazenando tanto programas quanto dados em sua expansão de memória interna. As instruções eram armazenadas eletronicamente em um tubo de mercúrio. Cristais dentro do tubo geravam pulsos eletrônicos que se refletiam para frente e para trás, tão lentamente que podiam, de fato, reter a informação por um processo semelhante àquele pelo qual um desfiladeiro retém um eco. Outro avanço: o EDVAC podia codificar as informações em forma binária em vez de decimal, o que reduzia substancialmente o número de válvulas necessárias.
Dois anos antes que ficasse pronto o EDVAC, o cientista inglês Maurice

Wilkens, baseado na descrição do armazenamento de programa, do projeto EDVAC, construiu o primeiro computador operacional em larga escala de programa armazenado do mundo, o EDSAC (Eletronic Delay Storage Automatic Calculator).
Em 1951 foi construído o primeiro computador destinado ao uso comercial, o UNIVAC-I (Universal Automatic Computer), uma máquina eletrônica de programa armazenado que recebia instruções de uma fita magnética de alta velocidade em vez de cartões perfurados. No ano seguinte foram construídos os computadores MANIAC-I (Mathematical Analyser Numerator, Integrator and Computer), MANIAC-II, UNICAC-II (sendo este último com memórias de núcleos de ferrite). Com o surgimento destas máquinas acaba a pré-história da informática.
Desde o surgimento do UNIVAC-I como o primeiro computador comercial, até hoje, quase todas as transformações no desenvolvimento de computadores foram impulsionadas por descobertas e/ou avanços na área da eletrônica. Tudo começou com a válvula a vácuo e a construção de dispositivos lógicos biestáveis (flip-flop). Um biestável é um dispositivo capaz de ter dois estados estáveis e de comutar de um para o outro conforme lhe seja ordenado.
Os avanços da física do estado sólido provocaram a grande evolução na história dos computadores. Esses progressos podem ser resumidos em:

1. Invenção da válvula a vácuo, que foi utilizada como elemento de controle para integrar dispositivos biestáveis, em 1904.

2. Descoberta, na década de 50, dos semicondutores, além do surgimento do diodo e do transistor. Esse último, inventado por Walter Brattain e John Barden nos laboratórios Bell em 1945. O transistor a princípio era feito de germânio, mas em 1954, aperfeiçoou-se um transistor feito de silício, o que baixou consideravelmente seu preço, popularizando sua utilização. O primeiro computador transistorizado foi o TX-0 (Transistorized eXperimental computer 0), construído no MIT (Instituto de Tecnologia de Massachussets). O transistor substitui a válvula e permitiu a diminuição no tamanho dos computadores e uma maior confiabilidade nos equipamentos.

3. Baseados no transistor, foram construídos circuitos capazes de realizar funções lógicas, como portas lógicas e os circuitos derivados.

4. Criação dos Circuitos Integrados, por volta de 1958. Com a evolução dos transistores, os circuitos integrados surgiram da necessidade cada vez maior de miniaturização e economia de custo dos circuitos eletrônicos. Consistiam no tratamento físico-químico sobre uma

película de silício, permitindo configurar diferentes circuitos e portas lógicas. Com isso, teve início a ciência do projeto lógico de circuitos com baixa integração ou SSI (Short Scale Integration) que permitia integrar em cada circuito cerca de 10 portas lógicas.

5. Surgiu a integração em média escala ou MSI (Medium Scale Integration) onde passava-se a integrar entre 100 e 1.000 portas lógicas.

6. Anos mais tarde, chega a LSI (Large Scale Integration) integrando entre 1000 e 10.000 portas lógicas numa única pasta de silício.

7. Quando foram ultrapassados as 10.000 portas lógicas chega a VLSI (Very Large Scale Integration).

8. Em 1971 surge o microprocessador, com o qual conseguiu-se implementar toda a CPU (Unidade Central de Processamento) num único elemento integrado.

De acordo com a evolução, dividiu-se os computadores em gerações:

I Geração de Computadores (1940 - 1952): É constituída por todos os computadores construídos a base de válvulas a vácuo, e que eram aplicados em campos científicos e militares. Utilizavam como linguagem de programação a linguagem de máquina e a forma de armazenar dados era através de cartões perfurados.

II Geração de Computadores (1952 - 1964): Tem como marco inicial o surgimento dos Transistores. As máquinas diminuíram muito em tamanho e suas aplicações passam além da científica e militar a administrativa e gerencial. Surgem as primeiras linguagens de programação. Além do surgimento dos núcleos de ferrite, fitas e tambores magnéticos passam a ser usados como memória.

III Geração de Computadores (1964 - 1971): Tem como marco inicial o surgimento dos Circuitos Integrados. Grande evolução dos Sistemas Operacionais, surgimento da multiprogramação, real time e modo interativo. A memória agora é feita de semicondutores e discos magnéticos.

IV Geração de Computadores (1971 - 1981): Tem como marco inicial o surgimento do Microprocessador, que possibilitou a grande redução no tamanho dos computadores. Surgem muitas linguagens de alto-nível e nasce a teleinformática (transmissão de dados entre computadores através de rede).

V Geração de Computadores (1981 até a atualidade): Surgimento do VLSI. Inteligência artificial, altíssima velocidade de processamento, alto grau de interatividade.

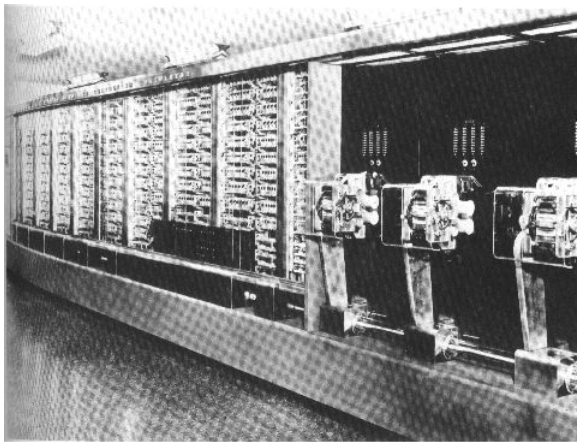


Figura 1.8: MARK I

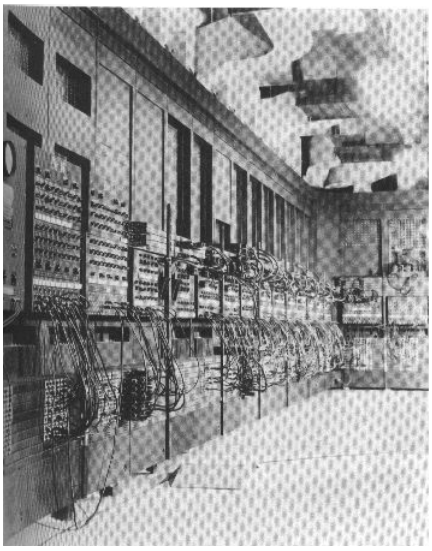


Figura 1.9: ENIAC

1.7 Início e Evolução das Linguagens de Programação

Ao desenvolver os projetos lógicos do EDVAC e da máquina do *Institute of Advanced Study* da Universidade de Princeton , von Neumann tinha também uma preocupação muito grande com a sua programação.
Existe um manuscrito de von Neumann que contém o que é quase certamente o primeiro programa escrito para um computador com programa armazenado na memória. Uma análise muito detalhada deste manuscrito e da sua história foi feita em 1970 por Donald E. Knuth no artigo intitulado Von Neumann's First Computer Program. O programa, escrito em 1945, propõe uma solução para o problema da classificação de uma série de dados em ordem não decrescente de uma chave.
A própria escolha do problema é muito significativa pois von Neumann queria mostrar que este tipo de máquina poderia realizar, de maneira muito eficiente, uma tarefa que era executada então pelas classificadoras de cartões da IBM, máquinas eletromecânicas especialmente projetadas para esta finalidade.
Ficaria demonstrada assim a aplicabilidade do EDVAC não apenas a cálculos científicos mas também para propósitos mais gerais.
O projeto da máquina do *IAS* por von Neumann teve uma descrição muito mais completa e mais divulgada do que a do EDVAC. Este projeto escrito por Goldstine e von Neumann, compõe-se de três volumes intitulados Planning and Coding of Problems for an Electronic Computing Instrument. Os três volumes constituem um verdadeiro manual de técnicas de programação com múltiplos exemplos. O primeiro volume é dedicado à metodologia de programação. Sugere que a tarefa seja separada em duas fases: uma primeira referente à parte lógica da solução a ser representada por diagramas de fluxo e uma segunda que é a codificação propriamente dita. Nota que o problema de codificação é um problema de tradução da linguagem matemática em que o problema e sua solução foram concebidos para uma outra linguagem, a da máquina. Explica a utilização de construções iterativas e de decisão e a correspondente notação em termos de diagramas. Explicita a conexão óbvia entre iteração e indução. O treino de von Neumann em lógica aparece na discussão de primeiros conceitos de linguagens de programação: constantes, variáveis livres (isto é, parâmetros) e variáveis ligadas (isto é, variáveis locais) de um programa. Outra conseqüência dos seus conhecimentos de lógica é a introdução de asserções indutivas para descrever o estado da computação em pontos selecionados dos diagramas. Através de pequenos

exemplos são introduzidos vários conceitos como por exemplo indexação e subrotinas. Novamente é dada ênfase à análise de eficiência de execução dos programas codificados. Finalmente, há uma seção com subrotinas para conversão entre as notações decimal e binária, bem como para aritmética de precisão dupla.
O segundo volume desta parte da documentação traz vários exemplos de programação. Os primeiros são de natureza numérica, envolvendo os problemas de integração pelo método de Simpson e de interpolação. É discutido o problema de erros de arredondamento e são apresentados alguns variantes conforme a maneira de representar dados. Como sempre, há análises de tempo de execução.
Uma grande parte do segundo volume é dedicada novamente ao problema de classificação por intercalação, apresentando de maneira mais completa e acabada o mesmo algoritmo codificado no manuscrito analisado por Knuth. Há uma justificativa explícita para a utilização deste problema a fim de testar a eficiência das partes não aritméticas da máquina: memória e controle lógico. O problema de intercalação é resolvido de maneira muito semelhante à apresentada anteriormente. O problema de classificação é apresentado então como um problema de repetição da intercalação com seqüências de comprimentos crescentes 1, 2, 4, 8, 16, ... A análise de tempo de execução produz o resultado que hoje é bem conhecido, com número de operações proporcional a $n \log(n)$. A descrição da implementação do algoritmo termina com uma comparação com a eficiência das máquinas classificadoras de cartões mostrando que, com hipóteses razoáveis sobre o tamanho dos registros classificados e das suas chaves, o computador deve ser de 15 a 30 vezes mais veloz, para uma massa de dados que caberia na memória. Finalmente, há considerações sobre a utilização do mesmo método para a classificação externa com dados em memória secundária como, por exemplo, numa fita magnética.
A escolha do problema de classificação e a solução adotada não podem ser subestimadas. Mesmo antes do advento dos computadores eletrônicos, classificadoras e intercaladoras eletromecânicas eram muito usadas em aplicações empresariais e em processamento de grandes volumes de dados. Durante muitos anos, as aplicações de computadores dependiam em boa parte de sua capacidade de classificação, principalmente de grandes arquivos de dados contidos em fitas magnéticas. Knuth menciona, em 1973, que, de acordo com as estimativas dos fabricantes de computadores daquela época, mais de 25% do tempo de uso dos seus computadores eram dedicados à classificação, chegando a mais de 50% em muitas instalações.

1.7.1 As linguagens de Alto Nível

O modelo de von Neumann para a arquitetura lógica dos computadores estabeleceu as seguintes características para os sistemas de computação:
Os computadores *Processam Dados* sob o controle de um conjunto de *Instruções* denominada *Programa*.
Os programas que *rodam* num computador são denominados *Softwares*
Os vários *devices* que compõe um sistema computacional como:teclado, monitor, discos, memória, unidade de processamento, etc. são denominados *Hardware*. Os programadores escrevem as *Instruções* em várias *linguagens de programação*.
Algumas são diretamente "entendidas" pelo computador outras requerem *traduções intermediárias*.
Com esse enfoque as linguagens podem ser divididas em:

- Linguagens de Máquina
- Linguagens Assembly
- Linguagens de Alto Nível

Os *programas tradutores* de linguagem Assembly para linguagem de máquina são denominados *Assemblers*.
Os *programas tradutores* de linguagem de alto nível para linguagem de máquina são denominados *Compiladores*

Fortran

Em 1954 ocorreu um simpósio sobre *computação automática*, e seu maior evento foi a apresentação do compilador algébrico de Laning e Zierler. Foi o primeiro *software* que permitiu como entrada de dados um código algébrico elegante, embora limitado.

Nesse mesma época John Backus montou um grupo de pesquisa dentro da IBM para trabalhar em um projeto sobre programação automática, para responder a uma questão fundamental: "(...) pode uma máquina traduzir uma linguagem matemática abrangente em um conjunto razoável de instruções, a um baixo custo, e resolver totalmente uma questão?". Em novembro de 1954 a equipe de Backus tinha criado o IBM Mathematical FORmula TRANslation System, o FORTRAN.

Com o FORTRAN apareceram as expressões simbólicas, subprogramas com parâmetros, mas principalmente ocorreu a primeira tentativa de

se definir rigorosamente a sintaxe de uma linguagem de programação. Um pouco mais tarde surgiu a notação BNF para a descrição sintática de uma linguagem de programação.

Lisp

A história do LISP remonta a Turing e Church. Pela análise de Turing nos anos de 1936 e 1937, após seu famoso artigo sobre o décimo problema de Hilbert, o cálculo-lambda de Church, apesar da sua sintaxe simples, era suficientemente poderoso para descrever todas as funções mecanicamente computáveis, ou seja, pode ser visto paradigmaticamente como uma linguagem de programação. No cálculo-lambda muitos problemas de programação, especificamente aqueles referentes às chamadas de procedimento, estão em sua forma mais pura, e isto influenciará diretamente linguagens como LISP e Algol.

Algol

Nos inícios da década de 1960, fruto do trabalho de americanos e europeus, surgiu uma linguagem projetada para representar algoritmos ao invés de se escrever programas simplesmente, o Algol-60. Ela implementava o conceito de estrutura de blocos, onde variáveis, procedimentos, etc., poderiam ser declarados onde quer que o programa os necessitasse. Algol-60 influenciou profundamente muitas linguagens que vieram depois e evoluiu para o Algol-68.

Pascal

Foi a linguagem Pascal entretanto que se tornou a mais popular das linguagens do estilo Algol, porque é simples, sistemática e facilmente implementável nos diferentes computadores. Junto com o Algol-68, está entre as primeiras linguagens com uma ampla gama de instruções para controle de fluxo, definição e construção de novos tipos de dados.

C

Em 1970, Ken Thompson, chefe da equipe que projetou o UNIX para o PDP11 do Bell Labs, implementou um compilador que batizou de *linguagem B*.

A linguagem *B* mostrou-se muito limitada, prestando-se apenas para certas classes de problemas. Na primeira mudança de computador, percebeu-se que o sistema teria de ser novamente reescrito numa outra linguagem.

Ficou claro então que seria conveniente que o sistema fosse baseado numa linguagem independente da máquina e que, ao mesmo tempo,

permitisse o desenvolvimento de sistemas operacionais.

Estes problemas levaram o projetista Dennis Ritchie, do Bell Labs, a projetar uma nova linguagem, sucessora da *B*, que viria então, a ser chamada de *C*.

A linguagem *C* buscou manter o "contato com o computador real"e mais ainda dar ao programador novas condições para o desenvolvimento de programas em áreas diversas, como comercial, científica e de engenharia.

Por muitos anos (aproximadamente 10) a sintaxe (formato) tida como padrão da linguagem *C* foi aquela fornecida com o UNIX versão 5.0 do Bell Labs. A principal documentação deste padrão encontra-se na publicação "The *C* Programming Language", de Brian Kernighan e Dennis Ritchie (K&R), tida como a "bíblia da linguagem *C*".

O mais interessante desta versão de *C* era que os *programas-fonte* criados para rodar em um tipo de computador podiam ser transportados e recompilados em outros sem grandes problemas. A esta característica dá-se o nome de *portabilidade*.

Em 1985, ANSI (American National Standards Institute) estabeleceu um padrão oficial da linguagem *C* chamada "*C* ANSI".

Smalltalk/C++/Delphi

No início da década de 1990 ocorreu um difusão intensa do paradigma da orientação a objeto. Este paradigma esteve em gestação por cerca de 30 anos e as novas tecnologias como a Internet, as necessidades geradas pela novas arquiteturas, tais como a de cliente-servidor e a do processamento distribuído, coincidiam com o paradigma da orientação a objeto: encapsulamento, mensagem, etc. O crescimento da Internet e o "comércio eletrônico"introduziram novas dimensões de complexidade no processo de desenvolvimento de programas. Começaram a surgir linguagens que buscam superar esses novos desafios de desenvolvimento de aplicações em um contexto heterogêneo (arquiteturas de hardware incompatíveis, sistemas operacionais incompatíveis, plataformas operando com uma ou mais interfaces gráficas incompatíveis, etc). "o próximo passo ou um paradigma completamente novo", surge a linguagem JAVA.

Java

A origem da linguagem Java está ligada a um grupo de pesquisa e desenvolvimento da Sun Microsystems formado em 1990, liderado por

Patrick Naughton e James Gosling, que buscava uma nova ferramenta de comunicação e programação, independente da arquitetura de qualquer dispositivo eletrônico. Em 1994, após o surgimento do NCSA Mosaic e a popularização da Internet, a equipe centrou seus esforços para criar uma linguagem para aplicações multimídia on line.

A linguagem Java foi inspirada por várias linguagens:

tem a concorrência da *Mesa*, tratamento de exceções como *Modula-3*, linking dinâmico de código novo e gerenciamento automático de memória como *LISP*, definição de interfaces como *Objective C*, e declarações ordinárias como *C*. Apesar destas qualidades, todas importantes, na verdade duas outras realmente fazem a diferença e tornam Java extremamente atrativa: sua portabilidade e o novo conceito de arquitetura neutra.

Portabilidade significa que Java foi projetada objetivando aplicações para vários sistemas heterogêneos que podem compor uma rede como a Internet, por exemplo, e as diferentes características dessa rede. Java procura obter os mesmos resultados de processamento nas diferentes plataformas.

Por arquitetura neutra entende-se que programas em Java são compilados para se obter um código objeto (byte code na terminologia Java) que poderá ser executado em um Power PC que use o sistema operacional OS/2, ou em um sistema baseado no chip Pentium debaixo do Windows 95 ou em um Macintosh usando MacOs, ou em uma estação de trabalho Sparc rodando Unix. Ou seja, em qualquer computador, desde que tal computador implemente o ambiente necessário para isso, denominado conceitualmente de Máquina Virtual Java.

Capítulo 2

Introdução

2.1 Algoritmos

Um algoritmo é um procedimento *bem definido* para solucionar um problema num número finito de passos.
O termo ALGORITMO é uma deturpação do nome *al-Khowarizmi* que foi um matemático árabe século IX.
Seu livro sobre os numerais hindus é a base da moderna notação decimal.
Originalmente a palavra foi utiliza como ALGORISMO e designava as regras para a aritmética usando a notação decimal.
O termo ALGORISMO evoluiu para ALGORITMO por volta do século XVIII.
Com o advento dos computadores o conceito inicial de ALGORITMO foi extendido para incluir todos procedimentos para solucionar problemas e não só aqueles envolvendo aritmética.

2.2 Algoritmos Computacionais

Em 1833, quando Charles Babbage concebeu as idéias de sua *máquina de cálculo universal*, idéias que viriam a ser retomadas em 1930 por Alan Turing, ele já tinha plena consciência que ela deveria seguir conjuntos mutáveis de instruções e, portanto, servir a diferentes funções. Babbage percebeu que para criar estas instruções precisaria de um tipo inteiramente novo de *linguagem* e a imaginou como números, flechas e outros símbolos. Essa linguagem serviria para *programar* a Máquina Analítica com uma longa série de instruções condicionais que lhe permitiriam modificar suas ações em resposta a diferentes situações.
Reconhecendo a importância de se terem resultados impressos, ele percebia

a necessidade de os resultados finais e os intermediários serem impressos.
Dispositivos de entrada e saída eram assim necessários.
A entrada de informações (dados) para a máquina seria feita através de três tipos de cartões: *cartões de números*, com os números das constantes de um problema; *cartões diretivos* para o controle do movimento dos números na máquina; e *cartões de operação* para dirigir a execução das operações tais como adições, subtrações, etc.
Mas o mais genial foram duas inovações conceituais *simples* mas que produziram um grande impacto.
A primeira era o conceito de "transferência de controle"que permitia à máquina comparar quantidades e, dependendo dos resultados da comparação, desviar para outra instrução ou seqüência de instruções.
A segunda característica era possibilitar que os resultados dos cálculos pudessem alterar outros números e instruções colocadas na máquina, permitindo que o *computador* modificasse seu próprio programa.
Nesses temas teve importante participação Ada Augusta Byron, Lady Lovelace.
De acordo com B.H. Newman, os trabalhos de Ada Byron demonstram que ela teve total compreensão dos princípios de um computador programado com um século de antecedência.
Mesmo não estando a máquina construída, Ada escreveu seqüências de instruções e propôs conceitos que seriam posteriormente essenciais na programação de computadores como subrotinas, loops e saltos.
Ela é reconhecida como a primeira *programadora*.
Com essas considerações fica claro que para **construir um algoritmo** para resolver um determinado problema, via computador, devemos levar em consideração os seguintes princípios:

1 - Quais são as *ordens* que um computador pode executar;

2 - Quais os tipos informações (dados) ele pode considerar;

3 - Quais são as entradas;

4 - Quais são as saídas.

Vamos denominar **algoritmos computacionais** os algoritmos escritos considerando-se os principios acima.
Por mais incrível que possa parecer, básicamente, as *ordens* que um computador pode executar continuam as mesmas desde a época de Babbage, ou seja: subrotinas, loops e saltos.

Assim para escrever **algoritmos computacionais** para solucionar determinados problemas não existe a necessidade de se fixar numa linguagem de programação mas apenas nesses princípios básicos.
Podemos usar uma simbologia para esses princípios e adotá-la como nossa **linguagem algorítmica**.
É claro que para executar esse algoritmo num computador vamos necessitar *traduzir* esse algoritmo da **linguagem algorítmica** para a **linguagem de programação** que é a linguagem na qual o computador pode executar os **algoritmos computacionais**.
Todas as **linguagens de programação** contem os mesmos princípios básicos enumerados acima. O que determina a escolha de uma ou outra linguagem, numa determinada situação, são as **estruturas de dados** e **entradas e saídas** da linguagem que podem ser mais conveniente e naturais para a situação.
Como já dissemos para executar os **algoritmos computacionais** num computador eles devem necessariamente estar escritos numa **linguagens de programação**.
No nosso entendimento essas duas tarefas podem ser abordadas concomitantemente mas sempre deixando claro que são independentes.
Também é essencial entender que o conhecimento da sintaxe de uma linguagem não é **condição suficiente** para a elaboração de um algoritmo mas é uma **condição necessária** para poder executá-lo num computador.

2.3 A escolha da linguagem *C*

A linguagem *C* é uma linguagem de alto nível muito poderosa. Ela foi, em principio, projetada para desenvolver sistemas operacionais e posteriormente foi adaptada para uso geral. Essa origem dotou-a de ferramentas que permitem um fácil acesso aos denominados componentes de *baixo nível* do computador. Por essa facilidade ela pode ser considerada também de baixo nível e essa característica a faz extremamente atraente para os experts, mas torna-a *um pouco perigosa* para os principiantes.
Ao longo do tempo a linguagem *C* tornou-se a ferramenta preferida para o desenvolvimento de *compiladores e softwares* sendo que praticamente todas as linguagens atualmente em uso foram desenvolvidas usando-a. Essa característica possibilita que a maioria dos comandos básicos nessas linguagens tenham a mesma sintaxe que o *C* - e.g. C++, Perl, JavaScript, Java, etc.
Achamos que esse é um dos muitos argumentos favoráveis ao aprendizado do *C* como primeira linguagem em que pese a demasiada "liberdade"que

propicia ao usuário neófito.
Uma outra vantagem é a seguinte: o *C* não sendo uma linguagem proprietária sempre será possível conseguir um compilador *C* free !!! (sem custo).
Iremos adotar a estratégia de introduzir à linguagem na sua forma mais básica tornando seu aprendizado sem vínculos com os diversos ambientes e plataformas atualmente no mercado. Isso tornará possível utilizar indiferentemente (no que se refere à linguagem) os sistemas Windows, Linux|Unix, Dos, etc.) como também não misturar o aprendizado do ambiente (editor de texto, gerenciador de projetos etc.) com o aprendizado da linguagem.

2.4 O mais simples Programa *C*

Vamos começar com o mais simples possível programa *C* e usá-lo para mostrar alguns fundamentos da linguagem bem como o processo de *compilação*.

1. Digite o programa abaixo (3.) usando um editor de texto standard (vi ou emacs no Linux|Unix, Notepad no Windows ou TeachText num Macintosh) ou o seu predileto;

2. Salve o texto (programa) num file chamado, digamos, **hello.c** e assegure-se de que seu texto foi corretamente criado com o nome (**hello**) e a extensão (**.c**) ;

3. Aqui está o primeiro programa:

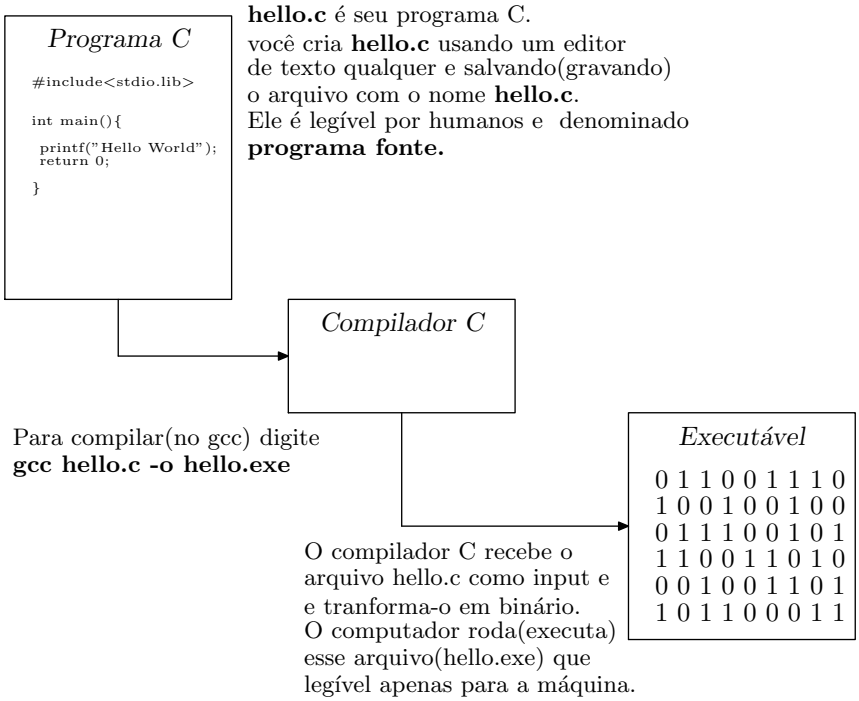
```
#include <stdio.h>
main(){
  printf("Hello World");
}
```

4. Quando executado, o programa irá *ordenar* ao computador para *escrever* no vídeo a linha **Hello World**

5. Para compilar este *código* são necessários os seguintes passos:

   - Numa máquina Linux|Unix, digite
     gcc hello.c -o hello
     (se **gcc** não funcionar tente **cc**). Este comando deve invocar o compilador *C*, chamado **gcc**, solicitar a compilação do programa fonte **hello.c** e dar ao **executável** o nome **hello**. Para executá-lo basta digitar **hello** (ou, em algumas máquinas Linux|Unix, ./hello).

- Numa máquina DOS or Windows usando DJGPP (www.delorie.com/djgpp/), no prompt do MS-DOS digite **gcc hello.c -o hello.exe**
  Este comando deve invocar o compilador *C*, chamado **gcc**, solicitar a compilação do programa fonte **hello.c** e dar ao **executável** o nome **hello.exe**. Para executá-lo basta digitar **hello.exe**
- Caso esteja usando um outro tipo de compilador ou sistema leia as instruções para compilar e executar programas.



2.5 Free Software Disponíveis para Download

DJGPP - A free 32-bit development system for DOS

ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2/readme.1st
Installation instructions 20 kb

ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2/djdev203.zip
DJGPP Basic Development Kit 1.5 mb

ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2apps/rhid149b.zip
RHIDE (similar to Borland's IDE, including a built-in editor and debugger) 2.4 mb

ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2gnu/bnu2112b.zip
Basic assembler, linker 2.6 mb

ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2gnu/gcc304b.zip
Basic GCC compiler 2.2 mb

ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2gnu/gdb500b.zip
GNU debugger 1.1 mb

ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2gnu/mak3791b.zip
Make (processes makefiles) 267 kb

ftp://ftp.simtel.net/pub/simtelnet/gnu/djgpp/v2gnu/txi40b.zip
Info file viewer 632 kb

LCC-Win32:A portable C compiler

Você pode fazer download do *lcc-win32 C compiler system* em:
ftp://ftp.cs.virginia.edu/pub/lcc-win32/lccwin32.exe
ftp://ftp.cs.virginia.edu/pub/lcc-win32/manual.exe

O sistema é auto-contido: você não necessita nada mais do que sua instalação para começar a programar em *C* num ambiente Win32. Você terá:

1. Gerador de Código (compiler, assembler, linker, resource compiler, librarian)
2. Ambiente de desenvolvimento integrado com: editor, debugger, make file generation, etc.
3. Manual do usuário - aproximadamente 240 pages no formato Word Microsoft.

Capítulo 3

Introduzindo as Estruturas e as Operações Básicas

3.1 Dados, Operadores e Expressões

3.1.1 Tipos de Dados Básicos

- int (Tipo Inteiro) 10;4367;-6253;-1
- float (Tipo Real) 0.54;-234.98;0.00654
- char (Tipo Caracter) 'a';'R';'@';'>'

3.1.2 Operadores Aritméticos

Um **operador**, também denominado **operador binário**, é uma função que a cada par de valores de um determinado conjunto associa um valor desse mesmo conjunto.
Na linguagem matemática podemos definir um operador como:

$$\begin{array}{rccc} op: & X \times X & \longmapsto & X \\ & (x,y) & \longrightarrow & op\,(x,y) \end{array}$$

Um operador com o qual já temos familiaridade desde os tempos da nossa infância é o conhecido operador **soma** ou seja:

$$\begin{array}{rccc} soma: & Z \times Z & \longmapsto & Z \\ & (x,y) & \longrightarrow & soma(x,y) \end{array}$$

O conjunto Z neste caso representa o conjunto dos números inteiros.
É claro que estamos muito mais acostumados com a seguinte notação:

$(x, y) \longrightarrow soma(x, y) = x + y$
Ou seja o símbolo + está representando o operador **soma**.
De modo semelhante temos os operadores de **subtração**, **multiplicação** e **divisão** que são denotados por:

Operador	símbolo na linguagem	símbolo na matemática
Soma	+	+
Subtração	-	-
Multiplicação	*	$\times$ ou $\cdot$
Divisão	/	$\div$

Exemplo 3.1.1

$7 + 3$	$\longrightarrow$	10
$9 - 13$	$\longrightarrow$	-4
$-2 * 3$	$\longrightarrow$	-6
$4/2$	$\longrightarrow$	2
$2/4$	$\longrightarrow$	0
$-6/3$	$\longrightarrow$	-2

▲

Devemos ainda considerar um operador que é muito utilizado nos algoritmos computacionais.
Esse operador, que é definido entre números do tipo **int** e cujo resultado também é do tipo **int**, é denominado **resto da divisão inteira** e será denotado por %.

Exemplo 3.1.2

$7\%1$	$\longrightarrow$	0
$1\%3$	$\longrightarrow$	1
$23\%5$	$\longrightarrow$	3
$-5\%2$	$\longrightarrow$	-1

▲

Do mesmo modo que temos os operadores definidos no conjunto Z temos aqueles que são definidos no conjunto R dos números reais. As operações nesse caso são definidas para os dados do tipo **float** ou seja:

$$\begin{array}{rccc} op: & R \times R & \longmapsto & R \\ & (x,y) & \longrightarrow & op\,(x,y) \end{array}$$

Operador	símbolo na linguagem	símbolo na matemática
Soma	+	+
Subtração	-	-
Multiplicação	*	$\times$ ou $\cdot$
Divisão	/	$\div$

Exemplo 3.1.3

$3.4 + 56.91$	$\longrightarrow$	60.31
$9.21 - 14.8763$	$\longrightarrow$	-5.6663
$2.1 * 32.12$	$\longrightarrow$	67.452
$1.0/2.0$	$\longrightarrow$	0.5
$1.0/3.0$	$\longrightarrow$	$0.3333\cdots$

▲

3.1.3 Operadores Relacionais

Considerando o tipo básico de dado **int** sabemos entre os dados desse tipo existe uma ordem intrínseca já definida.
Essa ordem é a ordem já devidamente estabelecida no conjunto Z dos números Inteiros.
A relação de ordem estabelecida em Z significa que:
Dados $m, n \in Z$ apenas uma das seguintes possibilidades pode ocorrer:

- m é maior do n
- m é igual a n
- m é menor do que n

Na verdade basta sabermos comparar qualquer $r \in Z$ com zero pois

- m é maior do $n \Longleftrightarrow r = (m - n)$ é maior do 0
- m é igual a $n \Longleftrightarrow r = (m - n)$ é igual a 0
- m é menor do $n \Longleftrightarrow r = (m - n)$ é menor do 0

O mesmo que discutimos para os dados do tipo **int** vale igualmente para o dados do tipo **float**, ou seja, entre esses dados existe uma relação de ordem pré-estabelecida que é herdada do conjunto R dos números Reais.
No caso dos dados do tipo **char** temos estabelecida a chamada **ordem lexicográfica ou ordem alfabética**, ou seja, dados dois dados do tipo **char** sabemos "quem vem antes de quem" na ordem alfabética. Sabemos por exemplo que 'A' vem antes do 'C'. Na linguagem C temos a seguinte ordenação para os dados do tipo **char**:

- Os caracteres minúsculos sempre sucedem os maiúsculos por exemplo: 'a' sucede 'A'.
- 'a' 'b' 'c' 'd' 'e' 'f' 'g' 'h' 'i' 'j' 'k' $\cdots$ 'z'
- 'A' 'B' 'C' 'D' 'E' 'F' 'G' 'H' 'I' 'J' 'K' $\cdots$ 'Z'

Vamos utilizar os seguintes operadores relacionais

Operador	símbolo na linguagem	símbolo na matemática
menor que	$<$	$<$
menor ou igual	$<=$	$\leq$
maior que	$>$	$>$
maior ou igual	$>=$	$\geq$
igual	$==$	$=$
diferente	$!=$	$\neq$

3.1.4 Operadores Lógicos ou Booleanos

Vamos considerar agora um conjunto que seja composto por apenas 2 elementos.
Esses elementos podem ser representados pelos símbolos V e F; 1 e 0 ou T e F.
De modo geral esses conjuntos representam fenômenos que admitem apenas dois estados antagônicos (Verdadeiro ou Falso) sendo que apenas um desses estados pode ocorrer ao mesmo tempo.
Como já fizemos anteriormente, também para esse caso, podemos definir um operador nesse conjunto da seguinte maneira.
$op : \{V, F\} \times \{V, F\} \longrightarrow \{V, F\}$
$(x, y) \longmapsto op\,(x, y)$
De modo concreto vamos definir os seguintes operadores:

Operador	símbolo na linguagem	símbolo na matemática
and	&&	e
or	||	ou

Ao contrário dos operadores de adição, subtração, multiplicação e divisão com os quais temos grande familiaridade esses novos operadores exigem uma definição.
Felizmente o conjunto onde eles estão definidos contem apenas 2 elementos e assim basta definir como atua cada um dos operadores para os seguintes pares de valores:
$(V, V); (V, F); (F, V)$ e (F, F). Evidentemente o resultado da operação deve assumir apenas um dos valores V ou F.

Agora, como geralmente se diz na literatura, vamos agora construir a "tabela verdade" de cada um dos operadores.

&&	V	F
V	V	F
F	F	F

||	V	F
V	V	V
F	V	F

É usual também se definir um outro operador, neste caso um **operador unário**, denominado **operador de negação**.
Para este caso teremos:

$$\begin{array}{rccc} op: & \{V, F\} & \longmapsto & \{V, F\} \\ & x & \longrightarrow & op\,(x) \end{array}$$

Operador	símbolo na linguagem	símbolo na matemática
not	!	não

O operador atua da seguinte maneira:

$!V = F$
$!F = V$

Observando a definição dos **operadores relacionais** podemos notar que seu resultado tem apenas uma das seguintes possibilidades lógicas:$true$ ou $false$ que representamos por V e F.
Esta propriedade como veremos posteriormente será muito importante na construção de algoritmos.
Vejamos alguns exemplos a titulo de esclarecimento.

Exemplo 3.1.4

$(2 < 7)$	$\longrightarrow$	V
$(32 == 23)$	$\longrightarrow$	F
$(234! = 234)$	$\longrightarrow$	F
$('b' >=' A')$	$\longrightarrow$	V
$('X'! =' x')$	$\longrightarrow$	V

▲

3.1.5 Expressões Aritméticas

Uma expressão aritmética nada mais é que a expressão envolvendo os dados do tipo **int** e do do tipo **float** e os operadores aritméticos.
Nosso objetivo é definir a ordem de precedência dos operadores pois isso, obviamente é essencial para o resultado final da expressão.
Em principio poderíamos definir uma ordem qualquer para a nossa linguagem de alto nível mas seguindo a tradição, no que se refere do desenvolvimento dos computadores e das linguagens, vamos acompanhar os conceitos já estabelecidos pela matemática.
Assim vamos definir a seguinte ordem de prioridade:

(1) Os parênteses mais internos

(2) Multiplicação e Divisão

(3) Soma e Subtração.

(*) No caso dos operadores com mesma prioridade efetuamos as operações da esquerda para a direita.

Exemplo 3.1.5

$$(2 \overset{(1)}{+} 5 \overset{(2)}{*} 3) \overset{(3)}{+} (3 \overset{(4)}{-} 2 \overset{(5)}{*} 7 \overset{(6)}{-} (2 \overset{(7)}{*} 2 \overset{(8)}{-} 1))$$

As operações serão executadas na seguinte ordem:
(7)(8)(2)(1)(5)(4)(6)(3) ▲

3.1.6 Expressões Lógicas ou Booleanas

Uma expressão booleana é uma expressão que envolve os **operadores lógicos** e o resultado da utilização dos **operadores relacionais**.
A ordem de prioridade dos operadores é a seguinte:

(1) Os parênteses mais internos

(2) operadores relacionais

(3) ! (not)

(4) && (and)

(5) || (or)

(*) No caso de mesma prioridade da esquerda para a direita.

Exemplo 3.1.6

$$(1 \overset{(1)}{<} 2) \overset{(2)}{\&\&} ((3 \overset{(3)}{==} 7)) \overset{(4)}{||} ((2 \overset{(5)}{>} 3) \overset{(6)}{||} (5 \overset{(7)}{!=} 1) \overset{(8)}{\&\&} \overset{(9)}{!} (2 \overset{(10)}{<} 1))$$

As operações serão executadas na seguinte ordem:
(3)(5)(7)(10)(9)(8)(6)(1)(2)(4) ▲

Capítulo 4

As instruções da Linguagem

Instruções são **diretivas** ou ordens que damos ao computador para que ele execute determinadas tarefas.
As **diretivas** devem ser sempre encerradas por um *ponto e virgula*(;).

4.1 Armazenando Dados

Como já foi dito, uma das mais importantes qualidades do computador é sua grande capacidade de *armazenar* informações (dados) e de *acessá-las* com eficiência.
Os dados ficam, de alguma maneira, armazenados nos *registradores* (memória) do computador.
Assim é de vital importância existirem **diretivas** para as seguintes tarefas:

- Informo que deve ser reservado um *espaço* para armazenar um dado;
- Informo que esse dado **não muda** seu valor durante a execução ou;
- Informo que esse dado **pode mudar** seu valor durante a execução;
- Informo que esse dado é do tipo **int** ou;
- Informo que esse dado é do tipo **float** ou;
- Informo que esse dado é do tipo **char**.

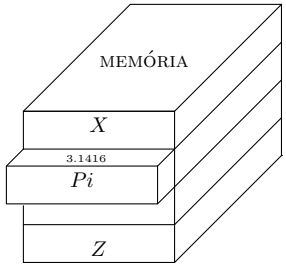
Em seguida veremos a sintaxe referente à essas diretivas.

a) Para dados que não têm seu valor alterado durante a execução.
Sintaxe: **const** tipo_do_dado nome_do_dado=valor_do_dado;
Execução:

- Reserva um espaço de memória para um dado cujo valor será constante durante a execução do programa;
- Esse dado será do tipo: **tipo_do_dado**;
- Ele será referenciado por seu nome: **nome_do_dado**;
- Atribua ao dado o valor: **valor_do_dado**;

Exemplo 4.1.1

const int j=10;
const float Pi=3.1416;
const char arroba='@'; ▲



b) Para dados que podem ter seu valor alterado durante a execução.
Sintaxe: tipo_do_dado nome_do_dado=valor_do_dado;

Execução:A mesma do caso anterior ressaltando que, eventualmente, o valor atribuido inicialmente pode ser alterado por uma diretiva posterior.


Os dados do tipo anterior, por motivos óbvios, são denominados **variáveis**.

Exemplo 4.1.2

int soma=0;
float produto=12.5;
char nome='Z'; ▲

4.2 Observações práticas sobre o nome dos dados

1. Não utilize acentos nos nomes dos dados.

2. Em que pese algumas linguagens de alto nível não fazerem distinção entre nomes de dados grafados em letras maiúsculas ou minúsculas, a linguagem *C* faz essa distinção. Por exemplo as diretivas abaixo são perfeitamente válidas.

$$\begin{cases} \text{float} \quad A = 10.1; \\ \text{const int} \quad a = 2; \end{cases}$$

3. Sempre que possível, escolha como nome do dado algo sugestivo ou que lhe diga respeito. Por exemplo: se uma variável vai armazer, digamos, a área de um círculo podemos usar:
   float area_do_circulo=0; ou
   float area=0;

4. O nome do dado não deve conter espaços em branco. Use o símbolo *underscore* (_) para simular os espaços nos nomes compostos.

5. Não utilize para nome de dado as chamadas *palavras reservadas* ou seja, aquelas que já têm algum significado para a linguagem. No nosso caso, até agora, as *palavras reservadas* são: **const, int, float e char**.

4.3 Os Comandos

Definimos um **comando** como sendo uma **diretiva** ou uma **seqüência de diretivas**.
Para o segundo caso será usada a seguinte sintaxe:

```
{   diretiva_1;
    diretiva_2;
        ⋮
    diretiva_n;
}
```

4.3.1 Comandos de Entrada e Saida de Dados

a) Entrada de Dados

Sintaxe: (1) scanf("%d", &var_int);
(2) scanf("%f", &var_float);
(3) scanf("%c", &var_caracter);

Execução: (1) O computador recebe, via teclado, um dado do tipo inteiro e armazena na variável var_int.
(2) O computador recebe, via teclado, um dado do tipo float e armazena na variável var_float.
(3) O computador recebe, via teclado, um dado do tipo caracter e armazena na variável var_caracter.

Exemplo 4.3.1

int idade=0;
scanf("%d", &idade); (suponha que o usuário tenha digitado 15)
Após a execução do comando scanf a variável idade terá o valor=15.
▲

b) Saida de Dados

Sintaxe: (1) printf("%s","um texto qualquer");
(2) printf("%i", var_int) ;
(3) printf("%f", var_float);
(4) printf("%c", var_caracter);

Execução: (1) escreve no vídeo o texto que está entre as aspas
(2) escreve no vídeo o valor do inteiro armazenado na variável var_int.
(3) escreve no vídeo o valor do float armazenado na variável var_float.
(4) escreve no vídeo o caracter armazenado na variável var_caracter.

Exemplo 4.3.2

float soma=12.54;
printf("%s%f", "Valor da soma=", soma);
Após a execução do comando write aparecerá no vídeo:
Valor da soma=12.54
▲

4.3.2 Comando de Atribuição de Valor

Esse comando nos possibilita alterar o valor de um dado do tipo variável.

Sintaxe: nome_da_variavel=expressão;

Execução: O computador avalia a expressão e esse valor é atribuído à variável nome_da_variavel.

Observação: Espera-se que a variável e a expressão sejam do mesmo tipo.

Exemplo 4.3.3

Comando	Estado da Variável
float $x = 0$;	$x \longleftarrow 0$
$x = 2.1 * 3 + 1$;	$x \longleftarrow 7.3$

▲

Exemplo 4.3.4

Comando	Estado da Variável
int $i = 0$;	$i \longleftarrow 0$
$i = 3$;	$i \longleftarrow 3$
$i = i + 1$;	$i \longleftarrow 4$
$i = 3 + 2 * i + i/8$;	$i \longleftarrow 11$

▲

4.3.3 Comando de Desvio Condicional

Um computador, em principio, executa os comando na ordem (seqüência) em que eles aparecem.
Não é difícil imaginar que algumas vezes será necessário que, sob determinada condição, algum(s) comandos dessa seqüência não sejam executados ou até que devam ser executados numa ordem diferente da estabelecida pela seqüência.
Assim existe a necessidade de um comando que nos possibilite atingir esse objetivos.

Sintaxe: (1) if (expressão_logica) $comando_1$

(2) if (expressão_logica) *comando*$_1$
else *comando*$_2$

Execução: (1) A expressão_logica é avaliada, se seu valor for ver dadeiro (true)então o comando *comando*$_1$ será exe cutado. Se o valor da expressão for (false) então o referido comando não será executado.

(2) A expressão_logica é avaliada. Se seu valor for ver dadeiro (true) então *comando*$_1$ será executado. caso contrário (false) será executado o comando_2

Exemplo 4.3.5

```
// Exemplo de Desvio Condicional
int logica=1;
if (logica) printf("Verdadeira");
else printf("Falsa");
```

Após a execução dos comandos teremos no vídeo a palavra:Verdadeira ▲

Exemplo 4.3.6

```
// Exemplo de Desvio Condicional
int j = 1;
if ((j + 3) <= 4) j = j + 1;
if (j > 2) printf("%s%d" , "valor de j=", j );
else printf("%s%d" , "valor calculado de j=" , j );
```

Após a execução dos comandos teremos no vídeo:valor calculado de j=2 ▲

4.3.4 Comando de Repetição - While

Em diversos algoritmos é necessário que comandos sejam executados repetidamente até que uma determinada condição seja atingida.

Sintaxe: while (expressão_logica) *comando*$_1$

Execução: Enquanto a expressão_logica for verdadeira (true) o o comando *comando*$_1$ é executado.

Se expressão_logica=false o comando *comando*$_1$ não será executado. Se o valor da expressão_logica não tornar-se false em alguma execução do *comando*$_1$, teremos então o "famoso" *loop infinito*.

Exemplo 4.3.7

```
// Exemplo usando While
int i = 1;
while (i < 3) i=i+1;
printf("%s%i", "valor de i=" , i );
```

Após a execução teremos a saída: valor de i=3 ▲

Exemplo 4.3.8

```
// Exemplo usando While
int entrei = 1;
int i = 1;
while (entrei){
    i = i * 2;
    if (i > 6) entrei = false;
}
printf("%s%i" , "valor de i=" , i );
```

Após a execução teremos a saída: valor de i=8 ▲

4.3.5 Comando Break

Esse comando é utilizado quando houver a necessidade de "forçar" o término da execução normal de um comando de repetição ou de um comando de escolha multipla **switch** que iremos abordar posteriormente.
Sintaxe: break;

Execução: "Aborta" a execução do *loop*, transferindo a execução para o próximo comando após o bloco de diretivas.

4.3.6 Comando de Repetição - Do

Como já foi discutido anteriormente se a expressão booleana na diretiva **while** for **false** o laço do while não é executado. Isso é muito interessante para determinadas *situações algoritmicas* mas não desejado em outras.
Não são raras as situações em que é interessante, e as vezes até necessário, que o laço de repetição seja executado ao menos uma vez.
É claro que poderíamos fazer um *malabarismo algoritmico* para conseguir isso usando o comando **while**, mas com isso tornaríamos o procedimento não natural e por conseqüência de difícil entendimento.

Sintaxe: Do $comando_1$ while (expressão_logica);

Execução: Executa o comando $comando_1$;
Avalia a expressão_logica;
Enquanto a expressão_logica = true executa o $comando_1$.

Independente do valor da expressão_logica o comando $comando_1$ é executado pelo menos uma vez.
Se o valor da expressão_logica não tornar-se false em alguma execução do $comando_1$ teremos um *loop infinito*

Exemplo 4.3.9

```
// Contagem de uma entrada de Caracteres
char caracter=' ';
int contador=0;
printf("Entre com caracter(es) (# para encerrar)");
do {
      scanf("%c" , &caracter);
      if(caracter == '#') break;
```

```
    contador=contador+1;
  } while (caracter != '#');
printf("%s%s%d%s","\n","voce digitou" , contador ,"caracter(s) valido(s)");
```

Input ⟵ As&Bcxy$#
Output ⟶ voce digitou 8 caracter(es) valido(s) ▲

Você notou que foi utilizada no inicio do comando printf a seqüência "\n"?
Esse texto representa uma diretiva para o comando printf.
Sua função é informar ao comando printf para abandonar a linha na qual ele esta "escrevendo" e mudar para a próxima linha.

4.3.7 Comando de Repetição - For

Esse comando de repetição é usado para o caso em que temos a seguinte situação:
Queremos repetir um comando uma quantidade de vezes conhecida à priori. Além disso sabemos o valor inicial e final da *variável contadora* das repetições.
Sintaxe: for (com_init ; exp_logica ; com_prox_exec) $comando_1$

Execução:

1 - O comando de inicialização com_init é executado;

2 - A expressão exp_logica é avaliada;

3 - Se a exp_logica=true são executados os comandos: $comando_1$ e em seguida o comando de preparação para proxima execução com_prox_exec.

4 - A expressão exp_logica é re-avaliada e volta-se à etapa 3.

5 - A repetição será encerrada quando exp_logica=false.

Se inicialmente exp_logica=false então os comandos:$comando_1$ e com_prox_exec não serão executados. Se exp_logica não tornar-se false em alguma execução do $comando_1$ ou do comando com_prox_exec teremos um *loop infinito*

Exemplo 4.3.10

```
// Calculando a Soma dos pares entre 1 e n dado.
int i=0,n=0,soma=0; write("Entre com um inteiro positivo");
scanf("%d%,&n );
for (i = 1; i <= n; i=i+1)
if (i%2 == 0) soma = soma+i;
printf("%s%d%s%d","A soma dos inteiros pares entre 1 e ",n,"eh=",soma);
```

Input ⟵ 10
Output ⟶A soma dos inteiros pares entre 1 e 10 eh=30 ▲

4.3.8 Comando de Múltipla Escolha - Switch

Quando um algoritmo necessitar de decisões lógicas baseadas em testes de diversas expressões lógicas esse é o comando adequado.

Sintaxe:

```
switch ( exp ){
  case     exp_1:  comando_1   break;
  case     exp_2:  comando_2   break;
  ...      ...     ...         ...
  case     exp_n:  comando_n   break;
  default  :       comando_d
};
```

Obs: A expressão exp e as expressões exp_1 , exp_2 , $\cdots$, exp_n devem ser do mesmo tipo.

Execução: A expressão exp é avaliada e seu valor é comparado com exp_1.
Se as expressões forem iguais o comando $comando_1$ é executado e o comando break finaliza o bloco.
Caso as expressões comparadas não sejam iguais a comparação será feita agora entre exp e exp_2 e assim sucessivamente.
Caso exp e exp_i sejam iguais para algum i o comando $comando_i$ será executado e o bloco encerrado.
Se exp for diferente de todas exp_i então o comando $comando_d$ que é o chamado comando default será executado.

Exemplo 4.3.11

```
// Determinando se uma letra recebida como input eh Vogal ou Consoante
char letra=' ';
printf("Entre com uma letra ... (minuscula por favor)");
scanf("%c", &letra);
switch (letra){
      case 'a': write(" A letra eh uma vogal ");break;
      case 'e': write(" A letra eh uma vogal ");break;
      case 'i': write(" A letra eh uma vogal ");break;
      case 'o': write(" A letra eh uma vogal ");break;
      case 'u': write(" A letra eh uma vogal ");break;
      default : write(" A letra eh uma consoante ");
}
```

Input ⟵ v
Output ⟶ A letra eh uma consoante ▲

4.3.9 Operadores de Incremento e Decremento

Considerando que comandos do tipo $i = i + 1$ e $i = i - 1$ aparecem com muita freqüência em diversos algoritmos O C fornece dois operadores para facilitar e otimizar os comandos acima.
O operador de incremento denotado ++ que adiciona 1 ao seu operando.
O operador decremento denotado por $--$ que subtrai 1 ao aperando.
Temos então as seguintes formas:

$n++$ O valor de n é incrementado de 1 depois de ser usado.
$++n$ O valor de n é incrementado de 1 antes de ser usado.
$n--$ O valor de n é decrementado de 1 depois de ser usado.
$--n$ O valor de n é decrementado de 1 antes de ser usado.

Analise com atenção o exemplo a seguir(não pense que tem algo errado nele!!). Ele deve ajudá-lo a entender os operadores de incremento e decremento.

Exemplo 4.3.12

```
#include <stdio.h>
main(){
 int i=5;
 printf("i = %d\n" ,i);
 printf("i++ = %d\n" , i++);
```

```
 printf("i = %d\n" , i);
 printf("----------\n");
 i=10;
 printf("i = %d\n" ,i);
 printf("++i = %d\n" , ++i);
 printf("i = %d\n" , i);
}
```

Output

$i = 5$
$i++ = 5$
$i = 6$
$----------$
$i = 10$
$++i = 11$
$i = 11$

4.4 As Funções Pré-Definidas

Nossa linguagem Pidgin-Algo_Dados possui as seguintes funções pré-definidas

Função	Matemática	Ação
sqrt :$float \mapsto float$	$sqrt(x) = \sqrt{x}$	raiz quadrada de $x \geq 0$
exp :$float \mapsto float$	$exp(x) = e^x$	exponencial de x
sin :$float \mapsto float$	$sin(x) = seno(x)$	seno de x em rad
cos :$float \mapsto float$	$cos(x) = cos(x)$	cosseno de x em rad
fabs :$float \mapsto float$	$fabs(x) = \|x\|$	módulo de x
floor :$float \mapsto int$	$floor(x) = \lfloor x \rfloor$	maior inteiro $\leq x$
ceil :$float \mapsto int$	$ceil(x) = \lceil x \rceil$	menor inteiro $\geq x$
log :$float \mapsto float$	$log(x) = ln(x)$	logaritmo natural de $x > 0$
pow :$float \times float \mapsto float$	$pow(x, y) = x^y$	x elevado a y onde $x > 0$

Capítulo 5

Algoritmos Propostos (I)

Lembre-se que um algoritmo computacional é um procedimento *bem definido* para solucionar um problema num *número finito de passos*.

ALGORITMO 5.1

Algoritmo: Dada uma temperatura na escala Celsius transforma-la para a escala Fahrenheit sabendo-se que

$$C = (5/9)(F - 32)$$

Input: Um inteiro representando a temperatura na escala Celsius.

Output: O inteiro que representa a temperatura correspondente na escala Fahrenheit

ALGORITMO 5.2

Algoritmo: Determinar a área e o comprimento de um circulo conhecido seu raio.
$area = \pi r^2$; $comprimento = 2\pi r$

Input: Um inteiro positivo representando o raio do circulo.

Output: A área e o comprimento do circulo.

ALGORITMO 5.3
Algoritmo: Dado um inteiro N, determinar se ele é par ou impar.

Input: Um inteiro.

Output: O número dado no input e uma mensagem informando se ele é par ou impar.

ALGORITMO 5.4
Algoritmo: Dados dois inteiros N e M, armazenar numa variável de nome **maior** o maior deles e numa variável de nome **menor** o menor deles.

Input: Dois números inteiros.

Output: O conteúdo da variável menor e da variável maior.

ALGORITMO 5.5
Algoritmo: Suponha que foram recebidas, via input, as variáveis do tipo float x e y.
Por motivo de necessidade, muito comum em determinadas classes de algoritmos, desejamos que os conteúdos das variáveis sejam trocados.

Input: Os dados do tipo float x e y.

Output: O conteúdo da variável x e da variável y.

ALGORITMO 5.6

Algoritmo: Dados os inteiros $N1$,$N2$,$N3$,$N4$ e $N5$, armazenar numa variável de nome **maior** o maior deles e numa variável de nome **menor** o menor deles.

Input: Os inteiros $N1$,$N2$,$N3$,$N4$,$N5$.

Output: O conteúdo da variável menor e da variável maior.

ALGORITMO 5.7

Algoritmo: Sejam l_1, l_2 e l_3 dados do tipo float representando o comprimento dos lados de um triângulo. A área desse triângulo pode ser calculada através da seguinte fórmula

$$area = \sqrt{p(p-l_1)(p-l_2)(p-l_3)}$$

onde p é o semi-perímetro do triângulo.

Input: Os lados do triângulo l_1,l_2,l_3.

Output: A área do triangulo.

ALGORITMO 5.8

Algoritmo: Determinar as raizes reais de uma equação do segundo grau.
Um equação do segundo grau é uma equação do tipo

$$ax^2 + bx + c = 0$$

onde a, b e c são constantes dadas.

Input: Os dados do tipo float a, b, c.

Output: Todos os possíveis valores da(s) raízes ou uma mensagem dizendo que a equação não tem raizes reais quando for o caso.

ALGORITMO 5.9

Algoritmo: Determinar **todas** as raizes da equação $ax^2 + bx + c = 0$, onde a, b e c são constantes dadas, e vamos supor que $a \neq 0$.
De acordo com a fórmula de Bhaskara as raizes são dadas por:

$$\text{Se } \Delta = b^2 - 4ac \geq 0 \begin{cases} x_1 = (-b + \sqrt{\Delta})/2a \\ x_2 = (-b - \sqrt{\Delta})/2a \end{cases}$$

$$\text{Se } \Delta < 0 \begin{cases} x_1 = (-b + i\sqrt{|\Delta|})/2a \\ x_2 = (-b - i\sqrt{|\Delta|})/2a \end{cases}$$

onde i é a chamada unidade imaginária, ou seja, $i^2 = -1 \therefore i = \sqrt{-1}$

Input: Os dados do tipo float a, b, c.

Output: Todas as raizes da equação.

ALGORITMO 5.10

Algoritmo: Dados os inteiros $N1$,$N2$ e $N3$, ordená-los de forma crescente.

Input: Os inteiros $N1$,$N2$,$N3$.

Output: Os números recebidos no input em ordem crescente.

ALGORITMO 5.11
Algoritmo: Dados os inteiros l_1,l_2 e l_3, onde $l_i > 0$.

a- Determine se esses dados podem representar os lados de um triângulo.

b- No caso afirmativo determine se esse triângulo é retângulo.

Input: Os dados l_1,l_2,l_3.

Output: Um texto informando as seguintes possíveis opções:
- Os dados não podem representar aos lados de um triângulo ou
- Os dados representam aos lados de um triângulo ou
- Os dados representam aos lados de um triângulo retângulo

ALGORITMO 5.12

Algoritmo: A seqüência de Fibonacci

O problema seguinte foi originalmente proposto por *Leonardo de Pisa*, também conhecido como **Fibonacci**, (filho de Bonacci) no século XVIII no seu livro *Liber Abaci*.
O problema pode ser formulado da seguinte maneira:
Suponha que um "casal" de coelhos jovens é colocado numa ilha.
Que um casal só se reproduz após completar 2 meses.
Que após a maturidade (2 meses) cada casal produz um novo casal a cada mês.
Deseja-se saber a quantidade de casais de coelhos haverá na ilha após n meses.
Obs: Estamos presupondo que não existem mortes.
Vamos denotar por F_n a quantidade de casais após n meses.
Observe que:

F_0=1 (começamos com um casal)

F_1=1 (o casal ainda é jovem para reproduzir)

F_2=2 (1 casal antigo + 1 novo)

F_3=3 (1 antigo+ 1 "amadurecendo" + 1 novo)

Pode ser provado que: $\begin{cases} F_0 = 1; F_1 = 1; \\ F_n = F_{n-1} + F_{n-2} \quad n \geq 2 \end{cases}$

Input: n do tipo int, representando os meses.

Output: A população de coelhos da ilha. (F_n)

ALGORITMO 5.13

Algoritmo: Determinar se um determinado ano, representado por um inteiro positivo N, é bissexto.
Um ano é bissexto se ele é um múltiplo de 400 ou se múltiplo 4 mas não de 100.

Input: Um inteiro N positivo representando um determinado ano.

Output: O ano recebido no input e uma mensagem indicando se ele é ou não bissexto.

ALGORITMO 5.14

Algoritmo: Determinar a soma de todos inteiros de 1 até um inteiro n positivo dado, ou seja

$$soma = \sum_{i=1}^{n} i = 1 + 2 + 3 + 4 + \cdots + n$$

Input: Um inteiro positivo n.

Output: A soma dos inteiros entre 1 o inteiro recebido como input.

ALGORITMO 5.15

Algoritmo: Dado um inteiro positivo n.
Determine o *fatorial* de n.

$$\begin{cases} fatorial = 1 & \text{se } n = 0, \\ fatorial = 1 \times 2 \times 3 \times \cdots \times n & \text{se } n > 0. \end{cases}$$

Input: Um inteiro positivo n.

Output: O fatorial do inteiro recebido como input.

ALGORITMO 5.16

Algoritmo: O objetivo do algoritmo a seguir é calcular a altura média de um grupo de pessoas com a seguinte sistemática:
Existe um digitador que é encarregado de digitar num computador a altura de cada pessoa do grupo.
Cada pessoa do grupo vai até a mesa do digitador e diz sua altura.
Quando todo grupo já tiver informado as respectivas alturas o digitador deverá informar à um supervisor a altura média do grupo.
Proponha um algoritmo para ajudar esse digitador. Lembre-se: digitador só sabe digitar.

Input: Dados do tipo float representando as alturas dos individuos.

Output: A altura média do grupo.
Obs: Supondo que foram "digitadas" m alturas então a altura média é dada por $altura\ media =$ (soma das alturas lidas)$/m$

ALGORITMO 5.17

Algoritmo: O problema a seguir simula um "computador" que calcula uma expressão aritmética que contem apenas uma operação.
A expressão será do tipo:dado operador dado
Os dados serão do tipo float e os operadores permitidos são:$+ - */$
O "computador" deve ser capaz de fazer as seguintes análises da expressão e emitir a respectiva mensagem:

a) Operador não válido, se o operador for diferente de $+ - * /$

b) Se a expressão estiver correta, executar a operação. (e.g. $2 * 3 = 6$);

c) Advertir para operações não permitidas (e.g. divisão por zero).

Input: Um float, um char, um float.

Output: Conforme discutido acima.

ALGORITMO 5.18

Algoritmo: O objetivo aqui é "ensinar ao computador" as operações aritméticas entre dados do tipo racional.
Um dado é do tipo racional se ele é da forma p/q, onde p e q são inteiros e $q \neq 0$.
Iremos então representar um dado racional como **int/int** e os operadores como **char**.
As operações seguem as regras já conhecidas, por exemplo:
$2/3 * 1/4 = 2/12$ não se preocupe com simplificações.
$1/3 + 1/4 = 7/12$.

Input: *int/int operador int/int*

Output: Conforme exemplificado acima.

ALGORITMO 5.19
Algoritmo: Dado um inteiro n. Determine 2^n (Não se esqueça do caso $n < 0$).

Input: Um inteiro n.

Output: O resultado 2^n.

ALGORITMO 5.20
Algoritmo: Dado um inteiro $n > 0$. Determine n^n.

Input: Um inteiro n.

Output: O resultado n^n.

ALGORITMO 5.21

Algoritmo: Uma das maneiras de "medir" eficiência de um algoritmo é contar a quantidade execuções de um determinado comando. Geralmente essa quantidade esta relacionada com algum parâmetro de input.
É evidente que medir "alguma coisa" significa comparar com algo já conhecido e determinado.
Podemos então comparar a eficiência de algoritmos do seguinte modo:
-Contamos as repetições de um determinado comando que ele necessita executar, para a solução de um problema.
-Comparamos esse número com a quantidade de repetições desse mesmo comando que um outro algoritmo executa na solução do mesmo problema.

Por exemplo: Dados x do tipo float e n do tipo int, podemos calcular x^n efetuando $n-1$ produtos. É claro que quanto maior o input n mais produtos devem ser efetuados.

O objetivo aqui é que você proponha um algoritmo que denominaremos "fast exponentiation" que execute a exponenciação, descrita acima, com menor número de produtos. E.g. para n=16 é possível realizar a tarefa com apenas 5 produtos.

Input: x do tipo float e n do tipo inteiro.

Output: - A quantidade de produtos efetuados
- O valor de x^n.

ALGORITMO 5.22

Algoritmo: Esse algoritmo tem por objetivo multiplicar inteiros sem, obviamente, utilizar o operador $(*)$. Estaremos assim "ensinando o computador" a multiplicar inteiros ou seja :
Dados n e m inteiros, determine $n \times m$.

Input: n e m do tipo int.

Output: O resultado $n \times m$.

ALGORITMO 5.23
Algoritmo: Algoritmo Euclideano - Algoritmo da Divisão

Euclides foi um matemático grego que viveu em 350 AC.
O algoritmo que discutiremos a seguir já era conhecido dos antigos chineses mas ficou conhecido como Euclideano em homenagem a Euclides que o incluiu na sua famosa obra ELEMENTOS.
O teorema de Euclides, que possibilita um algoritmo para a divisão de inteiros, afirma o que segue:
Dados m, n inteiros positivos, com $n \neq 0$.
Então existem únicos d e r inteiros, com $0 \leq r < n$ e satisfazendo $m = n \times q + r$.
O inteiro q é dito quociente da divisão inteira de m por n e r é o resto dessa divisão.
Dados n e m inteiros, utilize o teorema de Euclides para "ensinar ao computador" as operações: m/n e $m\%n$

i – Faça um algoritmo no qual você, caso necessite, poderá usar o operador $(*)$

ii – Faça outro algoritmo no qual você está proibido de usar o operador $(*)$

Input: n e m do tipo int.

Output: O resultado de m/n e o resto dessa divisão.

ALGORITMO 5.24

Algoritmo: **Descobrindo o dia da semana de uma data**

O objetivo desse problema é propor um algoritmo para, inicialmente, conhecendo-se uma data em um determinado mês e o dia da semana desta data, determinar qual será o dia da semana de uma outra data dentro do mesmo mês.

Podemos representar uma data através de 3 inteiros positivos, por exemplo: 10/7/1976 seria representado pelos inteiros $dia = 10, mes = 7$ e $ano = 1976$.

Podemos também representar os dias da semana da seguinte maneira:

Domingo= 0 ; Segunda-feira = 1 ; $\cdots$; Sábado= 6.

Sabendo-se, digamos, que hoje é 7/4, sábado desejamos saber em que dia da semana cairá o dia 27/4.

Sugestão: determine o número de dias entre as datas, calcule o resto da divisão por 7 e, ... pense um pouco.

Lembre-se que as possibilidades para o resto são: $0, 1, 2, 3, 4, 5, 6$

Input: Dois inteiros positivos *dia* e *dia_da_semana*:*dia* representando um dia de um determinado mês e *dia_da_semana* entre 0 e 6, representando o dia da semana desse tal dia.

Um inteiro *novo_dia* representando um outro dia do mesmo mês.

Output: O dia da semana que "cai"essa nova data.

ALGORITMO 5.25

Algoritmo: **Calculando a quantidade de dias entre duas datas**
O objetivo desse problema é determinar a quantidade de dias transcorridos entre duas datas.
Como no algoritmo anterior vamos representar cada uma das datas com três inteiros positivos representando dia, mês e ano.
Basta agora calcular a quantidade de dias lembrando-se dos seguintes fatos:

1) Cada ano tem 365 dias mas os bissextos têm 366.

2) Um mês pode ter 28, 29 30 ou 31 dias.

Input: *dia_old* *mes_old* *ano_old*
dia_new *mes_new* *ano_new*

Output: A quantidade de dias entre as datas.

IMPORTANTE

Você pode usar as idéias dos dois algoritmos anteriores para determinar em que dia da semana "caiu"uma determinada data sabendo-se que, por exemplo, 7 de abril de 2001 foi um sábado.
Na verdade podemos fazer o mesmo para datas futuras ou seja em que dia da semana "vai cair"uma determinada data.
Por curiosidade, descubra em que dia da semana foi descoberto o Brasil.

ALGORITMO 5.26

Algoritmo: Algoritmo de Eudoxo (408-355 AC)

Eudoxo foi um matemático grego que viveu no século IV antes de Cristo.

Seu método para calcular áreas e volumes denominado de **Método de Exaustão** influenciaria os matemáticos Newton, Leibniz e Riemann no desenvolvimento da análise infinitesimal e integral.

O algoritmo se propõe ao seguinte:

Dado um número $a > 1$, determinar um valor aproximado para $\sqrt{a}$.

Eudoxo propôs o seguinte Algoritmo.

passo 1 $\rightarrow$ $x_0 = a/2$

passo 2 $\rightarrow$ $x = (x_0 + a/x_0)/2$

passo 3 $\rightarrow$ $x_0 = x$

passo 4 $\rightarrow$ Volte ao passo 2

Pode ser provado que a cada etapa realizada o valor de x está cada vez mais próximo de $\sqrt{a}$.

Nosso objetivo agora é usar esse algoritmo para "ensinar o computador" calcular aproximações para raiz quadrada de números.

Use por exemplo 5 repetições e observe o quanto é boa a aproximação fazendo a seguinte comparação:

Se $x \approx \sqrt{a}$ então $x * x \approx a$

Na verdade esse é o algoritmo implementado na maioria das calculadoras por sua simplicidade e eficiência.

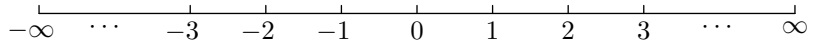
Input: a tipo float e n do tipo int (número de iterações)

Output: Uma aproximação para $\sqrt{a}$.

Capítulo 6

Aritmética Finita

No nosso dia a dia, até agora, fizemos uso da chamada *aritmética infinita* e suas peculiaridades. Uma dessas peculiaridades é que podemos representar um número inteiro por maior que ele seja. Representamos esse maior número inteiro pelo símbolo ∞ e o menor por $-\infty$.
Sabemos também que, dado um número n, seu *oposto*, *complemento* ou *simétrico* é definido como sendo o inteiro m tal que $n + m = 0$.
Este inteiro m, oposto de n, é representado por $-n$.
De um modo geral temos a seguinte representação dos inteiros e seus opostos:



Como veremos a seguir desde os primeiros computadores até os mais atuais utilizam a denominada *aritmética finita*. Essa aritmética é inevitável considerando-se a limitação, em termos de memória para os cálculos e armazenamentos, que todo equipamento eletrônico tem.

6.1 A Representação dos Inteiros

A unidade de armazenamento dos equipamentos eletrônicos é denominada *BIT* (BInary digiT) e representada por :

$\boxed{0}$ ou $\boxed{1}$

Os números inteiros positivos além de representar a si próprios serviram, desde os primeiros computadores, para identificar registros (posições de memória) além de servir para codificar as diretivas (comandos).

A unidade de armazenamento conduziu, de modo natural, os criadores dos computadores a adotar o *sistema binário* para representar os inteiros.
Foi comum, por um longo período, codificar as diretivas e os registros através de uma seqüência de 8 BITS.
Essa seqüência de **8 BITs** foi denominada **1 BYTE** ou **1 WORD** que foi o termo mais usado posteriormente.
Com a necessidade de diretivas cada vez mais complexas houve a necessidade da utilização de uma, ou mais, **WORDs** para codificá-las.
Durante um longo tempo os dados do tipo **Inteiro** foram representados utilizando-se **2 BYTEs=2 WORDs =16 BITs**.

6.2 A Representação Binária dos Inteiros

Do mesmo modo que o nosso conhecido **sistema decimal sistema binário** é um sistema posicional. Isso significa que um número inteiro positivo pode ser representado, no sistema decimal, como uma seqüência de dígitos onde o último digito à direita representa as unidades, o penúltimo as dezenas o antepenúltimo as centenas e assim por diante.
Usando a simbologia matemática podemos representar um inteiro positivo m no sistema decimal (também dito na base 10) como:

$$m = a_n a_{n-1} \cdots a_0 = a_n \times 10^n + a_{n-1} \times 10^{n-1} + \cdots + a_0 \times 10^0$$

onde $a_i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$.
Exemplo:
O inteiro $n = 3721$ pode ser representado por:
$n = 3 \times 10^3 + 7 \times 10^2 + 2 \times 10^1 + 1 \times 10^0$
$n = 3$ milhares + 7 centenas + 2 dezenas + 1 unidade

De maneira inteiramente análoga podemos representar os inteiros positivos no sistema binário (também dito na base 2) ou seja:

$$m = (a_n a_{n-1} \cdots a_0)_2 = a_n \times 2^n + a_{n-1} \times 2^{n-1} + \cdots + a_0 \times 2^0$$

onde $a_i \in \{0, 1\}$

Exemplo:
$(10101)_2 = 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 16 + 4 + 1 = 21$

6.3 Somando na Aritmética Binária

Como já foi dito, o sistema binário assim como o sistema decimal é um sistema posicional. Para somar números escritos como binários procedemos de modo análogo ao que fazemos no sistema decimal. Lembre-se que no sistema decimal adicionamos dois números da seguinte maneira:
somamos inicialmente os dígitos dos números que representam as unidades. Se essa quantidade ultrapassa uma dezena escrevemos o digito das unidades e somamos na coluna das dezenas o digito 1. Este procedimento é o conhecido "vai 1". Usamos o mesmo procedimento para a coluna das dezenas, centenas etc.
Vejamos um exemplo:

$$\begin{array}{cccccc} & {\scriptstyle 1} & {\scriptstyle 1} & {\scriptstyle 1} & \\ & 1 & 5 & 7 & 9 \\ + & 3 & 4 & 8 & 5 \\ \hline & 5 & 0 & 6 & 4 \end{array}$$

Para os números em binário basta observar o seguinte:

$$0 + 0 = 0$$

$$0 + 1 = 1$$

$$1 + 0 = 1$$

$1 + 1 = 2 = (10)_2$ ou seja 2 é o digito 0 e "vai 1".

Como o sistema binário é posicional, basta usar o mesmo procedimento que para os números decimais.
Vejamos um exemplo:

$$\begin{array}{cccccccccc} & {\scriptstyle 1} & {\scriptstyle 1} & & & & & & & \\ & 1 & 0 & 1 & 1 & 0 & 1 & \longrightarrow & 2^5 + 2^3 + 2^2 + 2^0 & = 45 \\ + & 0 & 1 & 1 & 0 & 1 & 0 & \longrightarrow & 2^4 + 2^3 + 2^1 & = 26 \\ \hline 1 & 0 & 0 & 0 & 1 & 1 & 1 & \longrightarrow & 2^6 + 2^2 + 2^1 + 2^0 & = 71 \end{array}$$

6.4 Multiplicando na Aritmética Binária

Novamente aqui nos "guiamos" pelo conhecimento já adquirido na multiplicação no sistema decimal. Como no caso da Adição basta observar que:

$$0 \times 0 = 0$$

$$0 \times 1 = 1$$

$1 \times 0 = 1$

$1 \times 1 = 1.$

Como o sistema binário é posicional, basta usar o mesmo procedimento que para os números decimais lembrando-se dos posicionamentos das unidades, dezenas etc. e da regra do "vai um" nas somas.
Vejamos um exemplo:

$$\begin{array}{r} 1\ 0\ 1\ 0 \\ \times\ 1\ 1\ 0\ 1 \\ \hline 1\ 0\ 1\ 0 \\ 0\ 0\ 0\ 0\ + \\ 1\ 0\ 1\ 0\ + \\ 1\ 0\ 1\ 0\ + \\ \hline 1\ 0\ 0\ 0\ 0\ 0\ 1\ 0 \end{array}$$

6.5 Os Inteiros na Aritmética Finita

Todos os computadores desde o mais potente *mainframe* até a mais simples calculadora digital utiliza um número finito e fixo de dígitos binários para representar um dado.
Essa limitação nos conduz à denominada aritmética finita.
É claro que é desejável, sempre que possível, que as regras e propriedades que são válidas na aritmética infinita continuem válidas na aritmética finita.
Algumas particularidades dessa aritmética finita ficam logo evidentes. Por exemplo: existe uma limitação para o maior número inteiro positivo e o menor negativo que podemos representar usando um número finito de BITs.

6.5.1 O maior inteiro usando 16 BITs

Para se representar um número inteiro foi adotada a seguinte estratégia:
Usar um BIT para representar o sinal do inteiro e os demais 15 para representar os digitos binários.

s	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1

Podemos adotar a convenção de representar os inteiros positivos colocando na posição 16 o BIT 0 e para os negativos o BIT 1 na posição 16.
Com essas considerações, qual deve ser o maior inteiro que podemos representar nessa aritmética? Obviamente esse número é aquele que tem o BIT

0 na posição 16 e o BIT 1 nas demais, ou seja

$$M = (0111111111111111)_2 = 1\times 2^{14} + 1\times 2^{13} + 1\times 2^{12} + \cdots + 1\times 2^1 + 1\times 2^0$$

Para determinar o valor de M "sem muito trabalho"vamos proceder da seguinte maneira:

$$M = 2^{14} + 2^{13} + 2^{12} + \cdots + 2^1 + 2^0 \tag{6.1}$$

$$2.M = 2^{15} + 2^{14} + 2^{13} + \cdots + 2^2 + 2^1 \tag{6.2}$$

Observe que as equações (6.1) e (6.2) tem os mesmos termos a menos do primeiro da segunda equação e do último da primeira.
Subtraindo 6.1 de 6.2 todos os termos serão cancelados a menos do termo 2^{15} e do termo $2^0 = 1$
Desse modo teremos $M = (2.M - M) = 2^{15} - 1 = 32767$ que é o maior inteiro que podemos representar usando **16 BITs**.

6.5.2 A Representação dos Inteiros Negativos em Binário

Vamos tentar, de um modo intuitivo e ingênuo, nos guiar através do nosso conhecimento da aritmética infinita para representar um inteiro negativo. Para facilitar o entendimento vamos supor que nossos números inteiros serão representados por apenas 3BITs.
Assim teremos os seguintes inteiros:
$(000)_2 = 0$
$(001)_2 = 1$
$(010)_2 = 2$
$(011)_2 = 3$
Lembrando que o BIT mais à esquerda representa o sinal do número, podemos fazer uma tentativa de representação dos inteiros negativos simplesmente trocando esse BIT ou seja:
$(101)_2 = -1$
$(110)_2 = -2$
$(111)_2 = -3$
Essa idéia de representação deve ser logo descartada pois como veremos a seguir a propriedade $m + (-m) = 0$ não é preservada.
Observe, por exemplo, que: $(001)_2 + (101)_2 = (110)_2 \neq (000)_2$.
Assim essa representação não nos serve.

Na tentativa de solucionar esse problema surgiu a o denominado *complemento* de um inteiro N representado por n dígitos binários. Esse complemento, representado por $2^n - N$ deveria satisfazer a equação:$(2^n - N) + N = 0$.
Foi proposto o seguinte algoritmo para achar o complemento de um número inteiro N representado em binário: Vamos supor que o número é representado do seguinte modo $N = (a_n a_{n-1} \cdots a_0)_2, a_i \in \{0, 1\}$

1– Construa um número binário L trocando todos os bits 1 por 0 e todos bits 0 por 1 na representação de N.

2– O complemento de N será dado por $L + 1$

Para exemplificar vamos achar o complemento do número $1 = (001)_2$ do nossa representação de 3BITs.

1– $N = (001)_2 \longrightarrow L = (110)_2$

2– O complemento de N será dado por $L + 1 = (110)_2 + (001)_2 = (111)_2$

Observe que nesse caso temos: $(001)_2 + (111)_2 = (000)_2$.
Esse complemento de N faz o papel na aritmética finita do $(-N)$ na aritmética infinita, sendo então uma representação para os inteiros negativos binários.

6.5.3 O menor inteiro usando 16 BITs

Vamos agora, à título de exercício, determinar o menor inteiro que pode ser representado com 16 BITs. É claro que esse número que denotaremos $(-M)$ é o complemento do maior inteiro positivo M.
$M = (0111111111111111)_2 \rightarrow L = (1000000000000000)_2$
$L + 1 = (1000000000000001)_2 = 1 \times 2^{15} + 1 \times 2^0 = 2^{15} + 1 = 32768$
Assim o menor inteiro é $(-M) = -32768$

6.6 Os Números Reais (float)

A representação dos números do tipo **float** na aritmética finita está baseada na seguinte representação dos números reais:
Todo $x \in R$, $x \geq 0$ pode ser representado por:

$$x = m \times 10^e \quad \text{com} \;\; 0 \leq m < 1$$

Essa forma é conhecida como representação exponencial do número na forma normalizada.
m é dito **mantissa** de x
e é dito **expoente** de x
Vejamos alguns exemplos:
$33.25 = 0.3325 \times 10^2$
$0.1284 = 0.1284 \times 10^0$
$0.00765 = 0.765 \times 10^{-2}$
Essa representação como veremos a seguir é muito adequada para representar os dados do tipo **float** na aritmética finita.

6.6.1 O maior float usando 3 Words

Nos tempos do inicio dos computadores e das linguagens de alto nível foi muito comum a seguinte estratégia para armazenar os dados do tipo float.
Usa-se uma seqüência de 3 **WORDs** contíguas para o armazenamento da seguinte maneira:

s	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	s	7	6	5	4	3	2	1
←					MANTISSA										→	←			EXPOENTE				→

A mantissa com seu sinal ocupam 16 BITs e o expoente e seu sinal os 8 BITs restantes.
Na aritmética binária esses 24 BITs são obviamente ocupados com zeros e uns.
Vamos agora determinar o maior dado do tipo **float** que podemos armazenar nessa representação.
Como a mantissa m é sempre menor do que 1 na representação exponencial normalizada a grandeza do número será dada pela grandeza de seu expoente.
Podemos estimar o expoente máximo da seguinte maneira:

0	1	1	1	1	1	1	1

$\rightarrow\ 2^0+2^1+2^2+2^3+2^4+2^5+2^6+2^7 = 2^7-1 = 127$

Assim o maior float usando 3 Words é da ordem de $2^{127} \approx 10^{38}$.
O próximo "desafio"para os pioneiros da computação era agora representar a mantissa em binário.
Ou seja dado um número m, $0 \leq m < 1$ representá-lo em binário. Como sempre a pista nos é dada pelo nosso conhecimento do sistema decimal.

É sabido que um número x do tipo real, $0 \leq m < 1$ pode ser escrito como

$$x = \frac{a_1}{10} + \frac{a_2}{10^2} + \cdots + \frac{a_n}{10^n} \quad \text{com } a_i \in \{0, 1, \ldots, 9\} \text{ e } a_1 \neq 0$$

Por exemplo
$x = 0.3721 = \frac{3}{10} + \frac{7}{10^2} + \frac{2}{10^3} + \frac{1}{10^4}$

Em binário usamos a seguinte representação

$$x = \frac{a_1}{2} + \frac{a_2}{2^2} + \cdots + \frac{a_n}{2^n} \quad \text{com } a_i \in \{0, 1\}$$

Exemplo
$x = (.1011)_2 = 1 \times \frac{1}{2} + 0 \times \frac{1}{2^2} + 1 \times \frac{1}{2^3} + 1 \times \frac{1}{2^4} + 0 \times \frac{1}{2^5} = \frac{11}{16} = .6875$

A transformação de decimal para binário é feita do seguinte modo:

$x = .6875$
$floor(.6875 \times 2^1)\%2 = 1$
$floor(.6875 \times 2^2)\%2 = 0$
$floor(.6875 \times 2^3)\%2 = 1$
$floor(.6875 \times 2^4)\%2 = 1 \quad \longrightarrow \quad x = (.1011)_2$

Capítulo 7

Arrays e Estruturas

7.1 Seqüências - Arrays Unidimensionais

Com freqüência em matemática e em inúmeros algoritmos temos a necessidade de utilizar uma seqüência de dados do mesmo tipo.
É desejável que possamos de alguma maneira identificar essa seqüência de dados através de seus índices.
Em Matemática essa **estrutura** é fornecida pelas chamadas seqüências, que são definidas como :
$x : N \longmapsto R$
$\quad i \longrightarrow x_i$
Temos desse modo uma seqüência de elementos dada por:
$x_0, x_1, x_2, \ldots, x_n, \ldots$

Exemplo 7.1.1

$x_i = 2i$
Temos nesse caso a seguinte seqüência: $0, 2, 4, 6, \ldots$
Observe que podemos referenciar qualquer elemento da seqüência através de seu índice. Por exemplo o elemento de índice $i = 200$ é $x_{200} = 400$. ▲

No caso dos computadores veremos que essa estrutura representa um papel fundamental, na verdade ela é parte intrínseca de qualquer computador. De fato a memória de um computador pode ser representada como uma seqüência.
Cada elemento dessa seqüência representaria uma posição de memória e seu respectivo endereço.

7.2 Definindo Seqüências

Na linguagem C as seqüências, que também são denominadas **arrays unidimensionais**, são definidas através da seguinte sintaxe:

Tipo Identificador[tamanho];

$$\begin{cases} \text{Tipo:} & \text{O tipo dos elementos da seqüência (int,float ou char)} \\ \text{Identificador:} & \text{O nome da seqüência} \\ \text{tamanho:} & \text{A quantidade de elementos da seqüência} \end{cases}$$

Os elementos da seqüência são referenciados através de seu índice, ou seja **Identificador[i]** é um elemento da seqüência com $i = 0, 1, \ldots, tamanho - 1$

Exemplo 7.2.1

int $a[10]$
Seqüência com 10 elementos do tipo inteiro.
Os elementos são: $a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7], a[8], a[9]$

índice	i	0	1	2	3	4	5	6	7	8	9
elemento	$a[i]$	$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$	$a[5]$	$a[6]$	$a[7]$	$a[8]$	$a[9]$

▲

Exemplo 7.2.2

char vogal[4] // Seq\"{u}\^{e}ncia com 5 elementos do tipo char.
Os elementos são: vogal [0], vogal [1], vogal [2], vogal [3], vogal [4]

▲

7.3 Inicializando Seqüências

(i) Tipo Identificador[tamanho]=dado_Tipo
Todos elementos da seqüência serão inicializados com o valor dado_Tipo.
É claro que o tipo do identificador e o tipo do dado devem ser os mesmos.

exemplo: int vet[5]=0.
Após essa inicialização teremos os seguintes valores para os elementos da seqüência vet
vet[0]=0, vet[1]=0, vet[2]=0, vet[3]=0, vet[4]=0.

(ii) int digito[ ]={0,1,2,3,4,5,6,7,8,9}
Teremos aqui as seguintes atribuições de valores:
digito[0]=0, digito[1]=1, digito[2]=2, ... , etc.

(iii) char vogal[ ]={'a','e','i','o','u'}
vogal[0]='a', vogal[1]='e', vogal[2]='i', vogal[3]='o', vogal[4]='u'.

7.4 Strings

Strings são seqüências de caracteres ou seja seqüência de dados do tipo char.
Podemos usar a seguinte sintaxe para sua inicialização:
char fruta[ ]="laranja";
Após a atribuição acima teremos:

fruta[0]='l'
fruta[1]='a'
fruta[2]='r'
fruta[3]='a'
fruta[4]='n'
fruta[5]='j'
fruta[6]='a'
fruta[7]='\0' (fim da string);

Observe que o fim da string é representado pelo símbolo \0

Exemplo 7.4.1

Vamos fazer um trecho de algoritmo para calcular o comprimento de uma string dada.

```
// Calculando o comprimento de uma String
   int i=0;
   char str[ ]="Essa e uma string de comprimento ainda nao conhecido";
   while ( str[i] != '\0' ) i = i + 1;
   write ("O comprimento da string eh = " , i);
```

Output→ O comprimento da string eh=52 ▲

7.5 Entrada/Saida de Strings via Console

Visando facilitar a entrada e saída de dados do tipo String, via console, a linguagem fornece as seguintes diretivas para esses fins.

Exige-se a inclusão da biblioteca:
<stdio.h> (**st**an**d**ard **i**nput **o**utput)

Saída **Sintaxe:** puts(s);

Significado: "Imprime "no video a string s

Entrada **Sintaxe:** gets(s);

Significado: "Recebe via teclado "a string s

Importante: O final da entrada da string é determinado pela digitação de Enter

Exemplo 7.5.1

```
// Entrada e Sa\'{\i}da de Strings
#include <stdio.h>
 char nome[30];
 gets(nome);
 puts(nome);
```

Input:Maria da Silva Xavier enter

Output:Maria da Silva Xavier
▲

7.6 Funções para manipulação de Strings

Abordaremos a seguir algumas funções pré-definidas que nos auxiliam a manipular dados do tipo String.

Para a utilização dessas funções é exigida a inclusão da biblioteca $< string.h >$

- **strlen**(string length - comprimento de uma string)
- **strcpy**(string copy - copia uma string)
- **strcmp**(string compare - comparação de strings)

Vamos denotar por S o conjunto das strings.

Função	Definição	Ação
$strlen : S \longmapsto int$	$strlen(s)$	comprimento da string s
$strcpy : S \times S \longmapsto S$	$strcpy(s1, s2)$	copia a string s2 na string s1
$strcmp : S \times S \longmapsto int$		$strcmp(s1, s2) = 0$ se $s1 = s2$ $strcmp(s1, s2) < 0$ se $s1 < s2$ $strcmp(s1, s2) > 0$ se $s1 < s2$

Exemplo 7.6.1

```
// Uso das funcoes strlen ( ), strcmp( ) e strcpy ( )
#include <iostream.h>
int l,n,m,k=0;
   char s1[ ]="abacate";
```

```
char s2[ ]="zabumba";
char s3[ ]="aqui ";
l=strlen(s1 ); n=strlen(s2 ); m=strlen(s3);
k=strcmp(s1,s2);
strcpy(s3,"uma string qualquer");
printf("%i\n",l); printf("%i\n",n);
printf("%i\n",m); printf("%i\n",k);
printf("%s\n",s3);
```

Output:7
7
5
-25
uma string qualquer

▲

7.7 Saida via Console Formatada

No sentido de facilitar e permitir que a saída de dados, via console, seja mais flexível detalharemos a seguir as diretivas necessárias para esse fim.
Essas diretivas que veremos a seguir exigem a inclusão da biblioteca

<stdio.h> (**st**an**d**ard **i**nput **o**utput)

Sintaxe: printf("string_de_controle", lista_de_dados);
Significado:

string_de_controle – caracteres de controle separados por vírgulas cujo significado é dado na tabela 7.1

lista_de_dados – os dados de entrada separados por vírgulas

Caracteres	Significado
%i	um inteiro no formato decimal
%md	é reservado um campo de tamanho m para o inteiro
%-md	como antes só que o sinal (-) significa alinhamento à esquerda
%f	um float no formato decimal
%m.nf	float num campo de tamanho m e n decimais
%-m.nf	float num campo de tamanho m e n decimais alinhado à esquerda
%c	um caracter
%mc	caracter num campo de tamanho m
%-mc	caracter num campo de tamanho m alinhado à esquerda
%s	uma string
%ms	string num campo de tamanho m
%-ms	string num campo de tamanho m alinhado à esquerda

Tabela 7.1: Strings de Controle

A **quantidade** e o **tipo** dos dados da lista_de_dados no comando **fprint()** deve necessariamente corresponder com os respectivos caracteres de controle da string_de_controle **sob pena de erro de execução não previsível**.

String	Significado
$\backslash n$	mudança de linha
$\backslash t$	tabulação horizontal
$\backslash b$	backspace
$\backslash r$	carriage return (impressora)
$\backslash f$	muda folha de impressão (impressora)

Tabela 7.2: Códigos de Barra Invertida

Exemplo 7.7.1

```
// Saida Formatada (String)
#include <stdio.h>
 char nome[40]="Antonio";
 char profissao[30]="Engenheiro";
  printf("%s%s" , nome , profissao);
  printf("\n");
  printf("%15s%15s" , nome , profissao);
```

```
 printf("\n");
 printf("%-15s%-15s" , nome , profissao);
 printf("\n");
```

Output:AntonioEngenheiro
Antonio Engenheiro
Antonio Engenheiro

▲

Exemplo 7.7.2

```
// Saida Formatada
#include <stdio.h> int n;
 n=65478;
 printf("%i",n);
 printf("\n");
 printf("%2d",n);
 printf("\n");
 printf("%8d",n);
 printf("\n");
 printf("%-8d",n);
```

Output:65478
65478
65478
65478

▲

Exemplo 7.7.3

```
// Saida Formatada
#include <stdio.h> float x=12.9832;
printf("%7.4f",x);
 printf("\n");
 printf("%10.4f",x);
 printf("\n");
 printf("%-15.6f",x);
```

Output:12.9832
12.9832
12.983200

▲

Exemplo 7.7.4

```
// Entrada e Saida Formatadas
#include <stdio.h>
 char nome[30];
  int n;
 printf("Entre com o nome : ");
 gets(nome);
 printf("Entre com sua idade : ");
 scanf("%i" , &n);
 printf("%s%s%s%i%s" , "Seu nome eh " , nome, "\nVoce tem " , n , " anos");
```

Input:Entre com o nome : Antonio da Silva enter
Entre com sua idade : 30 enter

Output:Seu nome eh Antonio da Silva
Voce tem 30 anos

▲

7.8 Observações sobre a Entrada de Dados

A entrada de dados na linguagem *C* pode se tornar, para um iniciante, uma fonte constante de erros de execução.
Vamos a seguir mostrar como funciona o **Buffer de Input** no compilador *C*. Isso com certeza ajudará o usuário não cometer, ou descobrir a causa, de diversos erros na entrada de dados.

Leitura de Strings:

(I)

```
// Leitura
char nome[20];
i=0;
scanf("%c",&nome[i]);
while (nome[i] !='\n') scanf("%c",&nome[++i]);
//Impressao
i=-1;
while(nome[++i] != '\0') printf("%c",nome[i]);
```

O trecho de programa acima pode ser usado para **ler** e **imprimir** uma string que cuja entrada é terminada por um enter

Repetindo o trecho de leitura o **buffer de input** pode ser visualizado por:

s_1	s_2	s_3	$\cdots$	s_n
$\uparrow$	$\uparrow$	$\uparrow$	$\cdots$	$\uparrow$

(II)

```
// Leitura
char nome[20];
scanf("%s",&nome);
```

O trecho acima lê uma string que é encerrada por um **espaço**.

Assim não serve para a leitura de uma string que contenha um **espaço**, e.g. Antonio Silva.

(III)

```
// Leitura
char nome[20];
//Leitura
gets(nome);
//Impressao
puts(nome);
```

O trecho acima lê uma string que é encerrada por um enter

Após a impressão da string sempre mudara de linha. Repetindo o trecho de leitura o **buffer de input** pode ser visualizado como:

s_1	$\backslash n$	s_2	$\backslash n$	$\cdots$	s_n	$\backslash n$
	$\uparrow$		$\uparrow$			$\uparrow$

Leitura de Dados Numéricos:

```
int i , j ;
scanf("%d%d",&i,&j);
```

O trecho acima lê dois dados do tipo inteiro.

A entrada de um dado do tipo numérico (**int** ou **float**) é encerrada por um enter

Uma coleção de dados numéricos pode ser lida como uma única entrada em que os dados são separados por **espaço** finalizando com enter

Nesse caso o **buffer de input** pode ser visualizado como:

m_1		m_2		$\cdots$	m_n	$\backslash n$
	↑		↑			↑

Leitura de Dado Numérico e String:

```
int i , j ;
char nome[20];
scanf("%d%s",&i,&nome);
```

O trecho acima lê um dado tipo inteiro e uma string.

Nesse caso o **buffer de input** pode ser visualizado como:

m_1	$\backslash n$	s_1	$\backslash n$
	↑		↑

Observe que nesse caso após a leitura do inteiro o **ponteiro do buffer** fica posicionado no caracter $\backslash n$. Assim quando a string for lida a ela será atribuído o caracter $\backslash n$ e não a string representando **nome** como seria a nossa intenção.
O trecho de programa a seguir é uma solução para esse problema.

```
int i;
char lixo;
char nome[20];
scanf("%d%c%s",&i,&lixo,&nome);
```

Observe agora que após a leitura do inteiro será lido um caracter (no caso teremos lixo=$\backslash n$) e em seguida a string será lida corretamente.

Leitura String e Dado Numérico:

```
int i , j ;
char nome[20];
gets(nome);
scanf("%d",&i);
```

O trecho acima lê uma string e em seguida um dado tipo inteiro.

Nesse caso o **buffer de input** pode ser visualizado como:

s_1	$\backslash n$	m_1	$\backslash n$
	$\uparrow$		$\uparrow$

Observe que nesse caso após a leitura da String o **ponteiro do buffer** fica posicionado no caracter $\backslash n$. Como o próximo dado a ser lido é numérico ele será lido corretamente pois como já foi visto anteriormente os dados numéricos são separados por $\backslash n$ no **buffer de input**.

Resumindo...

- Para leitura de strings use sempre **gets()**;
- Se após a leitura **dado numérico** tiver que ser lida uma string nunca se esqueça de **acertar a leitura** ou seja ler antes o caracter $\backslash n$;
- **NUNCA SE ESQUEÇA DO CARACTER &** no comando scanf.

7.9 Arrays Multi-dimensionais

Na linguagem *C* as estruturas que generalizam os arrays unidimensionais são os **Arrays Multi-dimensionais** ou **Matrizes**. Seus equivalentes de

ordem 2 na matemática são as conhecidas matrizes.
Como no caso unidimensional os arrays multi-dimensionais são seqüências de dados do mesmo tipo. Os **arrays multi-dimensionais**, são definidos através da seguinte sintaxe:

Tipo Identificador$[tamanho_1][tamanho_2][tamanho_3]\cdots[tamanho_n]$;

$$\begin{cases} \text{Tipo:} & \text{O tipo dos elementos do array (int,float ou char)} \\ \text{Identificador:} & \text{O nome do array} \\ tamanho_i\text{:} & \text{A quantidade de elementos do i-ésimo índice do array} \end{cases}$$

Os elementos do array são referenciados através de seus índices, ou seja **Identificador[i][j]...[n]** é um elemento do array onde:
$i = 0, 1, \ldots, (tamanho_1 - 1)$
$j = 0, 1, \ldots, (tamanho_2 - 1)$
.....
$n = 0, 1, \ldots, (tamanho_n - 1)$

Exemplo 7.9.1

int mat[10][5];
Nesse caso um array de **inteiros** cujo nome é **mat** e que tem 50 **elementos**.
O primeiro índice do array deve variar de 0 até 9
O segundo índice do array deve variar de 0 até 4
Assim são elementos do array: mat[2][4];mat[0][4];mat[9][4]

Caso os índices não estejam dentro dos limites da definição ao elemento correspondente à esses índices será atribuído um valor imprevisível (lixo). Cabe ao programador cuidar desses limites.
No exemplo acima se tentarmos "usar", por exemplo, o elemento mat[8][5] com certeza teremos problemas.

Exemplo 7.9.2

char linha[10][80];
Nesse caso temos um array de **caracteres** cujo nome é **linha** e que tem 800 **elementos**.

Exemplo 7.9.3

O programa abaixo imprime no vídeo a tabela das multiplicações("tabuadas").

```
#include <stdio.h>
// Tabela de Multiplica\c{c}\~{a}o
int main() {
 int mult[10][10];
 int i,j;
 for (i=1 ; i<=9 ; i++)
  for (j=1 ; j<=9 ; j++)
  mult[i][j]=i*j;
 for (i=1 ; i<=9 ; i++){
  for (j=1 ; j<=9 ; j++)
     printf("%3d" , mult[i][j]);
   printf("\n");
  }
return(0);
}
```

A saída será a seguinte:

```
 1   2   3   4   5   6   7   8   9
 2   4   6   8  10  12  14  16  18
 3   6   9  12  15  18  21  24  27
 4   8  12  16  20  24  28  32  36
 5  10  15  20  25  30  35  40  45
 6  12  18  24  30  36  42  48  54
 7  14  21  28  35  42  49  56  63
 8  16  24  32  40  48  56  64  72
 9  18  27  36  45  54  63  72  81
```

▲

Exemplo 7.9.4

O programa abaixo tem por objetivo calcular as médias de uma turma de 3 alunos.
Ele terá como input o nome de cada aluno e suas respectivas notas (p1,p2,p3).
Como saída devemos ter os nomes as notas e as medias dos alunos da turma.

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

```
// Medias de uma turma de 3 alunos
int main() {
 char nome[3][30]; // Nome ter\'{a} no m\'{a}ximo 30 caracteres
 float p1 [3] , p2 [3] , p3 [3] , media[3];
 int i , j , k;
 char lixo;
 printf("Entrada de Nomes e Notas \n");
 for (i=0 ; i<3 ; i++){
  gets(nome[i]);// lendo nomes
  scanf("%f%f%f" , &p1[i] , &p2[i] , &p3[i ]); // lendo notas
  scanf("%c",&lixo); // Acertando a leitura
  }
 for (i=0 ; i<3 ; i++)
 media[i]=(p1[i] + p2[i] + p3[i ])/3; // c\'{a}lculo das m\'{e}dias
 printf("%-20s %-s %-s %-s %-s %-s %-s",
        "Nome", "\t" ,"P1\t" ,"P2\t" ,"P3\t" ,"Media\t" ,"\n");
 for (i=0;i<2;i++)
 printf("%-20s %s %-4.1f %-s %-4.1f %-s %-4.1f %-s %-4.1f %-s",
        nome[i],"\t",p1[i ],"\t",p2[i ],"\t",p3[i ],"\t",media[i ],"\n")
return(0);
}
```

Input:

Mariana Freitas enter

8.5 9.5 7.7 enter

Bruno Freitas enter

9.0 7.9 8.5 enter

Joaquim Silva enter

5.3 4.2 3.5 enter

*

Output:

Nome	P1	P2	P3	Media
Mariana Freitas	8.5	9.5	7.7	8.6
Bruno Freitas	9.0	7.9	8.5	8.5
Joaquim Silva	5.3	4.2	3.5	4.3

▲

7.10 O Usuário pode definir Tipos

Até agora só nos foi possível utilizar dados dos chamados tipos básicos, ou seja, **int**, **float** e **char**. A diretiva a seguir vai nos permitir definir outros tipos de dados.

Sintaxe:

typedef tipo identificador;

typedef tipo identificador[const];

Exemplo 7.10.1
typedef int inteiro ;
inteiro i , j , k ;
Com as diretivas acima foram criados:
- Um tipo denominado inteiro, cujo tipo é **int**;
- As variáveis i , j , k do tipo inteiro.

Exemplo 7.10.2
typedef char string10[10];
string10 x,y,z ;
Com as diretivas acima foram criados:
- Um tipo denominado string10, cujo tipo é uma string de 10 elementos;
- As variáveis x , y ,z do tipo string10.

Exemplo 7.10.3
typedef int matriz5x5[5][5];
matriz5x5 a , b;
Com as diretivas acima foram criados:
- Um tipo denominado matriz5x5, cujo tipo é um array bi-dimensional de 25 elementos;
- As variáveis a , b do tipo matriz5x5.

7.11 Estruturas - Struct

A estrutura de Array nos deu a liberdade de utilizar coleções de elementos de **apenas um** dos *tipos básicos*.
A estrutura que veremos a seguir nos possibilitará definir dados cujo tipo poderá ser uma composição dos tipos básicos.

Sintaxe:

```
struct nome_da_estrutura {
        tipo_basico_1       nome_1 ;
        tipo_basico_2       nome_2 ;
        ..................................... ;
        tipo_basico_n       nome_n ;
};
```

Para criar uma variável do tipo acima usamos a sintaxe:

struct nome_da_estrutura nome_var;

Com a diretiva acima é criada uma variável cujo nome é **nome_var** do tipo **nome_da_estrutura**.
Essa variável é composta das seguintes partes (membros):

nome_var.nome_1 (uma variável do tipo tipo_basico_1)
nome_var.nome_2 (uma variável do tipo tipo_basico_2)
.....
nome_var.nome_n (uma variável do tipo tipo_basico_n)

Exemplo 7.11.1

```
struct registro {
        char nome[30];
        int dia_nasc;
        int mes_nasc;
        int ano_nasc;
};
struct registro pessoa;
```

Foram assim criados:

- Um tipo chamado registro;

- Uma variável chamada pessoa.

Os elementos ou variáveis da estrutura são denominados **membros da estrutura**.
No exemplo anterior **membros da estrutura** são os seguintes:

pessoa.nome;

pessoa.dia_nasc;

pessoa.mes_nasc;

pessoa.ano_nasc;

Podemos usar, por exemplo, os seguintes comandos para esses membros:

gets(pessoa.nome);

pessoa.dia_nasc=10;

pessoa.mes_nasc=12;

pessoa.ano_nasc=2000;

Geralmente usa-se a seguinte forma compacta das diretivas acima:

```
struct registro {
        char nome[30];
        int dia_nasc;
        int mes_nasc;
        int ano_nasc;
} pessoa ;
```

Inicialização:

```
struct registro pessoa ={
        "Antonio da Siva",
        21,
        6,
        1975};
```

Exemplo 7.11.2

```
struct reg_alunos {
        char nome[30];
        float p1;
        float p2;
        float media;
} aluno[50] ;
```

Nesse exemplo foi criada uma estrutura do tipo **reg_alunos** e uma variável do tipo array chamada **aluno** com no máximo **50 membros do tipo reg_alunos**.
Podemos por exemplo fazer referencia a:
aluno[10].nome = uma string com no máximo 30 caracteres representando o nome do décimo aluno numa turma.
aluno[5].p1 = um float representando uma nota do quinto aluno numa turma.
aluno[45].media = um float representando a média do quadragésimo quinto aluno numa turma.

Exemplo 7.11.3

```
int main(){
 struct registros_dos_alunos {
    char nome[30];
    int  faltas ;
    float  p;
} aluno = {"Antonio Carlos",10,7.7};
printf("Nome=%s faltas=%d  nota=%4.1f",aluno.nome,aluno.faltas,aluno.p);
}
```

O output desse programa será:
Nome=Antonio Carlos faltas=10 nota= 7.7

Exemplo 7.11.4
O sistema de numeração usado pelos romanos usava os mesmos símbolos que a sua escrita. Ele era um pouco diferente do que atualmente conhecemos como **sistema romano**. Até hoje esse sistema tem algumas utilizações como por exemplo: relógios, capítulos de livros, datas de copyright, nomes de reis e papas etc.
O programa a seguir faz a conversão de um número romano (válido) entre 1 e 3999 para seu equivalente decimal.
A equivalência dos símbolos é dada por:

Símbolo	I	V	X	L	C	D	M
Valor	1	5	10	50	100	500	1000

Tabela 7.3: Valores dos Símbolos Numéricos Romanos

A regra para determinar o equivalente decimal de uma número romano válido consiste em somar os valores dos símbolos.
O valor de um símbolo é determinado da seguinte maneira:

a) O valor de um símbolo é aquele da tabela 7.3 se o valor do símbolo sucessor for menor do que a do símbolo.

b) Se o valor do símbolo sucessor for maior que o do símbolo devemos atribuir à esse par (símbolo sucessor) a diferença: valor do sucessor menos valor do símbolo.

```
#include <string.h>
#include <stdio.h>
// Programa Romano\_to\_Decimal
int main(){
struct roman_number {
    char letter;
    int value;
};
//
static struct roman_number roman[] = {
    {'I', 1},
    {'V', 5},
    {'X', 10},
    {'L', 50},
    {'C', 100},
    {'D', 500},
    {'M', 1000},
```

```
    {'\0',0},
};
int num=0,i=-1,j=0;
char romano[10];
int idx[10];
printf("Entre com um numero romano: ");
gets(romano);
 while (romano[++i] != '\0'){
  j=0;
  // Procurando os simbolos no numero romano
  while ( (roman[j].value > 0) && ( roman[j].letter != romano[i] ) ) j++;
  idx[i] = roman[j].value;
 }
 //
j=0;
idx[strlen(romano)]=0;
while(j <= strlen(romano)){
    if (idx[j]< idx[j+1]){
      num = num + (idx[j+1] - idx[j]);
      j++;
    }
  else num = num + idx[j];
 j++;
}
//
j=0;
// Esse trecho imprime o resultado no formato desejado
    while (j <= strlen(romano)) printf("%c",romano[j++]);
    printf("%s %d"," = ",num);
return(0);
}
```

Input:
MMCMXLIX enter

Output:
MMCMXLIX=2949

Exemplo 7.11.5

```
#include <string.h>
#include <stdio.h>
int main(){
struct registros_dos_alunos {
     char nome[30];
     int  faltas ;
     float  p1;
     float  p2;
     float  p3;
     float  media;
     char mensagem[10];
};
struct registros_dos_alunos  aluno[50];// Definimos uma sequencia de, no maximo,
                                       // 50  variaveis  do tipo  registro_de_alunos
int  i=0 , qtde;
char lixo;
printf("%s","Entre com o nome do aluno (ou * para encerrar)\n");
printf("%s","Em seguida entre com as notas (p1 p2 p3) e faltas\n");
gets(aluno[0].nome);
while (strcmp(aluno[i].nome,"*") != 0){
  scanf("%f%f%f%d",&aluno[i].p1 , &aluno[i].p2 , &aluno[i].p3 , &aluno[i].faltas  );
  scanf("%c",&lixo); // Acertando a leitura
  gets(aluno[++i].nome);
}
qtde=i-1;// quantidade de alunos.
//Calculando as medias e atribuindo a mensagem
for (i=0 ; i<= qtde ; i++){
  aluno[i].media=(aluno[i].p1 + aluno[i].p2 + aluno[i].p3)/3;
  if (aluno[i].media >= 5.0 ) strcpy(aluno[i].mensagem,"Aprovado");
  else strcpy(aluno[i].mensagem,"Reprovado");
}
printf("%-20s %-s %-s %-s %-s %-s %-s %-s %-s",
       "Nome", "\t" ,"P1\t" ,"P2\t" ,"P3\t" ,"Faltas\t","Media\t" ,"MSG\t" ,"\n");
for (i=0;i<=qtde;i++)
printf("%-20s %s %-4.1f %-s %-4.1f %-s %-4.1f %-s %-3d %-s %-4.1f
         %-s %-s %-s",
        aluno[i].nome,"\t",aluno[i].p1,"\t",aluno[i].p2,"\t",aluno[i].p3,"\t",
        aluno[i].faltas ,"\t",aluno[i].media,"\t",aluno[i].mensagem,"\t","\n");

 return(0);
}
```

Input:
Mariana Freitas enter

8.5 9.5 7.7 5 enter

Bruno Freitas enter

9.0 7.9 8.5 2 enter

Joaquim Silva enter

5.3 4.2 3.5 15 enter

*

Output:

Nome	P1	P2	P3	Faltas	Media	MSG
Mariana Freitas	8.5	9.5	7.7	5	8.6	Aprovado
Bruno Freitas	9.0	7.9	8.5	2	8.5	Aprovado
Joaquim Silva	5.3	4.2	3.5	15	4.3	Reprovado

Capítulo 8

Algoritmos Propostos (II)

ALGORITMO 8.1

Algoritmo: Calcule a media de um conjunto de dados e determine quantos desses dados estão acima dessa média e quantos estão abaixo.

Input: - Um inteiro representando a quantidade de dados.
- Os dados.

Output: - A média dos dados.
- A quantidade de dados superiores à média.
- A quantidade de dados inferiores à média.

ALGORITMO 8.2

Algoritmo: Dadas duas strings s1 e s2 onde inicialmente s2 é uma string vazia "atribua o valor de s1 à s2"ou seja "copie" a string s1 em s2.

Input: Uma string s1.

Output: A string s2.

ALGORITMO 8.3

Algoritmo: Dada uma string s contendo o caracter '/' construa uma substring z de s que não contenha o referido caracter.

Input: Uma string s contendo o caracter '/'.

Output: A string z.

ALGORITMO 8.4

Algoritmo: Dado um inteiro menor ou igual à 999999, determine quantas unidades, dezenas, centenas, etc. tem esse número.

Input: - Um inteiro menor ou igual à 999999.

Output: - quantidade de unidades.
- quantidade de dezenas.
- quantidade de centenas.

⋮ ⋮

ALGORITMO 8.5

Algoritmo: Dado um inteiro positivo na base 10 determine seu correspondente em binário (base 2).

Input: Um inteiro.

Output: Um string do tipo (010111)2 que corresponde ao inteiro fornecido.

ALGORITMO 8.6

Algoritmo: Dada uma string representando um inteiro positivo na base 2 usando 16 Bits, determine seu correspondente na base 10.

Input: Uma string usando 16Bits.

Output: O inteiro correspondente na base 10.

ALGORITMO 8.7

Algoritmo: Dada um inteiro positivo menor do que 32767, determine usando a aritmética finita de 16 Bits, seu correspondente negativo em binário.

Input: Um inteiro positivo.

Output: O inteiro negativo correspondente em binário.

ALGORITMO 8.8

Algoritmo: Dada uma string representando um float $0 \leq x < 1$ na base 2 na aritmética finita de 16 Bits, determine seu correspondente na base 10.

Input: Uma string representando um float $0 \leq x < 1$ na base 2.

Output: O float correspondente na base 10.

ALGORITMO 8.9

Algoritmo: Dado um float $0 \leq x < 1$, determine usando a aritmética finita de 16 Bits, seu correspondente em binário.

Input: Um float $0 \leq x < 1$.

Output: O seu correspondente em binário.

ALGORITMO 8.10

Algoritmo: Dadas duas strings representando em binário dois inteiros positivos, determine usando a aritmética finita de 16 Bits, a soma desses dois inteiros em binário.

Input: Duas strings representando em binário dois inteiros positivos.

Output: Uma string representando a soma desses inteiros em binário.

ALGORITMO 8.11

Algoritmo: Dadas duas strings representando em binário dois inteiros positivos, determine usando a aritmética finita de 16 Bits, o produto desses dois inteiros em binário.

Input: Duas strings representando em binário dois inteiros positivos.

Output: Uma string representando o produto desses inteiros em binário.

ALGORITMO 8.12

Algoritmo: Dada uma string s1 de floats, determine uma string s2 que é s1 com os elementos ordenados em ordem crescente.

Input: A string s1.

Output: A string s2.

ALGORITMO 8.13

Algoritmo: Dada uma string de caracteres s, sem usar strings auxiliares armazene na própria s a string reversa de s.

Input: A string s.

Output: A string reversa s.

ALGORITMO 8.14

Algoritmo: O objetivo desse algoritmo é determinar a media final de cada aluno de uma turma de n alunos que fizeram 4 provas no ano.

Input: - A quantidade de alunos na turma
- O nome de cada aluno e suas respectivas notas

Output: - Nome de cada aluno P1 P2 P3 P4 Media
(ordenados de modo decrescente por média)

ALGORITMO 8.15

Algoritmo: O objetivo desse algoritmo ordenar alfabéticamente uma lista de nomes.

Input: - A quantidade nomes da lista
- Os nomes da lista

Output: - A lista de nomes ordenada

ALGORITMO 8.16

Algoritmo: O programa a seguir propõe a implementação do **jogo da vida** proposto por John H. Conway. Este jogo simula o desenvolvimento de uma população em um grid 12x12.
O usuário entra com uma configuração inicial, e gerações sucessivas são computadas de acordo com as seguintes regras:

- Nascimento: uma célula com três vizinhos torna-se viva;
- Sobrevivência: uma célula com 2 ou 3 vizinhos permanece viva;
- Morte por solidão: uma célula morre se tiver menos que 2 vizinhos;
- Morte por superpopulação: uma célula morre se tiver mais do que 3 vizinhos;

Input: - Os índices das celulas vivas na configuração inicial.
- Quantidade de gerações

Output: A posição no grid das células vivas que podem ser representadas por exemplo pelo caracter * a cada iteração. Considere a iteração 0 como a configuração inicial.

ALGORITMO 8.17

Algoritmo: O objetivo desse algoritmo é transformar um número inteiro entre 1 e 3999 em seu correspondente romano.

Input: Um inteiro entre 1 e 3999

Output: O numero inteiro lido e seu correspondente romano.

Capítulo 9

Programação Estruturada

Programação estruturada é uma técnica utilizada em computação onde um determinado problema é subdividido em sub-problemas que, espera-se, sejam mais fáceis de serem resolvidos.
Esse método ou técnica tem o seguinte objetivo fundamental ou premissa:
Determinar de modo claro e sucinto as partes fundamentais da solução deixando para uma segunda etapa o detalhamento e respectiva solução dessas partes.
Quase sempre, essas ditas partes fundamentais da solução, envolvem algum tipo de repetição de um mesmo procedimento ou de procedimento muito semelhante.
Uma das grandes vantagens dessa técnica é que ela facilita as tarefas de testar, *debugar* e modificar os programas.
Todas linguagens de programação fornecem ferramentas e estruturas de dados que facilitam a implementação de algoritmos que atendem à técnica da programação estruturada.
Desde que a primeira linguagem de programação começou a ser desenvolvida os desenvolvedores perceberam a grande necessidade de se permitir que determinados trechos dos algoritmos pudessem ser repetidos para diferentes parâmetros.
Esses trechos nas linguagens mais antigas recebiam o sugestivo nome de **rotinas** ou **subrotinas**.
Nessas antigas linguagens, onde a utilização da pouca memória disponível era muito valorizada, essas rotinas ficavam armazenadas em periféricos específicos e não na memória principal dando assim uma idéia de quão segmentados poderiam ser os algoritmos. Esse conceito deu origem às denominadas bibliotecas de subrotinas que nada mais eram, e ainda são, uma coleção de

procedimentos utilizados para realizar tarefas bem definidas.

9.1 Funções em C

Um **programa bem escrito** em C, ou seja, escrito com elegância, eficiência e de modo compreensível, deve necessariamente fazer uso das **funções**.
Um programa em C é uma ou uma coleção de funções.
As funções como veremos a seguir são **blocos de construção** que realizam uma **determinada tarefa** e que podem ser reusados.
Vamos dar um exemplo de um problema onde, esperamos, esclarecer as vantagens desse tipo de procedimento.
Suponha a seguinte *tarefa*:
Dado um inteiro $n >= 0$ determinar $s = 2! + 3! + 4! + \cdots + n!$
Caso tenhamos uma função que realize a tarefa acima, que na notação matemática pode ser representa por $f(i) = i!$, com certeza nossa tarefa ficará muito mais fácil.
De fato bastara um trecho do tipo:
$s = 0;$
$for\ (i = 2; i <= n; i++)\ s = s + f(i);$
para calcular o valor de s.

9.1.1 Funções que retornam um único valor

As funções desse tipo já nos são familiares pois são desse tipo todas as funções pré-definidas, vistas na secção 4.4. Uma função desse tipo pode receber um número ilimitado de **parâmetros** e devolve (retorna) sempre um **único resultado**.

Sintaxe:

```
tipo nome(declaração_de_parametros){
  diretiva_1;
  diretiva_2;
  ..............
  diretiva_n;
}
```

- **tipo:** É o **tipo da função** que é determinado pelo tipo do dado que a função devolve(retorna) após sua execução. Os tipos válidos são: **int**, **float**, **char**;
- **nome:** É o nome pelo qual a função será referenciada;
- **declaração_de_parametros:** O tipo é a quantidade de parâmetros;
- **diretivas:** São comandos ou declarações;
- Pelo menos uma das diretivas deve ser o comando **return(expr)** onde o resultado da expressão (expr) deve ser do mesmo **tipo da função**.
- O comando **return** tem duas importantes finalidades. A primeira é **retornar um valor** e a segunda é a **saída imediata** da função e seu retorno para a diretiva posterior à sua **chamada**.

Exemplo 9.1.1

Vamos definir uma função que: dado um inteiro $n \geq 0$, calcula $n!$

```
// Funcao Fatorial
int  fatorial (int n){
   int i, fat=1;
   if ( n == 0 ) return(1); // 0!=1
   else
     for (i = 1 ; i<= n ; i++) fat = fat * i;
   return(fat);
}
```

- tipo da função : int (a função retorna um valor do tipo **int**)
- nome da função : fatorial
- tipo e quantidade de parâmetros : 1 parâmetro do tipo inteiro (a função recebe um valor do tipo **int**

Uma vez definida a função devemos esclarecer os seguintes pontos:

- Onde ela deve ser colocada ?
- Como utilizá-la ou como se diz no *jargão computacional* como **chamá-la**?

Em principio vamos considerar que ela deve colocada num programa de modo a ficar antes de sua primeira utilização (chamada).
Posteriormente veremos que essa condição pode ser flexibilizada.
Para explicar como **chamar** a função vamos continuar o exemplo anterior.

Exemplo 9.1.2

Vamos *escrever* um programa que tem como **input** um inteiro $n \geq 1$ e como **output** $s = 1! + 2! + 3! + 4! + \cdots + n!$

```
#include <stdio.h>
// Definicao da Funcao Fatorial
int  fatorial (int n){
    int i, fat=1;
    if ( n == 0 ) return(1); // 0!=1
    else
      for (i = 1 ; i<= n ; i++) fat = fat * i;
    return(fat);
}
//Programa Principal
int main(){
  int s=0 , i , n;
   printf("Entre com um inteiro maior do que zero:");
   scanf("%d",&n);
   for( i=1 ; i<=n ; i++ ) s = s + fatorial(i);
   printf(" s = %d" , s);
   return(0);
}
```

Observe que a **chamada da função** foi feita no comando
$s = s + fatorial(i)$;
ou seja para **chamar uma função** basta referenciar seu nome com o(s) respectivo(s) parâmetro(s).

Exemplo 9.1.3

Vamos definir uma função que recebe dois inteiros e retorna o maior deles e utilizá-la num programa para imprimir o maior entre dois inteiros recebidos como entrada.

```
#include <stdio.h>
// Definicao da Funcao Maximo
int maximo(int a , int b){
```

```
  if ( a <= b ) return(b);
  else return(a);
}
//Programa Principal
main(){
  int i , j;
  printf("Entre com 2 inteiros:" );
  scanf("%d%d", &i , &j);
  printf("O maior eh = %d" , maximo(i,j));
}
```

Observe que nesse caso a função foi **chamada** num comando **print** o que é perfeitamente válido.

Um erro muito comum ao iniciante é usar a sintaxe **maximo(a,b)** no programa principal. É claro que o compilador vai acusar o erro pois o programa principal *desconhece* a e b. Observe que eles foram declarados na função e assim só ela os conhece.

Na chamada da função as variáveis i e j (que são conhecidas do programa principal) irão substituir os parâmetros a e b da definição da função.

Exemplo 9.1.4

A função desse exemplo retorna o comprimento de uma string.

```
#include <stdio.h>
// Definicao da Funcao str_compr
int str_ compr( char s[ ] ){
  int i=0;
  while ( s[i++] != '\0');
  return(i-1);
}
//Programa Principal
main(){
  char nome[30];
  printf("Entre com uma string:");
  gets(nome);
  printf("comprimento da string = %d" , str_ compr(nome));
}
```

Observe que na definição da função a string parâmetro (char s[ ]) não neces-

sita ser dimensionada. No programa principal é **necessário** dimensioná-la (char nome[30]).

9.1.2 Funções que não retornam valor

As funções desse tipo executam algum procedimento mas não retornam valor ao *código chamador*.

Sintaxe:

```
void nome(declaração_de_parametros){
  diretiva_1;
  diretiva_2;
  ..............
  diretiva_n;
}
```

Exemplo 9.1.5

```
#include <stdio.h>
void print_header()
{
    printf("Apostila de Algoritmos e Estrutura de Dados\n");
    printf("Prof. Sergio R. de Freitas\n");
    printf("Versao 1.0, disponibilizada em 03/04/2001\n");
}
//Programa Principal
main(){
print_header();
}
```

Exemplo 9.1.6

A função desse exemplo imprime uma matriz $n \times m$

```
#include <stdio.h>
typedef int matriznxm[3][3];
 matriznxm a , b ,c ;
// Definicao da Funcao imprime_matriz
 void imprime_matriz(int n , int m , int mat[ ][3]){
  int i,j;
  for( i=0 ; i < n ; i++ ){
```

```
    for( j=0 ; j < n ; j++ ) printf("%4d" , mat[i][j ]);
    printf("\n");
  }
}

//Programa Principal
main(){
  int m = 3, n = 3 , i , j;
  matriznxm a = {
                  {1,5,6},
                  {3,4,7},
                  {2,9,5}
                 };
  matriznxm b = {
                   {6,9,2},
                   {−3,4,10},
                   {−2,−1,7}
                 };
for( i=0 ; i < n ; i++ )
 for( j=0 ; j < n ; j++ ) c[i][j] = a[i][j] + b[i][j];
printf("    Matriz (a)      \n\n");
imprime_matriz (3,3,a);
printf("\n");
printf("    Matriz (b)      \n\n");
imprime_matriz (3,3,b);
printf("\n");
printf("    Matriz (a + b)  \n\n");
imprime_matriz (3,3,c);
}
```

9.1.3 Funções que retornam mais de um valor

Como você deve ter percebido, a *comunicação* entre uma função e seu *código chamador* é feita exclusivamente através dos *parâmetros* e do comando **return**. Como já foi visto, a função *recebe* dados através dos parâmetros e aí ou não devolve nada (tipo **void**) ou devolve um único valor através do **return**.

Uma das características do *C* é que os parâmetros, como se diz no *jargão computacional*, são **passados por valor**. Isso significa que a função recebe apenas uma **cópia temporária** de cada parâmetro que *duram* apenas enquanto a função está sendo executada.

Na prática significa que se alguma das diretivas que compõe a função alterar o valor dos parâmetros estará apenas alterando a **cópia temporária**.

Vamos exemplificar.

Exemplo 9.1.7

```
#include <stdio.h>
void troca(int i , int j){ // troca os valores de i e j
 int t;
    t=i;
    i=j;
    j=t;
    printf("dentro da funcao temos : i = %d ; j = %d\n" , i , j );
}
//Programa Principal
main(){
 int i=10 , j=20;
 printf("antes da execucao da funcao : i = %d ; j = %d\n" , i ,j  );
 troca ( i ,j  );
 printf("depois da execucao da funcao : i = %d ; j = %d\n" , i , j );
}
```

O output será:
antes da execucao da funcao : i = 10 ; j = 20
dentro da funcao temos : i = 20 ; j = 10
depois da execucao da funcao : i = 10 ; j = 20

Com o exemplo acima fica claro que não é possível alterar os valores dos parâmetros. Isso deve-se ao fato de eles serem passados à função *por valor* e não *por referência*. Como veremos na próxima seção, a passagem de parâmetros *por referência* permitirá *parâmetros variáveis*.
A passagem de parâmetros *por referência*(endereço) será conseguida através da utilização de apontadores (ponteiros) que serão discutidos a seguir.

9.2 Apontadores (Ponteiros) em *C*

Os **apontadores** são utilizados em toda parte na linguagem *C*. Desse modo se tivermos a pretensão de utilizar a linguagem em toda sua plenitude não podemos prescindir de um bom entendimentos dos **apontadores** que no *jargão computacional* são mais conhecidos por **ponteiros (pointers)**. Ponteiro, nesse caso, deve ser entendido no sentido de *apontar para* (daí o nome). Um programador em *C* tem que se *sentir confortável* com a utilização de apontadores apesar de que os principiantes possam sentir algum desconforto no seu entendimento e utilização.
C usa apontadores de três diferentes maneiras:

- *C* usa apontadores para criar estruturas de dados dinâmicas;
- *C* usa apontadores para possibilitar a *passagem de parâmetros variáveis* às funções;
- *C* usa os apontadores como uma alternativa de acesso às informações armazenadas em arrays. De fato existe uma íntima conexão entre arrays e apontadores em *C*.

Os programadores em *C* também usam apontadores para tornar seus códigos mais eficientes.

9.2.1 Introdução aos Apontadores

Para tentar facilitar o entendimento dos apontadores vamos tentar compará-los com uma *variável normal*.
Como já sabemos uma *variável normal* é uma posição na memória que pode armazenar um dado. Quando declaramos uma variável *i* como **int**, em algum lugar da memória, quatro bytes são reservados para armazená-la. No programa aquela posição na memória é referenciada pelo nome *i* e a *nível de maquina* por um *endereço de memória*.
Um apontador é diferente.
Um apontador é uma variável que aponta para outra variável. Isso significa que um apontador **armazena o endereço de memória de uma outra variável**. Dizemos que um apontador **aponta para** uma variável pois ele armazena seu endereço.
Pelo fato de armazenar um endereço, ao invés de um valor, um apontador é composto de duas partes: o apontador e o valor apontado por ele.

Esse fato no início pode causar alguma confusão e trazer algum desconforto até que nos habituemos e possamos *dominar* essa técnica extremamente poderosa.

9.2.2 Apontadores e Endereços

Como já dissemos um apontador contem o endereço de um objeto. Isso torna possível *acessar* esse objeto *indiretamente* através de seu apontador. **Os apontadores são sempre do tipo inteiro positivo**

endereço	1000045	1102200	2765228	110832
conteúdo	230	10.9	'c'	false

Tabela 9.1: Endereços e Conteúdos

Declaração de Apontadores

Para declarar uma variável que vai armazenar um **apontador** deve ser usada a seguinte sintaxe:

tipo *nome_da_varivel

Você pode criar um apontador para todo tipo de dado - **float**, **char**, **estrutura**, etc. - bastando para isso usar um * para indicar que se trata de um apontador e não uma variável normal.

Exemplo 9.2.1
int *p;

Aqui foi criado um apontador para um inteiro.
Como no caso das variáveis que, quando criadas, contem *lixo* como valor isso também vale para as variáveis do tipo apontador obviamente.

Exemplo 9.2.2
float *x;

Aqui foi criado um apontador para um dado do tipo **float**.

Operadores de Apontadores

Existem dois operadores especiais unários para os apontadores:

- O operador & que retorna o endereço de memória de seu operando.

- O operador * que é o complemento do & e retorna o conteúdo da variável localizada no endereço que o segue.

Exemplo 9.2.3

```
#include <stdio.h>
int main() {
 int i=10 ;   // Declara i do tipo inteiro e inicializa com 10
 int *ender ; // Declara um apontador(ender) para um inteiro
 ender=&i;    // Atribua ao apontador o endereco de i
 printf("endereco de i = %d\nvalor do dado no endereco = %d" , ender , *ender);
}
```

O output será:

endereco de i = 587580
valor do dado no endereco = 10

Os operadores & e * tem precedência máxima sobre todos os outros operadores.

Apontadores e Arrays

Como já foi dito existe uma forte relação entre **apontadores** e **arrays**. Qualquer operação que pode ser realizada através dos indices dos arrays também pode ser realizada usando apontadores só com muito maior velocidade na execução do código.

Exemplo 9.2.4

A declaração **int a[10]** define um array de tamanho 10 que é um bloco de 10 consecutivos objetos chamados $a[0], a[1], \cdots, a[9]$.
A notação $a[i]$ significa o elemento do array i posições do inicio.
Se p é um apontador para um inteiro declarado por
$int \quad *p \quad$ então o comando
$p = \&a[0]$
faz com que p aponte para o elemento inicial do array ou seja $a[0]$.
Uma diretiva do tipo
$x = *p$ irá atribuir a x o conteúdo de $a[0]$.
Se p aponta para um particular elemento do array então, por definição, $p+1$ aponta para o próximo elemento do array e $p-1$ para o anterior.

Prosseguindo como esse raciocínio temos $*(p+1)$ e $*(p-1)$ como os respectivos conteúdos daqueles elementos do array.

Exemplo 9.2.5

```
#include <stdio.h>
main() {
int i , *p ;
int a[5]={0,2,4,6,8};
p=&a[0];
for (i = 0 ; i < 5 ; i++)
printf("a[%d]=%d \n" , i , *(p++) );
}
```

Output

```
a[0]=0
a[1]=2
a[2]=4
a[3]=6
a[4]=8
```

- O nome de um array (sem o índice) é o *endereço* do elemento inicial do array. No exemplo anterior poderíamos ter usado a sintaxe p=a ao invés de p=&a[0];
- Todas as strings são terminadas pelo caracter \0 cujo valor é **falso**.

Exemplo 9.2.6

```
#include <stdio.h>
//funcao que calcula o comprimento de uma string
int str_len(char *s){ //O nome de um array \'{e} um apontador para o primeiro elemento do array
int i=0 ;
while(*s){i++;
s++;
}
return(i);
}
//principal
main() {
```

```
char a[20]="abacate";
printf("comprimento da sring = %d" , str_ len(a));
}
```

9.2.3 Passando Parâmetros por Referência (Endereço)

Como já dissemos anteriormente essa é a maneira que temos para alterar os valores de parâmetros passados à uma função.
Para isso será necessária a utilização dos apontadores.
Vamos retornar ao exemplo da **função troca** que nos serviu de exemplo para mostrar a impossibilidade de se alterar os valores dos parâmetros passados às funções.

Exemplo 9.2.7

```
#include <stdio.h>
void troca(int *i , int *j){
 int t;
    t=*i;
    *i=*j;
    *j=t;
}
main(){
 int i=10 , j=20;
 printf("antes da execucao da funcao : i = %d ; j = %d\n" , i ,j );
 troca ( &i ,&j );
 printf("depois da execucao da funcao : i = %d ; j = %d\n" , i , j );
}
```

Output:
antes da execucao da funcao : i = 10 ; j = 20
depois da execucao da funcao : i = 20 ; j = 10

- Os parâmetros da função são apontadores para inteiros;
- Lembre-se que &i e &j representam os endereços de i e j;
- Assim troca(&i,&j) na verdade está trocando os endereços de i e j.

Exemplo 9.2.8

```
#include <stdio.h>
void str_inversa(char s[ ]){
 int  i = 0 , j=strlen(s)-1;
 char temp;
while ( i < strlen(s)/2 ){
   temp=s[i];
   s[i]=s[j ];
   s[j]=temp;
   i++;
   j--;
}
}
//Programa Principal
main(){
 char s[ ]="cachorrada";
  printf("%s\n" , s);
  str_inversa(s);
  printf("%s\n" , s);
}
```

Output:
cachorrada
adarrohcac

- Você não está um pouco intrigado?
- Você percebeu que a função recebeu uma string, modificou-a e essa string modificada foi retornada ao código chamador?
- A explicação é que quando se manipula arrays estamos na verdade manipulando seus endereços!!!

Exemplo 9.2.9
Criptologia - Mensagens Secretas
O mais antigo método para criptografar(codificar) uma mensagem foi usado por Julius Caesar
Ele criou um código para mensagens secretas deslocando as letras do alfabeto três posições para frente, sendo que as três últimas corresponderiam as três primeiras.
O código de Julius Ceasar pode ser expresso matematicamente do seguinte modo:
A cada letra do alfabeto fazemos corresponder em seqüência os números $0, 1, \cdots, 25$ ou seja definimos:
$map : \{A, B, C, \cdots, Z\} \mapsto \{0, 1, 2, \cdots, 25\}$

A função codificadora é dada por:
$f : \{0, 1, 2, \cdots, 25\} \mapsto \{0, 1, 2, \cdots, 25\}$
$f(p) = (p + 3)\ (mod)\ 26$
A função decodificadora é dada por:
$f^{-1}(p) = (p - 3)\ (mod)\ 26$

Lembre-se que: m (mod) n é o resto da divisão inteira de **m** por **n**.

O programa abaixo é uma implementação do método.

```
#include <stdio.h>
struct juliuscaesar{
    char letra;
    int valor;
} caesar[]={
              {'A', 0},{'B', 1},{'C', 2},{'D', 3},{'E', 4},{'F', 5},
              {'G', 6},{'H', 7},{'I', 8},{'J', 9},{'K', 10},{'L', 11},
              {'M', 12},{'N', 13},{'O', 14},{'P', 15},{'Q', 16},{'R', 17},
              {'S', 18},{'T', 19},{'U', 20},{'V', 21},{'X', 22},{'W', 23},
              {'Y', 24},{'Z', 25},
            };
void encripta(char s[], int k, char t[]){
int i=-1, j;
while (s[++i]!='\0'){
  j=0;caesar[j].valor=0;
  // Procurando os simbolos no caesar
  while ( (caesar[j].valor <= 25) && ( caesar[j].letra != s[i]) ) j++;
  if (caesar[j].valor >= 26) t[i]=' '; // mantem os brancos.
  else t[i]=caesar[(j+k)%26].letra;
 }
 t[i]='\0';  //finalizando a string
 }
// ---------------------------
main(){
 char s[80], t[80];
 printf("Entre com o texto a ser codificado(letras maiusculas):\n");
 gets(s);
 encripta(s,3,t);
 printf("Texto codificado :%s\n",t);
 encripta(t,-3,s);
 printf("Texto decodificado :%s\n",s);
}
```

Output:
Entre com o texto a ser codificado(letras maiusculas):

TEMOS QUE DESCOBRIR A SENHA
Texto codificado :XHPRV TYH GHVFREULU D VHQKD
Texto decodificado :TEMOS QUE DESCOBRIR A SENHA

9.3 Recursividade

As funções em C podem ser usadas recursivamente, ou seja, uma função pode **chamar a si própria** direta ou indiretamente.
Um exemplo onde é natural entender esse conceito é na definição da função fatorial, que como sabemos satisfaz a seguinte propriedade:$n! = n * (n-1)!$
Podemos então definir a função fatorial recursivamente do seguinte modo:

```
1  // fatorial definido recursivamente
2  #include<stdio.h>
3  int fatorial (int n){
4      if (n > 0) return(n*fatorial(n-1));
5      else return(1);
6  }
7  main(){
8   int n;
9    printf("Entre com um inteiro positivo:");
10   scanf("%d",&n);
11   printf( " fatorial (%d) = %d",n,fatorial(n));
12 }
```

Vamos explicar como funciona a recursividade passo a passo.
Talvez uma tabela explique melhor que palavras.

$\Downarrow$	$fatorial(3)$ =?	$\Uparrow$	6
$\Downarrow$	$fatorial(3)$ =3*fatorial(2)=?	$\Uparrow$	3*2=6
$\Downarrow$	$fatorial(2)$ =2*fatorial(1)=?	$\Uparrow$	2*1=2
$\Downarrow$	$fatorial(1)$ =1*fatorial(0)=1	$\Uparrow$	1

Observe que o compilador vai *deixando pendente* as chamadas da função para valores que ele desconhece.
Assim vamos descendo ($\Downarrow$) até chegar no valor $n = 0$ cujo valor da função o compilador conhece (linha 5 $fatorial(0) = 1$).
Agora o compilador vai subindo ($\Uparrow$) e resolvendo as pendências até chegar na primeira chamada da função.

Exemplo 9.3.1
Seqüência de Fibonaci
Veja os detalhes da seqüência no Algoritmo 5.12

```
#include<stdio.h>
#include<stdio.h>
// Fibonaci
int Fibonaci (int n){
   if (n >= 2)return(Fibonaci(n-1) + Fibonaci(n-2));
   else return(1);
}
main(){
 int n;
 printf("Entre com um inteiro positivo:");
 scanf("%d",&n);
 printf( "Fibonaci(%d) %d",n,Fibonaci(n));
}
```

Exemplo 9.3.2
Busca Binária : O programa abaixo usa a técnica de busca binária para **determinar a posição** de um dado num array.
A função retorna -1 se o dado não está contido no array ou sua posição caso contrário.

```
#include<stdio.h>
#define MAX 7
int saida=-1;// definindo saida globalmente
void print_lista ( int a[]){
int i;
for (i=0; i < MAX; i++)
    printf("%-3d\t",a[i]);
   printf("\n");
}
// Busca Binaria
int Busca_binaria (int k, int l , int r , int a[]){
  int m ;
  if ((r-l) < 2 ){
    if (a[l]==k) saida=l;
    if (a[r]==k) saida=r;
  }
 else{
   m =(l+r)/2;
   if (k == a[m]) saida=m;
   if ( k < a[m--]) Busca_binaria (k,l,m,a);
   if ( k > a[m++]) Busca_binaria (k,m,r,a);
 }
 return(saida);
```

```
 }
int main(){
 int n;
 int key;
 int a[ ]={1,9,20,200,300,600,1000};
  print_lista (a);
 printf("Entre com o dado procurado:");
 scanf("%d",&n);
 key=Busca_binaria (n , 0 , MAX−1 , a);
 printf( "%d",key);
 return(0);
}
```

Capítulo 10

Entrada e Saída Usando Arquivos Texto

Até agora, nos nossos programas, a entrada e saída de dados têm sido feitas através do teclado e do vídeo respectivamente.
Esses são os chamados dispositivos *default* do *C* para I/O (input output). Eles também são denominados *stdin*(standard input) e *stdout*(standard output).
Existem inúmeras situações em que a entrada e saída através desses dispositivos *default* são contraproducentes e as vezes até inviáveis. Suponha, por exemplo, o problema de **dar entrada** em: nome, notas e faltas dos alunos de uma turma de tamanho 80. É óbvio que a *entrada interativa* via teclado desses dados é, no mínimo, uma falta de senso. Devemos ainda levar em consideração que uma saída via *stdout*(console de vídeo) é volátil, ou seja, só dura enquanto o vídeo estiver ativo não podendo ser *guardada*.
O que nós precisamos para esses casos é poder ler e escrever num arquivo.

10.1 Criando um arquivo texto

Sintaxe: FILE *f;
f=fopen(string1 , string2);

- FILE é um tipo definido na biblioteca < *stdio.h* >
- fopen() é uma função também definida na < *stdio.h* >

- A função fopen() tem dois parâmetros:
  string1: uma string cujo conteúdo será o nome do arquivo no sistema operacional associado à f.
  string2: uma string representando o *modo de abertura* do arquivo.
  Os *modos de abertura* mais comuns são:

"r"	:	abertura do arquivo para leitura
"w"	:	abertura do arquivo para escrita
"a"	:	abertura do arquivo para adicionar dados
"w+"	:	abertura do arquivo para leitura e escrita
"a+"	:	abertura do arquivo para leitura e adicionamento de dados

- A função fopen() retorna um apontador que tem o valor **NULL** caso tenha havido algum problema na abertura do arquivo!!!

Quando for encerrada a utilização de um arquivo, aberto por um modos vistos acima ("r","w","a"), ele deve ser **fechado** sob risco de perda de dados.
Para fechar um arquivo, aberto usando um apontador **f**, usamos a seguinte diretiva:

fclose(f);

fclose() é uma função da biblioteca < *stdio.h* > cujo parâmetro é um apontador para um dado do tipo FILE.

Exemplo 10.1.1

```
#include <stdio.h>
main(){
FILE *fp;
fp=fopen("saida.txt","w");
if (fp==NULL) printf("Erro na abertura do arquivo");
else printf("arquivo criado com exito! ");
fclose(fp);
}
```

- Caso o arquivo saida.txt **não exista** ele será criado;
- Caso o arquivo saida.txt **já exista** ele será **destruído !!** e um novo será criado;
- O arquivo saida.txt será criado no mesmo diretório em que se encontra o código (programa) que está sendo executado;
  Você pode usar a **forma completa do nome arquivo** caso haja necessidade e.g. "c:\dados\saida.txt"
- Pode não ser possível criar um arquivo por diversos motivos: "disco cheio","usuário sem permissão para criar arquivos"etc.

Exemplo 10.1.2

```
#include <stdio.h>
main(){
FILE *f;
fp=fopen("entrada.txt","r");
if (f==NULL) printf("Erro na abertura do arquivo");
fclose(f);
}
```

- Caso o arquivo entrada.txt **não exista** f==NULL ;
- No caso de sucesso na abertura do arquivo entrada.txt o apontador fica posicionado no inicio do arquivo esperando um comando de leitura;
- Cuide para que o arquivo entrada.txt esteja no mesmo diretório em que se encontra o código (programa) que está sendo executado ou use a **forma completa do nome arquivo** quando for o caso.

10.2 Comandos de Saida

FILE *f;

fputc(caracter , f)

Escreve um caracter no arquivo associado a f.

fputs(string , f)

Escreve uma string no arquivo associado a f.

fprintf(f , "string_de_formato", lista_de_variaveis)

Escreve no arquivo associado a f, usando os formatos já vistos para saída via console.

Exemplo 10.2.1

```
#include <stdio.h>
main(){
  FILE *f;
  char s1[ ]={"Abacate"};
  char s2[ ]={"Verde"};
  f=fopen("saida.txt","w");
  if (f==NULL){
    printf("Erro na criacao do arquivo");
    exit ();
  }
  fputs(s1,f);
  fputs(s2,f);
  fclose(f)
}
```

O output será o arquivo saida.txt com o conteúdo:
AbacateVerde

Exemplo 10.2.2

```
#include <stdio.h>
main(){
 FILE *f;
 int i;
 f=fopen("saida.txt","w");
 if (f==NULL){
```

```
   printf("Erro na criacao do arquivo");
   exit();
 }
 for (i=0 ; i<=42 ; i++){
    if ( i%11 == 10 ) fputc('\n',f);
    fputc(i+'0',f);
 }
 fclose(f);
}
```

O output será o arquivo saida.txt com o conteúdo:

```
0123456789
:;<=>?@ABCD
EFGHIJKLMNO
PQRSTUVWXYZ
```

Você pode ter ficado surpreso com a expressão $i +' 0'$.
Ela está correta em C e funciona do seguinte modo:
O inteiro i é transformado no caracter correspondente ao valor de i.
As letras maiúsculas correspondem aos inteiros no intervalo $[17, 43]$
e as minúsculas no intervalo $[17 + 32, 43 + 32]$;

Exemplo 10.2.3

```
#include <stdio.h>
main(){
 FILE *f;
 int i;
 f=fopen("saida.txt","w");
 if (f==NULL){
   printf("Erro na criacao do arquivo");
   exit();
 }
 for (i=17 ; i < 43 ; i++) fputc(i+'0',f);
 fputc('\n',f);
 for (i=17 ; i < 43 ; i++) fputc(i+32+'0',f);
 fclose(f);
}
```

O output será o arquivo saida.txt com o conteúdo:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
abcdefghijklmnopqrstuvwxyz
```

Exemplo 10.2.4

```
#include <stdio.h>
main(){
 FILE *fp;
 struct registros {
     char nome[40];
     int idade;
     char cpf[15];
}aluno[ ] = {{"Antonio Carlos",31,"120322972-00"},
              {"Jose da Silva",18,"223322472-11"},
              {"Maria Jose",17,"123628472-xx"}
            };
fp=fopen("saida.txt","w");
int i=-1;
while ( ++i < 3)
 fprintf (fp,"%-20s %d %s\n" , aluno[i].nome , aluno[i].idade , aluno[i ]. cpf  );
 fclose (fp );
}
```

O output será o arquivo saida.txt com o conteúdo:

```
Antonio Carlos       31 120322972-00
Jose da Silva        18 223322472-11
Maria Jose           17 123628472-xx
```

Se no programa acima trocarmos o comando fprintf(fp, $\cdots$);
por fprintf(stdout, $\cdots$);
O output será visto no "arquivo stdout"que, no caso, é o monitor de vídeo.

10.3 Comandos de Entrada

FILE *f;

feof(f)

Retorna o valor lógico **true** se o final do arquivo foi *alcançado*.

rewind(f)

Posiciona o apontador de leitura no inicio do arquivo associado a f.

ch=fgets(f)

Lê um caracter arquivo associado a f.

Quando a leitura atinge o final do arquivo (end-of-file) ch tem o valor **EOF**

fgets(string , m , f)

Lê uma string de comprimento m no arquivo associado a f.

fscanf(f , "string_de_formato", lista_de_variaveis)

Lê dados no arquivo associado a f usando os formatos já vistos para as entradas via console.
Em geral, não é uma boa idéia usar **fscanf** para leitura em arquivos. A menos que o arquivo esteja perfeitamente formatado, e o usuário conheça esse formato, não é fácil lidar com ele corretamente.
Uma idéia que em geral é válida é usar **fgets** para ler strings num arquivo que tenha um a formato conhecido.
No que segue iremos discutir diversos exemplos tentando abordar, e dar soluções, para alguns problemas envolvendo leitura formatada em arquivos texto.

Exemplo 10.3.1

```
1  #include <stdio.h>
2  //programa listagem.c
3  main(){
4      FILE *f;
5      char c;
6      f=fopen("listagem.c","r");
7      if (f==NULL){
8      printf("Erro na abertura do arquivo");
9      exit ();
10     }
11     while( (c=getc(f)) != EOF)
12     putc(c , stdout);
13     fclose (f);
14 }
```

Comentários:
Diretivas 6 - 10 - Verificam se houve erro na abertura do arquivo de leitura.
Diretiva 11 - Lê um caracter e testa se é o final de arquivo.
Diretiva 12 - Imprime no vídeo o caracter lido.

Observe que estamos lendo o arquivo **listagem.c**. Esse é o arquivo que contem o programa que está sendo executado.

Output: (No vídeo)

```
#include <stdio.h>
//programa listagem.c
main(){
    FILE *f;
    char c;
    f=fopen("listagem.c","r");
    if (f==NULL){
    printf("Erro na abertura do arquivo");
    exit();
    }
    while( (c=getc(f)) != EOF)
    putc(c , stdout);
    fclose(f);
}
```

Exemplo 10.3.2

```
1  #include <stdio.h>
2  //programa le_string.c
3  main(){
4      FILE *f; char s[1000];
5      f = fopen (" le_string .c","r" );
6      if ( f==NULL){ printf ("Erro na abertura do arquivo") ; exit() ; }
7      while (fgets(s,1000,f ) != NULL)
8          printf ("%s",s);
9      fclose ( f );
10 }
```

Comentários:
Diretivas 6 - 10 - Verificam se houve erro na abertura do arquivo de leitura.
Diretiva 11 - Lê até 1000 caracteres, ou o final de arquivo (NULL).
Diretiva 12 - Imprime no vídeo essa longa string.

Observe que:
Estamos lendo o arquivo **le_string.c** que é o programa que está sendo executado.

Na leitura da string são lidos todos caracteres inclusive os *caracteres especiais* e.g. $\backslash n$, $\backslash t$, etc.

Output: (No vídeo)

```
#include <stdio.h>
//programa le_string.c
main(){
    FILE *f;
    char s[1000];
    f=fopen("le_string.c","r");
    if (f==NULL){
    printf("Erro na abertura do arquivo");
    exit();
    }
    while (fgets(s,1000,f)!=NULL)
        printf("%s",s);
    fclose(f);
}
```

Vamos agora analisar um exemplo usando **fscanf**.
Para isso vamos utilizar o arquivo de saida do exemplo 10.2.4.
O mais óbvio seria escrever um programa cujo formato no comando **fscanf** fosse o mesmo utilizado para **fprintf** daquele exemplo, ou seja:

```
fscanf(fp,"%-20s %d %s\n" , &aluno[i].nome , &aluno[i].idade ,
                            &aluno[i].cpf );
```

Caso você tente irá ver que as strings não são lidas corretamente. Não é difícil entender a razão disso pois, você deve se lembrar que, no comando **scanf** os espaços em branco nas strings são considerados **fim de string**.

Exemplo 10.3.3

```
1  #include <stdio.h>
2  main(){
3       FILE *f;
4       int i=-1;
5         struct registros {
6         char nome[20];
7         int idade;
8         char cpf[12];
9         } aluno[100];
10 f = fopen ( "reg.txt" ,"r" );
11      if ( f == NULL){
12      printf("Erro na abertura do arquivo");
13      exit ();
14      }
15 while ((!feof(f))){
16   fgets ( aluno[++i].nome , 20 , f);
17   fscanf ( f,"%d" , &aluno[i].idade) ;
18   fgets (aluno[i].cpf , 12 , f);
19   fscanf(f,"\n");
20   printf("%s%d%s\n" , aluno[i].nome , aluno[i].idade , aluno[i].cpf);
21  }
22 fclose(f);
23 }
```

Comentários:
Diretivas 15 - Testa se o final de arquivo foi alcançado.
Diretiva 16 - Lê exatamente 20 caracteres.
Diretiva 17 - Lê um inteiro.
Diretiva 18 - Lê exatamente 12 caracteres.
Diretiva 19 - Abandona (Descarta) o resto da linha.
Diretiva 20 - Imprime no vídeo usando o formato especificado no comando printf.

Observe que o arquivo de leitura tem que, necessariamente, ter o formato do arquivo saida.txt ou seja:

```
Antonio Carlos       31 120322972-00
Jose da Silva        18 223322472-11
Maria Jose           17 123628472-xx
```

Exemplo 10.3.4

```
1  #include <stdio.h>
2  #include <string.h>
3  void troca_underscore(char s[]){
4   int i=-1;
5   while (s[++i]!='\0')
6     if( s[i]=='_' ) s[i]=' ';
7  }
8  main(){
9       FILE *f;
10      int i=0;
11      struct registros {
12        char nome[30];
13        float n1,n2,n3,media;
14        int  faltas;
15        char msg[3];
16      } aluno[30];
17 f=fopen("notas.txt","r");
18      if (f==NULL){
19      printf("Erro na abertura do arquivo");
20      exit();
21      }
22 while ((!feof(f))){
23  fscanf(f,"%s%f%f%f%i%f%s\n" , &aluno[i].nome , &aluno[i].n1 , &aluno[i].n2 ,
24             &aluno[i].n3 , &aluno[i]. faltas , &aluno[i].media , &aluno[i].msg);
25   aluno[i].media = (aluno[i].n1 +aluno[i].n2 +aluno[i].n3 )/3;
26   if ( aluno[i].media >= 7.0) strcpy( aluno[i].msg,"AP");
27   else  strcpy( aluno[i].msg,"RN");
28   if ( aluno[i]. faltas > 10) strcpy( aluno[i].msg,"RF");
29   troca_underscore(aluno[i].nome);
30   fprintf(stdout,"%-25s %-5.1f %-5.1f %-5.1f %-2i %-5.1f %s \n" ,
31                  aluno[i].nome , aluno[i].n1 , aluno[i].n2 ,
32                  aluno[i].n3 , aluno[i]. faltas , aluno[i].media , aluno[i].msg);
33   i++;
34  }
35 fclose(f);
36 }
```

Arquivo notas.txt

```
Marta_Lins 4.0 4.0 5.5 3
Mariana_Sousa_Freitas     10.0 6.0 8.3 10
Bruno_Sousa_Freitas  10.0 9.5 9.6 5
Marcia_de_Britto  7.0 6.5 6.6 12
```

As diretivas de 3-7 definem uma função que recebe como parâmetro uma string e troca o símbolo (_) por ( ) nessa string

As diretivas de 9-16 definem os tipos de dados que serão usados

As diretivas 16-21 verificam possíveis erros na abertura do arquivo

A diretiva 22 testa se o final de arquivo não foi alcançado

As diretivad 25-28 calculam medias e atribuem menções aos alunos

A diretiva 29 usa a função textbftroca_underscore para trocar *underscore* por *branco* nos nomes

A diretiva 30 Imprime no vídeo de acordo com o formato dado no fprintf

A diretiva 31 fecha o arquivo de leitura

Output:(Video)

```
Marta Lins               4.0   4.0   5.5   3  4.5   RN
Mariana Sousa Freitas    10.0  6.0   8.3   10 8.1   AP
Bruno Sousa Freitas      10.0  9.5   9.6   5  9.7   AP
Marcia de Britto         7.0   6.5   6.6   12 6.7   RF
```

10.4 Redirecionando Inputs e Outputs

A linguagem *C* disponibiliza uma diretiva singela mas de muita utilidade para o redirecionamento de **inputs** e **outputs**.

Vamos explicá-la com o exemplo a seguir.

Exemplo 10.4.1

```
#include <stdio.h>
//programa redir.c
main(){
    char c;
    while( (c=getchar()) != EOF)
    putchar(c);
}
```

No programa acima temos a entrada de um caracter através do **teclado** (input padrão) e sua impressão no vídeo (output padrão).

Digitando **enter** finalizamos a entrada de uma linha, pois o **enter** corresponde ao caracter $\backslash n$.
Digitando **CTRL-Z** é encerrada a entrada de dados pois **CTRL-Z** corresponde, no **DOS-Windows** ao **EOF**.
No **Unix-Linus** o **EOF** corresponde a **CTRL-D**.

Para usar como entrada o arquivo **ent.txt** e como saida o **sai.txt** usamos a seguinte sintaxe:

redir.exe < **ent.txt** > **sai.txt**

Por exemplo usando o comando:

redir.exe < **redir.c** > **sai.txt**

Teremos como output o arquivo **sai.txt** abaixo:

```
#include <stdio.h>
//programa redir.c
main(){
    char c;
    while( (c=getchar()) != EOF)
    putchar(c);
}
```

– Caso não exista o arquivo redir.c o sistema informa que o arquivo não foi encontrado.
– O arquivo sai.txt é criado automaticamente (caso já exista será sobrescrito)

Capítulo 11

Bibliotecas em *C*

As bibliotecas na linguagem *C* são muito importantes pois a linguagem fornece apenas os recursos mais básicos.
Você já deve ter percebido que até mesmo as funções básicas de I/O (leitura através do teclado e impressão no vídeo) são fornecidas por bibliotecas, ou seja para utilizá-las devemos usar a diretiva `#include<stdio.h>`.
Assim a linguagem foi constituída de *pedaços de códigos* que são armazenados em bibliotecas tornando possível sua re-utilização e até mesmo sua compilação em plataformas diferentes.
Do mesmo modo que a **stdio** existem também bibliotecas padrão para: funções matemáticas, manipulação de strings, manipulação da hora etc.
O usuário também pode criar funções e armazenar em bibliotecas de modo a tornar seus programas modulares.
Isso torna mais fácil entendê-los, testá-los e depurá-los além de tornar possível a sua reutilização em outros programas.

11.1 Criando uma biblioteca

A criação de sua própria biblioteca é muito fácil.
Toda biblioteca consiste de duas partes:
– O arquivo cabeçalho (header file)
– O arquivo do código fonte (code file).

O header file, que geralmente é denotado pelo sufixo .h, contem as informações necessárias para os programas que utilizarão essa biblioteca.
Em geral o header file contem constantes e tipos, juntamente com os protótipos para as funções disponíveis na biblioteca.
Por exemplo, você pode criar o arquivo **util.h** e colocar nele as diretivas:

```
//util.h
extern int get_linha(char s[], int limite);
extern int indice(char s[], char t[]);
FILE *f;
```

A palavra **extern** serve para informar ao compilador que as funções **get_linha** e **indice** serão *linkadas* posteriormente.
Em seguida é informado ao compilador o **protótipo** da função ou seja:
O nome, os parâmetros e o tipo de retorno finalizados com (;)
Vamos continuar com o exemplo.
Crie agora o arquivo **util.c** e coloque nele:

```
//  util.c
//  Encontra todas linhas de um arquivo
//  que contem uma string dada.
//  Retorna o comprimento da linha e a linha que contem a string.

#include <stdio.h>
#include "util.h"
int get_linha(char s[], int limite){
  int i=0;
  char ch;
  while(--limite > 0 && (ch = getc(f)) != EOF && ch != '\n')
     s[i++]=ch;
  if (ch == '\n') s[i++]=ch;
  s[i]='\0';
  return(i);
}

//   funcao indice
//   Retorna um inteiro positivo indicando a posicao
//   da string t na string s ou -1 se t nao esta contida em s.

int indice(char s[], char t[]){
 int i,j,k;
 for (i = 0; s[i] != '\0' ; i++){
    for (j=i , k=0 ; t[k] != '\0' && s[j] == t[k] ; j++ , k++ );
    if (t[k] == '\0') return(i+1);
 }
 return(-1);
}
```

Esses são os códigos das funções e como já dissemos util.c é o *code file*.

11.2 Compilando e Executando com uma Biblioteca

Para compilar uma biblioteca, supondo que você está o compilador gcc, você deve usar a seguinte diretiva:

gcc -c -g util.c

O parâmetro -c informa ao compilador para produzir um **arquivo objeto** para a biblioteca. O arquivo objeto contem o código de máquina da biblioteca. Ele não pode ser executado até que seja *linkado* a um arquivo que contenha uma **main function**. O arquivo **util.o** é o arquivo que contem o código de máquina da biblioteca.

```
//    main.c
#include<stdio.h>
#include "util.h"
#define MAXLINE 80
main(){
     char c;int i;
     char nome[30];
     char patt[10];
     char linha[MAXLINE];
     printf("Entre com o nome do arquivo alvo:");
     gets(nome);
     printf("Entre com o padrao de procura:");
     gets(patt);
     f=fopen(nome,"r");
     if (f==NULL){
     printf("Erro na abertura do arquivo: %s",nome);
     exit ();
     }
     while( get_linha (linha,MAXLINE) > 0 )
      if ( indice(linha, patt) >= 0 ) printf( "coluna:%d %s",indice(linha, patt),linha);
     fclose(f);
    }
```

Para compilar o programa principal (main program) use a diretiva:

gcc -c -g main.c

Essa diretiva cria o arquivo main.o contem o código de máquina do programa principal.

Para criar o **executável final** (o código de máquina para o programa inteiro) devemos *linkar* os dois arquivos objetos.
Para isso usamos a diretiva:

gcc -o main.exe main.o util.o

Para executar o programa basta digitar **main.exe**

O nome **main.exe** é geralmente usado para as plataformas DOS/Windows, para uma outra plataforma poderia ser usada a diretiva:

gcc -o main main.o util.o

Agora o executável teria o nome **main**

Vamos executar o programa main.exe

```
main.exe
Entre com o nome do arquivo alvo:main.c
Entre com o padrao de procura:ar
coluna:7     char c;int i;
coluna:7     char nome[30];
coluna:7     char patt[10];
coluna:7     char linha[MAXLINE];
coluna:33     printf("Entre com o nome do arquivo alvo:");
coluna:33     printf("Erro na abertura do arquivo: %s",nome);
```

11.3 Makefiles

Para facilitar a vida do usuário e evitar a aborrecida tarefa de repetir a digitação de diversas diretivas, toda vez que for feita alguma modificação nos códigos, existe uma facilidade denominada **make**.
O **make** processa um arquivo denominado makefile que contem as seguintes diretivas:

```
main: main.o util.o
    gcc -o main.exe main.o util.o
main.o: main.c util.h
    gcc -c -g main.c
util.o: util.c util.h
    gcc -c -g util.c
```

Digitando **make** será criado o executável **main.exe**

Observe que os comandos gcc devem ser precedidos por um **tab** (espaços em branco não são suficientes). As outras linhas devem ser alinhadas à esquerda.
O **makefile** tem dois tipos de linhas:
As que são alinhadas à esquerda indicam as dependências, as precedidas por um **tab** são executáveis.
Por exemplo, **main.o: main.c util.h** diz que o arquivo main.o depende dos arquivos main.c e util.h .

Observe que para disponibilizar uma biblioteca para utilização pública é suficiente disponibilizar apenas o **header file** e o **object file** da biblioteca ou seja os arquivos (.h) e (.o).
Não é necessário fornecer código fonte da biblioteca.

Agora você está em condições de entender porque você necessita incluir o arquivo **stdio.h** nos seus programas.
Ele é simplesmente uma biblioteca que foi criada e disponibilizada por alguém para facilitar a vida dos programadores.

Capítulo 12

Parâmetros na Linha de Comando

Você já deve ter observado que até agora usamos a seguinte sintaxe para o programa principal:

```
main(){
  .....
}
```

Na verdade essa é uma versão simplicada da sintaxe.
Como o programa principal é também uma função ele também pode **receber** e **retornar** dados.
A sintaxe completa é:

```
main(int argc, char *argv[]){
  .....
return();
}
```

Os parâmetros de entrada tem o seguinte significado:

argc: indica o número de parâmetros passados ao programa;
argv: é um apontador para um array de strings que contem os parâmetros de entrada;
Por convenção **argv[0]** contem o nome do programa que está sendo executado.

Exemplo 12.0.1
Suponha que o arquivo arg.c contenha código abaixo.

```
//  arg.c
#include<stdio.h>
main(int argc, char *argv[]){
 while (--argc>=0)
    printf("argv[%d]\t%s\n",argc,argv[argc]);
}
```

Usando os comandos:
gcc -c arg.c
gcc -o arg.exe arg.o
Será criado o executável arg.exe
Na linha de comandos (DOS ou outro ambiente) digite:

arg.exe parametro1 parametro2 parametro3 parametro4

teremos o seguinte output:

```
argv[4] parametro4
argv[3] parametro3
argv[2] parametro2
argv[1] parametro1
argv[0] c:arg.exe
```

Observe que os parâmetros são separados por brancos.
Quando houver a necessidade de usar um parâmetro que contenha brancos ele deve ser colocado entre aspas e.g. "isso é um único parâmetro"

Exemplo 12.0.2
Neste exemplo iremos utilizar as funções definidas na biblioteca util.h (veja 11.1)

```
//  find.c
#include<stdio.h>
#include "util.h"
#define MAXLINE 80
main(int argc, char *argv[]){
  char linha[MAXLINE];
  f=fopen(argv[2],"r");
    if (f==NULL){
    printf("Erro na abertura do arquivo: %s",argv[2]);
    exit ();
    }
  if (argc != 3){
    printf("Sintaxe: find.exe string arquivo");
    exit ();
   }
  while( get_linha (linha,MAXLINE) > 0 )
      if ( indice(linha, argv[1] ) >= 0 )
       printf( "coluna:%d %s",indice(linha, argv[1]), linha);
}
```

Usando os comandos:
gcc -c find.c
gcc -o find.exe find.o util.o
Será criado o executável **find.exe**

Na linha de comandos digite:

find.exe LINE find.c

teremos então o seguinte output:

```
coluna:12 #define MAXLINE 80
coluna:17   char linha[MAXLINE];
coluna:30   while( get_linha (linha,MAXLINE) > 0 )
```

Uma convenção que já é tradição nos programas *C* com parâmetros é considerar os parâmetros que iniciam com o sinal - são **parâmetros opcionais ou flags**. Vejamos um exemplo usando essa convenção.

Exemplo 12.0.3
O programa lista.c a seguir tem o seguinte objetivo:

Listar um **input** usando as seguintes condições:
Se o parâmetro **-n estiver presente** na linha de comando as linhas listadas serão **numeradas**, caso contrário as linhas **não serão numeradas**;
Se o parâmetro **-x estiver presente** na linha de comando serão listadas todas linhas que **não contiverem** a string, caso contrario serão listadas todas linhas que **contiverem** a string.

```
//   lista .c
#include<stdio.h>
#include "util.h"
#define MAXLINE 80
main(int argc, char *argv[]){
  char linha[MAXLINE], *s ;
  int numline=0 , exceto=0 , numera=0 ;
  f=stdin;
  while ( --argc > 0 && (*++argv)[0] == '-')
   for ( s=argv[0]+1 ; *s != '\0' ; s++ )
     switch(*s){
       case 'x' : exceto=1;
                  break;
       case 'n' : numera=1;
                  break;
       default  :
                   printf("Opcao ilegal %c",*s);
                  argc=0;
                  break;
     }
  if (argc !=1)
     printf ("Sintaxe : lista -x -n string \n");
  else
  while( get_linha (linha,MAXLINE) > 0 ){
       numline++;
     if ( indice(linha, *argv) >= 0 != exceto ){
       if (numera)
      printf("%1d: ",numline);
      printf("%s",linha);
     }
  }
}
```

teste (a)

```
lista.exe  -n case < lista.c

12:        case 'x' : exceto=1;
14:        case 'n' : numera=1;
```

teste (b)

```
lista.exe  arg < lista.c

main(int argc, char *argv[]){
  while ( --argc > 0  && (*++argv)[0] == '-')
   for ( s=argv[0]+1 ; *s != '\0' ; s++  )
                 argc=0;
  if (argc !=1)
     if ( indice(linha, *argv) >= 0 != exceto ){
```

teste (c)

```
lista.exe  num < lista.c

  int numline=0 , exceto=0 , numera=0  ;
      case 'n' : numera=1;
      numline++;
     if (numera)
    printf("%1d: ",numline);
```

teste (d)

```
lista.exe  -nx a < lista.c

2: #include<stdio.h>
3: #include "util.h"
4: #define MAXLINE 80
8:   f=stdin;
11:      switch(*s){
20:      }
23:   else
25:       numline++;
28:       printf("%1d: ",numline);
30:      }
31:   }
32: }
```

Você deve analisar esse exemplo com atenção pois ele tem muitos detalhes importantes.

Observe por exemplo que definimos **f** como **stdin** ou seja o input é o teclado, mas você deve ter percebido que usamos, por exemplo, a sintaxe:

lista.exe -n case < lista.c

< lista.c significa que estamos **redirecionando o input** para o arquivo **lista.c**

Também seria válida a seguinte sintaxe:

lista.exe -n case < lista.c > saida.txt

Nesse caso também o output está redirecionado para o arquivo saida.txt

Capítulo 13

Algoritmos Propostos (III)

ALGORITMO 13.1

Algoritmo: Defina uma função que dado x do tipo float retorne m onde m é o maior inteiro menor ou igual à x.
Obviamente você não pode usar a função pré-definida $floor$.

ALGORITMO 13.2

Algoritmo: Defina uma função que tem como parâmetros dois inteiros e que retorna o máximo divisor comum (mdc) entre esses inteiros.

ALGORITMO 13.3

Algoritmo: Para mostrar que um numero $p > 2$ é primo basta mostrar que ele não é divisível por nenhum inteiro no intervalo $[2, \sqrt{p}]$.
Defina uma função que determina se um inteiro dado é ou não primo.
A função recebe um inteiro p e retorna 1 se p é primo ou o caso contrário.

ALGORITMO 13.4

Algoritmo: O objetivo da função a ser definida a seguir é: Dada uma string e um caracter, determinar em que posição da string se encontra o caracter. A função deve ter como parâmetros a string e o caracter.
Deve retornar um inteiro positivo que indica a posição do caracter na string ou zero se a string não contiver o caracter.

ALGORITMO 13.5

Algoritmo: O objetivo da função a ser definida a seguir é:
Dada uma string, determinar uma substring dela.
A função deve ter como parâmetros a string e dois inteiros m e n.
Deve retornar uma substring da string que começa na posição m e tem comprimento n.
Exemplo: substr("maravilha",2,3)="ara"

ALGORITMO 13.6

Algoritmo: O objetivo da função a ser definida a seguir é comparar duas strings.
A função deve ter como parâmetros duas strings, digamos s e t, que vão ser comparadas.
Deve retornar: um inteiro < 0 se $s < t$.
um inteiro > 0 se $s > t$
0 se $s = t$

ALGORITMO 13.7

Algoritmo: **Busca Binária**
O objetivo da função a ser definida a seguir é encontrar, de modo eficiente, numa seqüência m de inteiros um inteiro k.
Vamos considerar que a seqüência m está ordenada de modo crescente.
A busca binária consiste em subdividir a seqüência m em duas subseqüências ($m1$ e $m2$) de aproximadamente mesmo tamanho. Podemos agora verificar se k está em $m1$ ou em $m2$ bastando para isso determinar se $k <$ que o último elemento de $m1$ ou se $k >$ que o primeiro elemento de $m2$.

Uma vez determinado a qual seqüência ele pertence podemos repetir o argumento anterior para essa seqüência que tem aproximadamente metade dos elementos da seqüência inicial. O processo deve continuar até que o comprimento da seqüência torne-se 1 (um único elemento). Agora temos duas chances:ou k é igual a esse elemento(foi encontrado) ou não significando que k não está na seqüência.

Definia uma função para realizar essa tarefa que deve ter como parâmetros:
- o tamanho da seqüência (inteiro n);
- a seqüência de inteiros ordenada de modo crescente(m[ ]).
- a chave procurada (o inteiro k);
Deve retornar: um inteiro indicando a posição(índice) do dado k ou -1(não está na seqüência).

ALGORITMO 13.8

Algoritmo: Criptografia

Considerando-se os problemas de criptografia (veja exemplo 9.2.9) podemos conseguir um pouco mais de segurança no código de Julius Caesar considerando funções da forma

$$f(p) = (ap + b)\ (mod)\ 26$$

onde a é um inteiro satisfazendo $mdc(a, 26) = 1$ e b um inteiro qualquer.
Defina uma função para codificar e outra para decodificar considerando o exposto acima com $b = 3$ e $a = 7$.
Para esses dados temos que

$$f^{-1}(p) = (21p + 7)\ (mod)\ 26$$

Teste suas funções num programa.

ALGORITMO 13.9

Algoritmo: Números pseudo-randomicos

O números pseudo-randomicos são assim denominados porque são gerados através de um método sistemático.

Dada uma *semente*, os métodos usando *congruência linear*, geram uma mesma seqüência de números aleatórios (randomicos) distintos num intervalo dado.

Considere, por exemplo, os seguintes dados:

- inteiro m (denominado módulo da congruência);
- inteiro a onde $2 \leq a < m$ (multiplicador);
- inteiro c (incremento);
- inteiro x_0 onde $2 \leq x_0 < m$ (semente);

Então a seqüência definida recursivamente através de

$$x_{n+1} = (ax_n + c)\ (mod)\ m \qquad n \geq 0$$

nos fornece m números randomicos distintos x_n onde $0 \leq x_n < m$.

Defina uma função que use como parâmetros de entrada os inteiros m, a, c, x_0 definidos acima e um inteiro n definindo o tamanho da seqüência de aleatórios.

A função deve retornar uma seqüência de tamanho n de inteiros aleatórios entre 0 e m.

Teste sua função para os dados $a = 7^5$, $m = 2^{15} - 1$ e $c = 0$.

ALGORITMO 13.10

Algoritmo: Dada uma string (pattern) determine a ocorrência da string num arquivo do disco.

Input: Uma string (pattern) e o nome do arquivo onde ela será procurada.

Output: A linha onde a string aparece.

ALGORITMO 13.11

Algoritmo: Compare dois arquivos.

Input: O nome dos dois arquivos.

Output: A primeira linha e a posição do caracter que os arquivos diferem caso contrário uma mensagem dizendo que são iguais.

ALGORITMO 13.12

Algoritmo: Escreva um programa para imprimir um conjunto de arquivos.

Input: O nome dos arquivos a serem impressos.

Output: Imprimir cada arquivo começando numa nova página contendo como header o nome do arquivo que está sendo listado.

Capítulo 14

Organização de Dados

Como já foi dito anteriormente uma das maiores vantagens na utilização dos computadores é tirar proveito da sua fantástica capacidade de poder **armazenar** e **acessar** grande quantidade de informações.
Assim desde os primórdios da utilização dos computadores ficou evidente a importância das técnicas de **busca** (searching) e **organização** (sorting) das informações armazenadas.
Do mesmo que na **vida real** rotinas eficientes de busca de informações pressupõe uma organização no armazenamento dessas informações. Você há de convir que é muito mais prático procurar as suas meias na *gaveta de meias* do que pela casa toda. Claro, isso caso exista a *gaveta de meias* e você seja suficientemente *organizado* para sempre guardá-las lá.
A **organização** é o preço que se paga para ter **facilidade** na procura.
Caso ainda não esteja convencido suponha como seria quase inviável a busca de palavras num dicionário que não estivesse em ordem alfabética.

14.1 Armazenamento e Acesso em Arrays e Arquivos Texto

Até agora as estruturas que dispomos para o armazenamento *quantidades razoáveis* de dados são os **arrays** e os **arquivos textos**. Eles têm características bem distintas, senão vejamos:

- Um array é uma estrutura de **armazenamento interno**, ou seja as informações são armazenadas na memória do computador. O acesso a elas é muito rápido e feito através do índice do elemento, também

denominada **key (chave)** do elemento. O tipo de acesso por ela proporcionado é denominado **acesso direto** pois qualquer dado pode ser acessado sem a necessidade de passar pelos dados anteriores.
A desvantagem é que além da limitação da quantidade de informações que pode ser armazenada, ou seja, a quantidade de memória disponível, ainda temos a necessidade de informar ao compilador a quantidade máxima de informações que iremos usar.
Isso é uma imposição muito séria para determinados tipos de aplicações, como veremos posteriormente. Obviamente esse tipo de armazenamento é **volátil** ou seja só dura enquanto o programa que o utiliza estiver executando.

- Um arquivo texto é uma estrutura onde o acesso às informações é feita de modo seqüêncial ou seja para acessar uma informação armazenada na posição n é necessário *passar* pelas $n-1$ posições anteriores, tornando o acesso às informações muito mais lento.
  Em contrapartida podemos armazenar uma quantidade imensa de informações sem a necessidade de saber essa quantidade à priori.
  Os arquivos texto são armazenadas nos chamados dispositivos de **armazenamento secundário** ou seja discos rígidos, discos flexíveis, cd-rom etc.
  Você não deve confundir o acesso aos arquivos que é feito de modo direto através da *tabela de alocação dos diretórios* com o acesso ao **conteúdo dos arquivos texto** que é seqüêncial.
  Dois dispositivos nos quais diferença entre esses tipos de acesso (direto e seqüêncial) fica bem claro são: os DVDs (digital video disk) e as fitas de vídeo. No caso do DVD você pode escolher uma determinada cena e ir direto à ela enquanto que no caso da fita de vídeo você tem que "ver" todas as cenas anteriores até chegar numa cena específica.

14.2 Listas

Uma lista, ou tabela, é uma seqüência de $n >= 0$ elementos $l[0], l[1], \cdots, l[n-1]$ onde tem-se:

(i) $l[0]$ é o primeiro elemento da lista;

(ii) o elemento $l[k]$ sempre é precedido pelo elemento $l[k-1]$.

Um elemento de uma lista linear pode ser visualizado do seguinte modo:

k	$Dado[k]$

A lista completa então será:

0	$Dado[0]$	1	$Dado[1]$	$\cdots$	$n-1$	$Dado[n-1]$

Na linguagem C podemos representar uma lista da seguinte maneira:

```
#define MAX 7
main(){
 typedef int tipo_dado; // voce pode mudar o tipo_dado.
 struct lista {
    tipo_dado dado;
 };
 struct lista l [ ] = {32,27,42,4,321,5,-234};
 int i=-1;
 while ( ++i < MAX )
 printf("l[%d].dado=%d\n", i ,l[i].dado  );
}
```

Output

```
l[0].dado=32
l[1].dado=27
l[2].dado=42
l[3].dado=4
l[4].dado=321
l[5].dado=5
l[6].dado=-234
```

14.3 Busca em Listas

```
1  // definicoes globais
2  #define MAX 50
3  typedef int tipo_dado;
4    struct lista {
5      tipo_dado dado;
6  };
7  struct lista l[MAX];
8  // Funcao Find retorna o indice ou -1 (nao encontrado)
9  int Find (tipo_dado x , struct lista l [], int n){
10  int i=-1 , busca=-1;
11  while ( ++i <= n )
12    if ( l[i].dado == x ){
13      busca=i;
14      i=n+1;
15  }
16  return(busca);
17 }
18 main(){
19   int i , n=10 , ind;
20   tipo_dado x;
21   for ( i=0 ; i < n ; i++ )
22     l[i].dado=2*i+3;
23   printf("Entre dado a ser procurado:");
24   scanf("%d" , &x);
25   for ( i=0 ; i < n ; i++ )
26     printf("l[%d] = %d\n" , i , l[i].dado );
27   ind=Find ( x , l , n);
28   printf( "\nindice = %d" , ind);
29 }
```

Observe que, na função Find, os comandos 11 e 12 são executados, *no máximo*, n vezes e assim são necessárias $2n$ comparações.
Podemos melhorar o desempenho fazendo, *no máximo*, $n+1$ comparações com a seguinte modificação na função

```
1  // Funcao Find retorna a chave ou −1 (nao encontrado)
2  int Find (tipo_dado x , struct lista  l [ ],  int n){
3   int i=−1 , busca=−1;
4   l[n+1].dado=x;
5   while ( l[ ++i ].dado != x);
6   if  ( i != n + 1 )  busca = i;
7   return(busca);
8  }
```

Observe que nesse caso o comando 5 é executado no máximo n vezes e o 6 apenas uma vez. Assim teremos, no máximo, $n+1$ comparações.

14.4 Busca em Listas Ordenadas

Vamos supor agora que a lista linear esteja ordenada através de sua chave, ou seja

$$l[i].dado <= l[i+1].dado$$

Podemos agora usar a seguinte função:

```
1  int Find_Ord (tipo_dado x , struct lista  l [ ],  int n){
2   int i=−1 , busca=−1;
3   l[n+1].dado=x;
4   while ( l[++i].dado < x);
5   if  ( i != n + 1 )  busca=i;
6   return(busca);
7  }
```

Observe que nesse caso o laço 4 é encerrado quando $l[++i].dado >= x$

Vamos agora discutir o **algoritmo mais eficiente** para buscas em listas ordenadas.
Ele baseia-se no artifício de recursivamente dividir a lista em *sub-listas* com metade dos elementos que a anterior.
Ele é conhecido como algoritmo de *busca binária*(binary search).
Na primeira iteração a dimensão da lista é n ;
Na segunda iteração a dimensão da lista (sub-lista) é $n/2$;
Na i-ésima iteração a dimensão da lista (sub-lista) é $n/2^{i-1}$;
Na m-ésima iteração a dimensão da lista é 1.

```
int Find (tipo_dado x , struct lista l [ ] , int n){
 int i=−1 , busca=−1 , med;
 int inf=0 , sup=n ;
 while ( inf <= sup ){
   med = (inf + sup)/2;
   if ( l[med].dado == x ){
    busca = med;
    inf = sup + 1;     // finaliza   while
   }
   else
     if (l[med].dado > x ) sup = med−1;
     else inf = med + 1;
 }
return(busca);
}
```

14.5 Inserções e Remoções em Listas

No caso das **inserções** devemos ter os seguintes cuidados:

(i) Verificar se o elemento a ser inserido já pertence a lista. Isso nos obrigará a usar um **método de busca**;

(ii) Verificar se há espaço na lista para inserções ou seja se a lista já não está *cheia*.

No caso das **remoções** o elemento a ser removido deve ser *encontrado* na lista. Assim teremos que, também nesse caso, envolver um **método de busca**. É claro que as **inserções** podem sempre ser feitas no **final da lista**, desde que a lista não esteja ordenada ou que não nos importemos em perder essa propriedade.
Por outro lado as **remoções** deixam *buracos* na lista o que nos obriga a re-ordenar os índices.
Esses problemas mais a necessidade que temos de alocar, à priori, espaço para as listas nos obrigará a definir outros tipos de estruturas e.g. **Pilhas (Stacks) e Filas (Queues)** , que possibilitarão um melhor tratamento desses problemas.

14.6 Ordenação em Listas

O problema de ordenação em listas consiste em:
Dada uma lista $l[0], l[1], \cdots, l[n]$ determinar uma permutação dos items de modo que : $l[k_0], l[k_1], \cdots, l[k_n]$ estejam ordenados.
Matematicamente isso é equivalente à determinar:

$$k : \{0, 1, \cdots, n\} \longmapsto \{0, 1, \cdots, n\} \quad \text{de modo que}$$
$$l[k(i)] <= l[k(i+1)] \qquad i = 0, \cdots, n-1$$

Uma boa *medida* da eficiência de um algoritmo de ordenação e contar o número de **comparações** e **transposições** dos items.
Esses números são função de n que está associado à quantidade de items na lista.
Usualmente, nos métodos de ordenação, iremos ordenar a própria lista ao invés de produzir a permutação que indica a ordem dos elementos da lista.
Os métodos de ordenação em listas podem ser classificados em três categorias principais:

- Ordenação por Inserção;
- Ordenação por Seleção;
- Ordenação por Transposição.

14.6.1 Ordenação por Inserção Direta

Este é o método usado pelos jogadores de cartas para ordenar as cartas na sua mão. As cartas (dados) são conceitualmente divididos em duas listas $l[0], \cdots, l[i-1]$ dita lista de destino e $l[i], \cdots, l[n]$ dita lista fonte. Em cada etapa, iniciando com $i = 1$ e incrementando uma unidade, o i-ésimo elemento da lista fonte é escolhido e transferido para a lista destino sendo inserido no lugar apropriado.

Exemplo 14.6.1
No exemplo a seguir iremos imprimir a lista a cada passo para que fique mais fácil de entender como funciona o algoritmo.

```
// definicoes globais
#define MAX 50
typedef int tipo_dado;
  struct lista {
   tipo_dado dado;
};
struct lista l[MAX];
void Print_Lista ( struct lista l [ ] , int n){
int i;
for (i = 0 ; i < n ; i++)
    printf("%-3d\t",l[i].dado);
printf("\n");
}
// Ordena uma Lista de n elementos
void Sort_Straight_Insertion (struct lista l [ ] , int n){
 int i , j;
 tipo_dado x;
 for (i = 1 ; i < n ; i++){
   x = l[i].dado;
   l [0]. dado=x;
   j=i-1;
   while ( x < l[j].dado) l[j+1].dado = l[j--].dado;
   l[j+1].dado = x;
  printf(" i = %d " , i);
  Print_Lista (l , n );
 }
}
main(){
 int i , n=8 ;
  l [0]. dado = 44 ; l [1]. dado = 55 ; l [2]. dado = 12 ; l [3]. dado = 42;
  l [4]. dado = 94 ; l [5]. dado = 18 ; l [6]. dado = 6 ;  l [7]. dado = 67;
  printf("    -->\t " ) ; Print_Lista ( l , n );
  Sort_Straight_Insertion ( l , n );
}
```

Output

```
   -->   44    55    12    42    94    18    6     67
 i = 1   55    55    12    42    94    18    6     67
 i = 2   12    12    55    42    94    18    6     67
 i = 3   42    12    42    55    94    18    6     67
 i = 4   94    12    42    55    94    18    6     67
 i = 5   18    12    18    42    55    94    6     67
 i = 6   6     6     12    18    42    55    94    67
 i = 7   67    6     12    18    42    55    67    94
```

Esse método pode ser melhorado usando uma **busca binária** para a inserção do item.

Exemplo 14.6.2

```
// definicoes globais
#define MAX 50
typedef int tipo_dado;
  struct lista {
   tipo_dado dado;
};
struct lista l[MAX];
int semente=65;
//  Retorna um numero randomico entre 0 e 32767
int rand()
{
    semente = semente * 1103515245 +12345;
    return (unsigned int)(semente / 65536) % 32768;
}
void Print_Lista ( struct lista l [ ] , int n){
int i;
for (i = 0 ; i <= n ; i++)
   printf("%-3d\t",l[i].dado);
printf("\n");
}
//
// Ordena uma Lista de n elementos
void Sort_Straight_Section (struct lista l [ ] , int n){
 int i , j , k ;
 tipo_dado x;
 for (i = 0 ; i <= n-1 ; i++){
    k = i ; x = l[i].dado;
    for (j = i + 1 ; j <= n ; j++)
      if ( l[j].dado < x ){
       k = j;
       x = l[j].dado;
       l[k].dado = l[i].dado ;
       l[i].dado = x;
     }
 printf(" i = %d " , i);
 Print_Lista ( l , n );
 }
}
main(){
```

```
 int i , n=5 ;
 for (i = 0 ; i <= n ; i++)
    l[i].dado = rand();
   printf("    -->\t " );
   Print_Lista ( l , n );
 Sort_Straight_Section ( l , n );
}
```

Output

```
    -->   24107  16553   12125   9428    13153   21441
 i = 2    16553  24107   12125   9428    13153   21441
 i = 3    12125  16553   24107   9428    13153   21441
 i = 4    9428   12125   16553   24107   13153   21441
 i = 5    9428   12125   13153   16553   24107   21441
 i = 6    9428   12125   13153   16553   21441   24107
```

14.6.2 Ordenação por Seleção Direta

O Método é baseado no seguinte principio:

1 − Selecione o menor item da lista;

2 − Troque-o com o item na primeira posição da lista;

3 − Repita 1 e 2 para os $n-1$ termos restantes;

Exemplo 14.6.3

```
// definicoes globais
#define MAX 50
typedef int tipo_dado;
  struct lista {
   tipo_dado dado;
};
struct lista l[MAX];
int semente=65;
//  Retorna um numero randomico entre 0 e 32767
int rand()
{
    semente = semente * 1103515245 +12345;
    return (unsigned int)(semente / 65536) % 32768;
}
void Print_Lista ( struct lista l [ ] , int n){
```

```
int i;
for (i = 0; i <= n; i++)
    printf("%d\t",l[i].dado);
printf("\n");
}
//
// Ordena uma Lista de n elementos
void Sort_Straight_Section (struct lista l [], int n){
 int i , j , k ;
 tipo_dado x;
 for (i = 0; i <= n-1; i++){
     k = i ; x = l[i].dado;
     for (j = i + 1; j <= n; j++)
        if ( l[j].dado < x ){
         k = j;
         x = l[j].dado;
         l[k].dado = l[i].dado ;
         l[i].dado = x;
       }
  printf(" i = %d " , i);
  Print_Lista ( l , n );
 }
}
main(){
 int i , n=5 ;
 for (i = 0; i <= n; i++)
    l[i].dado = rand();
   printf("    -->\t " );
   Print_Lista ( l , n );
  Sort_Straight_Section( l , n );
}
```

Output

```
    -->   13147  16165   18161   17931   24990   3346
 i = 0    3346   16165   18161   17931   24990   13147
 i = 1    3346   13147   18161   17931   24990   16165
 i = 2    3346   13147   16165   18161   24990   17931
 i = 3    3346   13147   16165   17931   24990   18161
 i = 4    3346   13147   16165   17931   18161   24990
```

14.6.3 Ordenação por Transposição Direta

O Método é baseado no seguinte principio:

A partir do inicio da lista testamos a seguinte condição:

$$l[i].dado <= l[i+1].dado$$

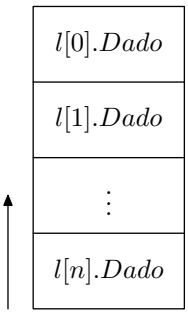
Caso essa condição não esteja satisfeita trocamos os valores de $l[i].dado$ e $l[i+1].dado$, ou seja fazemos uma transposição nesses dados.
Ao atingir o final da lista teremos na posição 0 o menor dado da lista, ou seja:

$$l[0].dado <= l[i].dado \quad i = 1 \cdots, n$$

Repetimos o processo para cada elemento subseqüente da lista.
Na i-ésima etapa teremos:

$$l[0].dado <= l[1].dado <= \cdots <= l[i]dado, \cdots, l[n].dado$$

Esse método ficou conhecido como ordenação das bolhas (**Bubble Sort**) pois se imaginarmos a lista disposta verticalmente e os dados como bolhas de diferentes pesos, essas bolhas vão ficar posicionadas numa altura que dependerá de seu peso. Podemos imaginar a mais "leve"no topo da lista e a mais pesada no fundo dela.



Exemplo 14.6.4

```
// definicoes globais
#define MAX 50
typedef int tipo_dado;
  struct lista {
   tipo_dado dado;
};
struct lista l[MAX];
int semente=321;
//  Retorna um numero randomico entre 0 e 32767
```

```
int rand()
{
    semente = semente * 1103515245 +12345;
    return (unsigned int)(semente / 65536) % 32768;
}
void Print_Lista ( struct lista l [ ] , int n){
int i;
for (i = 0 ; i < n ; i++)
   printf("%-3d\t",l[i].dado);
printf("\n");
}
// Ordena uma Lista de n elementos
void Sort_Bubble_Sort (struct lista l [ ] , int n){
 int i , j , k ;
 tipo_dado x;
 for (i = 1 ; i <= n ; i++){
   j = n  ;
   while (j-- > i)
   if ( l[j-1].dado > l[j].dado ){
        x = l[j-1].dado;
        l[j-1].dado = l[j].dado;
        l[j].dado = x;
   }
 printf(" i = %d " , i);
 Print_Lista ( l , n );
 }
}
main(){
 int i , n=6 ;
 for (i = 0 ; i < n ; i++)
   l[i].dado = rand();
 printf("    -->\t " );
 Print_Lista ( l , n );
 Sort_Bubble_Sort( l , n );
}
```

Output

```
   -->   31144  25024   17582   19594   743     26985
 i = 1   743    31144   25024   17582   19594   26985
 i = 2   743    17582   31144   25024   19594   26985
 i = 3   743    17582   19594   31144   25024   26985
 i = 4   743    17582   19594   25024   31144   26985
 i = 5   743    17582   19594   25024   26985   31144
 i = 6   743    17582   19594   25024   26985   31144
```

Observe que no algoritmo **Bubble-Sort**, mesmo que a lista de entrada já esteja ordenada, todos os passos do algoritmo serão executados. Isso obviamente não é razoável, mas pode ser melhorado. Para tanto iremos introduzir uma *flag* que nos possibilitará, uma vez finalizada uma etapa, decidir se a lista já se encontra ordenada. Isso será possível observando que se numa determinada etapa do algoritmo não houver nenhuma transposição de dados é por que a lista já está ordenada.
No exemplo a seguir introduziremos essa idéia.

Exemplo 14.6.5
Bubble-Sort com flag

```
1  // definicoes globais
2  #define MAX 50
3  typedef int tipo_dado;
4    struct lista {
5     tipo_dado dado;
6  };
7  struct lista l[MAX];
8  void Print_Lista ( struct lista l [ ] , int n){
9  int i;
10 for (i = 0 ; i <= n ; i++)
11     printf("%-d\t",l[i].dado);
12 printf("\n");
13 }
14 //
15 // Ordena uma Lista de n elementos
16 void Sort_Bubble_Sort_Flag (struct lista l [ ] , int n){
17  int i , j , k , flag ;
18  tipo_dado x;
19  for (i = 0 ; i <= n ; i++){
20     j = n + 1;
21     flag = 0;
22       while (--j > i)
23       if ( l[j-1].dado > l[j].dado ){
24         x = l[j-1].dado;
25         l[j-1].dado = l[j].dado;
26         l[j].dado = x;
27         flag = 1 ;
28       }
29    if ( flag == 0 ) {
```

```
30        printf(" i = %d  " , i);
31        Print_Lista ( l , n );
32        i = n + 1;
33     }
34   }
35 }
36 main(){
37   int i , n=7 ;
38   for (i = 0 ; i <= n ; i++)
39     l[i].dado = i;
40     l[6].dado=7;
41     l[7].dado=6;
42    printf("    -->\t " );
43    Print_Lista (l , n );
44    Sort_Bubble_Sort_Flag( l , n );
45 }
```

Output

```
   -->   0       1       2       3       4       5       7       6
 i = 1   0       1       2       3       4       5       6       7
```

O comando 19 determina as etapas do método.
O comando 21 inicializa a flag = 0
O bloco 22-28 executa as transposições de dados, caso necessárias, e se for o caso modifica o estado da flag.
O comando 29 testa o estado da flag: se for 0 (não houve nenhuma troca) imprime a lista e encerra o laço fazendo $i = n + 1$.
Observe ainda que no exemplo anterior apenas os dois últimos elementos da lista não estavam em ordem e assim bastou uma etapa do algoritmo para ordená-la.

14.7 Pilhas - Stacks

Uma pilha é uma lista onde as operações de **inserção** e **remoção** são sempre efetuadas num mesmo extremo da lista. Esse extremo é denominado **topo** da pilha.
Podemos representar uma pilha graficamente da seguinte maneira:

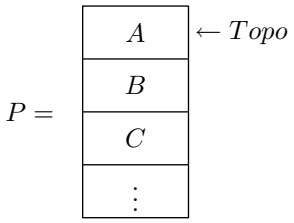


Figura 14.1: Pilha P

A operação que **remove** um item da pilha é denominado **Pop**.

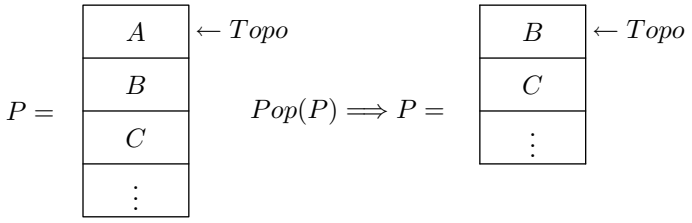


Figura 14.2: Pop(P)

A operação que **insere** um item na pilha é denominado **Push**.

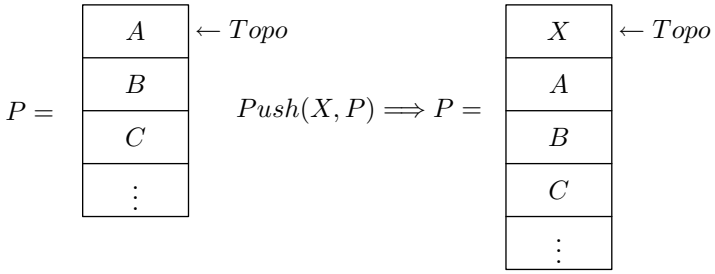


Figura 14.3: Push(X,P)

Como as operações de *Pop* e *Push* são sempre feitas no topo da pilha é fácil ver que os primeiros elementos a serem retirados de uma pilha são sempre os últimos que foram inseridos.
Por esse motivo as pilhas são também conhecidas como **LIFO** (**L**ast **I**n **F**irst **O**ut) Geralmente as pilhas na linguagem *C* são implementadas usando-se uma estrutura dinâmica. Faremos essa abordagem no capítulo seguinte, por enquanto iremos utilizar uma estrutura de array para implementar as pilhas.

Exemplo 14.7.1
No exemplo a seguir vamos implementar as operações de **Pop** e **Push** numa pilha usando para tanto a estrutura de array.
Observe que devemos nos *preocupar* ao dar um **Pop** numa pilha "vazia"ou um **Push** numa pilha "cheia".

```
#define MAX 100
#include<stdio.h>
typedef char tipo_dado; // tipo_dado pode ser trocado
tipo_dado pilha[MAX]; // array representando a pilha
//
tipo_dado Pop( tipo_dado pilha [ ], int *topo , int n){
    if ( *topo != 0 ) return(pilha[(*topo)--]);
    else printf("\nPilha Vazia");
return(' ');
}
void Push(tipo_dado x , tipo_dado pilha [ ], int *topo , int n){
    if ( *topo != n ) pilha[++(*topo)]=x;
    else printf("\nPilha Cheia");
}
void Inicializa_Pilha (tipo_dado pilha [ ] , int * topo , int n){
int i;
*topo=0;       // topo = 0 (pilha vazia)
for ( i = 0 ; i <= n ; i++ )pilha[i]=' ';
}
//
void main(){
int n=10; int *topo;
   Inicializa_Pilha (pilha, &topo, n);
  Push('A' , pilha , &topo , n);
  Push('+' , pilha , &topo , n);
  Push('&' , pilha , &topo , n);
  Push('Z' , pilha , &topo , n);
   printf("%c" , Pop(pilha , &topo , n));
   printf("%c" , Pop(pilha , &topo , n));
   printf("%c" , Pop(pilha , &topo , n));
   printf("%c" , Pop(pilha , &topo , n));
   printf("%c" , Pop(pilha , &topo , n));
}
```

Output

```
Z&+A
Pilha Vazia
```

Observe que demos 4 Pushs na pilha acima, assim ela possuia 4 elementos. Como demos 5 Pops a função fez a advertencia de "pilha vazia".

Exemplo 14.7.2

```
#define MAX 100
#include<stdio.h>
//int *topo;
typedef char tipo_dado;
tipo_dado pilha[MAX];
//
tipo_dado Pop( tipo_dado pilha [ ], int *topo , int n){
     if ( *topo != 0 ) return(pilha[(*topo)--]);
     else printf("\nPilha Vazia");
return(' ');
}
void Push(tipo_dado x , tipo_dado pilha [ ], int *topo , int n){
     if ( *topo != n ) pilha[++(*topo)]=x;
     else printf("\nPilha Cheia");
}
void Inicializa (tipo_dado pilha [ ] , int * topo , int n){
int i;
*topo=0; // topo = 0 (pilha vazia)
for ( i = 0 ; i <= n ; i++ )pilha[i]=' ';
}
//
void main(){
  int n=10,pot;
  int *topo;
  int total=-1 , num ;
  char s[20];
   Inicializa (pilha, &topo, n);
   printf("Entre com uma sequencia de digitos\n");
   gets( s );
   while ( s[++total] != '\0')
   Push(s[total ] , pilha , &topo , n);
   pot=1;num=0;
   while (--total >= 0 ) {
   num = num + pot*((Pop(pilha , &topo , n)-'0'));
   pot=pot*10;
}
```

printf("Numero = %d\n",num);
}

Output

```
Entre com uma sequencia de digitos
873087 Enter
Numero = 873087
```

14.8 Filas - Queues

Uma **fila** é uma lista onde as operações de **inserção** são efetuadas num extremo e as de **remoção** no outro extremo.
O extremo onde são feitas as inserções é denominado **entrada** da fila, o outro extremo é denominado **saida** da fila.
Uma fila pode ser representada graficamente por:

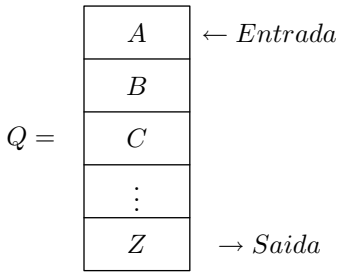


Figura 14.4: Fila Q

As **filas** são também conhecidas como **FIFO** (**F**irst **I**n **F**irst **O**ut)
Observe que as filas são as estruturas que representam as *filas do nosso cotidiano*. Numa fila *o primeiro que chega é o primeiro que sai.*
A operação que **insere** um item da fila é denominada **Push**.

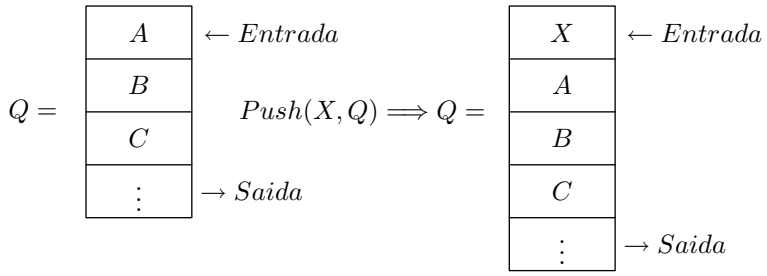


Figura 14.5: Push(X,Q)

A operação que **remove** um item da fila é denominada **Pop**.

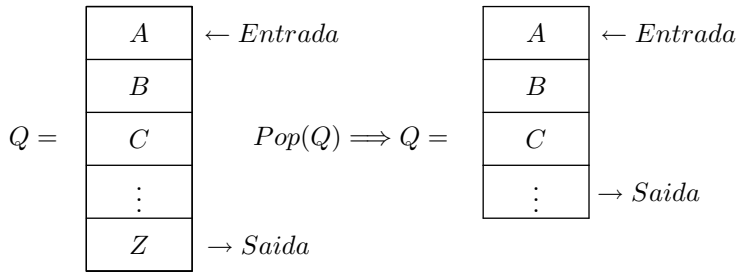


Figura 14.6: Pop(Q)

Exemplo 14.8.1
As operações de **Push** e **Pop** em Filas.

```
// definicoes globais
#define MAX 100
#include<stdio.h>
typedef char tipo_dado;
tipo_dado queue[MAX];
tipo_dado Pop( tipo_dado queue[] , int *entrada , int *saida , int n){
    if ( *saida <= *entrada ) return(queue[(*saida)++]);
    else printf("\nQueue Vazia");
return(' ');
}
void Push(tipo_dado x , tipo_dado queue[ ] , int *entrada , int n){
    if ( *entrada < n ) queue[++(*entrada)]=x;
    else{
     printf("\nQueue Cheia");
     exit (0);
    }
```

```
}
void Iniciliza_Queue(tipo_dado queue [] , int *entrada , int *saida , int n){
int i;
*entrada = -1 ;
*saida   =  0 ;
for ( i = 0 ; i < n  ; i++ )queue[i]='*';
}
void Print_Queue(tipo_dado queue[ ] , int entrada , int saida ){
int i;
printf("\nEntrada = %d<--\n",entrada);
for ( i = saida ; i <= entrada ; i++ )
printf("%c",queue[i]) ;
printf("\nSaida   = %d-->\n",saida);
}
int main(){
  int n=6;
  int entrada;
  int saida;
  tipo_dado lixo;
  Iniciliza_Queue(queue , &entrada , &saida , n);
  Print_Queue(queue , entrada , saida);
  Push('A' , queue , &entrada , n);
  Push('+' , queue , &entrada , n);
  Push('&' , queue , &entrada , n);
  Push('Z' , queue , &entrada , n);
  Print_Queue(queue , entrada , saida);
  lixo=Pop(queue , &entrada , &saida , n);
  lixo=Pop(queue , &entrada , &saida , n);
  Print_Queue(queue , entrada , saida);
  Push('T' , queue , &entrada , n);
  Push('M' , queue , &entrada , n);
  lixo=Pop(queue , &entrada , &saida , n);
  Print_Queue(queue , entrada , saida);
  Push('P' , queue , &entrada , n);
  Print_Queue(queue , entrada , saida);
  Push('X' , queue , &entrada , n);
  return(0);
}
```

Output

```
Entrada = -1<--

Saida   = 0-->
Entrada = 3<--
A+&Z
Saida   = 0-->
Entrada = 3<--
&Z
Saida   = 2-->
Entrada = 5<--
ZTM
Saida   = 3-->
Entrada = 6<--
ZTMP
Saida   = 3-->
Queue Cheia
```

Observe que na função **Push** implementada acima para uma fila de comprimento n o teste de *fila cheia* não implica que tenhamos as n posições do array ocupadas. Uma das maneiras de solucionar esse inconveniente seria mover os elementos na fila para não deixar *buracos* nela.
Esse procedimento é muito oneroso quando n for grande
Uma outra idéia é *colar os extremos da fila* considerando agora a fila como um **anel**

14.8.1 Filas Circulares

Conforme já observamos iremos representar a fila como um anel percorrido no sentido horário.
Quando $saida = entrada$ a fila estará **vazia**.
Quando $saida = n$ o próximo elemento a ser inserido é colocado na posição 1.
Se $saida - entrada = 1$ temos **fila cheia**.
O código abaixo implementa as operações de **Push** e **Pop** em filas circulares.

```
// definicoes globais
#define MAX 100
#include<stdio.h>
typedef char tipo_dado;
tipo_dado queue[MAX];

void Push(tipo_dado x , tipo_dado queue[ ] , int *entrada , int saida , int n){
 int aux;
     if ( ( saida - *entrada == 1 ) || ( (*entrada == n)&&( saida==1) ) ){
       printf("\nQueue Cheia");
       exit ();
     }
     else
      aux=*entrada%n + 1;
      queue[aux]=x;
      *entrada=aux;
}
tipo_dado Pop( tipo_dado queue[ ] , int entrada , int *saida , int n){
 int aux;
     if ( *saida == entrada ){
      printf("\nQueue Vazia");
      exit ();
     }
     else
      *saida = *saida % n + 1;
      aux = queue[*saida];
return(aux);
}
void Iniciliza_Queue(tipo_dado queue [ ] , int *entrada , int *saida , int n){
int i;
*entrada = -1 ;
*saida   = -1 ;
for ( i = 0; i < n ;i++ )queue[i]='*';
}
void Print_Array(tipo_dado queue[ ] , int entrada , int saida , int n ){
int i;
printf("\nEntrada = %d <--\nArray=",entrada);
for ( i = 0 ; i < n ;i++ )
printf("%c",queue[i]) ;
printf("\nSaida = %d -->\n",saida);
//printf("\n");
}
void main(){
  int n=4;
  int entrada;
```

```
  int saida;
  char lixo;
  Iniciliza_Queue(queue , &entrada , &saida , n);
  Print_Array(queue , entrada , saida , n);
  Push('A' , queue , &entrada , saida , n);
  Push('B' , queue , &entrada , saida , n);
  Print_Array(queue , entrada , saida , n);
  lixo=Pop(queue , entrada , &saida , n);
  printf("Elemento retirado = %c\n",lixo) ;
  Print_Array(queue , entrada , saida , n);
  Push('C' , queue , &entrada , saida , n);
  Push('D' , queue , &entrada , saida , n);
  Print_Array(queue , entrada , saida , n);
  lixo=Pop(queue , entrada , &saida , n);
  printf("Elemento retirado = %c\n",lixo) ;
  Push('U' , queue , &entrada , saida , n);
  Push('W' , queue , &entrada , saida , n);
}
```

OUTPUT

```
Entrada = -1 <--
Array=****
Saida = -1 -->

Entrada = 1 <--
Array=AB**
Saida = -1 -->
Elemento retirado = A

Entrada = 1 <--
Array=AB**
Saida = 0 -->

Entrada = 3 <--
Array=ABCD
Saida = 0 -->
Elemento retirado = B

Queue Cheia
```

Capítulo 15

Estruturas Dinâmicas

Estruturas de dados dinâmicas são estruturas que podem *alongar* ou *encolher* conforme as necessidades do programa.
Suponha por exemplo que você necessitasse construir um editor de texto. O mais comum seria considerar as linhas de seu texto como uma string de um comprimento fixo e o texto um array dessas strings.
Como sabemos para isso haveria a necessidade de alocar, à priori, o comprimento máximo da linha e a quantidade máxima de linhas. Isso obviamente causa um grande desperdício de *memória alocada* pois tanto podemos ter textos cujo comprimento da linha seja muito menor que esse máximo reservado como a quantidade de linhas no texto também pode ser muito menor que o máximo de linhas previstas.
Esse é um exemplo típico onde as estruturas que são dinâmicas devem ser usadas.
Como veremos a seguir poderemos *reservar* e *liberar* memoria em tempo de execução. Isso nos possibilitará usar apenas a quantidade de memória que é necessária.
As estruturas dinâmicas *reservam* memória no computador num lugar denominado *HEAP*. Os blocos de memória são reservados conectados utilizando-se apontadores. Quando a estrutura de dados não mais necessita do bloco de memória ele é *devolvido* à *HEAP* para ser reutilizado. Essa reciclagem torna muito eficiente o uso da memória.

15.0.2 Malloc, Free e Sizeof

A *Heap* é extremamente importante na linguagem *C* pois ela torna possível os programas, durante a execução, e com a utilização das funções **malloc**(memory allocation) e **free** reservar e liberar memória que for necessária

e não pré-reservando uma quantidade especificada como no caso dos arrays. Com o objetivo de tornar os programas *C portáveis* a linguagem disponibiliza uma função que retorna o tamanho de qualquer *objeto*.
Essa é a função **sizeof()** que é usada do seguinte modo:

Exemplo 15.0.2

```
#include<stdio.h>
struct obj {
 int  inteiro ;
 float  real ;
 char caracter;
 int a [3];
} objeto;
void main(){
printf ("Tamanho do inteiro = %d bytes\n",sizeof(int));
printf ("Tamanho do float = %d bytes\n",sizeof(float));
printf ("Tamanho do char = %d bytes\n",sizeof(int));
printf ("Tamanho da struct definida = %d bytes\n",sizeof(objeto));
}
```

OUTPUT

```
Tamanho do inteiro = 4 bytes
Tamanho do float = 4 bytes
Tamanho do char = 4 bytes
Tamanho da struct definida = 24 bytes
```

As funções **malloc** e **free** funcionam da seguinte maneira:
Suponha que você necessite reservar (alocar) uma certa quantidade de memória durante a execução de seu programa.
Para tanto você *invoca*, a qualquer tempo, a função **malloc** que requisita um bloco de memória, medido em bytes, da *heap*.
O sistema operacional irá reservar o bloco de memória solicitado e você poderá utilizá-lo do modo que lhe convier.
Após a utilização do bloco você pode liberá-lo novamente para o sistema operacional através da função **free**.

O exemplo abaixo mostra uma simples utilização desses conceitos.

```
1  #include<stdio.h>
2  #include <stdlib.h>
3  int main() {
4      int *p;
5      p = (int *)malloc(sizeof(int));
6      if (p == 0)
7      {
8          printf("Erro: memoria esgotada\n");
9          return(1);
10     }
11     *p = 5;
12     printf("Endereco do bloco: %d\n", p);
13     printf("Conteudo do bloco: %d\n", *p);
14     free(p);
15 return(0);
16 }
```

Output

```
Endereco do bloco: 5309392
Conteudo do bloco: 5
```

A linha (5) do programa invoca a função **malloc**.
Essa diretiva fará o seguinte:

A função examina a quantidade de mémoria disponível na *heap* e pergunta: a quantidade existente é suficiente para reservar o bloco do tamanho solicitado ?

Observe que o tamanho do bloco é passado através do parâmetro da função **malloc** e com o auxilio da função **sizeof()**

- Caso não haja memória suficiente na *heap* a função retorna o endereço 0 or NULL (que é sinônimo).
- Caso haja memória disponível o sistema reserva a quantidade de memória requerida e coloca na variável apontadora (neste caso p) o endereço do bloco reservado. A variável apontadora contem, ela própria, um endereço.
- Agora o bloco alocado está pronto para armazenar um valor do tipo especificado (int nesse caso) e o apontador estará apontando para ele.

  O comando (11) armazena o valor inteiro 5 no bloco de memória reservado.

  Os comandos (12) e (13) imprimem o endereço do bloco reservado e seu conteúdo.

  O comando (14) libera o bloco de memória para o sistema operacional.

15.1 Questões importantes

É realmente importante verificar se o apontador não é nulo após cada pedido de alocação de memória ?
Sim é de fundamental importância pois como *heap* varia de tamanho constantemente enquanto existem programas sendo executados, nunca poderemos ter a garantia que a solicitação teve sucesso.
Você deve checar sempre se a operação teve sucesso.
O que acontece se esquecermos de liberar um bloco de memoria após o termino de um programa que a requisitou ?
Quando o programa termina ele automaticamente faz a liberação dos blocos e memoria requisitados.
É no entanto considerada de má programação os códigos que não levam em consideração essa preocupação.
Note que podemos atribuir a um apontador o valor 0, ou seja ele aponta para 0. Nesse caso o deve ficar claro que ele não está apontando para um bloco, ele simplesmente contem o endereço 0. Isso é muito útil como *flag* como iremos ver posteriormente nas **listas ligadas**.
Podemos usar a seguinte sintaxe:

```
if (p == 0) {
    ...
}
// ou:
while (p != 0) {
    ...
}
```

Mas o trecho de código abaixo irá *travar* a execução do programa:

```
p = 0; *p = 5;
```

Observe que como dissemos p aponta para 0 e não para um bloco logo não é possível efetuar $*p = 5$.

Exemplo 15.1.1

```
#include <stdio.h>
struct rec {
    int i;
    float f;
    char c;
};
```

```
int main( ) {
    struct rec *p;
    p=(struct rec *) malloc (sizeof(struct rec));
    (*p).i=10;
    (*p).f=3.14;
    (*p).c='a';
    printf("%d %f %c\n",(*p).i,(*p).f,(*p).c);
    free(p);
    return 0;
}
```

Note que não é possível usar, por exemplo, a sintaxe $*p.i = 10$ pois como o operador . tem precedência sobre o operador * isso vai gerar um erro de sintaxe. Por isso a exigência do parentesis forçando a execução de * primeiro.
Na linguagem *C* é mais comum usar a seguinte sintaxe:

$p -> i = 10; \quad \Leftrightarrow \quad (*p).i = 10;$
$p -> f = 3.14; \quad \Leftrightarrow \quad (*p).f = 3.14;$
$p -> c =' a'; \quad \Leftrightarrow \quad (*p).c =' a';$

É claro que é muito mais confortável por exemplo criar uma variavel do tipo inteiro e trabalhar com ela do que criar e usar um ponteiro que aponta para um inteiro.
A seguir iremos explorar exemplos abordando algumas maneiras mais comuns e mneumônicas de definir e trabalhar com apontadores.
Uma técnica bastante comum é :

```
typedef int *IntPointer;
IntPointer p;
```

Que é o mesmo que:

```
int *p;
```

Iremos utilizar essa técnica nos exemplos a seguir pois ela torna a declaração de dados mais fácil de ler e entender.

15.2 Apontadores para Estruturas

Exemplo 15.2.1

```
#include<stdio.h>
typedef struct {
    char nome[21] ;
    char cidade[21] ;
    char estado[2] ;
} Registro ;
typedef Registro *RegPointer ;
RegPointer t ;
void main( ){
t = (RegPointer)malloc(sizeof(Registro)) ;
printf("Qtde de memoria usada pelo ponteiro t = %d bytes\n" , sizeof(t));
printf("Qtde de memoria usada pelo bloco = %d bytes\n" , sizeof(*t));
strcpy(t->nome , "Mariana");
strcpy(t->cidade , "Bonito");
strcpy(t->estado , "MS");
printf("%s\n%s\n%s\n" , t->nome , t->cidade , t->estado);
free(t);
}
```

Output

```
Qtde de memoria usada pelo ponteiro t = 4 bytes
Qtde de memoria usada pelo bloco  = 44 bytes
Mariana
Bonito
MS
```

15.3 Ligando Estruturas

É possível criar estruturas que são capazes de apontar para estruturas idênticas. Essa capacidade nos possibilitará criar uma **lista ligada** (linked list).
O exemplo abaixo mostra como fazer isso.

```
typedef int tipo_dado ; //pode ser mudado
typedef struct{
        tipo_dado dado;
        struct Addr *next;
} Addr;
Addr *first ; // apontador first para Addr
```

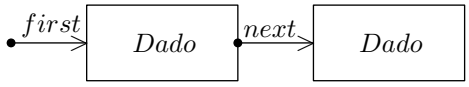


Figura 15.1: Lista Ligada

Exemplo 15.3.1
O exemplo a seguir utiliza a estrutura dinâmica de uma pilha para resolver o problema do labirinto.
Diz a lenda que o herói grego Teseu derrotou o Minotauro, monstro que habitava o célebre labirinto mantido pelo rei Minos, na ilha de Creta.
Antes de penetrar no labirinto do Minotauro ele recebeu de Ariadne, filha de Minos, rei de Creta, um novelo de lã para marcar o caminho de volta e desse modo ele conseguiu retornar de sua aventura.
Vamos modelar o problema da seguinte maneira:

- O labirinto será representado por uma matriz chamada *lab* de dimensão 10×10.
- $lab(i, j) = 1$ significa que (i,j) é uma *cela de passagem livre* enquanto que $lab(i, j) = 0$ significa uma *cela de passagem interrompida.*
- Vamos considerar os seguintes movimentos como possíveis a partir de uma cela (i, j).

  leste : $(i + 1, j)$
  oeste : $(i - 1, j)$
  norte : $(i, j + 1)$
  sul : $(i, j - 1)$

- O programa prevê uma configuração inicial indicando os possíveis caminhos ou seja as *celas de passagem livre.*
- As bordas do labirinto serão consideradas celas de passagem interrompida.
- O usuário fornece a *cela inicial* e a *cela final* e o programa verifica se existe um caminho ligando essas celas.
- A saída do programa será um caminho (uma seqüência de celas) ligando a *cela inicial* e a *cela final* ou uma mensagem indicando a não existência de caminho.

  No nosso caso uma pilha fará o papel do novelo de lã.

```
// Problema do Labirinto.
struct cela {
        int x;
        int y;
};

typedef struct cela stack_dado;
struct stack_rec {
    stack_dado dado;
    struct stack_rec *next;
};
struct stack_rec *top;
void stack_init () //  Inicializa  a pilha
{
    top=0;
}
void stack_push(stack_dado d){ /* Pushes the value d onto the
                                  stack. */
    struct stack_rec *temp;
    temp=(struct stack_rec *)malloc(sizeof(struct stack_rec));
    temp->dado=d;
    temp->next=top;
    top=temp;
}
stack_dado stack_pop(){
    struct stack_rec *temp;
    stack_dado d;
    if (top!=0)
    {
        d.x = top->dado.x;
        d.y = top->dado.y;
        temp = top;
        top = top->next;
        free(temp);
    }
    return(d);
}
void Move(int dir , struct cela u , struct cela *v ) {
switch ( dir ){
    case 0 : v->x = u.x - 1 ; v->y = u.y ; break ;
    case 1 : v->x = u.x + 1 ; v->y = u.y ; break ;
    case 2 : v->x = u.x     ; v->y = u.y - 1 ; break ;
    case 3 : v->x = u.x     ; v->y = u.y + 1 ; break ;
    default: printf("Erro no Move");break;exit(0);
}
```

```
}
int main() {
  struct cela dado , aux , inicio  , fim ;
  struct cela caminho[100];
  int lab [10][10];
  int i , j , achei = 1 , k , vizinho ;
  for (i = 0 ; i <= 9 ; i++)
  for (j = 0 ; j <= 9 ; j++) lab[i][j]=0;
// Inicializa Labirinto
   lab [1][1]=1 ; lab [1][2]=1 ; lab [2][1]=1;
   lab [3][1]=1 ; lab [3][2]=1 ; lab [3][3]=1;
    printf("Entre com a posicao inicial (xi,yi): " );
   scanf("%d%d" , &inicio.x , &inicio.y );
    printf("Entre com a posicao final   (xf,yf): " );
   scanf("%d%d" , &fim.x , &fim.y );
   lab[ inicio .x][ inicio .y]=1;  // marque a posicao inicial como possivel
   lab[fim.x][fim.y]=1;  // marque a posicao final como possivel
    stack_init ();
   i=-1;
    stack_push( inicio );
    while (top){
     dado=stack_pop();
     lab[dado.x][dado.y ]=0; // marcando como ja visitado
      caminho[++i] = dado; // marcando um possivel caminho
     if ( ( dado.x == fim.x) && ( dado.y == fim.y) ){
       achei=0;
       printf("Caminho:");
       for (j = 0 ; j < i ; j++)
          printf("(%d,%d)->" , caminho[j].x , caminho[j].y);
       printf("(%d,%d)" , fim.x , fim.y);
       break;
     }
     else{
     k=-1;vizinho=0;
     while (++k <= 3){
     Move( k , dado , &aux );
     if ( lab[aux.x][aux.y] == 1 ){
     lab[aux.x][aux.y ]=0; // marcando como ja visitado
     vizinho=1;
     stack_push(aux);
     }
    }
    if (vizinho==0) i-- ; // volte para a cela anterior
    }
   }
```

```
 if (achei==1)
 printf("Nao existe caminho ligando (%d,%d) e (%d,%d)" ,
          inicio.x, inicio.y , fim.x , fim.y);
 return(0);
}
```

Output:

```
Entre com a posicao inicial (xi,yi): 1 1
Entre com a posicao final   (xf,yf): 3 3
Caminho:(1,1)->(2,1)->(3,1)->(3,2)->(3,3)
```

Referências Bibliográficas

[1] The *C* Programing Language
Brian W. Kernighan & Dennis M. Ritchie
Prentice-Hall Softwares Series

[2] Algorithms + Data Structures = Programs
Niklaus Wirth
Prentice-Hall, Inc.
Englewood Cliffs, New Jersey.

[3] The Computer
from Pascal to von Neumann
Herman H. Goldstine
Princeton University Press.