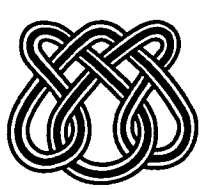
# UNIVERSIDADE DE SÃO PAULO

Introdução a Computação com a Linguagem C

André Carlos Ponce de Leon Ferreira de Carvalho

Nº 50

## NOTAS DIDÁTICAS



**Instituto de Ciências Matemáticas de São Carlos**

UNIVERSIDADE DE SÃO PAULO
Instituto de Ciências Matemáticas e de Computação
ISSN 0103-2585

# Introdução a Computação com a Linguagem C

André Carlos Ponce de Leon Ferreira de Carvalho

Nº 50

NOTAS

DIDÁTICAS

São Carlos
Jul./2001

# Introdução a Computação com a Linguagem C

Prof. André Carlos Ponce de Leon Ferreira de Carvalho
SCE-ICMC-USP

Este material contém os slides utilizados em cursos de Introdução a Computação utilizando a Linguagem C. Ao todo, são 10 aulas cobrindo os seguintes tópicos:

- Evolução histórica dos computadores
- Componentes básicos de um sistema computacional
  - Hardware
  - Software
- Noções básicas sobre sistemas operacionais
- Introdução à construção de algoritmos
- Linguagens de programação
- Linguagem C
- Variáveis e tipos de dados
  - Tipos básicos
  - Vetores e matrizes
  - Ponteiros
  - Registros
- Operadores
  - Simples
  - Compostos
- Funções
- Bibliotecas de funções
- Entrada e saída
- Aplicações

Prof. André Carlos Ponce de Leon Ferreira de Carvalho

# Introdução à Computação

## Fundamentos

Professor: André de Carvalho

## Sobre o Curso

- Objetivo
  - Familiarização com os conceitos da computação e com linguagens algorítmicas científicas
  - Mostrar como esse conhecimento pode ser utilizado na área de engenharia civil

## Sobre o Curso

- Conteúdo
  - Evolução histórica dos computadores
  - Componentes básicos de um sistema computacional
  - Noções básicas sobre operação de microcomputadores e Sistemas Operacionais
  - Introdução à construção de algoritmos; estruturas básicas de programação algorítmica; construção de algoritmos por refinamentos sucessivos
  - Tradução de algoritmos para a linguagem C
  - Prática de programação estruturada em C com aplicações

## Sobre o Curso

- Livros
  - SCHILDT, H. **C Completo e Total**, Makron Books1997
  - CARROL, D.W. **The Art of Programming, Computer Science with C**, West, 1996
  - DEITEL, H. M.; DEITEL, P.J. **C How to program**, Prentice Hall, 1994
  - KERNIGHAN, B. W. ; RITCHIE, D. M. **C - A linguagem de programação padrão ANSI**, Editora Campus, 1995

## Sobre o Curso

- Livros
  - ROBERTS, E. **Programmin Abstractions in C**, Addison Wesley, 1996
  - KERNIGHAN , B. W.; PIKE, R. **A Prática da Programação**, Editora Campus, 2000
  - TREMBLAY,J.P.; BUNT,R.B. **Ciências dos Computadores: Uma Abordagem Algorítmica**, São Paulo, McGraw-Hill, 1983
  - WIRTH,N. **Programação Sistemática**, Rio de Janeiro, Campus, 1978

## Aula de hoje

- Informações sobre o curso
- Informática
- Modalidades de Computadores
- Evolução dos Computadores
- Gerações modernas

## Informática

- Informática = informação automática
  - Tratamento da informaç ão de modo automáti co
- Informática pressupõe o uso de computadores eletrônicos no trato da informação
- Cabe à informática a tarefa de coletar, tratar e disseminar dados, gerando informação

## Informática

- Definições importantes
  - Dados: Elementos conhecidos de um problema
    - Ex.: Peso, altura, salário
  - Informação: Um conjunto estruturado de dados, transmitindo conhecimento
    - Ex.:Registro de informações de um emprega do

## Informática

- Computadores podem ser classificados
  - Pelo porte
  - Pela natureza do uso
  - Pelo número de processadores
    - Um único processador
    - Vários processadores
  - Pelos recursos disponíveis
    - Geral
    - Internet

## Informática

- Quanto ao porte
  - Grande porte ou mainframes
    - Muito usados em grande s empresas
  - Médio porte
    - Caindo de uso
  - Microcomputadores
    - Usados em redes, estão substituindo os mainframes

## Informática

- Quanto à utilização
  - Comerciais: permitem o trato rápido e seguro de problemas que trabalham com grande volume de dados
  - Industriais: utilizados para controlar processos na indústria
  - Científicos: usado nas áreas de cálculos e pesquisas científicas, onde são necessários processamentos mais precisos, rápidos e sofisticados

## História do Computador

- Homens começaram contando com os seus dedos
  - Precisa de formas de medir meses e estações para realizar festivais e cerimônias religiosas no tempo certo

## História do Computador

- Primeiro computador automático: abacus
  - Origem: China
  - Não é realmente uma "máquina automática"
  - Lembrava o usuário do estado atual do cálculo
    - Permitia a realização de operações matemáticas mais complexas que as que poderiam ser realizadas só com os pés e as mãos

## História do Computador

- Pascaline
  - Blaise Pascal, matemático francês desenvolveu em 1642 uma das primeiras calculadores mecânicas
  - Queria ajudar o pai no cálculo de impostos
  - Caixa cheia de engrenagens, rodas e visores
  - Calculava as quatro operações básicas
  - 50 pascalines foram produzidas

## História do Computador

- Engenho analítico
  - Proijetado1830 por Charles Babbage em
  - Era para ser uma máquina automaticamente sequenciada (programada) que poderia realizar vários cálculos – nunca foi construída
  - Sequências seriam controladas por cartões perfurados
  - Augusta Byron desenvolveu sequências sofisticadas para a máquina (primeira programadora)

## História do Computador

- Lord Kelvin (1872) - $1^0$ computador de grande porte
  - Analógico
  - Previa a altura das marés nos portos ingleses
  - Construído com pesos e polias que simulavam os efeitos do sol, da lua e dos ventos nas marés
  - Os resultados eram mostrados em gráficos

## História do Computador

- Máquina de tabulação
  - Desenvolvida em 1890 por Herman Hollerith para processar dados do censo americano
    - O Censo de 1880 demorou 12 anos para ter seus resultados compilados
    - Censo de 1890, com a máquina de Hollerith, foi processado em três anos
    - Uma das empresas que formou a IBM

## História do Computador

- ENIAC, máquina Atanasoff-Berry e Z3
  - Desenvolvidos em 1941
  - Primeiros computadores verdadeiramente programáveis
  - ENIAC
    - Pesava 30 toneladas, tinha 19000 válvulas e ocupava uma sala de 500 metros quadrados
    - Construído na University of Pensylania e usado pela primeira vez em 1945

## Gerações Modernas

- Primeira geração
  - Circuitos eletrônicos que usavam válvulas como seus principais componentes
    - Dispositivo que conduz a corrente elétrica em um só sentido
  - Operações internas em milisegundos ($10^{-3}$ segundos)
  - ENIAC, máquina Atanasoff-Berry e Z3

## Gerações Modernas

- Segunda geração
  - Válvulas foram substituídas por transistores
    - Amplificador de cristal, inventado nos EUA, em 1948, para substituir a válvula (Nobel de 1956)
    - Computadores se tornaram menores e mais rápidos, baratos e confiáveis
  - Operações em microsegundos ($10^{-6}$ segundos)
  - IBM 709 TX (introduzido em 1958)

## Gerações Modernas

- Terceira geração
  - Circuitos integrados (SSI e MSI)
    - Circuito eletrônico formado por um grande número de componentes arrumados em um chip (uma "pastilha" de semicondutor) de poucos centímetros ou milímetros quadrados
    - SSI - integração em pequena escala - menos de 10 elementos por chip
    - MSI - integração em média escala - 10 a 100 elementos por chip

## Gerações Modernas

- Terceira geração
  - Operações em nanosegundos ($10^{-9}$ segundos)
  - IBM 360 (introduzido em 1964)

## Gerações Modernas

- Quarta geração
  - Tecnologia de firmware, integração em escalas superiores (LSI,VLSI, SCSI, ULSI)
    - Firmware: Programa (ou software de modo geral) armazenado em chip
    - LSI - integração em grande escala - 100 a 500 elementos por chip
    - VLSI - integração em muito grande escala - 5.000 a 50.000 elementos por chip
    - SCSI - integração em super grande escala - 50.000 a 100.000 elementos por chip
    - ULSI - integração em ultra grande escala - mais de 100.000 elementos por chip

## Gerações Modernas

- Quarta geração
  - Operações em picosegundos ($10^{-12}$ segundos)
  - PDP-11 e IBM 370 (introduzidos em 1970 e 1971)

## Gerações Modernas

- Quinta geração
  - O que quer que seja, nos estamos ansiosamente esperando por ela
  - Avanços na computação ainda estão vindo rapidamente
    - Assim que um novo produto é liberado, sua substituição já esta na fase de projeto ou testes

## Gerações Modernas

- Se a indústria automobilística tivesse experimentado a mesma explosão tecnológica, um "carro popular" :
  - Seria capaz de carregar 100 pessoas
  - Seria capaz de andar a quase 1000 quilômetros por hora
  - Seria do tamanho de uma formiga
  - Custaria 50 centavos

## Conclusão

- Como vai ser o curso
- Objetivos e ementa
- Exames
- Alguns conceitos introdutórios
- História dos computadores
- Gerações modernas

## Agradecimentos


- Parte das informações destes slides foram obtidas de slides anteriores de:
  - Profa. Rosely Sanches
  - Profa. Sandra Aluísio
  - Profa. Solange Rezende
  - Profa. Renata Fortes

# Introdução à Computação

## Sistemas de Computação

Professor: André de Carvalho

## Aula de hoje

- Sistemas de Computação
  - Hardware
  - Software
- Memória
- Dispositivos de entrada e saída
- Conclusão

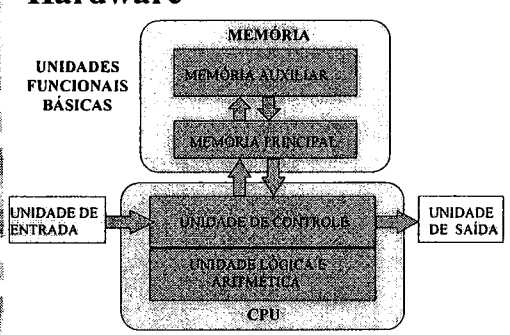
## Sistemas de Computação

- Hardware
  - As partes física, palpáveis de um computador
  - Ex.: Teclado, monitor, fios, placas
- Software
  - Programas e dados
  - Um programa é um conjunto de instruções
  - Ex.: Sistemas operacionais, Compiladores
- Um computador precisa de hardware e software
  - Cada um é inútil sem o outro

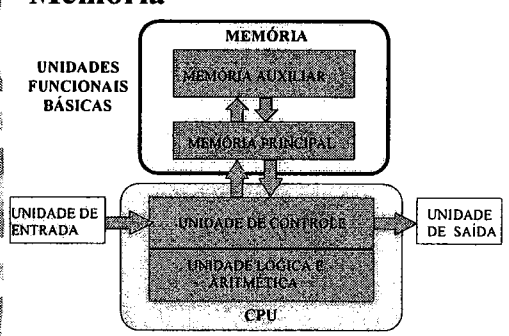
## Hardware

- Arquitetura de uma casa
  - Quartos, salas, cozinha, banheiros
- Arquitetura de um computador
  - CPU
  - Memória principal
  - Memória secundária
  - Dispositivos de entrada e saída

## Hardware




## Memória



## Analógico *versus* Digital

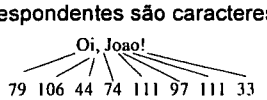
- Existem duas formas de armazenar e gerenciar dados
  - Analógica
    - Contínua, proporcional aos dados representados
    - Ex.: música em um disco de vinil
  - Digital
    - Informação é quebrada em pedaços (partes) e cada pedaço é representado separadamente
    - Ex.: música em um CD

## Informação Digital

- Computadores armazenam toda informação digitalmente
  - Números
  - Textos
  - Figuras e imagens
  - Sons
  - Vídeos
  - Instruções de um programa
  - Assim, tods informação é digitalizada (quebrada em pedaços e representada como números

## Representando Textos Digitalmente

- Por exemplo, todo caractere é armazenado como um número
  - Incluindo espaços, dígitos e sinais de pontuação
  - Letras maiúsculas e minúsculas correspondentes são caracteres diferentes




## Números Binários

- Ao ser digitalizada, a informação é representada e armazenada na memória usando o sistema de números binários
  - Um dígito binário (0 ou 1) é chamado de bit
    - Dispositivos que armazenam e movem informação são mais baratos e confiáveis se eles têm que representar apenas dois valores
    - Um bit pode representar dois estados possíveis
      - Como uma luz que está ligada (1) ou desligada (0)
    - Combinações de bits são usadas para armazenar valores

## Combinações de Bits

| 1 bit | 2 bits | 3 bits | 4 bits |
|---|---|---|---|
| 0 | 00 | 000 | 0000 1000 |
| 1 | 01 | 001 | 0001 1001 |
| | 10 | 010 | 0010 1010 |
| | 11 | 011 | 0011 1011 |
| | | 100 | 0100 1100 |
| | | 100 | 0101 1101 |
| | | 100 | 0110 1110 |
| | | 100 | 0111 1111 |

Cada bit adicional dobra o número de possíveis combinações

## Combinações de Bits

- Cada combinação pode representar um item distinto
  - Existem $2^N$ combinações de N bits
    - Assim, N bits são necessários para representar $2^N$ itens

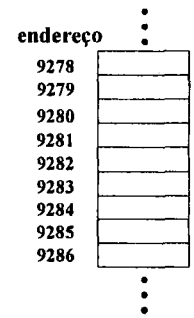
Quantos itens podem ser representados por:

| | |
|---|---|
| 1 bit ? | $2^1 = 2$ itens |
| 2 bits? | $2^2 = 4$ itens |
| 3 bits? | $2^3 = 8$ itens |
| 4 bits? | $2^4 = 16$ itens |
| 5 bits? | $2^5 = 32$ itens |

## Especificações de um computador

- Considere as seguintes especificações para um computador pessoal:
  - Processador Pentium IV 900 MHz
  - Memória RAM de 256 MB
  - Hard Disk (disco rígido) de 16 GB
  - CD ROM 24x
  - Video com recursos de multimídia de 17" com resolução de 1280 x 1024
  - Modem de 56 KB
- O que isso significa?

## Memória



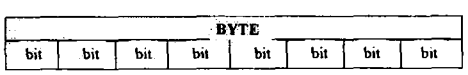

A memória é dividida em várias pequenas posições de memória do mesmo tamanho (palavras)

Cada posição de memória tem um endereço numérico que a identifica unicamente

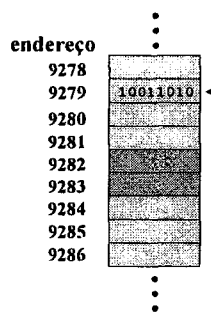
Os endereços são permanentes (vêm da fábrica) e não podem ser modificados pelo programador

## Memória

- Byte (binary term) : Unidade básica da informação
  - Composto por 8 bits




## Memória

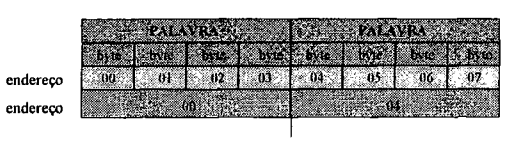



Toda posição de memória armazena um conjunto de bits (geralmente um byte)

Valores grandes são armazenados em posições consecultivas

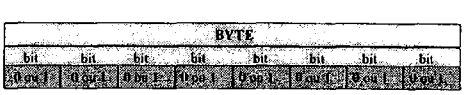
## Memória

- Uma palavra é formada por um grupo de 2, 4, 6 e até 8 bytes
  - Depende do modelo de computador
  - Exemplo: Palavra de 4 bytes




## Memória

- Os computadores armazenam informações e fazem todo seu tratamento baseado em fenômenos sobre sistemas biestáveis
  - Os símbolos básicos usados para representar os dois estágios são o 0 e o 1 (dígitos binários)



## Memória

- Todo dispositivo de memória tem uma capacidade de armazenamento
  - Indica quantos bits ele armazena
  - Capacidades são expressas em várias unidades

| Unidade | Símbolo | Número de Bytes |
|---|---|---|
| kilobyte | KB | $2^{10}$ = 1024 |
| megabyte | MB | $2^{20}$ (mais de 1 milhão) |
| gigabyte | GB | $2^{30}$ (mais de 1 bilhão) |
| terabyte | TB | $2^{40}$ (mais de 1 trilhão) |

## Memória

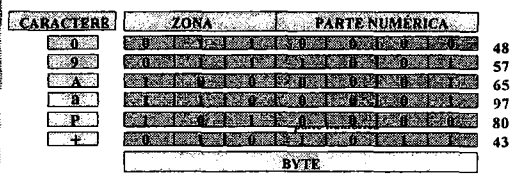
- Interligando todas as partes do computador, existem fios
  - Por onde "circulam" os bits
  - Computador de 16 bits ⇒ possui 16 fios para o transporte dos dados
  - Computador de 32 bits ⇒ possui 32 fios para o transporte dos dados

## Memória

- Armazenamento de Informações NÃO numéricas
  - Realizada através de um esquema de codificação
- Métodos de codificação mais populares na indústria de computadores:
  - Código EBCDIC (de 8 bits) - *Extended Binary Coded Decimal Interchange Code*
  - Código ASCII ( de 7 bits) - *American Standard Code for Information Interchange*

## Memória

- Código ASCII (7 bits)
  - Cada byte armazena um caractere: algarismo, letra, símbolo ou caractere de controle




## Memória

- Código ASCII (7 bits)
  - Possibilidade de $2^7$ representações diversas (128 caracteres)
    - Alfabeto inglês em letras minúsculas e maiúsculas (52)
    - Caracteres decimais numéricos (10)
    - Caracteres especiais e de operação (33)
    - Caracteres de controle (33)

## Memória

- Armazenamento de Informações numéricas
  - A representação de grandezas numéricas depende de:
    - Arquitetura do computador
    - Tipos de dados de cada linguagem
  - Linguagens voltadas para a área científica apresentam tipos de dados com maior precisão e complexidade
    - Possibilitam cálculos mais complexos

## Memória

A memória é dividida em camadas:

- Memória cache
- Memória principal
- Memória auxiliar

## Memória

Memória cache

- Camada mais próxima do processador
- Funcionamento muito rápido
- Custo elevado
- Pequena (por causa do custo)
- Dados são perdidos quando o computador é desligado (é volátil)

## Memória

- Memória principal
  - Armazena dados que não cabem na memória cache
  - Mais lenta
  - Maior que a cache
  - Custo inferior à cache
  - Dados são perdidos quando o computador é desligado (é volátil)

## Memória

- Memória auxiliar (discos magnéticos)
  - Armazenam os dados que não cabem na memória principal
  - Podem reter grande quantidade de dados
  - Os dados não são perdidos quando o computador é desligado (não é volátil)
  - Funcionamento muito lento

## Memória

- Memória auxiliar (discos magnéticos)
  - Existem dois tipos de disco:
    - discos magnéticos rígidos: winchesters, hard disk
    - discos magnéticos flexíveis: disquetes ou floppy
  - Os dados e programas devem primeiro ser transferidos para a memória principal antes de serem processados

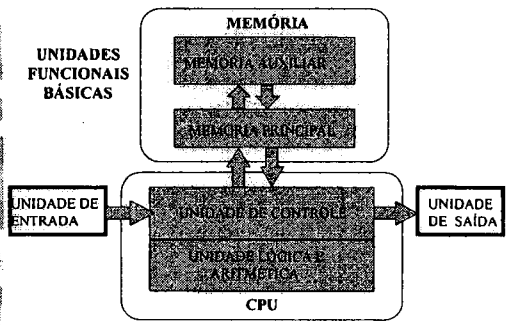
## Memória

- Observações
  - Memória principal e discos são dispositivos de acesso direto
    - Informação pode ser atingida diretamente
    - Tanto faz usar os termos acesso direto como acesso aleatório
  - Fita magnética é um dispositivo de acesso sequencial
    - Dados são armazenados em ordem linear
    - Pode ser necessário passar por dados intermediários para acessar outros dados

## Memória

- Observações
  - RAM: Randon Acess Memory (memória de acesso aleatório) – acesso direto
  - ROM: Read-Only Memory (memória só de leitura)
    - Chips de memória ou um dispositivo separado (CD-ROM)
  - Os termos RAM e memória principal se referem a mesma coisa
  - Tanto RAM quanto ROM são dispositivos de memória de acesso aleatório (direto)

## Unidades de entrada e saída




## Unidades de entrada e saída

- Monitor de vídeo
  - Formado por TRC (Tubo de Raios Catódicos) ou MCL (Monitor de Cristal Líquido)
  - Com relação a cor, os monitores podem ser
    - Monocromáticos (MDA): verde, âmbar ou branco
    - Policromáticos: pelo menos 4 cores

## Unidades de entrada e saída

- Monitor de vídeo
  - Resolução Gráfica (máxima): define a qualidade da imagem exibida pelo monitor
    - Indica o números de elementos de imagens (pixels) que ele pode exibir
    - Resolução mais alta (mais pixels) produz imagens com mais definição

## Unidades de entrada e saída

- Monitores Policromáticos:
  - CGA: processa até 4 cores simultaneamente e possui baixa resolução gráfica
  - EGA: processa até 16 cores simultaneamente e tem média resolução gráfica
  - VGA: processa mais de 16 cores simultaneamente, com alta resolução gráfica
  - Super VGA: mostra 256 cores simultaneamente, com altíssima resolução gráfica

## Unidades de entrada e saída

- Monitor
  - O tamanho de um monitor é medido diagonalmente (17"), como uma tela de televisão
  - A maioria dos monitores atuais tem recursos de multimídia
    - Som e imagem

## Unidades de entrada e saída

- Teclado
  - Semelhante ao teclado da máquina de escrever
    - Com algumas teclas incomuns na datilografia e um teclado numérico reduzido
- Impressora
  - Escreve dados em papel
  - Matriciais, de agulhas, jatos de tinta ou a laser

## Unidades de entrada e saída

- Unidade de disco magnético flexível
  - Destinadas a disquetes de alta densidade, dupla face e 3 1/2 polegadas de diâmetro (Disk Drive ou Drive)
- Tela sensível ao toque
  - Permite ao usuário selecionar opções através de indicações sobre o vídeo
    - Vídeo se torna um painel sensível a pressões

## Unidades de entrada e saída

- Leitora de código de barras
  - Formada, basicamente, por um foto detector e em um decodificador, acondicionados em um dispositivo (caneta ótica)
    - A caneta ótica fornece ao computador as informações contidas na etiqueta
  - Há diversos métodos de codificação de dados através de barras
    - Todos se baseiam em sucessão de listras escuras de largura variável intercaladas por espaços claros, passível de ser interpretada por processos óticos

## Unidades de entrada e saída

- CD-ROM (Compact-Disk Read-Only Memory)
  - Forma mais moderna de armazenamento de dados
    - Utiliza técnicas óticas de laser, em vez do eletro magnetismo
  - CD pode armazenar caracteres, sons e imagens
    - Elemento básico dos sistemas multimídias
      - Sistemas cujas unidades de entrada e saída trabalham com várias formas distintas de mídia ao mesmo tempo)

## Unidades de entrada e saída

- CD-ROM (Compact-Disk Read-Only Memory)
  - Utilizam a mesma tecnologia:
    - WORM (Write Once, Read Many) - discos óticos que podem ser gravados apenas uma vez, mas lidos inúmeras vezes
    - Discos apagáveis (Magneto Optical Erasable Disk) - discos regraváveis
      - Permitem inúmeras atualizações

## Unidades de entrada e saída

- Mouse
  - Apontador eletrônico
    - Seus movimentos são transmitidos a um cursor em forma de seta
    - Usado para entrada de dados e instruções
    - Substitui, com vantagem, o teclado em uma série de tarefas, mas não o torna indispensável (Ex: entrada de textos)
- Trackball ou ballpoint
  - Precursor do mouse, é usado no teclado de computadores portáteis (notebooks)

## Unidades de entrada e saída

- Scanner ou digitalizador
  - Dispositivo de entrada que digitaliza objetos escritos sobre papel ou qualquer outro meio
    - Desenhos (figuras) e fotografias
  - Armazena essas informações sob forma de sinais digitais, em arquivos
  - As mesas que suportam scanners são chamadas de mesas digitalizadoras

## Unidades de entrada e saída

- Plotter
  - Dispositivo de saída que produz desenhos, gráficos e diagramas baseados em linhas contínuas
  - Linhas são traçadas através de movimentos de elementos traçadores sobre superfície de papel ou outro meio

## Unidades de entrada e saída

- Joystick
  - Unidade de entrada utilizada para controle de cursor, deslocamento de imagens na tela e comandos de eventos.
  - É munido de hastes e botões

## Unidades de entrada e saída

- Modem
  - Dispositivo de transferência de dados
    - Modula e demodula sinais digitais
  - Permite que informação seja enviada e recebida entre computadores diferentes
    - Vários computadores incluem um modem, que permite que a informação seja transferida através de linha telefônica
  - Velocidade de transferência de informação é definida por uma taxa de transferência de dados
    - Ex.: 56.000 bps (bits por segundo)

## Unidades de entrada e saída

- Fax-modem
  - Placa de fax que pode ser encaixada em um slot de expansão do microcomputador
  - Proporciona recursos semelhantes ao de um fac-simile (fax) comum
- Sintetizador de voz
  - Dispositivo para saída de informações via voz
    - Funciona através da codificação e repetição de frases pré-comunicadas
      - Ou por meio de transformação de sinais digitais em sinais sonoros

## Unidades de entrada e saída

- Outros dispositivos
  - Capacete com recursos multimídia
  - Luva com recursos multimídia
  - Óculos com recursos multimídia
  - Nariz artificial

## Conclusão

- Sistemas de Computação
  - Hardware
  - Software
- Memória
- Dispositivos de entrada e saída

## Agradecimentos


- Parte das informações destes slides foram obtidas de slides anteriores de:
  - Profa. Solange Resende
  - Profa. Rosely Sanches
  - Profa. Sandra Aluísio
  - Profa. Renata Pontin Fortes
  - Prof. José Carlos Maldonado

# Introdução à Computação

## Sistemas de Computação

Professor: André de Carvalho

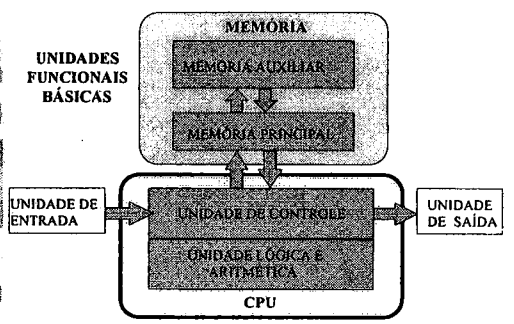
## Aula de hoje

- Sistemas de Computação
- Hardware
  - CPU
- Software
  - Linguagens de Programação
- Algoritmos
- Conclusão

## Sistemas de Computação

- Hardware
  - As partes física, palpáveis de um computador
  - Ex.: Teclado, monitor, fios, placas
- Software
  - Programas e dados
  - Um programa é um conjunto de instruções
  - Ex.: Sistemas operacionais, Compiladores
- Um computador precisa de hardware e software
  - Um é inútil sem o outro

## Unidade Central de Processamento

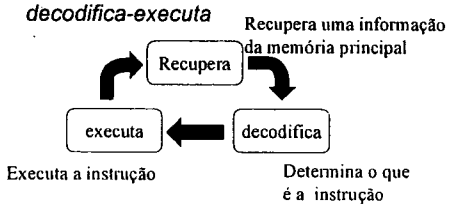



## Unidade Central de Processamento

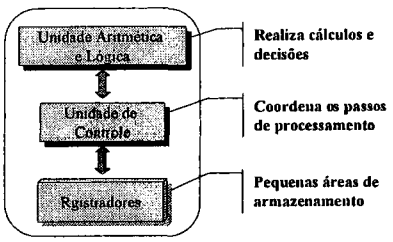
- A CPU executa as instruções
  - Instrução: comando que define completamente uma operação a ser executada
  - Programa: instruções ordenadas logicamente
- A CPU tem 2 unidades:
  - Unidade de controle: determina a execução e interpretação dos dados que estão sendo processados
  - Unidade lógica e aritmética: recebe dados da memória e executa quando uma instrução aritmética ou lógica sobre eles

## Unidade Central de Processamento

- CPU também é chamada de microprocessador
  - Segue continuamente o ciclo *Recupera-decodifica-executa*



## Unidade Central de Processamento




## Unidade Central de Processamento

- Velocidade da CPU
  - Define velocidade de processamento
  - É controlada pelo *clock* (relógio) do sistema
    - *Clock* gera um pulso eletrônico em intervalos de tempo regulares
    - Os pulsos coordenam as atividades da CPU
  - Medida em megahertz (MHz)
    - Medida de frequência de pulsos

## Unidade Central de Processamento

- Velocidade de processamento costuma ser expressa através de:
  - MIPS (milhões de instruções por segundo)
    - Ex: Pentium de 166 MIPS
  - MEGAFLOPS (milhões de operações de ponto flutuante por segundo)
    - Utilizada em sistemas voltados para aplicações numéricas (sistemas científicos e sistemas de computação gráfica)

## Software

- Um software é formado por:
  - Instruções que quando executadas produzem a função e o desempenho desejados
  - Estruturas de dados que possibilitam que os programas manipulem adequadamente a informação
  - Documentos que descrevem o funcionamento e o uso dos programas

## Software

- Classificação do softwares quanto ao tipo aplicação
  - Software básico
    - Coleção de programas escritos para dar apoio a outros programas
    - Ex.: Sistema operacional
  - Software de tempo real
    - Software que monitora, analisa e controla eventos do mundo real
    - Voltados para aplicações específicas
    - Ex.: controle de usinas, caixa eletrônico)

## Software

- Classificação do softwares quanto ao tipo aplicação
  - Software comercial
    - Sistemas de operações comerciais e tomadas de decisões administrativas
    - Ex.: Folha de pagamento
  - Software científico e de engenharia
    - Utilizado para processamento de números
    - Ex.: Cálculo estrutural

## Software

- Classificação do softwares quanto ao tipo aplicação
  - Software embutido
    - Usado para controlar produtos e sistemas industriais e de consumo
    - Ex.: Controle de máquina de lavar roup a
  - Software de computador pessoal
    - Auxilia atividades pessoais e de escritório
    - Ex.: processamento de textos, planilhas eletrônicas, diversões

## Software

- Software de inteligência artificial
  - Utilizado em problemas que exijam um comportamento semelhante ao de um especialista humano
  - Aplicado em problemas que exijam:
    - Raciocínio
    - Tomada de decisões
    - Análise de dados
    - Reconhecimento de padrões
  - Ex.: domótica

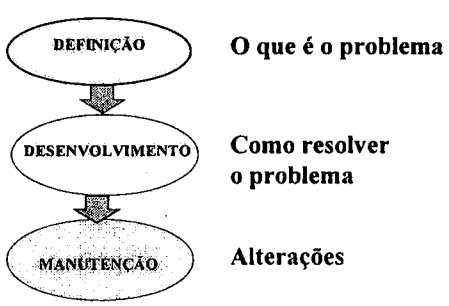
## Software

- Características
  - Desenvolvido ou projetado por engenharia, não manufaturado no sentido clássico
  - Não se desgasta, mas se deteriora
  - A maioria é feita sob medida
    - Em vez de ser montada a partir de componentes existentes

## Software

- Ciclo de vida
  - Formado pelas etapas do desenvolvimento de um software
    - As etapas envolvem **métodos**, **ferramentas** e **procedimentos** para a construção e manutenção do software
  - Contém 3 fases genéricas: **definição**, **desenvolvimento** e **manutenção**

## Fases de um ciclo de vida

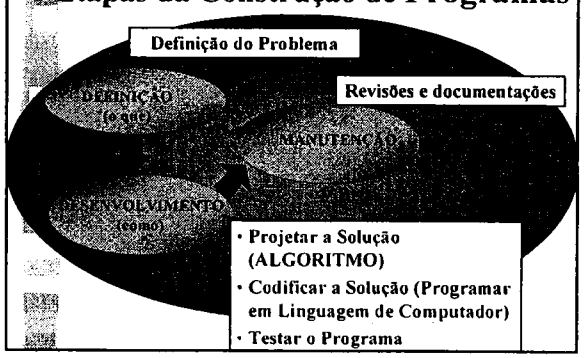



**O que é o problema**

**Como resolver o problema**

**Alterações**

## Etapas da Construção de Programas



## Sistemas Operacionais

- Sistemas Operacionais
  - Programa ou coleção de programas responsável pela supervisão dos processos executados em um computador
  - Podem ser caracterizados pelo tipo de serviço que prestam ao usuário
    - Sistemas do tipo lote (batch)
    - Sistemas de tempo compartilhado (time-sharing)
    - Sistemas de tempo real

## Sistemas Operacionais

- Sistemas do tipo lote (batch)
  - Primeiros computadores podiam realizar apenas uma tarefa de cada vez
- Sistemas de tempo compartilhado
  - Possibilita a utilização do computador por várias pessoas ao mesmo tempo
- Sistemas de tempo real
  - Podem ser utilizados continuamente
  - Respondem instantaneamente

## Sistemas Operacionais

- Possuem várias funções
  - Facilitar o uso do computador pelo usuário
  - Possibilitar o uso eficiente e controlado de vários componentes de hardware que constituem o sistema
    - Memórias principal e secundária
    - Dispositivos de entrada e saída (ex. impressora)
  - Possibilitar a diversos usuários o uso compartilhado e protegido

## Linguagens de Programação

- Computadores são ferramentas poderosas
  - Podem armazenar, organizar e processar uma enorme quantidade de informação
  - Mas eles não podem fazer nada, até que alguém lhes dê instruções detalhadas

## Linguagens de Programação

- A comunicação com computadores não é fácil
  - Eles requerem que as instruções sejam exatas e detalhadas
  - Não seria mais fácil se pudessemos escrever programas em português?
    - Ex.: Veja quanto eu preciso pagar a cada funcionário e informe a eles o que vão receber e quais descontos foram aplicados

## Linguagens de Programação

- A comunicação com computadores não é fácil
  - O problema é que português não é uma boa linguagem para das instruções precisas
    - Tente ler uma bula de remédio
  - Pode-se tentar escrever em português preciso
    - Explicar claramente cada instrução
    - Ter certeza de prever todo os casos que pudessem aparecer
    - Se trabalhassemos duro, nós conseguiríamos escrever instruções precisas em português?

## Linguagens de Programação

- Existe um grupo de pessoas que passa a maior parte do seu tempo tentando escrever em português preciso
  - As pessoas do governo
    - Os documentos que eles escrevem são chamados de regulamentos
  - Infelizmente, ao tentar fazer os regulamentos precisos, o governo os torna quase ilegíveis
  - O mesmo se aplica à manuais de instrução

## Linguagens de Programação

- Alguns anos atrás, foi aprovada uma lei na Califórnia proibindo motociclistas de andarem sem capacete
  - Logo depois de entrar em vigor, um policial parou um motociclista sem capacete
  - O motociclista sugeriu ao policial que lesse novamente a lei

## Linguagens de Programação

- A lei da Califórnia pede que:
  - Motoristas tenham um capacete à prova de quebra
  - Que o capacete seja apertado firmemente
- O guarda não pode multar o motorista, pois ele tinha o capacete apertado firmemente – ao seu joelho

## Linguagens de Programação

- Não dá para usar o português como linguagem de programação
  - Como então se comunicar com os computadores?

## Linguagens de programação

- Conjunto de comandos e estruturas de dados
  - Utilizado para informar ao computador os passos a serem executados para a realização de uma dada tarefa
- Diferentes níveis
  - Linguagens de máquina
  - Linguagens de montagem
  - Linguagens de alto nível

## Linguagens de programação

- Linguagens de máquina
  - Qualquer computador entende internamente apenas sua própria linguagem de máquina
    - Definido pelo hardware
    - Depende da máquina
  - Ex.: Programa que soma s alário base com hora extra

```
011 100110
000 100110
100 100100
```

Cada instrução é constituída de 2 partes:

| código da operação | operando |
|---|---|
| 011 | 100110 |

## Linguagens de programação

- Linguagem de montagem (*Assembly*)
  - Abreviações em inglês para representar as operações elementares dos computadores
    - Programas tradutores (*assemblers*) convertiam programas em linguagem *assembly* para programas em linguagem de máquina
  - Ex.:

    LOAD BASE
    ADD EXTRA
    STORE SALARIO

## Linguagens de programação

- Linguagem de alto nível
  - Linguagem de montagem requer várias instruções, mesmo para programas simples
  - Em linguagens de alto nível, comandos simples podem realizar um grande número de tarefas
  - Programas são convertidos para linguagem de máquina por programas tradutores
  - Ex.:
    - salario = base + extra

## Por que tantas linguagens?

- Propósitos diferentes
  - Sistemas comerciais, WWW, programas científicos, jogos
- Avanços tecnológicos
- Interesses comercias
- Cultura e background científico

## Tradutores

- Com o surgimento das linguagens de alto nível, tornaram-se necessários programas tradutores
  - Programas que traduzem programas fonte em programas objeto
  - Tradutores
    - Para linguagens de montagem ➡ montadores
    - Para linguagens de alto nível ➡ compiladores ou interpretadores

## Tradutores

- Compilador
  - Traduz o programa fonte para a linguagem de máquina, para depois executá-lo
- Interpretador
  - Decodifica unidades básicas do programa (ex. comandos) e as executa imediatamente
  - Pode ser ineficiente por decodificar repetidamente várias partes do programa fonte

## Padrões de Programação

- Nomes de variáveis com significado
- Código estruturado
- Código adequadamente tabulado
- Boa documentação
  - Nome do programador, contato
  - Descrição geral
  - Bons comentários

## Algoritmos

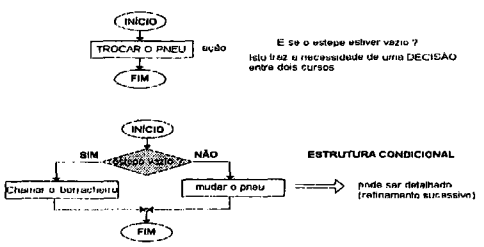
- Procedimento passo a passo para resolver um problema
- Pessoas tem inteligência e habilidade racional ⇒ fazem perguntas para esclarecer dúvidas
- Computador não tem senso próprio ⇒ deve receber instruções explícitas (algoritmos)

## Algoritmos

- Um algoritmo correto deve possuir 3 qualidades:
  - Cada passo do algoritmo deve ser uma instrução que possa ser realizada
  - A ordem dos passos deve ser precisamente determinada
  - O algoritmo deve ter fim

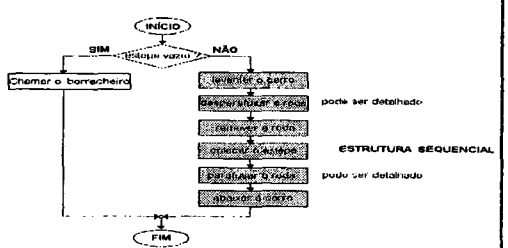
## Apresentação das Estruturas de Algoritmos

**ALGORITMO PARA TROCAR PNEU DE UM CARRO**



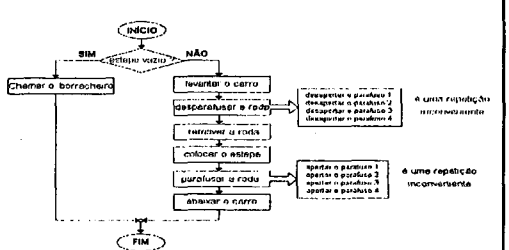

## Apresentação das Estruturas de Algoritmos

**ALGORITMO PARA TROCAR PNEU DE UM CARRO**



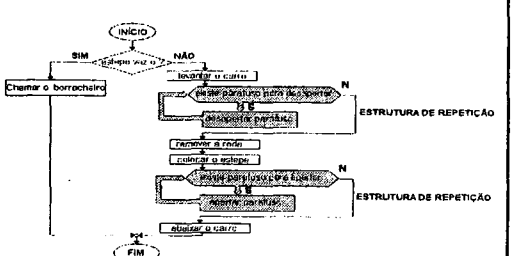

## Apresentação das Estruturas de Algoritmos

**ALGORITMO PARA TROCAR PNEU DE UM CARRO**




## Apresentação das Estruturas de Algoritmos

**ALGORITMO PARA TROCAR PNEU DE UM CARRO**



# Introdução à Computação

## Linguagem C

Professor: André de Carvalho



## Aula de hoje

- Expressões
- Operadores
- Constantes
- Bibliotecas
- Entrada e saída



## Expressões

- Expressões em C são compostas por termos e operadores
  - Ex.: *x = a + 5;*
  - Termo
    - Representa um simples valor de dado
    - Pode ser uma variável, constante ou chamada de função
      - Ex.: a, b, c, 2 e 4
  - Construída de acordo com a sintaxe da linguagem



## Expressões

- Servem para
  - Computar e atribuir valores à variáveis
  - Controlar o fluxo de execução de um programa
- Ao realizar operações, operador retorna um valor
  - Valor e seu tipo dependem do operador e do tipo dos seus operandos
    - Ex.: *x = (-b + sqrt (b * b - 4 * a * c)) / (2 * a)*

$$x = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$



## Expressões

```
/* Realiza calculo do valor de uma expressao*/
#include <stdio.h>

Main ()
   int operacao, valor;

   operacao = 1;
   valor = 0;
   while (valor < 10){
      valor = +1;
   }
   printf ("Valor da expressão eh: %d\n", valor);
}
```



## Operadores

- Executam funções sobre operandos
  - Unários: utilizam um operando
    - Pode vir antes do (*prefix*) ou após o (*postfix*) operando
    - Ex.: *count++, ++count*
  - Binários: utilizam dois operandos
    - Operador aparece entre os operandos (*infix*)
    - Ex.: *count = 0*
  - Ternários: utilizam três operandos (?:)
    - Cada parte do operador aparece entre dois operandos (*infix*)
    - Ex.: expr ? op1 : op2

## Operadores

- Operadores de C podem ser divididos em categorias
  - Aritméticos
  - Booleanos (relacionais e condicionais)
  - Bitwise
  - Atribuição



## Operadores

- Ordem como uma expressão é avaliada é importante
  - Exemplos: x * y * z; x + y / 100;
  - Ordem em que a expressão será avaliada pode ser definida explicitamente usando parênteses
    - Exemplo: x + (y / 100)
  - Operadores da linguagem C possuem uma ordem de precedência



## Operadores

| Operador | Associatividade |
|---|---|
| () [] -> . | esquerda |
| Operadores unários - ~ ++ \| & * ~ (type) sizeof | direita |
| * / % | esquerda |
| + - | esquerda |
| << >> | esquerda |
| < <= > >= | esquerda |
| == != | esquerda |
| & | esquerda |
| ^ | esquerda |
| \| | esquerda |
| && | esquerda |
| \|\| | esquerda |
| ?: | direita |
| = =op | direita |
| , | esquerda |

Maior precedência

Menor precedência

## Operadores

- Precedência cresce de baixo para cima
  - Operadores com maior precedência são avaliados primeiro
  - Operadores na mesma linha têm a mesma precedência
  - Operadores binários (exceto de atribuição) são avaliados da esquerda para a direita
  - Operadores de atribuição são avaliados da direita para a esquerda



## Operadores

- Definir valor das expressões abaixo:

a) *int y;*

*y = 7 + 4 % 2 - 5 * 4;*

b) *int y;*

*int x = 7;*

*y = (++x + 3) * 2 - 7 % 2 *(5 - 2);*



## Operadores

- Precedência
  - Informa ordem de aplicação de operadores na ausência de parênteses
    - Quando dois operadores competem pelo mesmo operando, aplica-se primeiro o de precedência mais alta
    - x = (-b + sqrt (**b * b** - **4 * a * c**)) / (2 * a)

## Operadores

- Associatividade
  - Quando dois operadores têm a mesma precedência, eles são aplicados na ordem especificada por sua associatividade
    - Indica se o operador segue para a direita ou esquerda
      - Operadores associativos-à-esquerda
        - » Maioria dos operadores de C
        - » Operador mais a esquerda é avaliado primeiro
      - Operador associativos-à-direita
        - » Operador mais a direita é avaliado primeiro

13

## Operadores

- Mistura de tipos
  - C permite incluir valores de tipos numéricos diferentes em uma expressão
    - Operandos são convertidos para um tipo comum
    - Resultado da operação é do mesmo tipo dos argumentos após a conversão
      - Garante que o resultado da operação é tão preciso quando possível
    - Ex.: sejam as variáveis *n* do tipo *int* e *x* do tipo *double*
      - *n + 1 = ?  x + 1 = ?*

14

## Hierarquia para conversão de tipos numéricos

Mais preciso

*long double*
*double*
*float*
*unsigned long*
*long*
*unsigned int*
*int*
*unsigned short*
*short*
*char*

Menos preciso

15

## Operadores aritméticos

- Operadores binários

| Operador | Uso | Descrição |
|---|---|---|
| + | *op1 + op2* | *Adiciona op1 e op2* |
| - | *op1 - op2* | *Subtrai op2 de op1* |
| * | *op1 * op2* | *Multiplica op1 por op2* |
| / | *op1 / op2* | *Divide op1 por op2* |
| % | *op1 % op2* | *Computa o resto da divisão de op1 por op2* |

16

## Operadores aritméticos

- Operador de divisão para inteiros (/)
  - Parte decimal é descartada se ambos os operandos forem do tipo inteiro (operação de truncamento)
    - Ex.: 9 / 4 = 2
  - Para que o resultado não seja truncado, pelo menos um dos operandos deve ser do tipo ponto flutuante
    - Ex. 9.0 / 4 = 9 / 4.0 = 9.0 / 4.0 = 2.25

17

## Operadores aritméticos

- Operador resto (%)
  - Computa o resto da divisão de um número inteiro por um outro número inteiro
    - Ex. 9 % 4 = 1
    - Útil para testar se um número é divisível por um outro (resto da divisão é igual a zero)
  - Cuidado com operações (/) e (%) quando pelo menos um dos operandos é negativo
    - Resultado depende da máquina
    - Geralmente a divisão é truncada para 0

18

## Operadores aritméticos

- Operadores unários

| Operador | Uso | Descrição |
|---|---|---|
| - | -op | troca o sinal de um operando |
| ++ | op++ | *Avalia o valor antes de incrementar; incrementa op de 1.* |
| | ++op | *Incrementa op de 1; avalia o valor após incrementar* |
| -- | op-- | *Avalia o valor antes de decrementar; Decrementar op de 1.* |
| | --op | *Decrementa op de 1; avalia o valor após decrementar* |

19

## Operadores aritméticos

- Exemplo:

```
main (){
   int x=0;
   printf ("x = %d\n", x++);
   printf ("x = %d\n", x);
   printf ("x = %d\n", ++x);
   printf ("x = %d\n", x);
}
```

Imprime:

x = 0
x = 1
x = 2
x = 2

20

## Operadores aritméticos

- Exemplo 2:

```
main (){
   int x, y;
   x = 5;
   y = ++x;
   printf ("x = %d, y = %d\n", x, y);
}
```

imprime: x = 6, y = 6

```
main (){
   int x, y;
   x = 5;
   y = x++;
   printf ("x = %d, y = %d\n", x, y);
}
```

imprime: x = 6, y = 5

21

## Operadores Booleanos

- C define três classes de operadores que manipulam dados Booleanos
  - Operadores relacionais
  - Operadores lógicos
  - Operador ?:

22

## Operadores Booleanos

- Vários comandos ou operadores da linguagem C utilizam valores lógicos
  - Falso ou verdadeiro
    - Valores booleanos
- Algumas linguagens de programação possuem um tipo específico para estes valores
  - Tipo booleano

23

## Operadores Booleanos

- A Linguagem C não utiliza um tipo booleano
  - Mas permite o uso do tipo *int* para expressar valores booleanos
    - Variável (constante) inteira possui valor igual a 0 → falso
    - Variável (constante) inteira possui valor diferente de 0 → verdadeiro

24

## Operadores Booleanos

- Expressões e funções lógicas retornam um valor inteiro
  - Quando o resultado de uma expressão (função) é falso, ela retorna um valor igual a 0
  - Quando o resultado de uma expressão (função) é verdadeiro, ela retorna um valor diferente de 0



## Operadores Booleanos

- Operadores relacionais
  - Comparam dois valores
  - Retornam um valor inteiro

| | |
|---|---|
| == | igual |
| != | não igual |
| > | maior |
| < | menor |
| >= | maior ou igual |
| <= | menor ou igual |



## Operadores Relacionais

- Comparam dois valores e determina o relacionamento entre eles
  - Retornam um valor inteiro

| Operador | Uso | Descrição |
|---|---|---|
| > | op1 > op2 | op1 é maior que op2 |
| >= | op1 >= op2 | op1 é maior ou igual a op2 |
| < | op1 < op2 | op1 é menor que op2 |
| <= | op1 <= op2 | op1 é menor ou igual a op2 |
| == | op1 == op2 | op1 é igual a op2 |
| != | op1 != op2 | op1 é diferente de op2 |



## Operadores Relacionais

```
main (){
   int i, j;
   printf ("digue dois numeros:");
   scanf ("%d%d", &i, &j);
   printf("%d == %d eh %d\n", i, j, i==j);
   printf("%d != %d eh %d\n", i, j, i!=j);
   printf("%d > %d eh %d\n", i, j, i>j);
   printf("%d < %d eh %d\n", i, j, i<j);
   printf("%d >= %d eh %d\n", i, j, i>=j);
   printf("%d <= %d eh %d\n", i, j, i<=j);
}
```



## Operadores Relacionais

```
/* Realiza calculo do valor de uma expressao*/
#include <stdio.h>

Main ()
   int operacao, valor;

   operacao = 1;
   valor = 0;
   if (operacao = 2){
      valor = valor + 4;
   }
   printf ("Valor da expressao eh: %d\n", valor);
}
```



## Operadores Relacionais

- Não confundir = (atribuição) com == (igual)
  - Um dos erros mais comuns
  - Compilador geralmente não pega estes erros
    - atribuição dentro de uma expressão vira uma atribuição encaixada
- Podem ser usados apenas para comparar valores de dados atômicos
  - *int, double, char, etc.*

# Operadores Lógicos

- Utilizam operandos inteiros e retornam resultado inteiro

Precedência

| Operador | Uso | Retorna verdade se |
|---|---|---|
| ! | ! op | op é falso |
| && | op1 && op2 | op1 e op2 são ambos verdade, avalia condicionalmente op2 |
| \|\| | op1 \|\| op2 | op1 ou op2 é verdade, avalia condicionalmente op2 |



# Operadores Lógicos

```
/* Realiza calculo do valor de uma expressao*/
#include <stdio.h>

Main ()
  int operacao, valor;

  operacao = 1;
  valor = 0;
  if (operacao < 2) && (valor != 3){
    valor = valor + 4;
  }
  printf ("Valor da expressao eh: %d\n", valor);
}
```



# Operadores Lógicos

- Avaliação preguiçosa
  - Em C as sub-expressões individuais de uma expressão lógica são avaliadas da esquerda para a direita
  - Avaliação termina assim que a resposta puder ser determinada
    - exp1 && exp 2
    - exp1 || exp2



# Operadores Booleanos

- Operador ?:
  - Escrito em duas parte e requer três operandos
    - (condição) ? exp1 : exp2 /* parênteses são opcionais */
      - Primeiro avalia a condição
      - Se condição é verdadeira, o resultado do operador é o valor da expressão exp1
      - Se a condição é falsa, o resultado do operador é o valor da expressão exp2
      - Ex.: max = (x > y) ? x : y



# Operadores Bitwise

- Permite manipular bits dos dados

| Operador | Uso | Descrição |
|---|---|---|
| >> | op1 >> op2 | move bits de op1 op2 posições p/ direita |
| << | op1 << op2 | move bits de op1 op2 posições p/ esquerda |
| & | op1 & op2 | bitwise and |
| \| | op1 \| op2 | bitwise or |
| ^ | op1 ^ op2 | bitwise xor |
| ~ | ~ op2 | Complemento de um |



# Operadores Bitwise

- Exemplo
  - Calcular o valor da expressão:
    - *13 >> 1;*
  - Resposta:
    - Representação binária do valor 13: 00001101
    - Movendo 00001101 uma posição para a direita: 00000110 = 6

## Operadores de Atribuição

- Comandos de atribuição
  - Atribuição de valores a variáveis em C é construída na forma de uma expressão
    - Ex.: resultado = 4;
  - Converte o tipo do valor do lado direito para o tipo da variável do lado esquerdo
    - Ex. sejam as variáveis n1 e n2 dos tipos *double* e *int*, respectivamente:
      - n1 = 0; /* n1 passa a armazenar o valor 0.0 */
      - n2 = 3.45; /* n2 passa a armazenar o valor 3 */

37

## Operadores de Atribuição

- Operador de atribuição
  - Pode conter uma expressão do lado direito
    - Ex.: z = (x = 4) * y = 3); /* equivale a: z = 12 */
    - Atribuições escritas como parte de uma expressão maior são chamadas atribuições encaixadas
      - Atribuições encaixadas devem ser usadas apenas em casos especiais, pois dificultam a leitura de programas
      - Ex.: atribuir o mesmo valor a diversas variáveis (atribuições múltiplas)
        - » n1 = n2 = n3 = 4; /* equivale a: n1 = (n2 = (n3 = 4)) */

38

## Operadores de Atribuição

- Operador de atribuição
  - Operador de atribuição básico atribui valor de uma expressão a uma variável
    - Ex1.: *int count = 0; i = i + 2;*
  - A linguagem C permite a combinação da atribuição com operadores binários (atribuição com atalho)
    - Ex. tot += val; /* equivale a: tot = tot + val */
    - Ex. res -= val; x /= 10; salario *= 2

39

## Operadores de Atribuição

| Operador | Uso | Equivalente a |
|---|---|---|
| += | | |
| -= | | |
| *= | | |
| /= | | |
| %= | | |
| &= | | |
| \|= | | |
| ^= | | |
| <<= | | |
| >>= | | |
| >>>= | | |

40

## Operadores de Atribuição

| Operador | Uso | Equivalente a |
|---|---|---|
| += | op1 += op2 | op1 = op1 + op2 |
| -= | op1 -= op2 | op1 = op1 - op2 |
| *= | op1 *= op2 | op1 = op1 * op2 |
| /= | op1 /= op2 | op1 = op1 / op2 |
| %= | op1 %= op2 | op1 = op1 % op2 |
| &= | op1 &= op2 | op1 = op1 & op2 |
| \|= | op1 \|= op2 | op1 = op1 \| op2 |
| ^= | op1 ^= op2 | op1 = op1 ^ op2 |
| <<= | op1 <<= op2 | op1 = op1 << op2 |
| >>= | op1 >>= op2 | op1 = op1 >> op2 |

41

## Conversão de tipos

- C permite a conversão explícita de tipos chamada conversão de tipo
  - Operador unário com o tipo desejado entre parênteses
    - Ex. supor que duas variáveis, numerador e denominador, são declaradas como do tipo int
    - res = (double) numerador / denominador
  - Quando conversor converte um tipo para um outro menos preciso, pode haver perda de informação
    - Ex.: (int) 2.87 = 2

42

## Diretivas de pre-processamento

- A linguagem C tem um pre-processador embutido nela
  - Linhas do código que começam com # são chamadas de diretivas de pre-processamento
  - São utilizadas para:
    - Definição de constantes (#define)
    - Inclusão de bibliotexas (#include)



## Constantes

- Uma vez definidas, têm o mesmo valor durante toda a execução de um programa
  - Ex.: *#define PI 3.14;*
  - Evitam ter que modificar um mesmo valor em vários locais de um mesmo programa



## Bibliotecas

- Difícil escrever programas interessantes sem usar bibliotecas de funções
  - A cada dia, os programadores dependem mais de bibliotecas de funções
    - 90% ou mais do código de um programa pode ser formado por códigos de bibliotecas
  - Programador deve saber como:
    - Escrever novos códigos
    - Evitar escrever novos códigos usando bibliotecas existentes



## Bibliotecas

- Coleções de programas ou funções
  - Escritas por outros programadores e disponibilizadas
  - Parte do ambiente da linguagem
- Geralmente cada biblioteca possui um conjunto de funções relacionadas
  - Funções matemáticas – *math.h*
  - Entrada e saída de dados – *stdio.h*
  - Funções gráficas



## Bibliotecas

- Para que as funções de uma biblioteca possam ser utilizadas em um programa, a biblioteca tem que ser incluída
  - Comando #include
  - Ex.: *#include <stdio.h>*
    - Permite que o programa utilize as funções da biblioteca *stdio.h*



## Bibliotecas

- Biblioteca de entrada e saída (I/O)
  - Inclui funções para a entrada (leitura) e saída (escrita) de dados
  - *printf()*
  - *scanf()*
  - *getchar()*
  - *putchar()*

# Bibliotecas

- Função *printf()*
  - Permite a escrita de dados *na* tela do computador
  - *Printf ("texto-de-controle", argumentos)*
  - Associa valores dos argumentos ao texto de controle

```
main (){
   int x=0;
   printf ("valor de x = %d\n", x);
}
```



# Bibliotecas

- Função *printf()*
  - Texto de controle pode conter
    - Caracteres que serão exibidos na tela do computador
    - Código de formatação
      - Indica os formatos usados para imprimir os argumentos
  - Argumentos
    - Separados por vírgula
    - Podem ser variáveis, constantes e/ou funções



# Bibliotecas

- Função *printf()*
  - Pode imprimir valores de qualquer tipo

| Código | Resultado |
|---|---|
| %d | Imprime um valor int no formato decimal |
| %f | Imprime um valor float no formato decimal |
| %c | Imprime um caractere |
| %s | Imprime um string (texto) |

```
main (){
   printf ("valor de x = %d\n", 223);
}
```



# Bibliotecas

| | |
|---|---|
| %d<br>%hd<br>%ld | Estes formatos mostram o valor como um número decimal dos tipos *int*, *short* e *long*, respectivamente.<br>O sinal % pode ser seguido por um número especificando a largura mínima a ser usada na impressão do número<br>Se o número é pequeno demais para preencher toda a largura, espaço extra é adicionado à esquerda, de forma que os números se alinhem à direita.<br>Ex.: *printf ("val = %4d\n", val); /* reserva 4 espaços para imprimir o valor da variável val */* |
| %f | Utilizado para valores dos tipos *float* ou *double*.<br>Ele exibe valores desses tipos com um ponto decimal.<br>A precisão pode ser especificada indicando quantos dígitos devem ser exibidos à direita do ponto decimal.<br>Ex.: *printf ("val = %8.3f\n, val"); /* imprime um número com 8 caracteres, 3 deles após o ponto */* |



# Bibliotecas

| | |
|---|---|
| %g | Este formato também é usado para valores dos tipos *float* ou *double*, sendo semelhante ao formato %f quando o número caiba em um espaço pequeno.<br>Números cuja magnitude é ou muito grande ou muito pequena (ex. 27000000.0 ou 00000000006) são representados mais compactamente usando notação científica (ex.: 2.7e+7 ou 6.0e-11).<br>O formato %g pode também incluir um campo largura e precisão, embora a precisão no formato %g especifique o número de dígitos significativos (ao invés dos números a direita do ponto decimal) |
| %c | Utilizado para valores do tipo *char*. *Exime apenas um caracter*<br>Ex.: *printf ("letra= %c\n, var"); /* imprime um caracter */* |



# Bibliotecas

- Alguns caracteres especiais:

| | |
|---|---|
| \n | passa para a próxima linha |
| \t | tab |
| \b | volta um caractere |
| \\ | barra |
| \" | imprime aspas |
| \f | passa para a próxima folha |
| \0 | nulo |

## Bibliotecas

- Função *scanf()*
  - Permite a leitura de dados do teclado
  - *scanf ("texto-de-controle", argumentos)*
  - Associa valores dos argumentos ao texto de controle

```
main (){
   int x;
   scanf("%d\n", &x);
}
```



## Bibliotecas

- Função *scanf()*
  - Complemento da função *printf()*
  - Texto de controle contém
    - Código de formatação
      - Indica os formatos usados para ler os argumentos
  - Argumentos
    - Separados por vírgula
    - Endereços de variáveis (&x)



## Bibliotecas

- Função *scanf()*
  - Operador de endereço: &
  - Endereço do primeiro byte ocupado pela variável

```
main(){
   int val;
   printf("Digite um numero: ");
   scanf("%d", &val);
   printf("\n o numero eh: %d, val);
   printf("\n o endereco eh: %d, &val);
}
```



## Bibliotecas

- Função *getchar()*
  - Função original para entrada de caracteres dos sistemas baseados em *Unix*
  - Armazena o caracter digitado

```
main (){
   char a;
   a = getchar ();
   printf("Caracter lido = %c\n", a);
}
```



## Bibliotecas

- Função *putchar()*
  - Escreve na tela um caractere

```
main (){
   char a;
   printf("Digite uma letra: ");
   a = getchar ();
   putchar (a);
   putchar ('\n');
}
```



## Exemplo

- O que faz o programa abaixo?

```
main (){
   long a;
   a = 0;
   while (getchar() != EOF)
      ++a;
   printf("%d\n", a);
}
```

## Conclusão

- Expressões
- Operadores
- Constantes
- Bibliotecas
- Entrada e saída

# Introdução à Computação

## Linguagem C

Professor: André de Carvalho

## Aula de hoje

- Comandos
  - Comandos simples
  - Comandos de controle
    - Condicionais
    - Repetitivos
- Funções
- Estratégia top-down

## Comandos

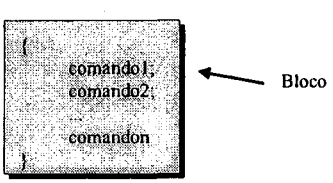
- Programas C são compostos por funções
  - Funções são formadas por comandos
- Comandos em C podem ser:
  - Comandos simples
    - Realizam alguma ação
  - Comandos de controle
    - Afetam a maneira como outros comandos são executados

## Comandos simples

- Maioria dos comandos em C são comandos simples
- Expressão seguida de ponto e vírgula (exp; )
- Pode envolver:
  - Atribuição
  - Chamada de função
  - Incremento (decremento) de variável

## Comandos simples

- Comandos de controle podem ser aplicados a uma sequência de comandos simples (bloco)




## Comandos simples

- Bloco de comandos
  - Tratado por comandos de controle como se fosse um único comando
  - Também são chamados de comandos compostos
    - C permite a declaração de variáveis antes dos comandos de qualquer bloco
    - Tabulação interior de um bloco faz parte de um bom estilo de programação

## Comandos simples

- Bloco de comandos
  - Conjunto de comandos legais da linguagem C
  - Comandos de um bloco são contidos entre "{" e "}"

```
while (c != -1){
  count++;
  printf("Contou " + count + " caracteres.");
}
```

## Comandos de controle

- Afetam a maneira como outros comandos são executados
  - Geralmente são aplicados a um único comando
  - Podem ser:
    - Condicionais
      - Condicionam a execução de um comando ao valor de uma expressão
    - Iterativos
      - Permite a execução de um comando repetidas vezes

## Comando de controle

- Comandos de controle condicionais
  - Muitas vezes, a avaliação de uma condição define a execução ou não de um comando (execução condicional)
  - A maneira mais fácil de fazer isso é através do comando *If*

    *if (condicao) comando*

    *if (condicao) comando1 else comando2*
    - comandos podem ser simples ou compostos (bloco)

## Comando *if-else*

```
main () {
      int n;

      printf ("Este programa diz se um numero");
      printf ("eh par ou impar \n");
      printf ("entre um numero: ");
      n = getinteger ();
      if (n % 2 == 0){
            printf ("este numero eh par \n");
      }
      else{
            printf ("este numero eh par \n");
      }
}
```

## Comando *if-else*

- Comando *if*
  - Algumas aplicações precisam de estruturas de decisão mais complicadas
    - Por exemplo, quando existem mais que duas possibilidades
    - Uma alternativa é utilizar uma cadeia de comandos *if*

```
if condicao1 comando1
else   if condicao2 comando2
       else   if condicao3 comando3
```

## Comando *if-else*

```
int pontosTeste;
char nota;

if (pontosTeste >= 90) {
    nota = 'A';
} else if (pontosTeste >= 80) {
    nota = 'B';
} else if (pontosTeste >= 70) {
    nota = 'C';
} else if (pontosTeste >= 60) {
    nota = 'D';
} else {
    nota = 'F';
}
```

- Apenas um *else*
- Se um *else if* for verdadeiro, os comandos *else if* e *else* abaixo dele não são avaliados
- Outra alternativa: utilizar o comando *switch*

## Comando *switch*

- Executa comandos condicionalmente
  - Baseado no valor de alguma expressão

```
Switch (expressao) {
    case constante-1: comando; break;
    case constante-2: comando; break;
    ...
    case constante-n: comando; break;
                default: comando; break;
}
```

## Comando *switch*

```
int mes;
...
switch (mes) {
    case 1: printf("January"); break;
    case 2: printf("February"); break;
    case 3: printf("March"); break;
    case 4: printf("April"); break;
    case 5: printf("May"); break;
    case 6: printf("June"); break;
    case 7: printf("July"); break;
    case 8: printf("August"); break;
    case 9: printf("September"); break;
    case 10: printf("October"); break;
    case 11: printf("November"); break;
    case 12: printf("December"); break;
}
```

## Comando *switch*

- Equivalente a um comando *if-else*

```
int mes;
...
if (mes == 1) {
    printf("January");
} else if (mes == 2) {
    printf("February");
...
```

*... da para ter uma ideia*

## Comando *switch*

- Qual comando usar? *if-else* ou *switch*?
  - Decisão do programador
    - Baseada em facilidade de leitura e outros fatores
- Cada comando *case* deve ser único
- Valor fornecido para cada comando *case*
  - Deve ser do mesmo tipo do valor retornado pela expressão do comando *switch*
  - Deve ser do tipo *char* ou *int*

## Comando *switch*

- Observações
  - Comando *break*
    - Faz com que o programa saia de dentro do comando *switch* e continue no próximo comando
    - Sem ele, todos os comandos *case* seriam executados

## Comando *switch*

- Observações
  - Em algumas situações, pode ser interessante executar mais de um comando *case*
  - Comando *default* pode ser usado para lidar com os valores que não foram definidos explicitamente nos comandos *case*

## Comando *switch*

```
int mês;
int numDias;
...
switch (mes) {
  case 1:
  case 3:
  case 5:
  case 7:
  case 8:
  case 10:
  case 12:
    numDias = 31;
    break;
  case 4:
  case 6:
  case 9:
  case 11:
    numDias = 30;
    break;
  case 2:
    if ( ((ano % 4 == 0) && ((ano % 100 == 0))
          || (ano % 400 == 0) )
      numDias = 29;
    else
      numDias = 28;
    break;
}
```

## Comando *switch*

```
int mes;
...
switch (mes) {
  case 1: printf("January"); break;
  case 2: printf("February"); break;
  case 3: printf("March"); break;
  case 4: printf("April"); break;
  case 5: printf("May"); break;
  case 6: printf("June"); break;
  case 7: printf("July"); break;
  case 8: printf("August"); break;
  case 9: printf("September"); break;
  case 10: printf("October"); break;
  case 11: printf("November"); break;
  case 12: printf("December"); break;
  default: printf("Opa, este não eh um mês valido!");
}
```

## Comando *switch*

- Cláusula *default* é selecionada quando nenhum dos rótulos casa com o valor da expressão
  - É opcional (mas é uma boa prática de programação)
    - *default*:
      - Pode incluir uma mensagem de erro
        - » Ex.: "valor de rótulo não esperado"

## Comandos iterativos

- Permitem a execução de partes de um programa mais de uma vez
- Comandos iterativos da linguagem C
  - Comando *while*
  - Comando *do while*
  - Comando *for*

## Comando while

- Comando iterativo mais simples
- Executa um comando repetidamente até uma condição se tornar falsa

```
while (expressão condicional) {
    comandos
}
```

## Comando while

- Teste condicional é executado antes de cada ciclo do *loop*
  - Se o primeiro teste resultar no valor 0 (Falso), o corpo do loop não é executado

## Comando while

O que faz o programa:

```
int quefaz (int n) {
   int soma;

   soma = 0;
   while (n > 0){
        soma += n % 10;
        n /= 10;
   }
   return (soma);
}
```

## Comando while

- Utilizado onde existe uma condição de teste que possa ser aplicado antes da execução do corpo
- Vários problemas de programação não se encaixam na estrutura do comando *while*
  - As vezes o teste seria mais natural em algum lugar no meio do loop
    - Ex.: leitura de dados do usuário até o recebimento de um valor especial (sentinela)

## Comando while

- *Loop* baseado em sentinela
  - Ler um valor
  - Se o valor é igual ao sentinela, sair do *loop*
  - Senão, executar o processamento requerido por este valor
  - Como sair do *loop* quando um dado valor for lido?
    - Problema do *loop* e meio
      - Estratégia 1:Por um comando *break* após a leitura
      - Estratégia 2: Copiar parte do código para fora do loop

## Comando while

- Utilização de comando *break* após a leitura
  - Estratégia mais eficiente

```
while (x > 0) {
   Apresentar prompt para usuário e ler um valor;
   if (valor == sentinela) {
        break;
   }
   Processa o valor do dado;
}
```

## Comando do-while

- Semelhante ao comando *while*
  - Exceto que a expressão é avaliada no final

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

- Comando de *loop* menos utilizado
  - Tem suas aplicações

## Comando do-while

- Exemplo

```
int c;

do {
    c = getInteger ();

} while (c != -1);
```

## Comando for

- Utilizado quando se deseja repetir uma operação um dado número de vezes

```
for (inicial; teste; passo) {
        comandos;
}
```

É equivalente a:

```
inicial;
while (test) {
        comandos;
        passo;
}
```

## Comando for

- O comando *for* é determinado pelas expressões:
  - inicial
  - teste
  - passo

*for (inicial; teste; passo)*

## Comando for

- Inicial
  - Expressão que indica como o *loop* do comando *for* deve ser inicializado
  - Executado uma única vez
    - No início do *loop*
  - Define o valor inicial da variável de indexação (contador)
    - Ex.: for (i = 0; .... for (i = -7; ...

## Comando for

- Teste
  - Expressão que indica quando o *loop* do comando *for* deve parar
  - Funciona como a condição de teste do *while*
  - Expressão é avaliada no topo de cada interação do *loop*
    - Compara valor do contador com um valor final
      - Quando o resultado da avaliação é *FALSO*, o *loop* termina
      - Enquanto teste é VERDADEIRO, o *loop* continua
    - Ex.: for (i = 0; (i < n); i++){

## Comando for

- Passo
  - Expressão chamada a cada interação do *loop* para atualizar o valor do contador
  - Indica quanto muda o valor da variável de indexação de um ciclo para outro
  - Especificações de passo mais comuns
    - index++, index--, index +=2

## Comando for

O que faz o programa abaixo?

```
main () {
        int i;
        for (i = 10; i >= 0; i--) {
                printf ("%2d\n", i);
        }
        printf ("fim da lista! \n");
}
```

## Comando for

```
main () {
        int t;
        for (t = 10; t >= 0; t--) {
                printf ("%2d\n", t);
        }
        printf ("fim da lista! \n");
}
```

Produz:

```
10
9
8
7
6
5
4
3
2
1
0
fim da lista!
```

## Comando for

- Expressões inicial,teste e passo são opcionais
  - Os separadores (;) devem aparecer
  - Falta de valor inicial ⇒ não é feita nenhuma inicialização do indexador
  - Falta de condição de teste ⇒ ela é assumida ser sempre VERDADE
  - Falta de passo ⇒ indexador não é alterado entre ciclos do loop
    - Mas pode ser alterado dentro do ciclo

## Exercício 1

- Re-escreva os trechos de programas abaixo usando o comando FOR

```
x = 14;
while ( x >= 3){
   printf ("%d\n", x);
   x -= 5;
}
```

```
y = 70;
while ( y <= 90){
   printf ("%d\n", y);
   y += 5;
}
```

## Exercício 2

- Qual é a saída do programa

```
int x = 6;
int y = 4;
while (x > y){
   for (i = 0; i < y--; i++) {
      x = y / 2;
   }
   x--;
}
```

## Exercício 3

- Escrever um programa C para calcular o máximo divisor comum entre dois números

## Exercício 3

```
int MDC (int a, b) {
   int aux;

   if (a < b) {
        aux = a;
        a = b;
        b = aux;
   }
   while (b != 0){
        aux = b
        b = a mod b;
        a = aux
   }
   return (a);
}
```

## Funções

- Função
  - Agrupa um conjunto de comandos e associa a ele um nome
    - O uso deste nome é uma chamada da função
  - Após sua execução, programa volta ao ponto do programa situado imediatamente após a chamada
    - A volta ao programa que chamou a função é chamada de retorno

## Funções

- Função
  - A chamada de uma função pode passar informações (argumentos) para o processamento da função
  - Argumentos = lista de expressões
    - Lista pode ser vazia
    - Lista aparece entre parênteses após o nome da função
      - Ex. *int Soma (int x, int y) {*

## Funções

- Retornando resultados de funções
  - No seu retorno, uma função pode retornar resultados ao programa que a chamou
    - *return (resultados);*
    - O valor da variável local *resultados* é passado de volta como o valor da função
    - Valores de qualquer tipo podem ser retornados
      - Funções predicado: funções que retornam valores
      - Procedimentos: funções que não retornam valores
        - » Exemplo: *void function (int x)*

## Funções

- Comando return
  - Procedimentos
    - return;
  - Funções
    - return (expressão);
    - Expressão não precisa estar entre parênteses
  - Encerra a execução da função (procedimento)
    - Função: valor da expressão retorna como o valor da função

## Funções

- Definições e protótipos de funções
  - Funções são definidas de acordo com a seguinte sintaxe:

```
Tipo_de resultado nome (lista de parâmetros):
{
        corpo de função
}
```

## Funções

- Definições e protótipos de funções
  - Tipo de resultado
    - Quando a função é um procedimento, usa-se a palavra chave *void*
    - Procedimento não retorna valor
  - Lista de parâmetros
    - Funcionam como variáveis locais com valores iniciais
    - Quando função não recebe parâmetros, a lista de parâmetros é substituída pela palavra *void*

## Funções

- Funcionamento de uma chamada:
  - Cada expressão na lista de argumentos é avaliada
  - O valor da expressão é convertido, se necessário, para o tipo de parâmetro formal
    - Este tipo é atribuído ao parâmetro formal correspondente no início do corpo da função
  - O corpo da função é executado

## Funções

- Funcionamento de uma chamada:
  - Se um comando *return* é executado, o controle é passado de volta para o trecho que chamou a função
    - Se um comando *return* inclui uma expressão, o valor da expressão é convertido, se necessário, pelo tipo do valor que a função retorna
      - O valor então é retornado para o trecho que chamou a função
      - Se um comando *return* não inclui uma expressão nenhum valor é retornado ao trecho que chamou a função

## Funções

- Funcionamento de uma chamada:
  - Se não existir um comando *return*, o controle é passado de volta para o trecho que chamou a função após o corpo da função ser executado

## Passagem de informações

- Ocorre através da declaração de argumentos dentro da declaração
  - Declaração de argumentos informa o número e tipo dos argumentos
  - Conjunto de argumentos forma uma lista de declarações de variáveis separadas por vírgulas
    - *Ex.: double bolsa, double notas, int idade*

## Passagem de informações

```
double mesada (double bolsa, double notas, int idade) {
   double total;

   if (idade > 10)
      return (idade * 20.0);
   else{
      total = notas*idade*20;
      return total;
   }
```

## Passagem de informações

- Argumentos são passados por valor
  - Quando chamada, a função recebe o valor da variável passada
    - Quando argumento é do tipo atômico, a passagem por valor significa que a função não pode mudar seu valor
  - Os argumentos deixam de existir após a execução do método

## Funções

- Protótipos
  - Antes de usar uma função em C, é aconselhável declará-la especificando seu protótipo
    - Tem a mesma forma que a função, só que substitui o corpo por um (;)
    - Nomes das variáveis de um parâmetro são opcionais
      - Fornecê-los ajuda a leitura do programa

## Funções

- Mecanismo do processo de chamada de função
  1. Valor dos argumentos é calculado pelo programa que está chamando a função
  2. Sistema cria nova espaço para todas as variáveis locais da função (estrutura de pilha)
  3. Valor de cada argumento é copiado na variável parâmetro correspondente na ordem em que aparecem
     3.1 Realiza conversões de tipo necessárias

## Funções

- Mecanismo do processo de chamada de função
  4. Comandos do corpo da função são executados até:
     4.1 Encontrar comando *return*
     4.2 Não existirem mais comandos para serem executados
  5. O valor da expressão *return*, se ele existe, é avaliado e retornado como valor da função
  6. Pilha criada é liberada
  7. Programa que chamou continua sua execução

## Funções

- Projeto top-down
  - Procedimentos e funções permitem dividir um programa em pedaços menores
    - Facilita sua leitura
  - É chamado de processo de decomposição
    - Estratégia de programação fundamental
    - Encontrar a decomposição certa não é fácil
      - Requer experiência

## Funções

- Projeto *top-down*
  - Melhor estratégia para escrever programas é começar com o programa principal
    - Pensar no programa como um todo
    - Identificar as principais partes da tarefa completa
      - Maiores pedaços são candidatos a funções
      - Mesmos estas funções podem ser decompostas em funções menores
      - Continuar até cada pedaço ser simples o suficiente para ser resolvido por si só

## Exercício 4

- Escrever um programa C para calcular o máximo divisor comum entre dois números desmembrando o programa em pelo menos três funções

## Conclusão

- Comandos
  - Comandos simples
  - Comandos de controle
    - Condicionais
    - Repetitivos
- Funções
- Estratégia *top-down*

# Introdução à Computação

## Linguagem C

Professor: André de Carvalho

```
int MDC (int a, b) {
    int aux;

    if (a < b) {
        aux = a;
        a = b;
        b = aux;
    }
    while (b != 0){
        aux = b
        b = a mod b;
        a = aux
    }
    return (a);
}
```

## Aula de hoje

- Introdução
- Vetores
- Strings
- Ponteiros
- Relacionamento entre vetores e ponteiros

## Tipo vetor

- Definição: coleção de valores de dados individuais com as seguintes características:
  - É ordenado: os elementos de um vetor podem ser contados de forma ordenada
  - É homogêneo: Todo valor armazenado em um mesmo vetor deve ser do mesmo tipo
    - Ex.: vetor de inteiros só pode ter elementos do tipo inteiro

## Tipo vetor

- Pode-se pensar em um vetor como uma sequência de caixas, uma para cada valor
  - Cada um dos valores é chamado de elemento
    - Ex.: 
- Vetor possui duas propriedades fundamentais
  - Tipo de elemento
  - Tamanho do vetor

## Tipo vetor

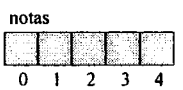
- Declaração
  - *type name[size]*
    - Ex.: int vetor[10];
  - Tamanho do vetor deve ser especificado como uma constante
    - Facilita mudança do tamanho
    - Ex.: *#define NElementos 10*
      *int vetor[NElementos];*

## Tipo vetor

- Declaração (cont.)
  - Cada elemento de um vetor é identificado por um índice
    - Começa com o valor 0 e termina com um valor igual ao número de elementos - 1
    - Ex. vetor de 4 elementos possui os índices 0, 1, 2, 3

## Tipo vetor

- Declaração (cont.)
  - Nome do vetor deve indicar ao usuário que tipo de valor está sendo armazenado
    - Ex. *#define NJuizes 5*
      *double notas[NJuizes];*




## Tipo vetor

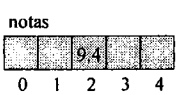
- Para se referir a um elemento específico de um vetor, devem ser fornecidos:
  - Nome do vetor
  - Índice correspondente à posição do elemento dentro do vetor
  - Ex.: A nota do segundo juiz é dada por *notas[1]*

## Tipo vetor

- Definições importantes
  - Seleção: processo de identificar um elemento
  - Expressão com seleção: resultado da seleção
    - Ex.: *vetor[indice];*

## Tipo vetor

- Expressão com seleção
  - Funciona como uma simples variável
    - Ex.: nota[2] = 9.4;
  - É importante distinguir entre índice de um elemento e valor de um elemento




Índice = 2
Valor = 9.4

## Tipo vetor

- Expressão com seleção
  - É possível mudar os valores em um vetor, mas nunca o seu tamanho
  - Valor do índice não precisa ser uma constante
    - Pode ser qualquer expressão cujo resultado é um tipo escalar (int, short, long, char)

```
for (i = 0; i < Njuizes; i++){
    notas[i] = 0.0;
}
```

## Tipo vetor

- Tamanho efetivo e tamanho alocado
  - Muitas vezes não se sabe quantos elementos o vetor vai conter
  - Estratégia: declarar tamanho como o número máximo de elementos possível (tamanho alocado)
    - Declarar um inteiro para indicar número de elementos utilizados (tamanho efetivo)
  - Obs.: Tamanho do vetor deve ser constante
    - Ex.: *int val_dados [n]; /* n não pode ser uma variável */*

## Tipo vetor

- Passagem de vetores como parâmetros
  - Em C, vetores inteiros podem ser passados como parâmetro
    - Tamanho do vetor não precisa ser informado na declaração da função
      - Passar junto uma variável com o tamanho
    - Funciona como passagem por referência
      - Alteração em um valor do parâmetro, muda automaticamente o valor correspondente no argumento

## Tipo vetor

```
double media (double ind_notas[], int n){
        int i;
        double total = 0.0;

        for (i = 0; i < n; i++){
                total += ind_notas[i];
        }
        return ((total) / n);
}
```

## Tipo vetor

- Inicialização de vetores
  - Valores iniciais podem ser atribuídos a uma variável do tipo vetor quando da sua declaração
    - Ex.: *int digitos[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }*
    - Neste caso, o tamanho do vetor pode ser omitido
      - Ex.: *int digitos[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }*
      - Compilador conta o número de inicializadores e reserva a mesma quantidade de elementos para o vetor
      - Facilita a manutenção do programa

## Tipo vetor

- Inicialização de vetores
  - Ex.: *static string CidadesGrandes [] = {"Rio", "Sao Paulo", "Belo Horizonte", "Porto Alegre", Salvador", "Curitiba", "Recife", "Fortaleza", };*
  - Fácil incluir outras cidades no futuro
    - Vírgula no final permite adicionar novas cidades sem alterar as entradas existentes na lista de inicializadores

## Exemplo

```
void main(){
  int val[100];
  int i;

  for (i = 0; i < 100; ++i){
    val[i] = i;
  }
}
```

## Tipo vetor

- Tamanho atual do vetor em bytes:
  - *sizeof(tipo base) * tamanho do vetor*

## Tipo vetor

- A Linguagem C não checa de você passou dos limites de um vetor
  - Se passar do fim e um vetor, você pode:
    - Escrever em cima dos conteúdos das posições nos extremos do *array*
    - Escrever no espaço reservado para outras variáveis
    - Escrever no espaço reservado para o código do programa

## Strings

- *Strings* são representados internamente como vetores de caracteres
  - Caracteres são armazenados em bytes consecutivos
  - Final de *string* é representado por '/0'
    - Ex.: Compilador C reserva 6 bytes para o string "Hello"


    - Declaração: char hello = {'H', 'e', 'l', 'l', 'o', '\o'}; ou char str[6] = "Hello";

## Strings

- Como *strings* são vetores, elementos individuais podem ser selecionados e manipulados

```
Static int ContaEspacos (char str[]) {
   int i, n_espacos;

   n_espacos = 0;
   For (i = 0; str[i] != '\0'; i++){
      If (str[i] == ' ) n_espacos++;
   }
   return (n_espacos);
}
```

## Strings

- Vetor de caracteres também pode ser interpretado como um ponteiro para o primeiro elemento

```
Static int ContaEspacos (char *str) {
   int n_espacos;
   char *p;

   n_espacos = 0;
   For (p = str; *p != '\0'; p++ ){
      if (*p == ' ) n_espacos++;
   }
   return (n_espacos);
}
```

## Strings

- Bibliotecas de operações sobre strings
  - Interface ANSI *string.h* para manipular strings
    - Fornece um conjunto de operações mais avançadas
    - Permitem trabalhar com o string inteiro utilizando uma simples chamada de função
    - Requerem que a representação seja considerada

## String

- Biblioteca ANSI *string.h*
  - Exporta operações que permitem a manipulação de *strings*
  - Biblioteca padrão da linguagem C

## String

Funções mais comuns de string.h

| Nome | Função |
|---|---|
| strcpy (s1, s2) | Copia s2 em s1 |
| strcat (s1, s2) | Concatena s2 ao final de s1 |
| strlen (s1) | Retorna o tamanho de s1 |
| strcmp (s1, s2) | Retorna 0 se s1 ==s2; menor que 0 se s1<s2; maior que 0 se s1>s2 |
| strchr (s1, ch) | Retorna um ponteiro para a primeira ocorrência de ch em s1 |
| strstr (s1, s2) | Retorna um ponteiro para a primeira ocorrência de s2 em s1 |

## String

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

void main(void)
{
  char s1[80], s2[80];

  gets(s1);
  gets(s2);
  printf("Comprimentos: %d %d\n", strlen(s1), strlen(s2));
  if(!strcmp(s1, s2)) printf("As strings sao iguais\n");
  strcat(s1, s2);
  printf("%s\n", s1);
  strcpy(s1, "Isso eh um teste\n");
  printf(s1);
  if(strchr("alo, 'o')) printf("o esta em alo\n");
  if(strstr("ola aqui", "ola")) printf("ol encontrado");
}
```

## String

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

void main(void)
{
  char s1[80], s2[80];

  gets(s1);
  gets(s2);
  printf("Comprimentos: %d %d\n", strlen(s1), strlen(s2));
  if(!strcmp(s1, s2)) printf("As strings sao iguais\n");
  strcat(s1, s2);
  printf("%s\n", s1);
  strcpy(s1, "Isso eh um teste\n");
  printf(s1);
  if(strchr("alo, 'o')) printf("o esta em alo\n");
  if(strstr("ola aqui", "ola")) printf("ol encontrado");
}
```

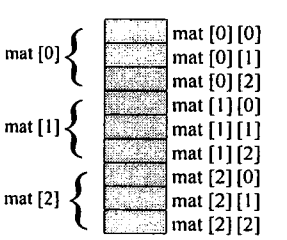
**Comprimentos: 33**
**As strings sao iguais**
**aloalo**
**Isso eh um teste**
**o esta em alo**
**ola encontrado**

## Tipo vetor

- Vetores multi-dimensionais (matrizes)
  - Quando os elementos de um vetores são vetores
  - Vetores bidimensionais (matrizes) são a forma mais comum
  - Ex.: *double mat [3][3];*

| | | |
|---|---|---|
| mat[0][0] | mat[0][1] | mat[0][2] |
| mat[1][0] | mat[1][1] | mat[1][2] |
| mat[2][0] | mat[2][1] | mat[2][2] |

## Tipo vetor




Internamente, C representa mat como um vetor de três elementos

Cada elemento é um vetor de três valores ponto-flutuantes

Na memória, estes nove valores formam uma lista unidimensional

## Tipo vetor

- Vetores multidimensionais
  - Podem ser passados como parâmetros
    - Deve ser especificado tamanho de cada índice (a menos do primeiro índice)
    - O mesmo vale para vetores estaticamente especificados
    - É um bom estilo de programação especificar o tamanho de todos os índices
  - Tamanho em bytes de um vetor bidimensional
    - *tamanho do indice1 * tamanho do indice2 * sizeof (tipo base)*

## Tipo vetor

- Vetores multidimensionais
  - Podem ser inicializados na declaração
    - Para enfatizar a estrutura geral, valores de cada vetor interno são inicializados entre chaves

```
static double ident [3][3] = {
        {1.0,  0.0,  0.0},
        {0.0,  1.0,  0.0},
        {0.0,  0.0,  1.0}
};
```

| | | |
|---|---|---|
| 1.0 | 0.0 | 0.0 |
| 0.0 | 1.0 | 0.0 |
| 0.0 | 0.0 | 1.0 |

## Exemplo

```
#include <stdio.h>

void main(void){
  int i, j, num[3][4];

  for(i=0; i<3; ++i)
    for(j=0; j<4; ++j)
      num[i][j] = (3*i)+j;
  for(i=0; i<3; ++i) {
    for(j=0; j<4; ++j)
      printf("%3d",
num[i][j]);
    printf("\n");
  }
}
```

## Exemplo

```
#include <stdio.h>
#define MAX 100
#define LEN 80
char text[MAX][LEN];
void main(void){
 int i, j;
 printf("Entre com uma linha vazia para sair \n");
 for(i=0; i<MAX; i++) {
  printf("%d: ", i);
  gets(text[i]);
  if(!*text[i]) break; /* sai com linha em branco */
 }
 for(i=0; i<i; i++) {
  for(j=0; text[i][j]; j++) putchar(text[i][j]);
    putchar('\n');
 }
}
```

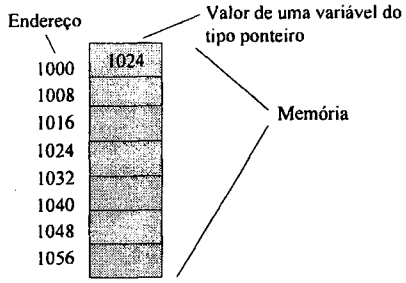
## Ponteiros e vetores

- Em C, para entender vetores é necessário entender ponteiros, e vice-versa
  - Vetores são implementados internamente como ponteiros
  - Operações sobre ponteiros só fazem sentido se forem consideradas em relação a vetores
  - É importante considerar estes dois conceitos juntos

## Ponteiros

- Variável que contém um endereço de memória
  - Este endereço é geralmente a posição de uma outra variável na memória
  - Quando uma variável contém o endereço de uma outra, diz-se que a primeira variável aponta para a segunda

## Ponteiros




## Ponteiros

- Variáveis ponteiros devem ser declaradas como tal
  - *tipo-base * nome-da-variavel*
  - O tipo base do ponteiro define que tipo de variáveis o ponteiro pode apontar
    - Tecnicamente, qualquer tipo de ponteiro pode apontar para qualquer lugar na memória
      - Aritmética de ponteiros é feita pelo tipo base, assim é importante declarar o ponteiro corretamente

## Exemplo

```
void main(){
  float x, y;
  int *p;

  p = &x;
  y = *p;
}
```

Não produz o resultado desejado

## Ponteiros

- Operadores de ponteiros
  - Operador unário & fornece o endereço na memória de seu operando (variável ou constante)
    - Ex.: m = &val;
    - & pode ser lido como "endereço de"
  - Operador unário * é o complemento de &
    - Devolve o valor da variável armazenada no endereço que o segue
    - Ex.: x = *val;
    - * pode ser lido como "no endereço"

## Exemplo

```
#include <stdio.h>

void main(void){
 int x;
 int *p1, *p2;

 p1 = &x;
 p2 = p1;

 printf("%p", p2); /* escreve o endereco de x! */
}
```

## Exemplo

```
#include <stdio.h>

void main(void){
 int x, y;
 int *p1;

 p1 = &x;
 y = *p1;

 printf("%d", y); /* escreve o valor de x! */
}
```

## Exemplo

```
#include <stdio.h>

void main(void){
  int x, *p, **q;

  x = 10;
  p = &x;
  q = &p;

  printf("%d", **q); /* imprime o valor de x */
}
```

## Ponteiros

- Cada tipo base ocupa um tamanho de memória diferente
  - *char: 8 bits (1 byte)*
  - *int:* 16 ou 32 bits *(2 ou 4 bytes)*
  - *short int:16 bits (2 bytes)*
  - *long int:* 32 bits *(4 bytes)*
  - *float 32 bits (4 bytes)*
  - *double 64 bits (8 bytes)*
  - *long double 80 bits (10 bytes)*

## Ponteiros e vetores

- Relacionamento entre ponteiros e vetores
  - Em C, o nome de um vetor tem como valor o endereço do primeiro elemento deste vetor
    - Ex.: *int lista[5]; /* lista é idêntico a &lista[0] */*
    - Nome de um vetor equivale a um ponteiro para o primeiro elemento do vetor
    - Nome de um vetor pode ser atribuído a uma variável do tipo ponteiro
      - Usado quando um vetor é passado de uma função para outra

## Ponteiros e vetores

```
sum = Soma (int vetor[], int n);

int Soma (int vetor[], int n) {
      int i, soma;

      soma = 0;
      for (i = 0; i < n; i++){
            soma += vetor[i];
      }
      return (soma);
}
```

```
int Soma (int *vetor, int n) {
```

## Ponteiros e vetores

```
int Soma (int *ip, int n) {
      int i, soma;

      soma = 0;
      for (i = 0; i < n; i++){
            soma += *p++;
      }
      return (soma);
}
```

## Ponteiros e vetores

- Relacionamento entre ponteiros e vetores
  - Para o compilador, não existe diferença entre vetor e ponteiro quando eles são passados como parâmetros
  - Dentro do computador, declarações *vetor[]* e **vetor* são equivalentes
  - Declaração de parâmetros deve refletir o seu uso

## Exemplo

- Se uma função recebe um vetor de uma dimensão, o parâmetro formal pode ser declarado de três formas:

```
func (int *x){
}
```

```
func (int x[10]){
}
```

```
func (int x[]){
}
```

## Exemplo

- Os três métodos de declaração têm o mesmo resultado
  - Cada um deles avisa ao compilador que um apontador para inteiro será recebido
    - Primeira declaração usa um ponteiro
    - Segunda declaração usa uma declaração padrão de *array*
    - Terceira declaração avisa que um *array* do tipo *int* de tamanho indefinido será recebido

## Exemplo

- Uma declaração do tipo *func (int val[32]){* teria o mesmo funcionamento
  - Compilador gera código instruindo *func()* a receber um ponteiro
    - Não cria um *array* com 32 elementos

## Ponteiros e vetores

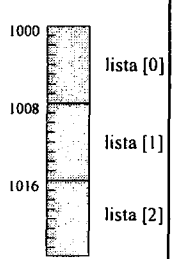
- Diferença entre ponteiros e vetores ocorre na alocação de memória durante a declaração
  - Ex.: *int vet[5];*
    - Reserva 5 posições consecutivas, cada uma capaz de armazenar um valor do tipo inteiro
  - Ex.: *int *p;*
    - Reserva apenas uma palavra
    - Suficiente para armazenar um endereço de memória

## Ponteiros e vetores

- Como usar um ponteiro como um vetor:
  - Atribuir ao ponteiro o endereço base do vetor
    - Ex. *p = vet; /*p e vetor* se tornam "sinônimos" */*
  - Por que fazer isso, se o próprio nome do vetor pode ser utilizado?
  - Ponteiro pode ser inicializado para uma nova memória que ainda não foi utilizada
    - *Permite a criação de novos vetores durante execução do programa (alocação dinâmica)*

## Ponteiros e vetores

- Funcionamento do tipo vetor
  - Ex.: *double lista [3];*
    - Reserva espaço para três variáveis do tipo *double* (tipo *double* ocupa 8 bytes)
  - Cada elemento de um vetor é um valor
    - O vetor lista tem um endereço
    - Endereço pode ser definido com o operador &



## Ponteiros e vetores

- Funcionamento do tipo vetor
  - Ex.: *&lista[1]* tem o valor ponteiro 1008, pois o elemento *lista[1]* é armazenado naquele endereço
    - Valor do índice não precisa ser constante
    - *lista[i]* é um valor
    - É permitido escrever *&lista[i]* (indica o endereço do i-ésimo elemento da lista)

## Ponteiros e vetores

- Funcionamento do tipo vetor
  - Endereço do *i-ésimo* elemento depende do valor da variável *i*
  - Compilador C não pode definir este endereço durante a compilação
    - Gera instruções que adicionam ao endereço base do vetor *i* vezes o tamanho de cada elemento em bytes
      - Ex.: *&lista [i] == 1000 + i*8*
      - Cálculo é feito automaticamente pelo compilador

## Ponteiros e vetores

- Aritmética de ponteiros
  - Em C, operadores + e - podem ser aplicados a ponteiros
  - Se *p* é um ponteiro para o elemento inicial de um vetor *vet* e *k* é um inteiro, então:
    - *p + k* é definido como *&vet[k]*
    - **(p + k) é o valor armazenado no endereço (p + k)*
      - **(p + 4) == vet[4]*

## Exemplo

```
int x = 1, y = 2, z[10];
int *ap;          /* ap eh um apontador */

ap = &x;          /* ap recebe o endereco de x */
                  /* ap aponta para x */

y = *ap;          /* y recebe o valor 1 */
*ap = 0;          /* x agora armazena o valor 0 */
ap = &z[2];       /* ap aponta agora para z[2] */
```

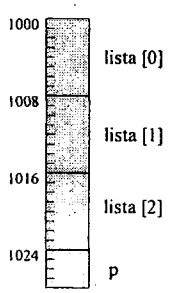
## Exemplo

```
int *p, val[10];

p = val;

p[5] = 20;        /* atribui usando indice */

*(p + 5) = 20;    /* atribui usando aritmética de ponteiros */
```

Ambas as operações atribuem o valor 20 ao sexto elemento de val

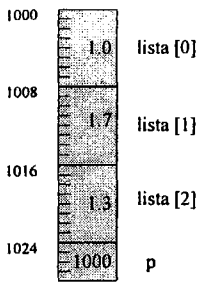
## Ponteiros e vetores

Sejam as declarações:
- *double lista [3];*
- *double *p;*



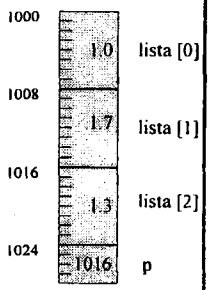
## Ponteiros e vetores

- Sejam as atribuições:
  - *lista [0] = 1.0;*
  - *lista [1] = 1.7;*
  - *lista [2] = 1.3;*
  - *p = &lista [0];*




## Ponteiros e vetores

- Seja a operação:
  - *p = p + 2;*
    - p passa a apontar para o elemento que aparece dois elementos depois no vetor
    - Para cada unidade adicionada a p, seu valor numérico interno é acrescido de 8
    - Aritmética leva em conta o tamanho do tipo base




## Ponteiros e vetores

- Aritmética de ponteiros
  - Operações /, * e % não podem ser usadas com operandos do tipo ponteiro
  - Ponteiros **não** podem ser somados
  - Ponteiros podem ser subtraídos
    - A operação de subtração ocorre de forma semelhante à subtração de inteiros
    - Operação retorna o número de elementos entre os dois ponteiros

## Exemplo

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

void imprime_string (char *p){
  int i;
  /* imprime o conteudo da string normal */
  /* e de tras para frente */
  printf(p);
  for(i=strlen(p)-1; i>-1; i--) printf("%c", p[i]);
}
```

## Ponteiros e vetores

- Ponteiros podem ser decrementados e incrementados
  - Ex.: **p++;*
  - De-referencia o apontador e retorna como valor o objeto para o qual ele aponta
  - Efeito colateral: incrementa o valor de *p*
    - Se o valor anterior era um elemento de um vetor, *p* passa a apontar para o próximo elemento do vetor

## Matrizes de Ponteiros

- Assim como qualquer outro tipo de dados, ponteiros podem ser organizados em matrizes
  - Ex.: int *x[10]
    - x[2] = &var /* atribui o endereco de *var* a x[2] */
    - *x[2] /* retorna o valor de *var* */
  - Pode ser passada como argumento para uma função

## Exemplo

```
void mostra_array (int *q[]){
  int i;
  for (i = 0; i < 10; i++)
    printf ("%d ", *q[i]);
}
```

Obs: q é um ponteiro para uma matriz de ponteiros para inteiros

## Matrizes de Ponteiros

- Matrizes de ponteiros são geralmente usadas como ponteiros para *strings*

```
void mensagem_erro (int num){
  static char *erro[] = {
        "Arquivo não pode ser aberto\n",
        "Erro de leitura\n",
        "Erro de escrita\n",
        "Erro da midia\n"
  };
  printf ("%s", erro[num]);
}
```

## Inicialização de Ponteiros

- Após um ponteiro ser declarado, enquanto um valor não lhe for atribuído, ele contém um valor desconhecido
  - Ponteiro recebe o valor nulo (equivale a 0)
    - Não aponta para nada
  - Tentar atribuir o "valor" desta variável provavelmente leva a erro de execução
    - Exceção: usar o valor nulo
      - Pode ser usado para indicar final de uma matriz de ponteiros

## Exemplo

```
/* Procura por um nome */
procura (char *p[], char *nome){
  int i;
  for (i = 0; p[i]; ++i)
    if (! Strcmp (p[i], nome)) return i;
  return -1; /* nome não foi encontrado */
}
```

## Exemplo

```
#include "string.h"
#include "stdio.h"

void main(void){
  char *p1;
  char s[80];

  do {
    p1 = s;
    gets(s); /* le uma string */
    /* imprime o equivalente decimal de cada caractere */
    while(*p1) printf("%d", *p1++);
  }while (strcmp(s, "fim"));
}
```

## Conclusão

- Introdução
- Vetores
- Strings
- Ponteiros
- Relacionamento entre vetores e ponteiros

# Introdução à Computação

## Linguagem C

Professor: André de Carvalho

## Aula de hoje

- Introdução
- Vetores
- Strings
- Ponteiros
- Relacionamento entre vetores e ponteiros

## Tipo vetor

- Definição: coleção de valores de dados individuais com as seguintes características:
  - É ordenado: os elementos de um vetor podem ser contados de forma ordenada
  - É homogêneo: Todo valor armazenado em um mesmo vetor deve ser do mesmo tipo
    - Ex.: vetor de inteiros só pode ter elementos do tipo inteiro

## Tipo vetor

- Pode-se pensar em um vetor como uma sequência de caixas, uma para cada valor
  - Cada um dos valores é chamado de elemento
    - Ex.: 
- Vetor possui duas propriedades fundamentais
  - Tipo de elemento
  - Tamanho do vetor

## Tipo vetor

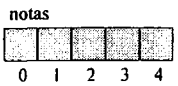
- Declaração
  - *type name[size]*
    - Ex.: int vetor[10];
  - Tamanho do vetor deve ser especificado como uma constante
    - Facilita mudança do tamanho
    - Ex.: *#define NElementos 10*
      *int vetor[NElementos];*

## Tipo vetor

- Declaração (cont.)
  - Cada elemento de um vetor é identificado por um índice
    - Começa com o valor 0 e termina com um valor igual ao número de elementos - 1
    - Ex. vetor de 4 elementos possui os índices 0, 1, 2, 3

## Tipo vetor

- Declaração (cont.)
  - Nome do vetor deve indicar ao usuário que tipo de valor está sendo armazenado
    - Ex. *#define NJuizes 5*
      *double notas[NJuizes];*




## Tipo vetor

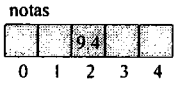
- Para se referir a um elemento específico de um vetor, devem ser fornecidos:
  - Nome do vetor
  - Índice correspondente à posição do elemento dentro do vetor
  - Ex.: A nota do segundo juiz é dada por *notas[1]*

## Tipo vetor

- Definições importantes
  - Seleção: processo de identificar um elemento
  - Expressão com seleção: resultado da seleção
    - Ex.: *vetor[indice];*

## Tipo vetor

- Expressão com seleção
  - Funciona como uma simples variável
    - Ex.: nota[2] = 9.4;
  - É importante distinguir entre índice de um elemento e valor de um elemento




Índice = 2
Valor = 9.4

## Tipo vetor

- Expressão com seleção
  - É possível mudar os valores em um vetor, mas nunca o seu tamanho
  - Valor do índice não precisa ser uma constante
    - Pode ser qualquer expressão cujo resultado é um tipo escalar (int, short, long, char)

```
for (i = 0; i < NJuizes; i++){
    notas[i] = 0.0;
}
```

## Tipo vetor

- Tamanho efetivo e tamanho alocado
  - Muitas vezes não se sabe quantos elementos o vetor vai conter
  - Estratégia: declarar tamanho como o número máximo de elementos possível (tamanho alocado)
    - Declarar um inteiro para indicar número de elementos utilizados (tamanho efetivo)
  - Obs.: Tamanho do vetor deve ser constante
    - Ex.: *int val_dados [n]; /* n não pode ser uma variável */*

## Tipo vetor

- Passagem de vetores como parâmetros
  - Em C, vetores inteiros podem ser passados como parâmetro
    - Tamanho do vetor não precisa ser informado na declaração da função
      - Passar junto uma variável com o tamanho
    - Funciona como passagem por referência
      - Alteração em um valor do parâmetro, muda automaticamente o valor correspondente no argumento

## Tipo vetor

```
double media (double ind_notas[], int n){
        int i;
        double total = 0.0;

        for (i = 0; i < n; i++){
                total += ind_notas[i];
        }
        return (total / n);
}
```

## Tipo vetor

- Inicialização de vetores
  - Valores iniciais podem ser atribuídos a uma variável do tipo vetor quando da sua declaração
    - Ex.: *int digitos[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }*
    - Neste caso, o tamanho do vetor pode ser omitido
      - Ex.: *int digitos[] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 }*
      - Compilador conta o número de inicializadores e reserva a mesma quantidade de elementos para o vetor
      - Facilita a manutenção do programa

## Tipo vetor

- Inicialização de vetores
  - Ex.: *static string CidadesGrandes [] = {"Rio", "Sao Paulo", "Belo Horizonte", "Porto Alegre", Salvador", "Curitiba", "Recife", "Fortaleza", };*
  - Fácil incluir outras cidades no futuro
    - Vírgula no final permite adicionar novas cidades sem alterar as entradas existentes na lista de inicializadores

## Exemplo

```
void main(){
   int val[100];
   int i;

   for (i = 0; i < 100; ++i){
      val[i] = i;
   }
}
```

## Tipo vetor

- Tamanho atual do vetor em bytes:
  - *sizeof(tipo base) * tamanho do vetor*

## Tipo vetor

- A Linguagem C não checa de você passou dos limites de um vetor
  - Se passar do fim e um vetor, você pode:
    - Escrever em cima dos conteúdos das posições nos extremos do *array*
    - Escrever no espaço reservado para outras variáveis
    - Escrever no espaço reservado para o código do programa

## Strings

- *Strings* são representados internamente como vetores de caracteres
  - Caracteres são armazenados em bytes consecutivos
  - Final de *string* é representado por '/0'
    - Ex.: Compilador C reserva 6 bytes para o string "Hello"


    - Declaração: char hello = {'H', 'e', 'l', 'l', 'o', '\o'}; ou char str[6] = "Hello";

## Strings

- Como *strings* são vetores, elementos individuais podem ser selecionados e manipulados

```
Static int ContaEspacos (char str[]) {
   int i, n_espacos;

   n_espacos = 0;
   For (i = 0; str[i] != '\0'; i++){
      if (str[i] == ' ') n_espacos++;
   }
   return (n_espacos);
}
```

## Strings

- Vetor de caracteres também pode ser interpretado como um ponteiro para o primeiro elemento

```
Static int ContaEspacos (char *str) {
   int n_espacos;
   char *p;

   n_espacos = 0;
   For (p = str; *p != '\0'; p++){
      if (*p == ' ') n_espacos++;
   }
   return (n_espacos);
}
```

## Strings

- Bibliotecas de operações sobre strings
  - Interface ANSI *string.h* para manipular strings
    - Fornece um conjunto de operações mais avançadas
    - Permitem trabalhar com o string inteiro utilizando uma simples chamada de função
    - Requerem que a representação seja considerada

## String

- Biblioteca ANSI *string.h*
  - Exporta operações que permitem a manipulação de *strings*
  - Biblioteca padrão da linguagem C

# String

Funções mais comuns de string.h

| Nome | Função |
|---|---|
| strcpy (s1, s2) | Copia s2 em s1 |
| strcat (s1, s2) | Concatena s2 ao final de s1 |
| strlen (s1) | Retorna o tamanho de s1 |
| strcmp (s1, s2) | Retorna 0 se s1 ==s2, menor que 0 se s1<s2, maior que 0 se s1>s2 |
| strchr (s1, ch) | Retorna um ponteiro para a primeira ocorrência de ch em s1 |
| strstr (s1, s2) | Retorna um ponteiro para a primeira ocorrência de s2 em s1 |

# String

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

void main(void)
{
   char s1[80], s2[80];

   gets(s1);
   gets(s2);
   printf("Comprimentos: %d %d\n", strlen(s1), strlen(s2));
   if(!strcmp(s1, s2)) printf("As strings sao iguais\n");
   strcat(s1, s2);
   printf("%s\n", s1);
   strcpy(s1, "Isso eh um teste.\n");
   puts(s1);
   if(strchr("alo, o")) printf("o esta em alo \n");
   if(strstr("ola aqui", "ola")) printf("ol encontrado");
}
```

# String

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

void main(void)
{
   char s1[80], s2[80];

   gets(s1);
   gets(s2);
   printf("Comprimentos: %d %d\n", strlen(s1), strlen(s2));
   if(!strcmp(s1, s2)) printf("As strings sao iguais\n");
   strcat(s1, s2);
   printf("%s\n", s1);
   strcpy(s1, "Isso eh um teste.\n");
   puts(s1);
   if(strchr("alo, o")) printf("o esta em alo \n");
   if(strstr("ola aqui", "ola")) printf("ol encontrado");
}
```

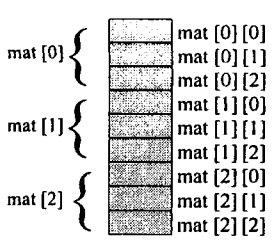
**Comprimentos: 33**
**As strings sao iguais**
**aloalo**
**Isso eh um teste**
**o esta em alo**
**ola encontrado**

# Tipo vetor

- Vetores multi-dimensionais (matrizes)
  - Quando os elementos de um vetores são vetores
  - Vetores bidimensionais (matrizes) são a forma mais comum
  - Ex.: *double mat [3][3];*

| | | |
|---|---|---|
| mat[0][0] | mat[0][1] | mat[0][2] |
| mat[1][0] | mat[1][1] | mat[1][2] |
| mat[2][0] | mat[2][1] | mat[2][2] |

# Tipo vetor




Internamente, C representa mat como um vetor de três elementos

Cada elemento é um vetor de três valores ponto-flutuantes

Na memória, estes nove valores formam uma lista unidimensional

# Tipo vetor

- Vetores multidimensionais
  - Podem ser passados como parâmetros
    - Deve ser especificado tamanho de cada índice (a menos do primeiro índice)
    - O mesmo vale para vetores estaticamente especificados
    - É um bom estilo de programação especificar o tamanho de todos os índices
  - Tamanho em bytes de um vetor bidimensional
    - *tamanho do indice1 * tamanho do indice2 * sizeof (tipo base)*

## Tipo vetor

- Vetores multidimensionais
  - Podem ser inicializados na declaração
    - Para enfatizar a estrutura geral, valores de cada vetor interno são inicializados entre chaves

```
static double ident [3][3] = {
        {1.0,  0.0,  0.0 },
        {0.0,  1.0,  0.0 },
        {0.0,  0.0,  1.0 }
};
```

| | | |
|---|---|---|
| 1.0 | 0.0 | 0.0 |
| 0.0 | 1.0 | 0.0 |
| 0.0 | 0.0 | 1.0 |

## Exemplo

```
#include <stdio.h>

void main(void) {
  int i, j, num[3][4];

  for(i=0; i<3; ++i)
    for(j=0; j<4; ++j)
      num[i][j] = (i*4)+j+1;
  for(i=0; i<3; ++i) {
    for(j=0; j<4; ++j)
      printf("%3d",
    num[i][j]);
      printf("\n");
  }
}
```

## Exemplo

```
#include <stdio.h>
#define MAX 100
#define LEN 80
char text[MAX][LEN];
void main(void){
  int t, i, j;
  printf("Entre com uma linha vazia para sair.\n");
  for(t=0; t<MAX; t++) {
    printf("%d: ", t);
    gets(text[t]);
    if(!*text[t]) break; /* sai com linha em branco */
  }
  for(i=0; i<t; i++) {
    for(j=0; text[i][j]; j++) putchar(text[i][j]);
      putchar('\n');
  }
}
```

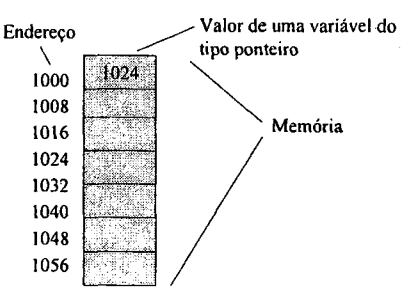
## Ponteiros e vetores

- Em C, para entender vetores é necessário entender ponteiros, e vice-versa
  - Vetores são implementados internamente como ponteiros
  - Operações sobre ponteiros só fazem sentido se forem consideradas em relação a vetores
  - É importante considerar estes dois conceitos juntos

## Ponteiros

- Variável que contém um endereço de memória
  - Este endereço é geralmente a posição de uma outra variável na memória
  - Quando uma variável contém o endereço de uma outra, diz-se que a primeira variável aponta para a segunda

## Ponteiros



## Ponteiros

- Variáveis ponteiros devem ser declaradas como tal
  - *tipo-base * nome-da-variavel*
  - O tipo base do ponteiro define que tipo de variáveis o ponteiro pode apontar
    - Tecnicamente, qualquer tipo de ponteiro pode apontar para qualquer lugar na memória
      - Aritmética de ponteiros é feita pelo tipo base, assim é importante declarar o ponteiro corretamente

## Exemplo

```
void main(){
  float x, y;
  int *p;

  p = &x;
  y = *p;
}
```

Não produz o resultado desejado

## Ponteiros

- Operadores de ponteiros
  - Operador unário & fornece o endereço na memória de seu operando (variável ou constante)
    - Ex.: m = &val;
    - & pode ser lido como "endereço de"
  - Operador unário * é o complemento de &
    - Devolve o valor da variável armazenada no endereço que o segue
    - Ex.: x = *val;
    - * pode ser lido como "no endereço"

## Exemplo

```
#include <stdio.h>

void main(void){
  int x;
  int *p1, *p2;

  p1 = &x;
  p2 = p1;

  printf("%p", p2); /* escreve o endereco de x! */
}
```

## Exemplo

```
#include <stdio.h>

void main(void){
  int x, y;
  int *p1;

  p1 = &x;
  y = *p1;

  printf("%d", y); /* escreve o valor de x! */
}
```

## Exemplo

```
#include <stdio.h>

void main(void){
  int x, *p, **q;

  x = 10;
  p = &x;
  q = &p;

  printf("%d", **q); /* imprime o valor de x */
}
```

## Ponteiros

- Cada tipo base ocupa um tamanho de memória diferente
  - *char: 8 bits (1 byte)*
  - *int:* 16 ou 32 bits *(2 ou 4 bytes)*
  - *short int:16 bits (2 bytes)*
  - *long int:* 32 bits *(4 bytes)*
  - *float 32 bits (4 bytes)*
  - *double 64 bits (8 bytes)*
  - *long double 80 bits (10 bytes)*

## Ponteiros e vetores

- Relacionamento entre ponteiros e vetores
  - Em C, o nome de um vetor tem como valor o endereço do primeiro elemento deste vetor
    - Ex.: *int lista[5]; /* lista é idêntico a &lista[0] */*
    - Nome de um vetor equivale a um ponteiro para o primeiro elemento do vetor
    - Nome de um vetor pode ser atribuído a uma variável do tipo ponteiro
      - Usado quando um vetor é passado de uma função para outra

## Ponteiros e vetores

```
sum = Soma (int vetor[], int n);

int Soma (int vetor[], int n) {
    int i, soma;

    soma = 0;
    for (i = 0; i < n; i++){
        soma += vetor[i];
    }
    return (soma);
}
```

```
int Soma (int *vetor, int n) {
```

## Ponteiros e vetores

```
int Soma (int *ip, int n) {
    int i, soma;

    soma = 0;
    for (i = 0; i < n; i++){
        soma += *p++;
    }
    return (soma);
}
```

## Ponteiros e vetores

- Relacionamento entre ponteiros e vetores
  - Para o compilador, não existe diferença entre vetor e ponteiro quando eles são passados como parâmetros
  - Dentro do computador, declarações *vetor[]* e **vetor* são equivalentes
  - Declaração de parâmetros deve refletir o seu uso

## Exemplo

- Se uma função recebe um vetor de uma dimensão, o parâmetro formal pode ser declarado de três formas:

```
func (int *x){
    .
    .
}
```

```
func (int x[10]){
    .
    .
}
```

```
func (int x[]){
    .
    .
}
```

## Exemplo

- Os três métodos de declaração têm o mesmo resultado
  - Cada um deles avisa ao compilador que um apontador para inteiro será recebido
    - Primeira declaração usa um ponteiro
    - Segunda declaração usa uma declaração padrão de *array*
    - Terceira declaração avisa que um *array* do tipo *int* de tamanho indefinido será recebido

## Exemplo

- Uma declaração do tipo *func (int val[32]){* teria o mesmo funcionamento
  - Compilador gera código instruindo *func()* a receber um ponteiro
    - Não cria um *array* com 32 elementos

## Ponteiros e vetores

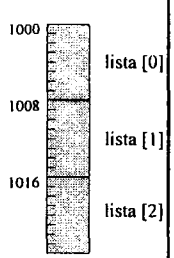
- Diferença entre ponteiros e vetores ocorre na alocação de memória durante a declaração
  - Ex.: *int vet[5];*
    - Reserva 5 posições consecutivas, cada uma capaz de armazenar um valor do tipo inteiro
  - Ex.: *int *p;*
    - Reserva apenas uma palavra
    - Suficiente para armazenar um endereço de memória

## Ponteiros e vetores

- Como usar um ponteiro como um vetor:
  - Atribuir ao ponteiro o endereço base do vetor
    - Ex. *p = vet; /*p e vetor* se tornam "sinônimos" */*
  - Por que fazer isso, se o próprio nome do vetor pode ser utilizado?
  - Ponteiro pode ser inicializado para uma nova memória que ainda não foi utilizada
    - Permite a criação de novos vetores durante execução do programa (alocação dinâmica)

## Ponteiros e vetores

- Funcionamento do tipo vetor
  - Ex.: *double lista [3];*
    - Reserva espaço para três variáveis do tipo *double* (tipo *double* ocupa 8 bytes)
  - Cada elemento de um vetor é um valor
    - O vetor lista tem um endereço
    - Endereço pode ser definido com o operador &




## Ponteiros e vetores

- Funcionamento do tipo vetor
  - Ex.: *&lista[1]* tem o valor ponteiro 1008, pois o elemento *lista[1]* é armazenado naquele endereço
    - Valor do índice não precisa ser constante
    - *lista[i]* é um valor
    - É permitido escrever *&lista[i]* (indica o endereço do i-ésimo elemento da lista)

## Ponteiros e vetores

- Funcionamento do tipo vetor
  - Endereço do *i-ésimo* elemento depende do valor da variável *i*
  - Compilador C não pode definir este endereço durante a compilação
    - Gera instruções que adicionam ao endereço base do vetor *i* vezes o tamanho de cada elemento em bytes
      - Ex.: *&lista [i] == 1000 + i*8*
      - Cálculo é feito automaticamente pelo compilador

## Ponteiros e vetores

- Aritmética de ponteiros
  - Em C, operadores + e - podem ser aplicados a ponteiros
  - Se *p* é um ponteiro para o elemento inicial de um vetor *vet* e *k* é um inteiro, então:
    - *p + k* é definido como *&vet[k]*
    - **(p + k) é o valor armazenado no endereço (p + k)*
      - **(p + 4) == vet[4]*

## Exemplo

```
int x = 1, y = 2, z[10];
int *ap;        /* ap eh um apontador */

ap = &x;        /* ap recebe o endereco de x */
                /* ap aponta para x */

y = *ap;        /* y recebe o valor 1 */
*ap = 0;        /* x agora armazena o valor 0 */
ap = &z[2];     /* ap aponta agora para z[2] */
```

## Exemplo

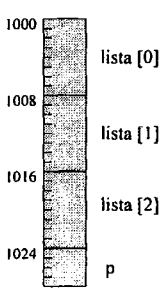
```
int *p, val[10];

p = val;

p[5] = 20;      /* atribui usando indice */

*(p + 5) = 20;  /* atribui usando aritmetica de ponteiros */
```

Ambas as operações atribuem o valor 20 ao sexto elemento de val

## Ponteiros e vetores

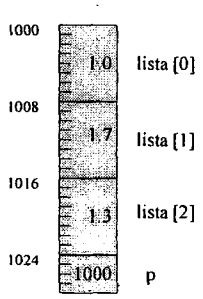
Sejam as declarações:

- *double lista [3];*
- *double *p;*



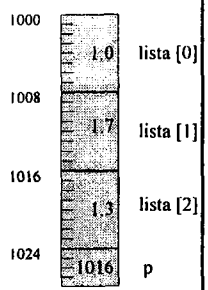

## Ponteiros e vetores

- Sejam as atribuições:
  - *lista [0] = 1.0;*
  - *lista [1] = 1.7;*
  - *lista [2] = 1.3;*
  - *p = &lista [0];*



## Ponteiros e vetores

- Seja a operação:
  - *p = p + 2;*
    - p passa a apontar para o elemento que aparece dois elementos depois no vetor
    - Para cada unidade adicionada a p, seu valor numérico interno é acrescido de 8
    - Aritmética leva em conta o tamanho do tipo base




## Ponteiros e vetores

- Aritmética de ponteiros
  - Operações /, * e % não podem ser usadas com operandos do tipo ponteiro
  - Ponteiros **não** podem ser somados
  - Ponteiros podem ser subtraídos
    - A operação de subtração ocorre de forma semelhante à subtração de inteiros
    - Operação retorna o número de elementos entre os dois ponteiros

## Exemplo

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

void imprime_string (char *p){
  int t;
  /* imprime o conteudo da string normal */
  /* e de tras para frente */
  printf(p);
  for(t=strlen(p)-1; t>=1; t--) printf("%c", p[t]);
}
```

## Ponteiros e vetores

- Ponteiros podem ser decrementados e incrementados
  - Ex.: **p++;*
  - De-referencia o apontador e retorna como valor o objeto para o qual ele aponta
  - Efeito colateral: incrementa o valor de *p*
    - Se o valor anterior era um elemento de um vetor, *p* passa a apontar para o próximo elemento do vetor

## Matrizes de Ponteiros

- Assim como qualquer outro tipo de dados, ponteiros podem ser organizados em matrizes
  - Ex.: int *x[10]
    - x[2] = &var /* atribui o endereco de *var* a x[2] */
    - *x[2] /* retorna o valor de *var* */
  - Pode ser passada como argumento para uma função

## Exemplo

```
void mostra_array (int *q[]){
  int t;
  for (t = 0; t < 10; t++)
    printf ("%d ", *q[t]);
}
```

Obs: q é um ponteiro para uma matriz de ponteiros para inteiros

## Matrizes de Ponteiros

- Matrizes de ponteiros são geralmente usadas como ponteiros para *strings*

```
void mensagem_erro (int num){
  static char *erro[] = {
        "Arquivo não pode ser aberto\n",
        "Erro de leitura\n",
        "Erro de escrita\n",
        "Erro da midia\n"
  };
  printf ("%s", erro[num]);
}
```

## Inicialização de Ponteiros

- Após um ponteiro ser declarado, enquanto um valor não lhe for atribuído, ele contém um valor desconhecido
  - Ponteiro recebe o valor nulo (equivale a 0)
    - Não aponta para nada
  - Tentar atribuir o "valor" desta variável provavelmente leva a erro de execução
    - Exceção: usar o valor nulo
      - Pode ser usado para indicar final de uma matriz de ponteiros

## Exemplo

```
/* Procura por um nome */
procura (char *p[], char *nome){
  int t;
  for (t = 0; p[t]; ++t)
    if (!Strcmp (p[t], nome)) return t;
  return -1; /* nome não foi encontrado */
}
```

## Exemplo

```
#include "string.h"
#include "stdio.h"

void main(void){
  char *p1;
  char s[80];

  do {
     p1 = s;
     gets(s); /* le uma string */
     /* imprime o equivalente decimal de cada caractere */
     while(*p1) printf("%d", *p1++);
  }while (strcmp(s, "fim"));
}
```

## Conclusão

- Introdução
- Vetores
- Strings
- Ponteiros
- Relacionamento entre vetores e ponteiros

# Introdução à Computação

## Linguagem C

Professor: André de Carvalho

## Aula de hoje

- Introdução
- Tipos compostos
- Tipo estrutura
- Tipo enumeração
- Tipo união
- Conclusão

## Tipos compostos

- A Linguagem C permite criar tipos de dados definíveis pelo usuário
  - Tipo estrutura
  - Tipo união
  - Tipo enumeração
- Neste curso serão vistos os tipos estrutura e enumeração

## Tipo estrutura

- Estrutura (ou registro): coleção de variáveis referenciadas por um nome
  - Também chamado agregado ou registro
  - Fornece uma forma conveniente de agrupar informações
  - Definição de estrutura forma um modelo que pode ser usado para criar variáveis de estruturas

## Tipo estrutura

- Tipo de dado estrutura (*struct*)
  - Formado por um conjunto de componentes individuais (campo, elemento ou membro)
    - Não precisam ser do mesmo tipo
    - Geralmente, os elementos de uma estrutura são logicamente relacionados
    - Ex.: Dados de cada empregado de uma empresa

| Nome | Número | Função | Salário | Admissão |
|---|---|---|---|---|
| João | 324 | operario | 800 | 200295 |
| Pedro | 15 | engenheiro | 2000 | 201192 |

## Tipo estrutura

- Criação de novas estruturas
  - Definição de um novo tipo estrutura
    - Define campos da estrutura
      - Nome de cada campo
      - Tipo de dado de cada campo
    - Define um modelo para todas as variáveis que sejam do novo tipo, mas não aloca espaço para elas
  - Declaração de variáveis do novo tipo
    - Permite armazenar valores do tipo estrutura definido

## Tipo estrutura

- Definição de um novo tipo estrutura

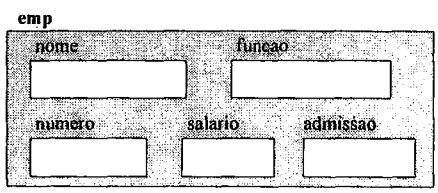
```
struct nome {
        declaração-de-campos
} ;
```

A declaração de um campo é semelhante à declaração de variável

```
struct reg_emp {
        char nome [30];
        int numero;
        char funcao [20];
        int salario;
        char admissao[20];
} x, y;
```

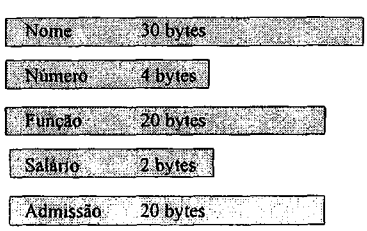
## Tipo estrutura

- Declaração de variáveis do tipo estrutura
  - Ex.: *reg_emp emp;*




## Tipo estrutura

- A variável *emp* aparece da seguinte forma na memória




## Tipo estrutura

- Atribuição
  - Uma vez declarada uma variável de um tipo estrutura, é possível referenciar toda a estrutura, usando apenas nome da variável
    - Ex.: *x = emp;*
- Seleção de campo
  - Campos específicos da estrutura, podem ser referenciados usando operador "."
    - Ex.: *y = emp.nome;*
    - Operador "." retorna um valor

## Exemplo

```
#include <stdio.h>

void main(void){
  struct {
    int a;
    int b;
  } x, y;

  x.a = 10;
  y = x; /* atribui uma estrutura a outra */
  printf("%d", y.a);
}
```

## Tipo estrutura

- Se apenas uma variável estrutura vai ser utilizada, não é necessário dar um nome ao tipo

```
struct {
    char nome [30];
    char rua[40];
    char cidade[20];
    char estado[2]
    unsigned long int cep;
} endereco;
```

## Tipo estrutura

- Inicialização de uma variável do tipo estrutura
  - Basta atribuir valores aos seus componentes
  - Ex.:

```
emp.nome = "Wilson";
emp.salario = 1200;
emp.funcao = "porteiro";
emp.admissao = "100998";
emp.numero = 31;
```

## Tipo estrutura

- Inicialização de uma variável do tipo estrutura
  - Pode ser feita na declaração da variável
  - Inicializadores devem ser fornecidos na ordem em que os respectivos campos aparecem na definição da estrutura

```
Ex: static reg_emp emp = {"Asdrubal", 98, "servente",
                          500, "121098"};
```

## Tipo estrutura

- Vetores de estruturas
  - Uso mais comum de estruturas
  - Ex.: struct reg_emp registros[100];
    - Cria 100 conjuntos de variáveis organizadas conforme definido em reg_emp
  - Para acessar uma estrutura específica, usar indexar o vetor
    - Ex.: printf ("%d", registros[3].salario);

## Tipo estrutura

- Passagem de um elemento de uma variável estrutura para uma função, passa apenas o valor desse elemento
  - Alterações realizadas na função não afetam elemento original
  - Exceção: se o elemento for um vetor
    - Neste caso, o endereço é que é passado

## Tipo estrutura

- Exemplo

```
struct pessoa{
  char sexo;
  int idade;
  float peso;
  char nome[20];
} joao;
```

```
func (joao.sexo);     /* valor     */
func (joao.idade );   /* valor     */
func (joao.peso );    /* valor     */
func (joao.nome );    /* endereco  */
func (joao.nome[2]);  /* valor     */
```

## Tipo estrutura

- Estruturas inteiras podem ser passadas como argumentos para funções
  - Tipo do argumento deve coincidir com tipo do parâmetro
  - mudanças feitas no conteúdo da estrutura dentro da função não mudam conteúdo da estrutura original

## Tipo estrutura

```
#include <stdio.h>

struct struct_type { /* Define um tipo de estrutura. */
  int a, b;
  char ch;
};

void f1(struct struct_type parm);

void main(void){
  struct struct_type arg;
  arg.a = 1000;
  f1(arg);
}
```

## Tipo estrutura

- Observação
  - O tipo do argumento TEM que coincidir com o tipo do parâmetro
  - Não é suficiente que os tipos sejam semelhantes (tenham os mesmos campos)

## Tipo estrutura

- Da mesma forma que permite ponteiros para outros tipos de variáveis, C permite ponteiros para estruturas
  - Algumas peculiaridades devem ser consideradas
  - Exemplo: *struct endereco *apont_end*

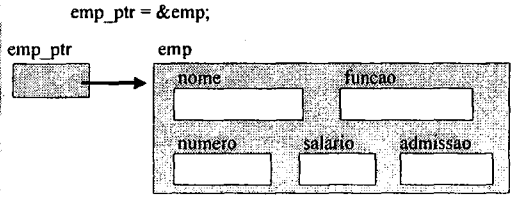
## Tipo estrutura

- Ponteiros para estruturas
  - Variável que armazenam dados estruturados é geralmente declarada como ponteiro para estruturas
    - Menor e mais facilmente manipulada
    - Permite passagem por referência (endereço)
  - Declaração de uma variável do tipo ponteiro para estrutura só aloca espaço para o ponteiro
    - Deve ser alocado espaço para os campos da estrutura

## Tipo estrutura

Ex. Seja o tipo estrutura reg_emp definida anteriormente:

```
struct reg_emp  *emp_ptr;
struct reg_emp  emp;
emp_ptr = &emp;
```




## Tipo estrutura

- Ponteiros para estruturas
  - Como referenciar campos de uma estrutura usando ponteiros?
    - *emp_ptr.salario
    - Desejado: (*emp_ptr).salario
    - Produzido: *(emp_ptr.salario)
  - Linguagem C define o operador especial -> para este caso
    - emp_ptr->salario

## Tipo enumeração

- C permite criar novos tipos listando os elementos que constituem seu domínio (tipo enumeração)
  - Forma sintática:

*enum nome { lista-de-elementos } lista de variaveis;*

Nome do novo tipo

Lista de identificadores (constantes de enumeração)

## Tipo enumeração

- Conjunto de constantes inteiras que especifica todos os valores legais que uma variável desse tipo pode ter
- Enumerações são comuns no dia-a-dia
  - Ex.: domingo, segunda, terça, quarta, quinta, sexta e sabado
- Enumerações são definidas de forma semelhante a estruturas
  - A palavra chave *enum* indica o início de um tipo enumeração

## Tipo enumeração

- Exemplo

```
enum {
    Norte, Leste, Sul, Oeste
} direcao;

direcao dir;
```

## Tipo enumeração

- Representação interna
  - Valores do tipo *enum* são armazenadas internamente como int (inteiros)
    - Podem ser usados em qualquer lugar onde um inteiro possa ser usado
  - Compilador atribui números inteiros consecutivos às constantes de enumeração
    - Começa com o valor 0
    - Ex.: *enum {Norte, Leste, Sul, Oeste} direcao;*

0 1 2 3

## Tipo enumeração

- Representação interna
  - É possível controlar o valor associado a cada símbolo
  - Exemplo:

```
enum {
        Semana = 7,
        Mes = 30,
        Ano = 365
} num_dias;
```

## Tipo enumeração

- Representação interna
  - Definição de uma enumeração é semelhante à definição de várias constantes

```
typedef enum {
  janeiro = 1, fevereiro, marco, abril,
  maio, junho, julho, agosto, setembro,
  outubro, novembro, dezembro
} mes;
```

```
#define janeiro     1
#define fevereiro   2
#define marco       3
...
#define dezembro   12
```

## Tipo enumeração

- Representação interna
  - Quando o valor de qualquer constante de enumeração não é especificado, o compilador adiciona um ao valor da constante anterior
  - Exemplo:

```
typedef enum {
   janeiro = 1, fevereiro, marco, abril, maio, junho, julho,
   agosto, setembro, outubro, novembro, dezembro
} mes;
```

## Tipo enumeração

- Representação interna
  - Símbolos que aparecem após os inicializadores recebem valores maiores do que a inicialização anterior
  - Exemplo

```
enum cores {amarelo, azul, branco, verde = 8,
            marron, preto};
```

## Tipo enumeração

```
enum cores {amarelo, azul, branco, verde = 8,
            marron, preto};
```

| | |
|---|---|
| Amarelo | 0 |
| azul | 1 |
| branco | 2 |
| verde | 8 |
| marron | 9 |
| amarelo | 10 |

## Exemplo

```
#include <stdio.h>
void main(void){
  int n, count, aux;
  printf("Entre um inteiro positivo: ");
  scanf("%d", &n);
  for ( ; n > = 1; --n){
    printf(" %3d ", n);
    aux = 1;
    for (count = 1; count <= n; ++count)
      aux *= count;
    printf("%d\n", aux);
  }
}
```

## Exemplo

```
#include <stdio.h>
#include <string.h>
void main(void)
{ char nome1[12] = "Jose", nome2[12] = "Maria";
  strncpy(nome1, nome2);
  printf("%s\n", nome1);
}
```

## Exemplo

```
#include <stdio.h>
void xxx (int *x, int *y);
void main(void){
  int a = 125, b = 37;
  printf("Antes de XXX a: %3d  b: %3d\n", a, b);
  xxx (&a, &b);
  printf("Apos XXX  a: %3d  b: %3d\n", a, b);
}
void xxx (int *x, int *y){
  int temp;
  temp = *y;
  *y = *x;
  *x = temp;
}
```

## Exemplo

```
#include <stdio.h>
void main(void){
  char *letter;
  letter = "Universo";
  while (*letter != '\0')
  { printf("%c", *letter);
    ++letter;
  }
  printf("\n");
}
```

## Exemplo

```
#include <stdio.h>
#define TAM_NOME 30
struct emp_reg{
  char name[TAM_NOME];
  int dependentes;
  float pagamento;
};
void main(void){
  struct emp_reg turno_comp = {"Joao da Silva", 4, 12.63};
  printf("Tamanho do tipo : %i.\n", sizeof(struct emp_reg));
  printf("Tamanho da var.: %i.\n", sizeof(turno_comp));
}
```

## Conclusão

- Introdução
- Tipos compostos
- Tipo estrutura
- Tipo enumeração
- Exemplos
- Conclusão

# Introdução à Computação

## Linguagem C

Professor: André de Carvalho

## Aula de hoje

- Introdução
- Arquivos
- Biblioteca de entrada e saída
- Funções de entrada e saída
- Outras bibliotecas
- Conclusão

## Introdução

- Suponha que você deseje manter as informações sobre os funcionários de sua empresa em um computador
  - Você não deseja re-digitar todas as informações cada vez que ligar o computador
  - É desejável que, após digitadas, as informações fiquem guardadas para uso posterior
    - Inclusão, alteração, consulta ou remoção
  - Você precisa armazenar estas informações em um arquivo

## Arquivos de dados

- Arquivo de dados
  - Frequentemente, é necessário armazenar dados por um tempo maior que a execução de um programa
  - Geralmente consiste de texto (arquivo texto)
    - Sequência unidirecional de caracteres armazenada em um meio permanente e identificada por um nome de arquivo
    - Semelhantes a *strings*
      - Coleção ordenada de caracteres com um ponto final especificado

## Arquivos de dados

- Um exemplo de arquivo seria então uma coleção de estruturas, uma para cada funcionário da sua empresa
  - Esta estrutura poderia conter campos como:
    - Nome
    - Número funcional
    - Função
    - Idade
    - Data de admissão
    - Salário

## Arquivos de dados

- O seu programa pode precisar:
  - Incluir novos funcionários no arquivo
  - Remover funcionários do arquivo
  - Alterar dados de um ou mais funcionários
  - Consultar informações sobre um ou mais funcionários
- Estas operações incluem:
  - Leitura de dados do arquivo (op. de entrada)
  - Escrita de dados no arquivo (op. de saída)

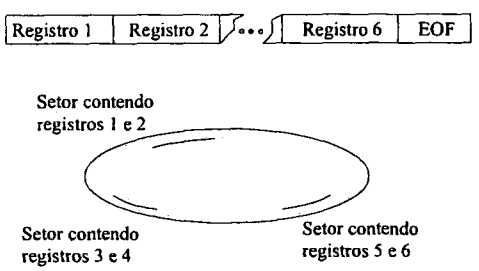
## Tipos de arquivos

- A organização sequencial pode ser apenas conceitual
  - Dependendo do das características físicas do dispositivo de armazenamento utilizado, um arquivo pode ser armazenado em uma outra forma
    - É apresentado ao usuário na forma sequencial

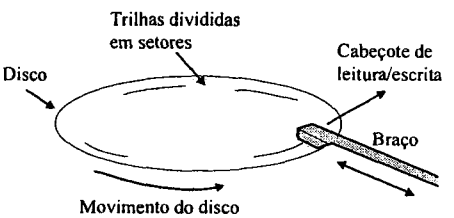
## Tipos de arquivos

- Dispositivo de memória pode ser
  - Sistema de fita megnética
    - Armazena os dados em ordem sequencial
  - Sistema de disco
    - Dados são armazenados nos setores disponíveis
    - Registros consecutivos não precisam ser armazenados na mesma ordem
      - Maioria dos Sist. Operacionais mantem um registro dos setores onde o arquivo esta armazenado e a ordem a ordem na qual esses setores devem ser lidos

## Tipos de arquivos




## Disco




## Tipos de arquivos

- Em C,um arquivo pode ser qualquer coisa
  - Disquete, teclado, impressora, vídeo
- Acesso a dados em um arquivo pode ser:
  - Aleatório
    - Discos
  - Sequencial
    - Fitas

## Tipos de arquivos

- Indiferente do tipo de sistema de memória utilizado, o final de um arquivo sequencial é sempre indicado por uma marca EOF (End of File)
  - Algumas implementações registram o tamanho do arquivo no seu inicio
  - Outras implementações usam um registro especial (sentinela) como último registro

## Arquivos de dados

- A linguagem C não contém comandos de entrada e saída de dados
  - Operações de entrada e saída ocorrem através de chamadas de funções da biblioteca de entrada e saída
  - Ex.: *scanf(), printf()*
  - Biblioteca padrão de entrada e saída (I/O) da linguagem C: *stdio.h*

## Biblioteca padrão de I/O

- Biblioteca *stdio.h*
  - Biblioteca mais utilizada da coleção ANSI de bibliotecas
  - Inclui várias funções para entrada e saída de valores
    - Lêem (escrevem) de (em) arquivos de dados
  - Possui várias funções para a manipulação de arquivos

## Biblioteca padrão de I/O

- Utilização de arquivos em C
  - Declarar arquivo
    - Ex.: *FILE *infile, *outfile;*
  - Abrir arquivo
    - Ex.: *outfile = fopen ("resultado.txt", w); /* w, r, a */*
  - Transferir dados do (para) arquivo
    - Ex.: *printf (outfile, "O resultado gerado foi\n");*
  - Fechar arquivo
    - Ex.: *fclose (outfile);*

## Abrindo um arquivo

```
outfile = fopen ("resultado.txt", w); /* w, r, a */
```

Nome do arquivo

Modo de abertura

| Modo | Significado |
|---|---|
| r | Abre um arquivo texto para leitura |
| w | Cria um arquivo texto para escrita |
| a | Anexa ao final de um arquivo texto |

## Abrindo um arquivo

- Função *fopen()* retorna um ponteiro para uma arquivo
- Ao usar o comando *fopen()* para abrir um arquivo para escrita (w)
  - Se o arquivo a ser aberto já existir, todo o conteúdo dele será perdido
  - Se este arquivo não existir, ele será criado
  - Dados podem ser adicionados ao final de um arquivo usando o modo *anexa (a)*

## Exemplo

```
FILE *fp;
fp = fopen ("teste", w);
```

```
FILE *fp;
if ((fp = fopen ("teste", w)) == NULL){
  printf ("Não pode abrir o arquivo teste\n");
  exit(1);
}
```

## Biblioteca padrão de I/O

- Arquivos padrão
  - A linguagem C fornece constantes do tipo * FILE, chamadas de arquivos padrão
    - Arquivo padrão de entrada: *stdin*
      - Representa o dispositivo padrão de onde a entrada é lida (teclado)
    - Arquivo padrão de saída: *stdout*
      - Representa o dispositivo padrão para saída de dados (tela)
    - Arquivo padrão de erros: stderr
      - Apresenta as mensagens de erro que o usuário deve ver (tela)

## Biblioteca padrão de I/O

- I/O orientada a caracteres
  - A forma mais simples de processar arquivos é ir através dos arquivos caracter a caracter
  - *stdio.h* fornece a função *getc ()* para ler caracteres
    - Ler o próximo caracter do arquivo e retorna seu código
      - Se retornasse caracter tornaria impossível a detecção de EOF
    - Protótipo: *int getc (FILE *infile);*

## Biblioteca padrão de I/O

- I/O orientada a caracteres
  - *stdio.h* fornece a função *putc ()* para escrever caracteres
    - Ex.: *putc (ch, outfile);*

## Biblioteca padrão de I/O

```
void CopiaArquivo (FILE *infile, FILE *outfile) {
  int ch;

  while ((ch = getc (infile)) != EOF) {
    putc (ch, outfile);
  }
}
```

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

void main(void){
 char str[80];
 FILE *fp;
 if((fp = fopen("TEST", "w"))==NULL) {
  printf("arquivo não pode ser aberto\n");
  exit(1);
 }
 do {
  printf("entre uma string (ENTER para
        sair):\n");
  gets(str);
  strcat(str, "\n"); /* acrescenta nova linha */
  fputs(str, fp);
 } while(*str!='\n');
}
```

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

void main(int argc, char *argv[]){ /* Do teclado para o disco */
 FILE *fp;
 char ch;

 if(argc!=2) {
  printf("voce esqueceu de entrar o nome do arquivo\n");
  exit(1);
 }
 if((fp=fopen(argv[1],"w"))==NULL) {
  printf("arquivo não pode ser aberto\n");
  exit(1);
 }
 do {
  ch = getchar();
  putc(ch,fp);
 } while(ch!='$');
 fclose(fp);
}
```

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

void main(int argc, char *argv[]){
 FILE *in, *out;
 char ch;

if(argc!=3) {
  printf("Voce não informou nome do arquivo\n");
  exit(1);
 }
 if((in=fopen(argv[1], "r"))==NULL) {
  printf("arquivo fonte nao pode ser aberto\n");
  exit(1);
 }
 if((out=fopen(argv[2], "w"))==NULL) {
  printf("arquivo destino nao pode ser aberto\n");
  exit(1);
 }
```

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

void main(int argc, char *argv[]){
 FILE *in, *out;
 char ch;

if(argc!=3) {
  printf("Voce não informou nome do arquivo\n");
  exit(1);
 }
 if((in=fopen(argv[1], "r"))==NULL) {
  printf("arquivo fonte nao pode ser aberto\n");
  exit(1);
 }
 if((out=fopen(argv[2], "w"))==NULL) {
  printf("arquivo destino nao pode ser aberto\n");
  exit(1);
 }
```

```
/* Este codigo copia de fato o arquivo */
 while(!feof(in)) {
  ch = getc(in);
  if(!feof(in)) putc(ch, out);
 }
 fclose(in);
 fclose(out);
}
```

## Biblioteca padrão de I/O

- Relendo caracteres de um arquivo de entrada
  - Em algumas aplicações pode ser desejável ler o mesmo caracter mais de uma vez
  - *stdio.h* fornece a função *ungetc (ch, infile);*
    - Manda o caracter de volta ao arquivo de entrada
    - Caracter é retornado na próxima chamada à *getc ()*
    - Apenas um caracter pode ser mandado de volta de cada vez

## Biblioteca padrão de I/O

- Atualização do conteúdo de um arquivo
  - Não é possível abrir um arquivo para saída se ele já estiver sendo utilizado para entrada

## Biblioteca padrão de I/O

- Atualização do conteúdo de um arquivo
  - Alternativa:
    - Abrir o arquivo original para entrada
    - Abrir arquivo temporário com nome diferente para saída
    - Copiar o arquivo de entrada atualizado para o temporário
    - Fechar ambos os arquivos
    - Deletar o arquivo original
    - Renomear o arquivo temporário para o nome do original

## Biblioteca padrão de I/O

- Atualização do conteúdo de um arquivo
  - Interface *stdio.h* fornece três funções
    - Função *tmpnam (NULL)*: gera nome de um arquivo temporários adequando para a máquina em uso
      - Retorna o nome de arquivo gerado
    - Função *delete (nome)*: deleta o arquivo nome
    - Função *rename (nome_velho, nome_novo)*: troca o nome de um arquivo de nome_velho para nome_novo
      - Funções *delete()* e *rename()* retornam 0 se bem sucedidas

## Biblioteca padrão de I/O

- I/O orientada a linha
  - Arquivos são geralmente subdivididos em linhas individuais
  - É vantajoso ler e escrever uma linha inteira de uma vez
  - Interface stdio.h duas funções para isso:
    - Função *fgets()*: lê uma linha inteira
    - Função *fputs()*: escreve uma linha inteira

## Biblioteca padrão de I/O

- I/O orientada a linha
  - Função *fgets ()*
    - Protótipo: *string fgets (char buffer[], int tam, FILE *infile)*
    - Copia a próxima linha no vetor de caracteres *buffer*
    - Para depois de ler o primeiro caracter newline
      - Para antes se o *buffer* encher
    - Retorna o ponteiro para o vetor de caracteres dado como primeiro parâmetro
    - Se chamado no final do arquivo, retorna NULL

## Biblioteca padrão de I/O

- I/O orientada a linha
  - Função *fputs ()*
    - Protótipo: *void fputs (char *str, FILE *outfile);*
    - Copia caracteres do *string* para o arquivo de saída até atingir o final do *string*
    - *String* não precisa conter caracter de fim de linha (*newline ou \n*)
    - Pula linha automaticamente

## Biblioteca padrão de I/O

```
static void CopiaArquivo (FILE *infile, FILE *outfile) {
    char buffer[MaxLinha];

    while (fgets (buffer, MaxLinha, infile) != NULL) {
        fputs (buffer, outfile);
    }
}
```

## Biblioteca padrão de I/O

- I/O formatado
  - Funções *printf ()*
    - Facilitam saída de dados
  - Funções *scanf ()*
    - Facilitam entrada de dados

## Biblioteca padrão de I/O

- Funções *printf ()*
  - Três formatos:
    - *printf* (string de controle, ...);
      - Escreve sua saída para a saída padrão
    - *fprintf* (arquivo de saída, string de controle, ...);
      - Escreve sua saída em um arquivo de saída
    - *sprintf* (vetor de caracteres, string de controle, ...);
      - Escreve sua saída em um vetor de caracteres
      - Usuário tem que garantir que exista espaço suficiente no vetor
  - Ex.: *printf ("x = %d\n", valor);*

## Biblioteca padrão de I/O

- Funções *scanf ()*
  - Três formatos
    - *scanf* (string de controle, ...);
      - Lê a entrada da entrada padrão
    - *fscanf* (arquivo de entrada, *string* de controle, ...);
      - Lê a entrada de um arquivo de entrada
    - *sscanff* (vetor de caracteres, *string* de controle, ...);
      - Lê a entrada a partir de um *string* de caracteres
  - Ex.: *fscanf (file,"%2d %f %*d %8s", &x, &y, nome);*

## Biblioteca padrão de I/O

- Funções *scanf ()*
  - Como tem que retornar informação a quem a chamou, usa chamada por referência
  - Cada um dos argumentos depois do *string* de controle deve ser um ponteiro
    - Geralmente se inclui o & na frente do nome da variável para convertê-la a seu endereço
    - Lembre-se: nome de um vetor já é um ponteiro

## Biblioteca padrão de I/O

- Funções *scanf ()*
  - Não têm trabalhado bem na prática e são usadas com pouca frequência em aplicações comerciais
  - Especificações de conversão não são tão completas como as especificações de conversão das funções *printf ()*

## Outras bibliotecas ANSI

- Outras bibliotecas ANSI muito utilizadas:
  - Biblioteca *stdlib*
    - Interface *stdlib.h* exporta várias definições padrão (NULL, malloc, free, abs...)
  - Biblioteca *ctype*
    - Interface *ctype.h* exporta funções para operações com valores alfanuméricos
  - Biblioteca *math*
    - Interface *math.h* exporta funções matemáticas

## Exercício

- Escreva um programa que, usando informações sobre 10 alunos (10 registros) imprima as seguintes informações
  - Nota média, desvio padrão das notas, menor nota, maior nota
  - Nome e nota dos alunos em ordem alfabética

## Exercício

- Cada registro de um aluno tem os campos
  - Nome
  - Idade
  - Nota

## Conclusão

- Introdução
- Arquivos
- Biblioteca de entrada e saída
- Funções de entrada e saída
- Outras bibliotecas