Estrutura de Dados

Claudio Cesar de Sá, Alessandro Ferreira Leite, Lucas Hermman Negri, Gilmário Barbosa

Departamento de Ciência da Computação
Centro de Ciências e Tecnológias
Universidade do Estado de Santa Catarina

10 de agosto de 2017

Sumário (1)

Sumário (2)

Agradecimentos

Vários autores e colaboradores ...

- Ao Google Images ...

Disciplina

Estrutura de Dados – EDA001

- ***Turma:***
- ***Professor:*** Claudio Cesar de Sá
  - `claudio.sa@udesc.br`
  - Sala 13 Bloco F
- ***Carga horária:*** 72 horas-aula • Teóricas: 36 • Práticas: 36
- ***Curso:*** BCC
- ***Requisitos:*** LPG, Linux, sólidos conhecimentos da linguagem C – há um documento específico sobre isto
- ***Período:*** 2º semestre de 2017
- ***Horários:***
  - 3ª 15h20 (2 aulas) - F-205 – aula expositiva
  - 5ª 15h20 (2 aulas) - F-205 – lab

Ementa

Ementa

Representação e manipulação de tipos abstratos de dados. Estruturas lineares. Introdução a estruturas hierárquicas. Métodos de classificação. Análise de eficiência. Aplicações.

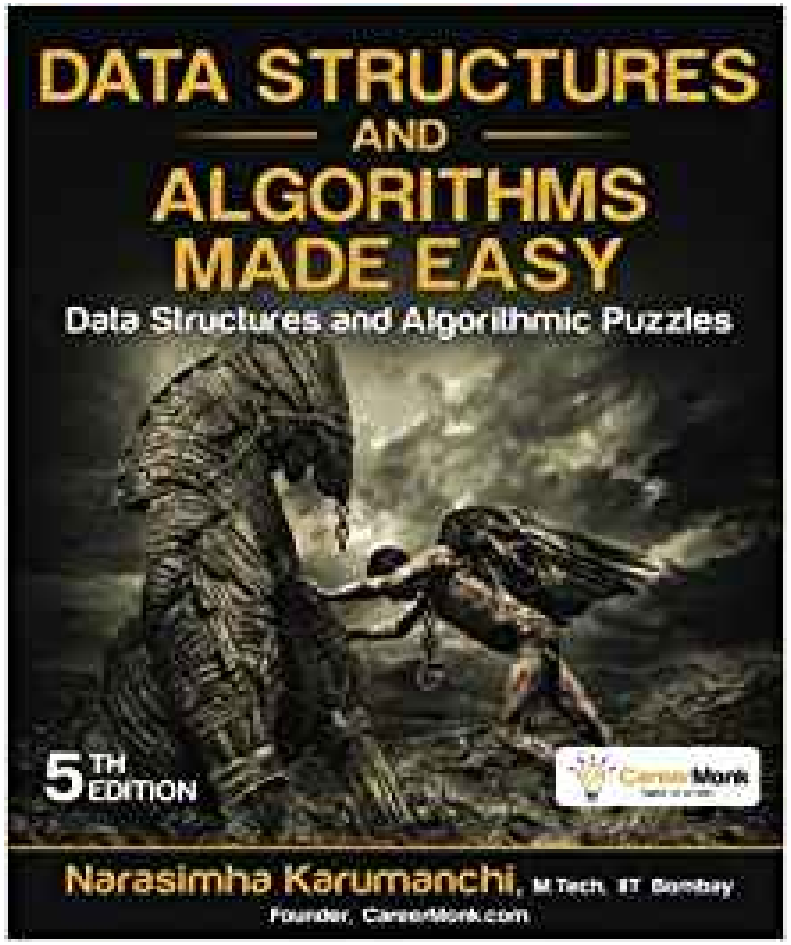
Objetivos (1)

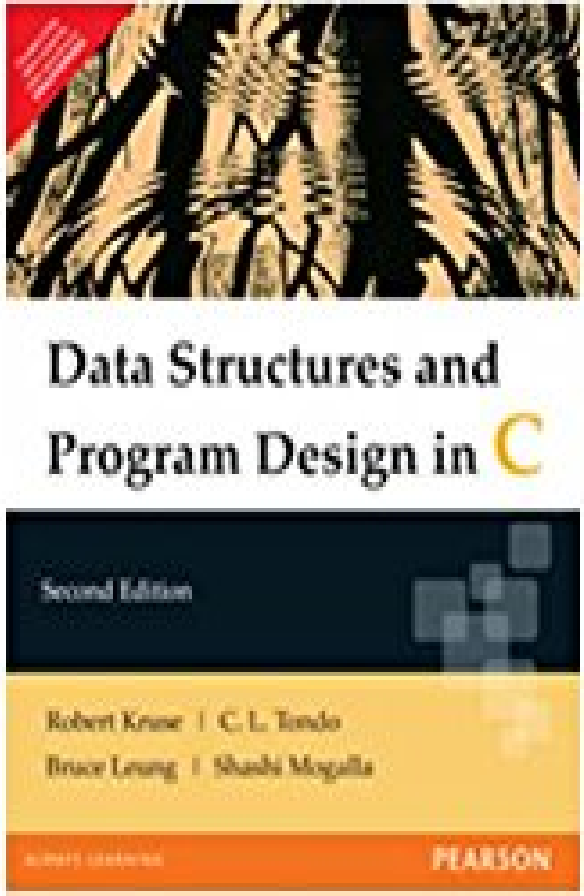
- ***Geral:***
  Há um documento específico sobre isto = Plano de Ensino

Objetivos (2)

- ***Específicos:***
  Há um documento específico sobre isto = Plano de Ensino

Livros que estarei usando ...





Conteúdo programático

Há um documento específico sobre isto = Plano de Ensino

Bibliografia UDESC

Há um documento específico sobre isto = Plano de Ensino

Conteúdo programático

Há um documento específico sobre isto = Plano de Ensino

Ferramentas ... nesta ordem

- Linux
- Linguagem C
- Codeblock

Metodologia e avaliação (1)

Metodologia:

As aulas serão expositivas e práticas. A cada novo assunto tratado, exemplos são demonstrados utilizando ferramentas computacionais adequadas para consolidar os conceitos tratados.

Metodologia e avaliação (2)

Avaliação

- Três provas – $\approx 90\%$
  - P_1: xx/set
  - P_2: yy/out
  - P_F: zz/nov

  (provão: todo conteúdo)
- Exercícios de laboratório – $\approx \%$
- Presença e participação: 75% é o mínimo obrigatório para a UDESC. Quem quiser faltar por razões diversas, ou assuntos específicos, trate pessoalmente com o professor.
- Tarefas extras que geram pontos por excelência
- Média para aprovação: 6,0 (seis)
  Nota maior ou igual a 6,0, repito a mesma no Exame Final. Caso contrário, regras da UDESC se aplica.

Dinâmica de Aula (1)

- Há um monitor na disciplina
- Há uma lista de discussão (para avisos e dúvidas gerais): `eda-lista@googlegroups.com`
- $\approx$ Teoria na 3a. feira
- $\approx$Prática na 5a. feira
- E/ou 50% do tempo em teoria, 50% implementações
- Onde tudo vai estar atualizado?

Dinâmica de Aula (2)

- `https://github.com/claudiosa/CCS/tree/master/estrutura_dados_EDA`
- Ou seja, tudo vai estar *rolando* no GitHub do professor
- No Google: `github + claudiosa`
- Finalmente ...

Dinâmica de Aula (3)

- Questões específicas (leia-se: notas, dor-de-dente, etc) venha falar pessoalmente com o professor!

Bibliografia (1)

Básica:

- Há um documento específico sobre isto = Plano de Ensino ... veja em detalhes tudo que foi escrito aqui
- Mais uma vez: `https://github.com/claudiosa/CCS/tree/master/estrutura_dados_EDA`

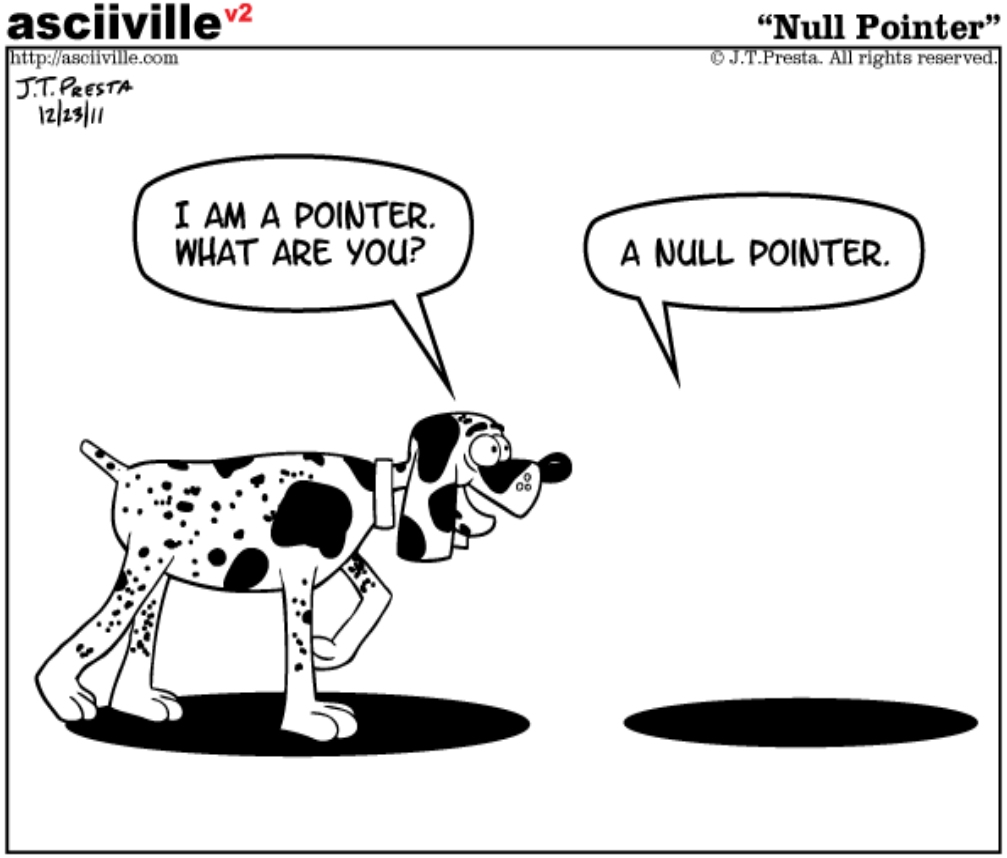
Antes de Começarmos .... (1)

- Todos os cursos de Estrutura de Dados começam com uma motivação em torno da área para Ciência
- Vou omitir ... mas reflita se ela é ou não onipresente no nosso cotidiano?
- Exemplos: bancos eletronicos, web, smartphones, etc

Capítulo 01 – Ponteiros (1)

Pontos fundamentais a serem cobertos:

1. **Pré-requisito: prática na linguagem C**
2. Exemplos – lúdicos
3. Ponteiros aos diversos tipos de dados
4. Uso de Memória
5. Alocação de memória Estática x Dinâmica
6. Funções para alocação de memória
7. Utilizando as funções para alocação de memória
8. Alocação de memória e estruturas em C



Motivação aos Ponteiros (1)

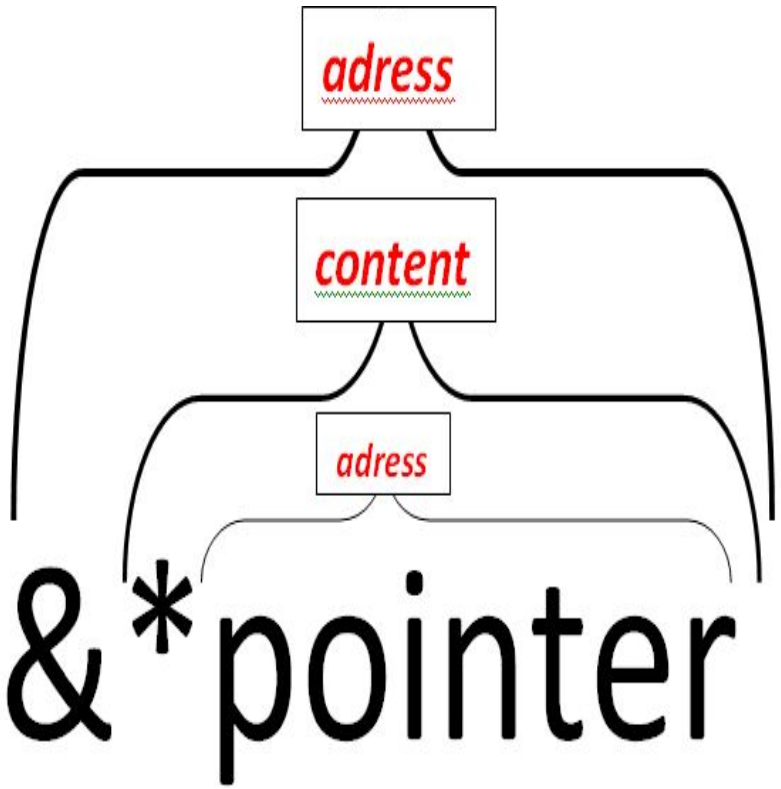


Figura: A história está por vir ...

Exemplos Lúdicos

```c
#include <stdio.h>
int main()
{
  int a;
  int *ptr;  // declara um ponteiro -- ptr-- para um inteiro
             // um ponteiro para uma variavel do tipo inteiro
  a = 90;
  ptr = &a;
  printf("Valor de A: %d\n", a);
  printf("Valor de ptr: %d \t Conteudo via ptr: %d\n", ptr, *ptr);
  return 1;
}
```

Exemplos Lúdicos

```c
#include <stdio.h>
int main()
{
  int a;
  int *ptr;  // declara um ponteiro -- ptr-- para um inteiro
             // um ponteiro para uma variavel do tipo inteiro
  a = 90;
  ptr = &a;
  printf("Valor de A: %d\n", a);
  printf("Valor de ptr: %d \t Conteudo via ptr: %d\n", ptr, *ptr);
  return 1;
}
```

- Ao executarmos esse código, qual será a sua saída?

Exemplos e agora?

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int a;
  int *ptr;  // declara um ponteiro -- ptr-- para um inteiro
             // um ponteiro para uma variavel do tipo INTEIRO
  a = 2017;
  ptr = &a;
  system("clear");
  printf("Valor de A: %d\t Endereco de A: %x\n", a, &a);
  printf("Valor de ptr: %d \t Conteudo via ptr: %d\n", ptr, *ptr);
  printf("Endereco de PTR: %x\n", & ptr);
  return 1;
}
```

Exemplos e agora?

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int a;
  int *ptr;  // declara um ponteiro -- ptr-- para um inteiro
             // um ponteiro para uma variavel do tipo INTEIRO
  a = 2017;
  ptr = &a;
  system("clear");
  printf("Valor de A: %d\t Endereco de A: %x\n", a, &a);
  printf("Valor de ptr: %d \t Conteudo via ptr: %d\n", ptr, *ptr);
  printf("Endereco de PTR: %x\n", & ptr);
  return 1;
}
```

Exemplos e agora?

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int a;
  int *ptr;  // declara um ponteiro -- ptr-- para um inteiro
             // um ponteiro para uma variavel do tipo INTEIRO
  a = 2017;
  ptr = &a;
  system("clear");
  printf("Valor de A: %d\t Endereco de A: %x\n", a, &a);
  printf("Valor de ptr: %d \t Conteudo via ptr: %d\n", ptr, *ptr);
  printf("Endereco de PTR: %x\n", & ptr);
  return 1;
}
```

- Qual é a saída do código acima?
- Quando não souber como funciona um comando em C?
- Várias respostas ... pense nelas e veja a melhor para voce!

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador $*$ (**estrela** – afinal é uma estrela em C) era quase desconhecido

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador $*$ (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador $*$ (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador $*$ (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente
- Reflita sobre ponteiros na vida real ...

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador $*$ (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente
- Reflita sobre ponteiros na vida real ...
- Quanto o $*$ é a essência das linguagens C e C++

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador * (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente
- Reflita sobre ponteiros na vida real ...
- Quanto o * é a essência das linguagens C e C++
- Entendendo isto vais entender o que há nas bibliotecas, como STL, etc

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador * (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente
- Reflita sobre ponteiros na vida real ...
- Quanto o * é a essência das linguagens C e C++
- Entendendo isto vais entender o que há nas bibliotecas, como STL, etc
- Resumindo:

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador * (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente
- Reflita sobre ponteiros na vida real ...
- Quanto o * é a essência das linguagens C e C++
- Entendendo isto vais entender o que há nas bibliotecas, como STL, etc
- Resumindo:

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador * (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente
- Reflita sobre ponteiros na vida real ...
- Quanto o * é a essência das linguagens C e C++
- Entendendo isto vais entender o que há nas bibliotecas, como STL, etc
- Resumindo:
  - O nome do ponteiro retorna o endereço para o qual ele aponta

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador * (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente
- Reflita sobre ponteiros na vida real ...
- Quanto o * é a essência das linguagens C e C++
- Entendendo isto vais entender o que há nas bibliotecas, como STL, etc
- Resumindo:
  - O nome do ponteiro retorna o endereço para o qual ele aponta
  - O operador (&) junto ao nome do ponteiro retorna o endereço do ponteiro

Reflexões Iniciais sobre Ponteiros

- O operador & era conhecido
- O operador * (**estrela** – afinal é uma estrela em C) era quase desconhecido
- Este último é como um catálogo telefônico, acessa um conteúdo de um assinante via uma página específica
- Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente
- Reflita sobre ponteiros na vida real ...
- Quanto o * é a essência das linguagens C e C++
- Entendendo isto vais entender o que há nas bibliotecas, como STL, etc
- Resumindo:
  - O nome do ponteiro retorna o endereço para o qual ele aponta
  - O operador (&) junto ao nome do ponteiro retorna o endereço do ponteiro
  - O operador (*) junto ao nome do ponteiro retorna o conteúdo da variável apontada

Ponteiro

- O correto entendimento e uso de ponteiros é crítico para uma programação bem-sucedida em C.

Ponteiro

- O correto entendimento e uso de ponteiros é crítico para uma programação bem-sucedida em C.
- Ponteiros são muitos utilizados em C, em parte porque eles são, às vezes, a única forma de expressar uma computação.

Ponteiro

- O correto entendimento e uso de ponteiros é crítico para uma programação bem-sucedida em C.
- Ponteiros são muitos utilizados em C, em parte porque eles são, às vezes, a única forma de expressar uma computação.
- Em alguns casos, o uso de ponteiro resulta em um código mais compacto e eficiente que obtido de outras formas.

Ponteiro

- O correto entendimento e uso de ponteiros é crítico para uma programação bem-sucedida em C.
- Ponteiros são muitos utilizados em C, em parte porque eles são, às vezes, a única forma de expressar uma computação.
- Em alguns casos, o uso de ponteiro resulta em um código mais compacto e eficiente que obtido de outras formas.
- Ponteiros e vetores são intimamente, relacionados.

Ponteiro

- O correto entendimento e uso de ponteiros é crítico para uma programação bem-sucedida em C.
- Ponteiros são muitos utilizados em C, em parte porque eles são, às vezes, a única forma de expressar uma computação.
- Em alguns casos, o uso de ponteiro resulta em um código mais compacto e eficiente que obtido de outras formas.
- Ponteiros e vetores são intimamente, relacionados.
- Aquela parte de vetores terem dimensões especificadas e fixas, é conhecida como **alocação estática**.

Ponteiro

- O correto entendimento e uso de ponteiros é crítico para uma programação bem-sucedida em C.
- Ponteiros são muitos utilizados em C, em parte porque eles são, às vezes, a única forma de expressar uma computação.
- Em alguns casos, o uso de ponteiro resulta em um código mais compacto e eficiente que obtido de outras formas.
- Ponteiros e vetores são intimamente, relacionados.
- Aquela parte de vetores terem dimensões especificadas e fixas, é conhecida como **alocação estática**.
- Os ponteiros irão servir para contornar esta limitação

Ponteiro

- Basicamente há três razões para utilizar ponteiros:

Ponteiro

- Basicamente há três razões para utilizar ponteiros:
  1. Ponteiros fornecem os meios pelos quais as funções podem modificar seus argumentos;

Ponteiro

- Basicamente há três razões para utilizar ponteiros:
  1. Ponteiros fornecem os meios pelos quais as funções podem modificar seus argumentos;
  2. Ponteiros são usados para suportar as rotinas de **alocação dinâmica** em C;

Ponteiro

- Basicamente há três razões para utilizar ponteiros:
  1. Ponteiros fornecem os meios pelos quais as funções podem modificar seus argumentos;
  2. Ponteiros são usados para suportar as rotinas de **alocação dinâmica** em C;
  3. O uso de ponteiros pode aumentar a eficiência de certas rotinas.

Ponteiro

- Basicamente há três razões para utilizar ponteiros:
  1. Ponteiros fornecem os meios pelos quais as funções podem modificar seus argumentos;
  2. Ponteiros são usados para suportar as rotinas de **alocação dinâmica** em C;
  3. O uso de ponteiros pode aumentar a eficiência de certas rotinas.
- Por outro lado, ponteiros podem ser comparados ao uso do comando **goto**, como uma forma diferente de escrever códigos impossíveis de entender.

Ponteiros e endereços

- Em um máquina típica, a memória é organizada como um vetor de células consecutivas numeradas ou endereçadas, que podem ser manipuladas individualmente ou em grupos contínuos.

Ponteiros e endereços

- Em um máquina típica, a memória é organizada como um vetor de células consecutivas numeradas ou endereçadas, que podem ser manipuladas individualmente ou em grupos contínuos.
- Uma situação comum é que qualquer *byte* pode ser um *char*, um par de células de um *byte* pode ser tratado como um inteiro short, etc.

Ponteiros e endereços

- Em um máquina típica, a memória é organizada como um vetor de células consecutivas numeradas ou endereçadas, que podem ser manipuladas individualmente ou em grupos contínuos.
- Uma situação comum é que qualquer *byte* pode ser um *char*, um par de células de um *byte* pode ser tratado como um inteiro short, etc.
- Um ponteiro é um grupo de células que podem conter um endereço.

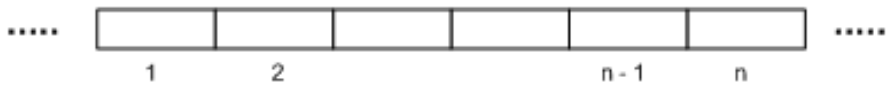


Figura: Representação da memória de uma máquina típica

Ponteiro

Definição

- É uma **variável que contém um endereço de memória**. Esse endereço é normalmente a posição de memória de uma outra variável.

Ponteiro

Definição

- É uma **variável que contém um endereço de memória**. Esse endereço é normalmente a posição de memória de uma outra variável.
- Se uma variável contém o endereço de uma outra, então a primeira é dita um ponteiro para a segunda.

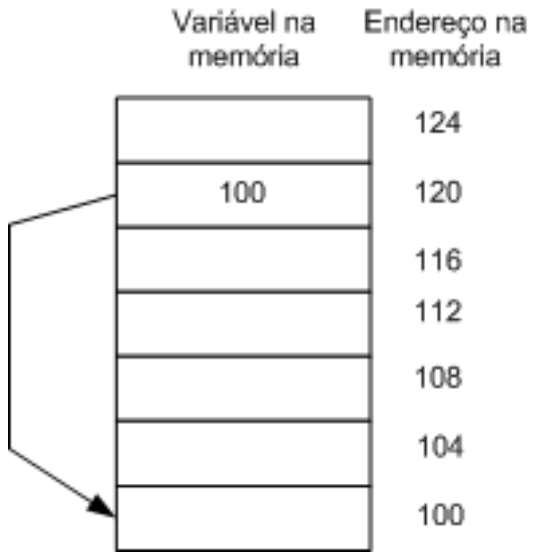


Figura: Representação de ponteiro

Variáveis Ponteiros

1. A linguagem C permite o armazenamento e a manipulação de valores de endereço de memória.

Variáveis Ponteiros

1. A linguagem C permite o armazenamento e a manipulação de valores de endereço de memória.
2. Para cada tipo existente (`int long float double char`), há um tipo ponteiro capaz de armazenar endereços de memória em que existem valores do tipo correspondente armazenados.

Variáveis Ponteiros

1. A linguagem C permite o armazenamento e a manipulação de valores de endereço de memória.
2. Para cada tipo existente (`int long float double char`), há um tipo ponteiro capaz de armazenar endereços de memória em que existem valores do tipo correspondente armazenados.
3. Um ponteiro proporciona um modo de acesso a variáveis sem referenciá-las diretamente.

Algumas das razões para utilizar ponteiros são:

1. Ponteiros fornecem maneiras com as quais as funções podem realmente modificar os argumentos que recebem.

Algumas das razões para utilizar ponteiros são:

1. Ponteiros fornecem maneiras com as quais as funções podem realmente modificar os argumentos que recebem.
2. Para criar estruturas de dados complexas, como listas encadeadas e árvores binárias, onde uma estrutura de dados deve conter referências sobre outra.

Algumas das razões para utilizar ponteiros são:

1. Ponteiros fornecem maneiras com as quais as funções podem realmente modificar os argumentos que recebem.
2. Para criar estruturas de dados complexas, como listas encadeadas e árvores binárias, onde uma estrutura de dados deve conter referências sobre outra.
3. Para comunicar informações sobre a memória, como na função **malloc** que retorna a localização de memória livre através do uso de ponteiro.

Algumas das razões para utilizar ponteiros são:

1. Ponteiros fornecem maneiras com as quais as funções podem realmente modificar os argumentos que recebem.
2. Para criar estruturas de dados complexas, como listas encadeadas e árvores binárias, onde uma estrutura de dados deve conter referências sobre outra.
3. Para comunicar informações sobre a memória, como na função **malloc** que retorna a localização de memória livre através do uso de ponteiro.
4. Notações de ponteiros compilam mais rapidamente tornando o código mais eficiente.

Algumas das razões para utilizar ponteiros são:

1. Ponteiros fornecem maneiras com as quais as funções podem realmente modificar os argumentos que recebem.
2. Para criar estruturas de dados complexas, como listas encadeadas e árvores binárias, onde uma estrutura de dados deve conter referências sobre outra.
3. Para comunicar informações sobre a memória, como na função **malloc** que retorna a localização de memória livre através do uso de ponteiro.
4. Notações de ponteiros compilam mais rapidamente tornando o código mais eficiente.
5. Para manipular matrizes mais facilmente através de movimentação de ponteiros para elas (ou parte delas), em vez de a própria matriz.

Variáveis do Tipos de Ponteiros

- A linguagem C não reserva uma palavra especial para a declaração de ponteiros.

Variáveis do Tipos de Ponteiros

- A linguagem C não reserva uma palavra especial para a declaração de ponteiros.
- As variáveis do tipo ponteiro são declaradas da seguinte forma: *tipo* com os nomes das variáveis precedidos pelo caractere *.

```
1     int *a, *b;
2
```

Variáveis do Tipos de Ponteiros

- A linguagem C não reserva uma palavra especial para a declaração de ponteiros.
- As variáveis do tipo ponteiro são declaradas da seguinte forma: *tipo* com os nomes das variáveis precedidos pelo caractere *****.

```
1     int *a, *b;
2
```

- A instrução acima declara que ***a** e ***b** são do tipo **int** e que ***a** e ***b** são ponteiros, isto é **a** e **b** contém endereços de variáveis do tipo **int**.

Operadores de Ponteiros

- A linguagem C oferece dois operadores unários para trabalharem com ponteiros.

Operador	significado
&	(“endereço de”)
*	(“conteúdo de”)

Operadores de Ponteiros

- O operador & (“endereço de”), aplicado a variáveis, resulta no endereço da posição da memória reservada para a variável. Por exemplo,

```
1          a = &b;
2
```

Operadores de Ponteiros

- O operador & ("endereço de"), aplicado a variáveis, resulta no endereço da posição da memória reservada para a variável. Por exemplo,

```
1          a = &b;
2
```

- coloca em **a** o endereço da memória que contém a variável **b**.

Operadores de Ponteiros

- O operador & ("endereço de"), aplicado a variáveis, resulta no endereço da posição da memória reservada para a variável. Por exemplo,

```
1          a = &b;
2
```

- coloca em **a** o endereço da memória que contém a variável **b**.
- O endereço não tem relação algum com o valor da variável **b**.

Operadores de Ponteiros

- O operador & ("endereço de"), aplicado a variáveis, resulta no endereço da posição da memória reservada para a variável. Por exemplo,

```
1          a = &b;
2
```

- coloca em **a** o endereço da memória que contém a variável **b**.
- O endereço não tem relação algum com o valor da variável **b**.
- Após as declarações as duas variáveis armazenam "lixos", pois não foram inicializadas.

Operadores de Ponteiros

- O operador * (“conteúdo de”), aplicado a variáveis do tipo ponteiro, acessa o conteúdo do endereço da memória pela variável ponteiro, isto é, devolve o conteúdo da variável apontada pelo operando. Por exemplo:

```
1         a = *b;
2
```

Operadores de Ponteiros

- O operador * ("conteúdo de"), aplicado a variáveis do tipo ponteiro, acessa o conteúdo do endereço da memória pela variável ponteiro, isto é, devolve o conteúdo da variável apontada pelo operando. Por exemplo:

```
1         a = *b;
2
```

- Coloca o valor de **b** em **a**, ou seja, **a** recebe o valor que está no endereço **b**.

Atribuição e acessos de endereço

```
1   /* a recebe o valor 5*/
2   a = 5;
3
4   /* p recebe o endereco de a (p aponta para a). */
5    p = &a;
6
7    /* conteudo de p recebe o valor 10 */
8    *p = 10;
9
```

Atribuição e acessos de endereço

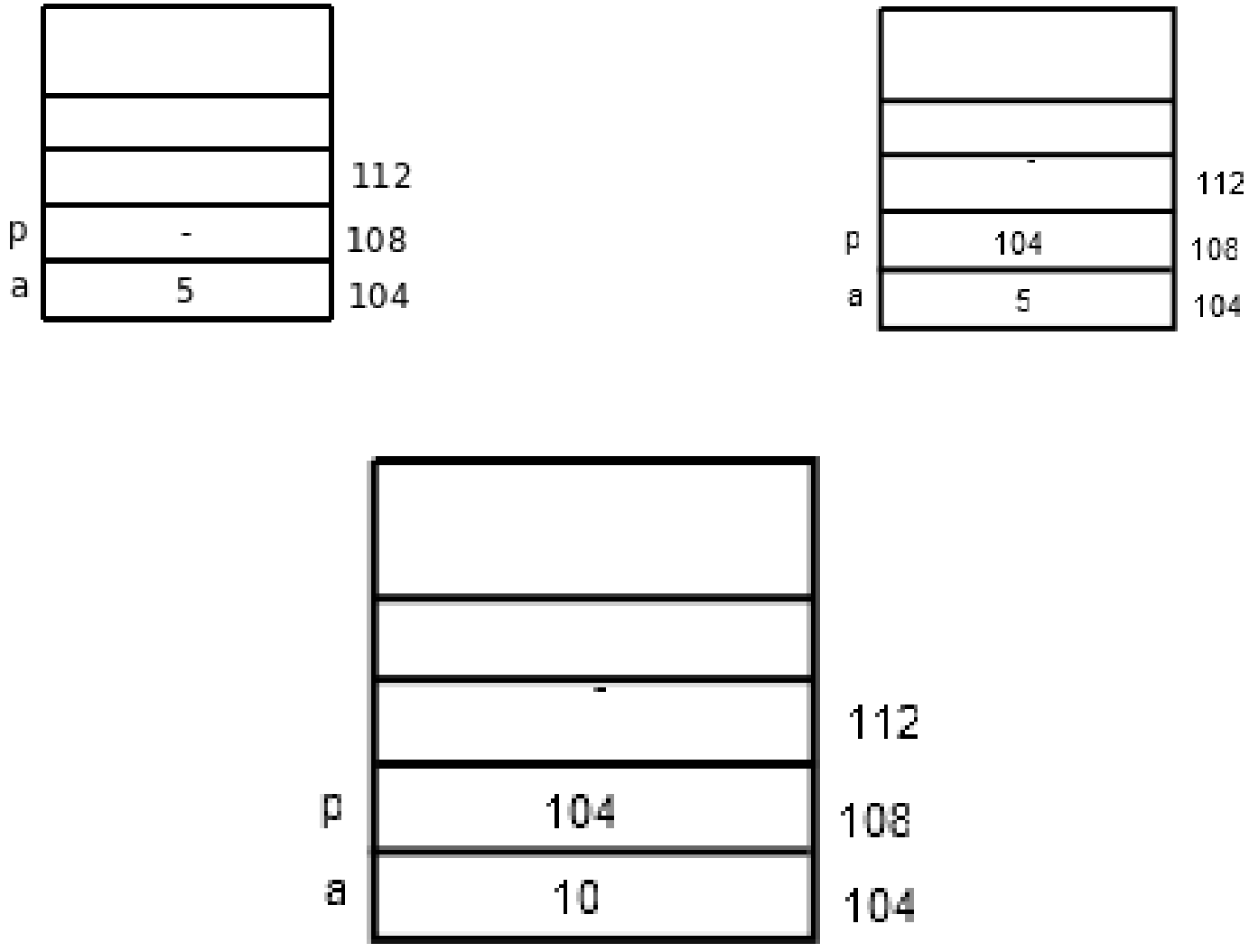
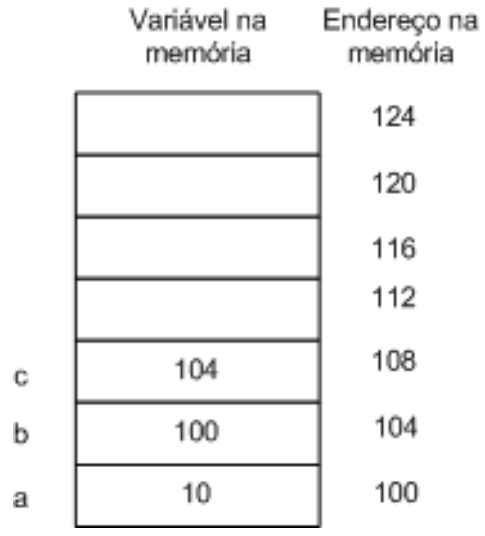


Figura: Efeito da atribuição de variáveis na pilha de execução

Atribuição de Ponteiros

- Assim como ocorre com qualquer variável, também podemos atribuir um valor a um ponteiro. Exemplo:

```
1  void main(void) {
2    int a = 10;
3    int *b, *c;
4    b = &a;
5    c = b;
6    printf("%p", c);
7  }
8
```



Operações com Ponteiros

- **Incremento/Decremento**:

Operações com Ponteiros

- **Incremento/Decremento**:
  - Podemos incrementar um ponteiro através da adição regular ou pelo operador de incremento ($++$).

Operações com Ponteiros

- **Incremento/Decremento**:
  - Podemos incrementar um ponteiro através da adição regular ou pelo operador de incremento ($++$).
  - Incrementar um ponteiro acarreta na movimentação do mesmo para o próximo tipo apontado.

Operações com Ponteiros

- **Incremento/Decremento**:
  - Podemos incrementar um ponteiro através da adição regular ou pelo operador de incremento ($++$).
  - Incrementar um ponteiro acarreta na movimentação do mesmo para o próximo tipo apontado.
  - Se **pa** é um ponteiro para inteiro com valor 200, depois de executada a instrução: **pa++**, o valor de **pa** será 202 e não 201.

Operações com Ponteiros

- **Incremento/Decremento**:
  - Podemos incrementar um ponteiro através da adição regular ou pelo operador de incremento ($++$).
  - Incrementar um ponteiro acarreta na movimentação do mesmo para o próximo tipo apontado.
  - Se **pa** é um ponteiro para inteiro com valor 200, depois de executada a instrução: **pa++**, o valor de **pa** será 202 e não 201.
  - Cuidar com o compilador e arquitetura em questão: 16, 32, 64 ou 128 bits

Operações com Ponteiros

- **Incremento/Decremento**:
  - Podemos incrementar um ponteiro através da adição regular ou pelo operador de incremento ($++$).
  - Incrementar um ponteiro acarreta na movimentação do mesmo para o próximo tipo apontado.
  - Se **pa** é um ponteiro para inteiro com valor 200, depois de executada a instrução: **pa++**, o valor de **pa** será 202 e não 201.
  - Cuidar com o compilador e arquitetura em questão: 16, 32, 64 ou 128 bits
  - Com isso, cada vez que incrementamos **pa** ele apontará para o próximo tipo apontado.

Operações com Ponteiros

- **Incremento/Decremento**:
  - Podemos incrementar um ponteiro através da adição regular ou pelo operador de incremento ($++$).
  - Incrementar um ponteiro acarreta na movimentação do mesmo para o próximo tipo apontado.
  - Se **pa** é um ponteiro para inteiro com valor 200, depois de executada a instrução: **pa++**, o valor de **pa** será 202 e não 201.
  - Cuidar com o compilador e arquitetura em questão: 16, 32, 64 ou 128 bits
  - Com isso, cada vez que incrementamos **pa** ele apontará para o próximo tipo apontado.
  - O mesmo é verdadeiro para o operador de decremento ($--$)

Operações com Ponteiros

- **Incremento/Decremento**:
  - Podemos incrementar um ponteiro através da adição regular ou pelo operador de incremento ($++$).
  - Incrementar um ponteiro acarreta na movimentação do mesmo para o próximo tipo apontado.
  - Se **pa** é um ponteiro para inteiro com valor 200, depois de executada a instrução: **pa++**, o valor de **pa** será 202 e não 201.
  - Cuidar com o compilador e arquitetura em questão: 16, 32, 64 ou 128 bits
  - Com isso, cada vez que incrementamos **pa** ele apontará para o próximo tipo apontado.
  - O mesmo é verdadeiro para o operador de decremento ($--$)
  - Cuidar ainda: associatividade (esq $\Leftrightarrow$ dir) e precedência (ver manual da linguagem) $\Rightarrow$ fazer os exercícios e ir anotando as respostas

Operações com Ponteiros

- **Comparações entre Ponteiros**:

Operações com Ponteiros

- **Comparações entre Ponteiros**:
  - Ponteiros podem ser comparados:

```
1   if (pa <> pb)
```

Operações com Ponteiros

- **Comparações entre Ponteiros**:
  - Ponteiros podem ser comparados:

```
1   if (pa <> pb)
```

  - Testes relacionais com $>=, <=, >, <$ são aceitos entre ponteiros, desde que os operandos sejam ponteiros.

Operações com Ponteiros

- **Comparações entre Ponteiros**:
  - Ponteiros podem ser comparados:

```
1   if (pa <> pb)
```

  - Testes relacionais com $>=, <=, >, <$ são aceitos entre ponteiros, desde que os operandos sejam ponteiros.
  - O tipo dos operandos devem ser o mesmo, para não obter resultados sem sentido.

Operações com Ponteiros

- **Comparações entre Ponteiros**:
  - Ponteiros podem ser comparados:

```
1   if (pa <> pb)
```

  - Testes relacionais com $>=, <=, >, <$ são aceitos entre ponteiros, desde que os operandos sejam ponteiros.
  - O tipo dos operandos devem ser o mesmo, para não obter resultados sem sentido.
  - Variáveis ponteiros podem ser testadas quanto à igualdade ($==$) ou desigualdade($!=$) onde os operandos são ponteiros, ou um dos operandos NULL.

Operações com Ponteiros

- **Atribuição**:

Operações com Ponteiros

- **Atribuição**:
  - Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1   pa = &a;
```

Operações com Ponteiros

- **Atribuição**:
  - Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1   pa = &a;
```

- **Leitura de valores**:

Operações com Ponteiros

- **Atribuição**:
  - Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1   pa = &a;
```

- **Leitura de valores**:
  - O operador (*) devolve o valor guardado no endereço apontado.

Operações com Ponteiros

- **Atribuição**:
  - Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1   pa = &a;
```

- **Leitura de valores**:
  - O operador (*) devolve o valor guardado no endereço apontado.
- **Leitura do endereço do ponteiro**:

Operações com Ponteiros

- **Atribuição**:
  - Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1   pa = &a;
```

- **Leitura de valores**:
  - O operador (*) devolve o valor guardado no endereço apontado.
- **Leitura do endereço do ponteiro**:
  - Os ponteiros variáveis também têm um endereço e um valor.

Operações com Ponteiros

- **Atribuição**:
  - Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1   pa = &a;
```

- **Leitura de valores**:
  - O operador (*) devolve o valor guardado no endereço apontado.
- **Leitura do endereço do ponteiro**:
  - Os ponteiros variáveis também têm um endereço e um valor.
  - O operador (&) retorna a posição da memória onde o ponteiro está localizado. Em resumo:

Operações com Ponteiros

- **Atribuição**:
  - Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1   pa = &a;
```

- **Leitura de valores**:
  - O operador (*) devolve o valor guardado no endereço apontado.
- **Leitura do endereço do ponteiro**:
  - Os ponteiros variáveis também têm um endereço e um valor.
  - O operador (&) retorna a posição da memória onde o ponteiro está localizado. Em resumo:
    1. O nome do ponteiro retorna o endereço para o qual ele aponta.

Operações com Ponteiros

- **Atribuição**:
  - Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1   pa = &a;
```

- **Leitura de valores**:
  - O operador (*) devolve o valor guardado no endereço apontado.
- **Leitura do endereço do ponteiro**:
  - Os ponteiros variáveis também têm um endereço e um valor.
  - O operador (&) retorna a posição da memória onde o ponteiro está localizado. Em resumo:
    1. O nome do ponteiro retorna o endereço para o qual ele aponta.
    2. O operador (&) junto ao nome do ponteiro retorna o endereço do ponteiro.

Operações com Ponteiros

- **Atribuição**:
  - Um endereço pode ser atribuído a um ponteiro através do operador unário & junto a uma variável simples. Exemplo:

```
1   pa = &a;
```

- **Leitura de valores**:
  - O operador (*) devolve o valor guardado no endereço apontado.
- **Leitura do endereço do ponteiro**:
  - Os ponteiros variáveis também têm um endereço e um valor.
  - O operador (&) retorna a posição da memória onde o ponteiro está localizado. Em resumo:
    1. O nome do ponteiro retorna o endereço para o qual ele aponta.
    2. O operador (&) junto ao nome do ponteiro retorna o endereço do ponteiro.
    3. O operador (*) junto ao nome do ponteiro retorna o conteúdo da variável apontada.

Chamadas por valor $\times$ referência

- A passagem de argumentos para funções em C são feitas por valor ("chamada por valor").
- Na passagem de parâmetro por valor a função chamada não pode alterar uma variável da função que fez a chamada.
- Sim, a chamada por valor cópia protege o conteúdo
- Mas, muitas vezes a duplicação do valor da variável deve ser evitado, ai precisamos da chamada por referência.
- Uso de ponteiros

Atribuição e acessos de endereço

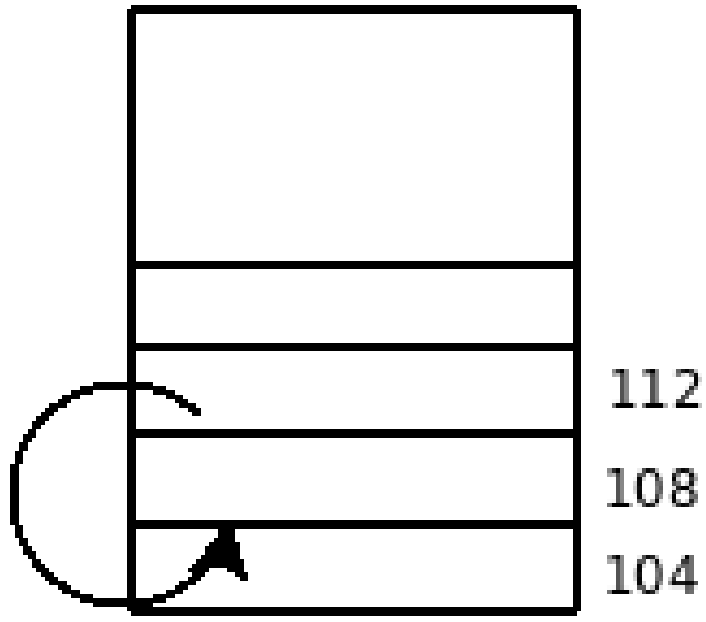


Figura: Representação gráfica do valor de um ponteiro

Chamada por referência

```
1  /*Ordena o vetor v de tamanho n, v[0 .. n-1]
2   * em ordem crescente.
3   */
4  void ordenacaoSelecao(int n, int v[]){
5   int i, j;
6   for(i = 0; i < n -1; i++)
7     for(j = i + 1; j < n; j++)
8       if (v[j] < v [i])
9         troca(v[i],v[j]);
10 }
11
12 void troca(int x, int y) {
13   int tmp =x;
14   x = y;
15   y = tmp;
16 }
```

Chamada por referência

- O código anterior cumpre o seu objetivo?

Chamada por referência

- O código anterior cumpre o seu objetivo?
- Por causa da chamada por valor, a função **troca** não afeta os argumentos **x** e **y** da rotina que chama, ou seja, o código está ERRADO.

Chamada por referência

- O código anterior cumpre o seu objetivo?
- Por causa da chamada por valor, a função **troca** não afeta os argumentos **x** e **y** da rotina que chama, ou seja, o código está ERRADO.
- Para obter o efeito desejado é necessário passar os endereços dos valores a serem permutados:

Chamada por referência

- O código anterior cumpre o seu objetivo?
- Por causa da chamada por valor, a função **troca** não afeta os argumentos **x** e **y** da rotina que chama, ou seja, o código está ERRADO.
- Para obter o efeito desejado é necessário passar os endereços dos valores a serem permutados:
  - **troca**(&v[i], &v[j]);

Chamada por referência

- O código anterior cumpre o seu objetivo?
- Por causa da chamada por valor, a função **troca** não afeta os argumentos **x** e **y** da rotina que chama, ou seja, o código está ERRADO.
- Para obter o efeito desejado é necessário passar os endereços dos valores a serem permutados:
  - **troca**(&v[i], &v[j]);
- O que muda na função troca?

Chamada por referência

```c
/* Ordena o vetor v de tamanho n, v[0 .. n-1]
 * em ordem crescente.
 */
void ordenacaoSelecao(int n, int v[]){
 int i, j;
 for(i = 0; i < n -1; i++)
   for(j = i + 1; j < n; j++)
     if (v[j] < v [i])
       troca(&v[i],&v[j]);
}
/*Permuta x e y*/
void troca(int *x, int *y) {
  int tmp = *x;
  *x = *y;
  *y = tmp;
}
```

Passagem por referência

- No código anterior os argumentos da função **troca** foram declarados como ponteiros.
- Os parâmetros ponteiros da função **troca** são ditos como de **entrada** e **saída**.
- Dessa forma, qualquer modificação realizada em **troca** fica visível à função que chamou.
- Para que uma função gere o efeito de chamada por referência, os ponteiros devem ser utilizados na declaração dos argumentos e a função chamadora deve mandar endereços como argumentos.

Ponteiros e matrizes

- Em C, há um estreito relacionamento entre ponteiros e matrizes.

Ponteiros e matrizes

- Em C, há um estreito relacionamento entre ponteiros e matrizes.
- O compilador transforma matrizes em ponteiros após a compilação do código.

Ponteiros e matrizes

- Em C, há um estreito relacionamento entre ponteiros e matrizes.
- O compilador transforma matrizes em ponteiros após a compilação do código.
- Qualquer operação que possa ser feita com índices de uma matriz pode ser feita com ponteiros.

Ponteiros e matrizes

- Em C, há um estreito relacionamento entre ponteiros e matrizes.
- O compilador transforma matrizes em ponteiros após a compilação do código.
- Qualquer operação que possa ser feita com índices de uma matriz pode ser feita com ponteiros.
- O nome de uma matriz é um endereço, ou seja, um ponteiro.

Ponteiros e matrizes

- Em C, há um estreito relacionamento entre ponteiros e matrizes.
- O compilador transforma matrizes em ponteiros após a compilação do código.
- Qualquer operação que possa ser feita com índices de uma matriz pode ser feita com ponteiros.
- O nome de uma matriz é um endereço, ou seja, um ponteiro.
- Ponteiros e matrizes são idênticos na maneira de acessar a memória.

Ponteiros e matrizes

- Em C, há um estreito relacionamento entre ponteiros e matrizes.
- O compilador transforma matrizes em ponteiros após a compilação do código.
- Qualquer operação que possa ser feita com índices de uma matriz pode ser feita com ponteiros.
- O nome de uma matriz é um endereço, ou seja, um ponteiro.
- Ponteiros e matrizes são idênticos na maneira de acessar a memória.
- Um **ponteiro variável** é um endereço onde é armazenado um outro endereço.

Exemplos de programas com matrizes

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

int strtam(char   *  );

int main(void)
{   char vetor[] = "ABCDEFG" ;
   //  char *lista = "Ola como vai?"; ... HUM ...
   char *pt;
   pt = vetor; //mais saudavel => pt = &vetor[0];

   system("clear");
   printf("O tamanho de \"%s\" e %d caracteres.\n", vetor, strtam( pt ));
   printf("\n ... Acabou ....");
   return 1;
}

int strtam(char *s){
    int tam=0;
    //while(*(s + tam++) != '\0');
    while(*s != '\0')
    {   tam++;
                s++;
        }
    return  tam; //tam-1; --> \0
}
```

Exemplos de programas com matrizes

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

int strtam(char   *  );

int main(void)
{   char vetor[] = "ABCDEFG" ;
   //  char *lista = "Ola como vai?"; ... HUM ...
   char *pt;
   pt = vetor; //mais saudavel => pt = &vetor[0];

   system("clear");
   printf("O tamanho de \"%s\" e %d caracteres.\n", vetor, strtam( pt ));
   printf("\n ... Acabou ....");
   return 1;
}

int strtam(char *s){
    int tam=0;
    //while(*(s + tam++) != '\0');
    while(*s != '\0')
    {   tam++;
                s++;
        }
    return  tam; //tam-1; --> \0
}
```

Vários elementos neste código

Exemplo de programas com matrizes

```c
/*imprime os valores da matriz*/
main(){
    int nums [5] = {100,200,90,20,10};
    int d;
    for(d = 0; d < 5; d++)
      printf("%d\n", nums[d]);
}

/*usa ponteiros para imprimir os valores da matriz*/
main(){
    int nums [5] = {100,200,90,20,10};
    int d;
    for(d = 0; d < 5; d++)
      printf("%d\n", *(nums + d));
}
```

Ponteiros e matrizes

- O segundo programa é idêntico ao primeiro, exceto pela expressão ***(nums + d)**.

Ponteiros e matrizes

- O segundo programa é idêntico ao primeiro, exceto pela expressão ***(nums + d)**.
- O efeito de ***(nums + d)** é o mesmo que **nums[d]**.

Ponteiros e matrizes

- O segundo programa é idêntico ao primeiro, exceto pela expressão ***(nums + d)**.
- O efeito de ***(nums + d)** é o mesmo que **nums[d]**.
- A expressão ***(nums + d)** é o endereço do elemento de índice **d** da matriz.

Ponteiros e matrizes

- O segundo programa é idêntico ao primeiro, exceto pela expressão ***(nums + d)**.
- O efeito de ***(nums + d)** é o mesmo que **nums[d]**.
- A expressão ***(nums + d)** é o endereço do elemento de índice **d** da matriz.
- Se cada elemento da matriz é um inteiro e d $= 3$, então serão pulados 6 bytes para atingir o elemento de índice 3.

Ponteiros e matrizes

- O segundo programa é idêntico ao primeiro, exceto pela expressão ***(nums + d)**.
- O efeito de ***(nums + d)** é o mesmo que **nums[d]**.
- A expressão ***(nums + d)** é o endereço do elemento de índice **d** da matriz.
- Se cada elemento da matriz é um inteiro e d = 3, então serão pulados 6 bytes para atingir o elemento de índice 3.
- Assim, a expressão ***(nums + d)** não significa avançar 3 *bytes*, além *nums* e sim 3 elementos da matriz.

Indireção múltipla

O que é *indireção* múltipla?

- *Indireção múltipla* ou **ponteiro de ponteiros** é quando temos um ponteiro apontando para outro ponteiro que aponta para o valor final.

Indireção múltipla

O que é *indireção* múltipla?

- *Indireção múltipla* ou **ponteiro de ponteiros** é quando temos um ponteiro apontando para outro ponteiro que aponta para o valor final.
- O valor normal de um ponteiro é o endereço de uma variável que contém o valor desejado.

Indireção múltipla

O que é *indireção* múltipla?

- *Indireção múltipla* ou **ponteiro de ponteiros** é quando temos um ponteiro apontando para outro ponteiro que aponta para o valor final.
- O valor normal de um ponteiro é o endereço de uma variável que contém o valor desejado.
- No caso de um ponteiro para um ponteiro, o primeiro contém o endereço do segundo que aponta para a variável que contém o valor desejado.

Indireção múltipla

O que é *indireção* múltipla?

- *Indireção múltipla* ou **ponteiro de ponteiros** é quando temos um ponteiro apontando para outro ponteiro que aponta para o valor final.
- O valor normal de um ponteiro é o endereço de uma variável que contém o valor desejado.
- No caso de um ponteiro para um ponteiro, o primeiro contém o endereço do segundo que aponta para a variável que contém o valor desejado.
- A *indireção múltipla* pode ser levada a qualquer dimensão desejada.

Indireção múltipla

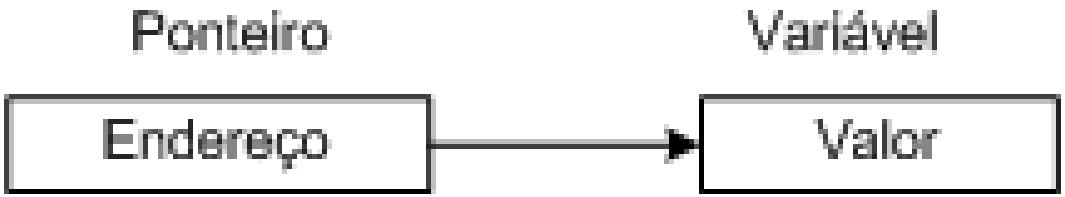


Figura: Indireção simples

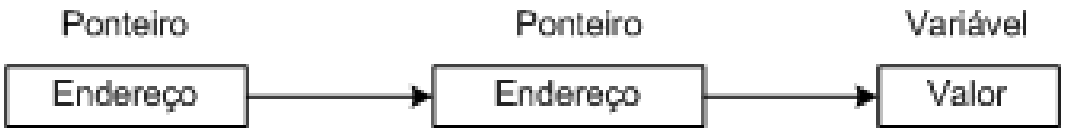


Figura: Indireção múltipla

Indireção múltipla

Declaração

- Uma variável que é um ponteiro para um ponteiro deve ser declarada como tal.

Indireção múltipla

Declaração

- Uma variável que é um ponteiro para um ponteiro deve ser declarada como tal.
- A declaração de ponteiro de ponteiro é realizada colocando-se um * adicional na frente do nome da variável. Exemplo:

```
1 int **a;
2 float **b;
```

Indireção múltipla

Declaração

- Uma variável que é um ponteiro para um ponteiro deve ser declarada como tal.
- A declaração de ponteiro de ponteiro é realizada colocando-se um * adicional na frente do nome da variável. Exemplo:

```
1 int **a;
2 float **b;
```

- No primeiro exemplo, temos a declaração de um ponteiro para um ponteiro de inteiro (**int**).

Indireção múltipla

Declaração

- Uma variável que é um ponteiro para um ponteiro deve ser declarada como tal.
- A declaração de ponteiro de ponteiro é realizada colocando-se um * adicional na frente do nome da variável. Exemplo:

```
1 int **a;
2 float **b;
```

- No primeiro exemplo, temos a declaração de um ponteiro para um ponteiro de inteiro (**int**).
- É importante salientar que **a** não é um ponteiro para um número inteiro, mas um ponteiro para um ponteiro inteiro.

Exemplo de indireção múltipla

```
1 int main(void) {
2   int x, *a, **b;
3   x = 4;
4   a = &x;
5   b = &a;
6   printf("%d ", **b);
7 }
```

Ponteiro para funções

- Um recurso muito poderoso da linguagem C, é o ponteiro para função.

Ponteiro para funções

- Um recurso muito poderoso da linguagem C, é o ponteiro para função.
- Embora uma função não seja uma variável, ela tem uma posição física na memória que pode ser atribuída a um ponteiro. Exemplo:

```c
int main(){
  int (*func)(const char*, ...);
  func = printf;
  (*func)("%d\n", 1);
}
```

Ponteiro para funções

- Um recurso muito poderoso da linguagem C, é o ponteiro para função.
- Embora uma função não seja uma variável, ela tem uma posição física na memória que pode ser atribuída a um ponteiro. Exemplo:

```c
int main(){
  int (*func)(const char*, ...);
  func = printf;
  (*func)("%d\n", 1);
}
```

- No código acima, a instrução

```c
int (*func)(const char*, ...);
```

declara uma função do tipo **int**. Nesse exemplo, estamos declarando que **func** é uma função do tipo ponteiro para inteiro que aponta para o endereço da função *printf*, através da instrução:

```c
func = printf;
```

Ponteiro para funções

- A declaração anterior é possível e válida, porque quando compilamos uma função, o código-fonte é transformado em código-objeto e um ponto de entrada é estabelecido.

Ponteiro para funções

- A declaração anterior é possível e válida, porque quando compilamos uma função, o código-fonte é transformado em código-objeto e um ponto de entrada é estabelecido.
- Quando é feita uma chamada à função, enquanto o programa está sendo executado, é efetuado uma chamada em linguagem de máquina para esse ponto de entrada.

Ponteiro para funções

- A declaração anterior é possível e válida, porque quando compilamos uma função, o código-fonte é transformado em código-objeto e um ponto de entrada é estabelecido.
- Quando é feita uma chamada à função, enquanto o programa está sendo executado, é efetuado uma chamada em linguagem de máquina para esse ponto de entrada.
- Portanto, se um ponteiro contém o endereço do ponto de entrada de uma função, então ele pode ser usado para chamar essa função.

Ponteiro para funções

- A declaração anterior é possível e válida, porque quando compilamos uma função, o código-fonte é transformado em código-objeto e um ponto de entrada é estabelecido.
- Quando é feita uma chamada à função, enquanto o programa está sendo executado, é efetuado uma chamada em linguagem de máquina para esse ponto de entrada.
- Portanto, se um ponteiro contém o endereço do ponto de entrada de uma função, então ele pode ser usado para chamar essa função.
- Dessa forma, ao executarmos a instrução:

```
1 (*func)("\%d\n", 1);
```

estamos executando a função **printf**, logo, será apresentado na saída, o valor 1.

Ponteiro para funções

- O endereço de uma função é obtido usando o nome da função sem parênteses ou argumentos.

Ponteiro para funções

- O endereço de uma função é obtido usando o nome da função sem parênteses ou argumentos.
- Observe que não colocamos parênteses junto ao nome da função. Se eles estiverem presentes como em:

```
1 func = printf();
```

, estaríamos atribuindo a ***func*** o valor retornado pela função e não o endereço dela.

Ponteiro para funções

- O endereço de uma função é obtido usando o nome da função sem parênteses ou argumentos.
- Observe que não colocamos parênteses junto ao nome da função. Se eles estiverem presentes como em:

```
1 func = printf();
```

, estaríamos atribuindo a ***func*** o valor retornado pela função e não o endereço dela.

- O nome de uma função desacompanhado de parênteses é o endereço dela.

Ponteiro para funções

- Nesse exemplo,nada é obtido e bastante confusão é introduzida. Porém, há momentos em que é vantajoso passar funções arbitrárias para procedimentos, ou manter uma matriz de funções.

Ponteiro para funções

- Nesse exemplo,nada é obtido e bastante confusão é introduzida. Porém, há momentos em que é vantajoso passar funções arbitrárias para procedimentos, ou manter uma matriz de funções.
- Por exemplo, escolher o melhor algoritmo para resolver um problema.

Ponteiro para funções

```c
/**
 * Ordena o vetor v de tamanho n, utilizando
 * o algoritmo de ordenacao implementado pela
 * funcao: algOrdenacao.
 */
void ordenar(int v[], int n,
   void (*algOrdenacao)(int v[], int n)){
   (*algOrdenacao)(v,n);
}
```

Ponteiro para funções

- Em resumo podemos:
  1. Declarar um ponteiro para uma função.
  2. Atribuir o endereço de uma função a um ponteiro.
  3. Chamar a função através do ponteiro para ela.
- Mas não podemos:
  1. Incrementar ou decrementar ponteiros para funções.
  2. Incrementar ou decrementar nomes de funções.

Alocação dinâmica

- Na declaração de um vetor visto até agora é necessário conhecer e informar o número de elementos do vetor.

Alocação dinâmica

- Na declaração de um vetor visto até agora é necessário conhecer e informar o número de elementos do vetor.
- Esse pré-dimensionamento é um fator limitante, pois, nos obriga a conhecer de antemão a quantidade de elementos do vetor.

Alocação dinâmica

- Na declaração de um vetor visto até agora é necessário conhecer e informar o número de elementos do vetor.
- Esse pré-dimensionamento é um fator limitante, pois, nos obriga a conhecer de antemão a quantidade de elementos do vetor.
- Uma solução é dimensionar o vetor com um número muito alto, para não termos limitações no momento de utilização do programa.

Alocação dinâmica

- Na declaração de um vetor visto até agora é necessário conhecer e informar o número de elementos do vetor.
- Esse pré-dimensionamento é um fator limitante, pois, nos obriga a conhecer de antemão a quantidade de elementos do vetor.
- Uma solução é dimensionar o vetor com um número muito alto, para não termos limitações no momento de utilização do programa.
- Essa solução leva a um desperdício de memória.

Alocação dinâmica

- Na declaração de um vetor visto até agora é necessário conhecer e informar o número de elementos do vetor.
- Esse pré-dimensionamento é um fator limitante, pois, nos obriga a conhecer de antemão a quantidade de elementos do vetor.
- Uma solução é dimensionar o vetor com um número muito alto, para não termos limitações no momento de utilização do programa.
- Essa solução leva a um desperdício de memória.
- Qual a solução?

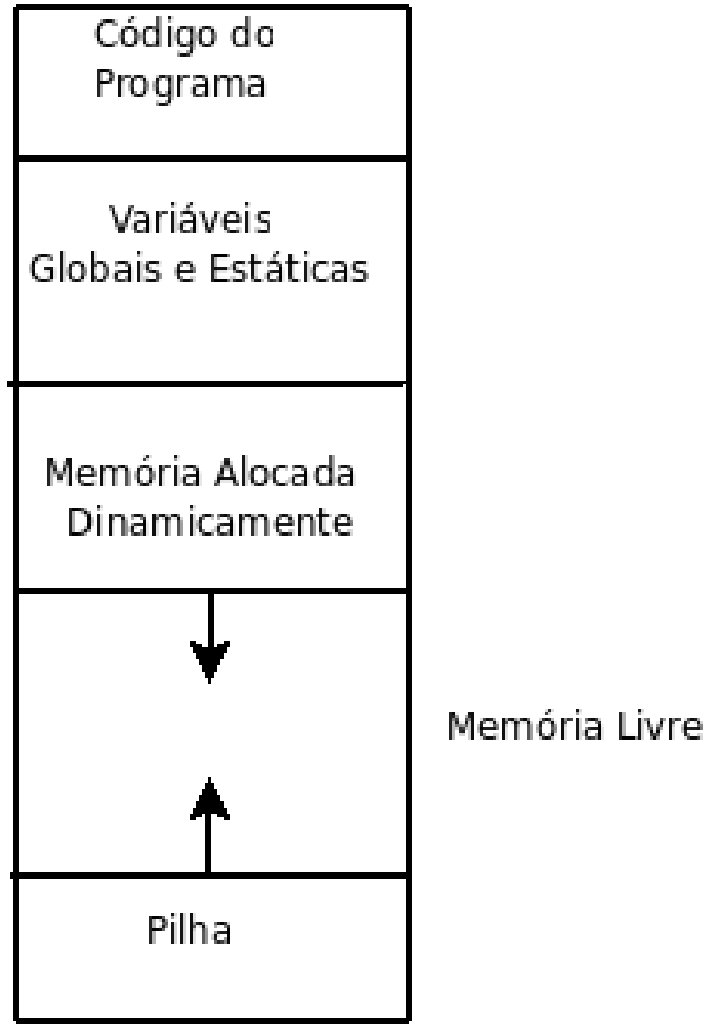
Alocação dinâmica

- Na declaração de um vetor visto até agora é necessário conhecer e informar o número de elementos do vetor.
- Esse pré-dimensionamento é um fator limitante, pois, nos obriga a conhecer de antemão a quantidade de elementos do vetor.
- Uma solução é dimensionar o vetor com um número muito alto, para não termos limitações no momento de utilização do programa.
- Essa solução leva a um desperdício de memória.
- Qual a solução?
  - Utilizar alocação dinâmica.

Alocação dinâmica

- Na declaração de um vetor visto até agora é necessário conhecer e informar o número de elementos do vetor.
- Esse pré-dimensionamento é um fator limitante, pois, nos obriga a conhecer de antemão a quantidade de elementos do vetor.
- Uma solução é dimensionar o vetor com um número muito alto, para não termos limitações no momento de utilização do programa.
- Essa solução leva a um desperdício de memória.
- Qual a solução?
  - Utilizar alocação dinâmica.
  - Isto é, requisitar espaços de memória em tempo de execução.

Uso da memória



Funções para alocação dinâmica de memória

- A função básica para alocar memória é **malloc**.

Funções para alocação dinâmica de memória

- A função básica para alocar memória é **malloc**.
- A função **malloc** recebe como parâmetro o número de *bytes* que se deseja alocar e retorna o endereço inicial da área de memória alocada.

Funções para alocação dinâmica de memória

- A função básica para alocar memória é **malloc**.
- A função **malloc** recebe como parâmetro o número de *bytes* que se deseja alocar e retorna o endereço inicial da área de memória alocada.
- O código seguinte realiza a alocação dinâmica de um vetor de inteiros com 10 elementos.

```
1  int *v;
2  v = malloc(10*4);
```

Funções para alocação dinâmica de memória

- A função básica para alocar memória é **malloc**.
- A função **malloc** recebe como parâmetro o número de *bytes* que se deseja alocar e retorna o endereço inicial da área de memória alocada.
- O código seguinte realiza a alocação dinâmica de um vetor de inteiros com 10 elementos.

```
1  int *v;
2  v = malloc(10*4);
```

- Após a execução, se a alocação for bem-sucedida, **v** armazenará o endereço inicial de uma área contínua de memória suficiente para armazenar 10 valores inteiros.

Funções para alocação dinâmica de memória

- A função básica para alocar memória é **malloc**.
- A função **malloc** recebe como parâmetro o número de *bytes* que se deseja alocar e retorna o endereço inicial da área de memória alocada.
- O código seguinte realiza a alocação dinâmica de um vetor de inteiros com 10 elementos.

```
1  int *v;
2  v = malloc(10*4);
```

- Após a execução, se a alocação for bem-sucedida, **v** armazenará o endereço inicial de uma área contínua de memória suficiente para armazenar 10 valores inteiros.
- Podemos tratar **v** como tratamos um vetor declarado estaticamente.

Funções para alocação dinâmica de memória

- A função básica para alocar memória é **malloc**.
- A função **malloc** recebe como parâmetro o número de *bytes* que se deseja alocar e retorna o endereço inicial da área de memória alocada.
- O código seguinte realiza a alocação dinâmica de um vetor de inteiros com 10 elementos.

```
1  int *v;
2  v = malloc(10*4);
```

- Após a execução, se a alocação for bem-sucedida, **v** armazenará o endereço inicial de uma área contínua de memória suficiente para armazenar 10 valores inteiros.
- Podemos tratar **v** como tratamos um vetor declarado estaticamente.
- Para ficarmos independente de compilador e máquinas, usamos o operador **sizeof()**.

```
1  int *v;
2  v = malloc(10*sizeof(int));
```

Funções para alocação dinâmica de memória

- A função **malloc** pode ser utilizada para alocar espaço para armazenar valores de qualquer tipo.

Funções para alocação dinâmica de memória

- A função **malloc** pode ser utilizada para alocar espaço para armazenar valores de qualquer tipo.
- O retorno da função **malloc** é um ponteiro genérico, para um tipo qualquer, representado por **void***, que pode ser convertido para o tipo apropriado da atribuição.

Funções para alocação dinâmica de memória

- A função **malloc** pode ser utilizada para alocar espaço para armazenar valores de qualquer tipo.
- O retorno da função **malloc** é um ponteiro genérico, para um tipo qualquer, representado por **void***, que pode ser convertido para o tipo apropriado da atribuição.
- É comum fazer a conversão explicitamente, realizando um cast para o tipo correto. Exemplo:

```
1 int *v;
2 v = (int*) malloc(10 * sizeof(int));
```

Funções para alocação dinâmica de memória

- A função **malloc** pode ser utilizada para alocar espaço para armazenar valores de qualquer tipo.
- O retorno da função **malloc** é um ponteiro genérico, para um tipo qualquer, representado por **void***, que pode ser convertido para o tipo apropriado da atribuição.
- É comum fazer a conversão explicitamente, realizando um cast para o tipo correto. Exemplo:

```
1  int *v;
2  v = (int*) malloc(10 * sizeof(int));
```

- Se porventura, não houver espaço livre suficiente para realizar a alocação, a função retorna um endereço nulo, representado por NULL, definido em *stdlib.h*.

Funções para alocação dinâmica de memória

- A função **malloc** pode ser utilizada para alocar espaço para armazenar valores de qualquer tipo.
- O retorno da função **malloc** é um ponteiro genérico, para um tipo qualquer, representado por **void***, que pode ser convertido para o tipo apropriado da atribuição.
- É comum fazer a conversão explicitamente, realizando um cast para o tipo correto. Exemplo:

```
1  int *v;
2  v = (int*) malloc(10 * sizeof(int));
```

- Se porventura, não houver espaço livre suficiente para realizar a alocação, a função retorna um endereço nulo, representado por NULL, definido em *stdlib.h*.
- Podemos verificar se a alocação foi realizada adequadamente, testando o retorno da função **malloc**. Exemplo:

```
1  v = (int*) malloc(10 * sizeof(int));
2  if (v == NULL)
3    printf("Memoria insuficiente.\n");
```

Funções para alocação dinâmica de memória

- Para liberar um espaço de memória alocado dinamicamente, usamos a função **free()**.

Funções para alocação dinâmica de memória

- Para liberar um espaço de memória alocado dinamicamente, usamos a função **free()**.
- A função **free()** recebe como parâmetro o ponteiro da memória a ser liberado.

```
1 free(v);
```

Funções para alocação dinâmica de memória

- Para liberar um espaço de memória alocado dinamicamente, usamos a função **free()**.
- A função **free()** recebe como parâmetro o ponteiro da memória a ser liberado.

```
1 free(v);
```

- Só podemos passar para a função **free()** um endereço de memória que tenha sido alocado dinamicamente.

Funções para alocação dinâmica de memória

- Para liberar um espaço de memória alocado dinamicamente, usamos a função **free()**.
- A função **free()** recebe como parâmetro o ponteiro da memória a ser liberado.

```
1 free(v);
```

- Só podemos passar para a função **free()** um endereço de memória que tenha sido alocado dinamicamente.
- Não podemos acessar o espaço de memória depois de liberado.

Funções para alocação dinâmica de memória

Função	**Descrição**
malloc(<*qtd. bytes*>)	Aloca uma área da memória e retorna a referência para o endereço inicial, se existir memória disponível. Caso contrário, retorna **NULL**.
sizeof(<*tipo*>)	Retorna a quantidade de memória necessária para para alocar um determinado tipo.
free(<*variável*>)	Libera o espaço de memória ocupado por uma variável alocada dinamicamente.

Capítulo xxxxx – Pilha

Pontos fundamentais a serem cobertos:

1. Contexto e motivação
2. Definição
3. Implementações
4. Exercícios

Introdução

- Uma das estruturas de dados mais simples.
- É a estrutura de dados mais utilizada em programação.
- É uma metáfora emprestada do mundo real, que a computação utiliza para resolver muitos problemas de forma simplificada.

Definição

Definição

Um conjunto ordenado de itens no qual novos itens podem ser inseridos e a partir do qual podem ser eliminados em uma extremidade denominada topo da pilha.

Definição

Definição

Um conjunto ordenado de itens no qual novos itens podem ser inseridos e a partir do qual podem ser eliminados em uma extremidade denominada topo da pilha.

Definição

Uma seqüência de objetos, todos do mesmo tipo, sujeita às seguintes regras de comportamento:

1. Sempre que solicitado a remoção de um elemento, o elemento removido é o último da seqüência.
2. Sempre que solicitado a inserção de um novo elemento, o objeto é inserido no fim da seqüência (topo).

Pilha

- Uma pilha é um objeto dinâmico, constantemente mutável, onde elementos são inseridos e removidos.
- Em uma pilha, cada novo elemento é inserido no topo.
- Os elementos da pilha só podem ser retirado na ordem inversa à ordem em que foram inseridos
  - O primeiro que sai é o último que entrou.
  - Por essa razão, uma pilha é dita uma estrutura do tipo: LIFO(*last-in, first* ou UEPS último a entrar é o primeiro a sair.)

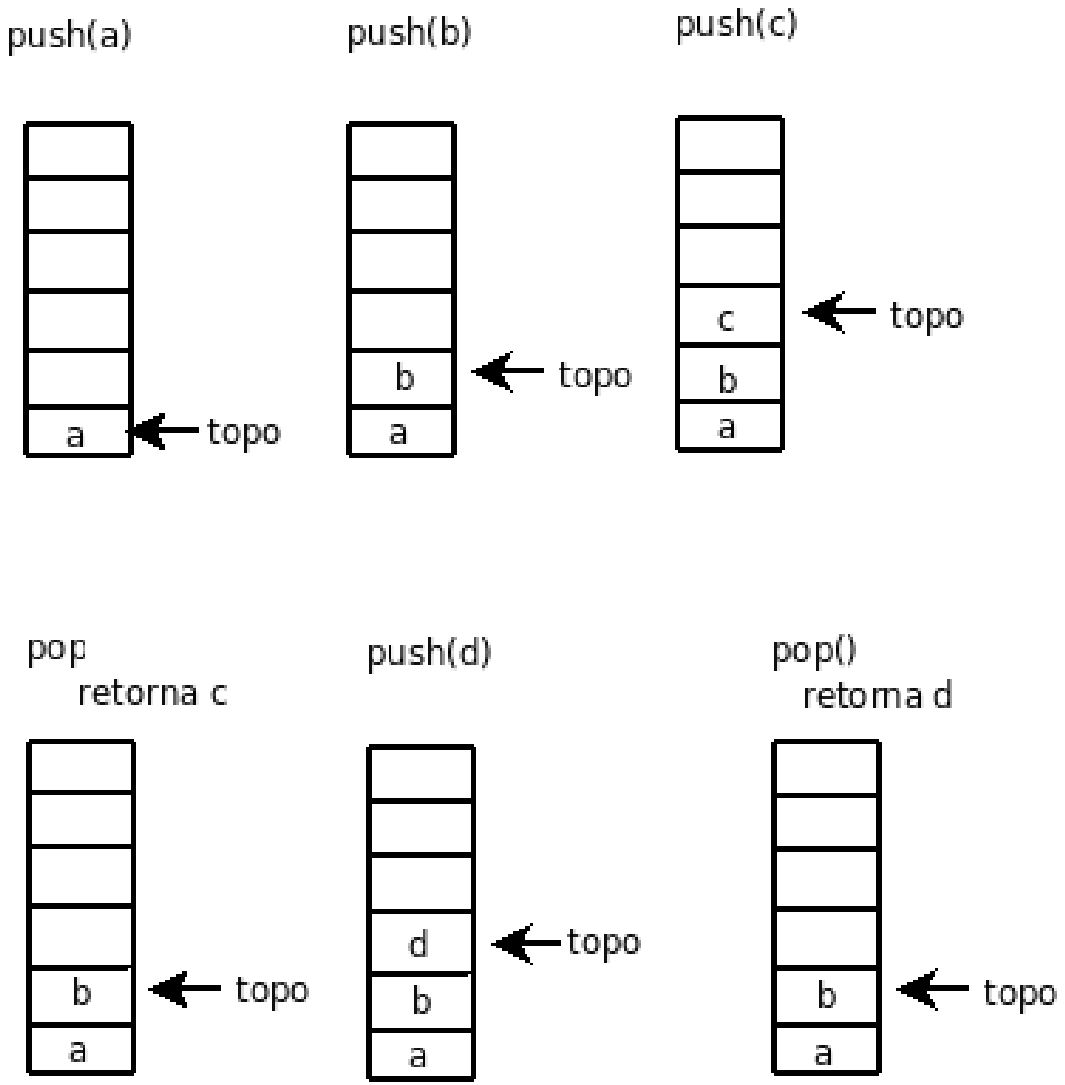
Operações básicas

- As operações básicas que devem ser implementadas em uma estrutura do tipo pilha são:

Operação	**Descrição**
push(p, e)	empilha o elemento *e*, inserindo-o no topo da pilha *p*.
pop(p)	desempilha o elemento do topo da pilha *p*.

Tabela: Operações básicas da estrutura de dados pilha.

Exemplo


push(a)
a
topo
push(b)
b
a
topo
push(c)
c
b
a
topo
pop
retorna c
b
a
topo
push(d)
d
b
a
topo
pop()
retoma d
b
a
topo

Operações auxiliares

- Além das operações básicas, temos as operações "*auxiliares*". São elas:

Operação	**Descrição**
create	cria uma pilha vazia.
empty(p)	determina se uma pilha p está ou não vazia.
free(p)	libera o espaço ocupado na memória pela pilha p.

Tabela: Operações auxiliares da estrutura de dados pilha.

Interface do Tipo Pilha

```c
/* Definicao da estrutura */
typedef struct pilha Pilha;
/*Aloca dinamicamente a estrutura pilha, inicializando
 *seus campos e retorna seu ponteiro.*/
Pilha* create(void);

/*Insere o elemento e na pilha p.*/
void push(Pilha *p, int e);

/*Retira e retorna o elemento do topo da pilha p*/
int pop(Pilha *p);

/*Informa se a pilha p esta ou nao vazia.*/
int empty(Pilha *p);
```

Implementação de Pilha com Vetor

- Normalmente as aplicações que precisam de uma estrutura pilha, é comum saber de antemão o número máximo de elementos que precisam estar armazenados simultaneamente na pilha.
- Essa estrutura de pilha tem um limite conhecido.
- Os elementos são armazenados em um vetor.
- Essa implementação é mais simples.
- Os elementos inseridos ocupam as primeiras posições do vetor.

Implementação de Pilha com Vetor

- Seja p uma pilha armazenada em um vetor VET de N elementos:
  1. O elemento vet[topo] representa o elemento do topo.
  2. A parte ocupada pela pilha é vet[0 .. topo - 1].
  3. A pilha está vazia se topo = -1.
  4. Cheia se topo = N - 1.
  5. Para desempilhar um elemento da pilha, não vazia, basta

$$x = vet[topo - -]$$

.

  6. Para empilhar um elemento na pilha, em uma pilha não cheia, basta

$$vet[t + +] = e$$

.

Implementação de Pilha com Vetor

```
1  #define N 20 /* numero maximo de elementos*/
2  #include <stdio.h>
3  #include "pilha.h"
4
5  /*Define a estrutura da pilha*/
6  struct pilha{
7    int topo; /* indica o topo da pilha */
8    int elementos[N]; /* elementos da pilha*/
9  };
10
11 Pilha* create(void){
12   Pilha* p = (Pilha*) malloc(sizeof(Pilha));
13   p->topo = -1; /* inicializa a pilha com 0 elementos */
14   return p;
15 }
16
```

Implementação de Pilha com Vetor

- Empilha um elemento na pilha

```
1 void push(Pilha *p, int e){
2      if (p->topo == N - 1){ /* capacidade esgotada */
3         printf("A pilha esta cheia");
4         exit(1);
5      }
6      /* insere o elemento na proxima posicao livre */
7      p->elementos[++p->topo] = e;
8 }
```

Implementação de Pilha com Vetor

- Desempilha um elemento da pilha

```
1  int pop(Pilha *p)
2  {
3        int e;
4        if (empty(p)){
5           printf("Pilha vazia.\n");
6           exit(1);
7        }
8
9        /* retira o elemento do topo */
10       e = p->elementos[p->topo--];
11       return e;
12 }
```

Implementação de Pilha com Vetor

```
1 /**
2  * Verifica se a pilha p esta vazia
3  */
4 int empty(Pilha *p)
5 {
6     return (p->t == -1);
7 }
```

Exemplo de uso

- Na área computacional existem diversas aplicações de pilhas.
- Alguns exemplos são: caminhamento em árvores, chamadas de sub-rotinas por um compilador ou pelo sistema operacional, inversão de uma lista, avaliar expressões, entre outras.
- Uma das aplicações clássicas é a conversão e a avaliação de expressões algébricas. Um exemplo, é o funcionamento das calculadoras da HP, que trabalham com expressões pós-fixadas.