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

19 de setembro de 2017

Sumário I

Sumário II

Agradecimentos

Vários autores e colaboradores ...

- Ao Google Images ...

Onde estamos ...

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.

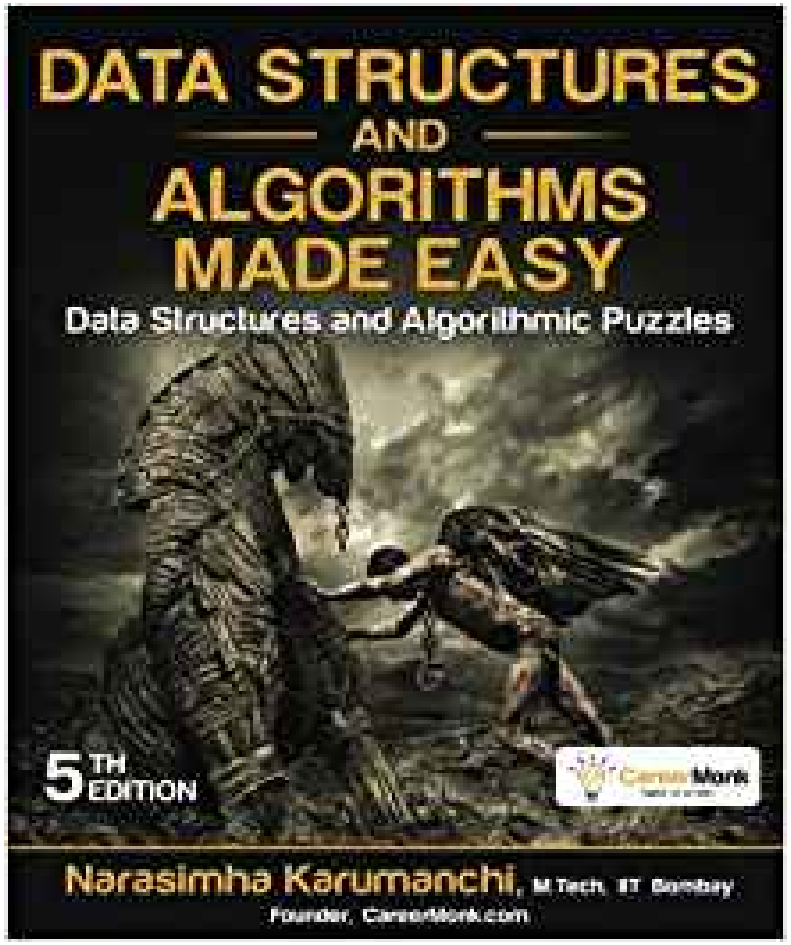
Objetivos I

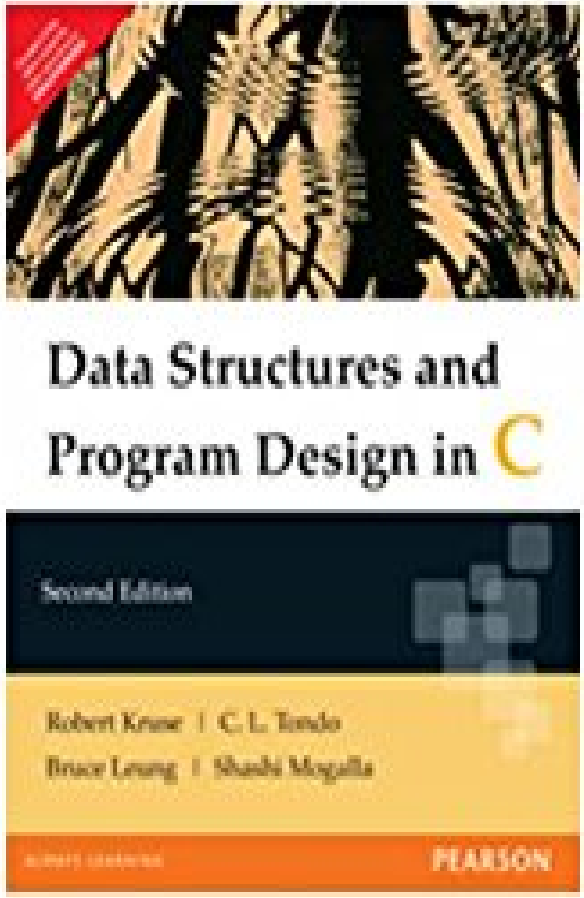
- ***Geral:***
  Há um documento específico sobre isto = Plano de Ensino

Objetivos II

- ***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 (ora o compilador g++)
- Codeblock

Metodologia e avaliação I

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 II

Avaliação

- Três provas – $\approx 90\%$
  - P_1: xx/ago
  - P_2: xx/set
  - P_3: xx/set
  - P_4: yy/out
  - P_5: yy/nov
  - 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

Metodologia e avaliação III

- 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 aplicam.
- Sitio das avaliações: `https://run.codes/Users/login` código da disciplina: **GEPZ**

Dinâmica de Aula I

- Há um monitor na disciplina – Lucas – ver no site de monitoria da UDESC os horários
- 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 II

- 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 III

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

Bibliografia I

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`

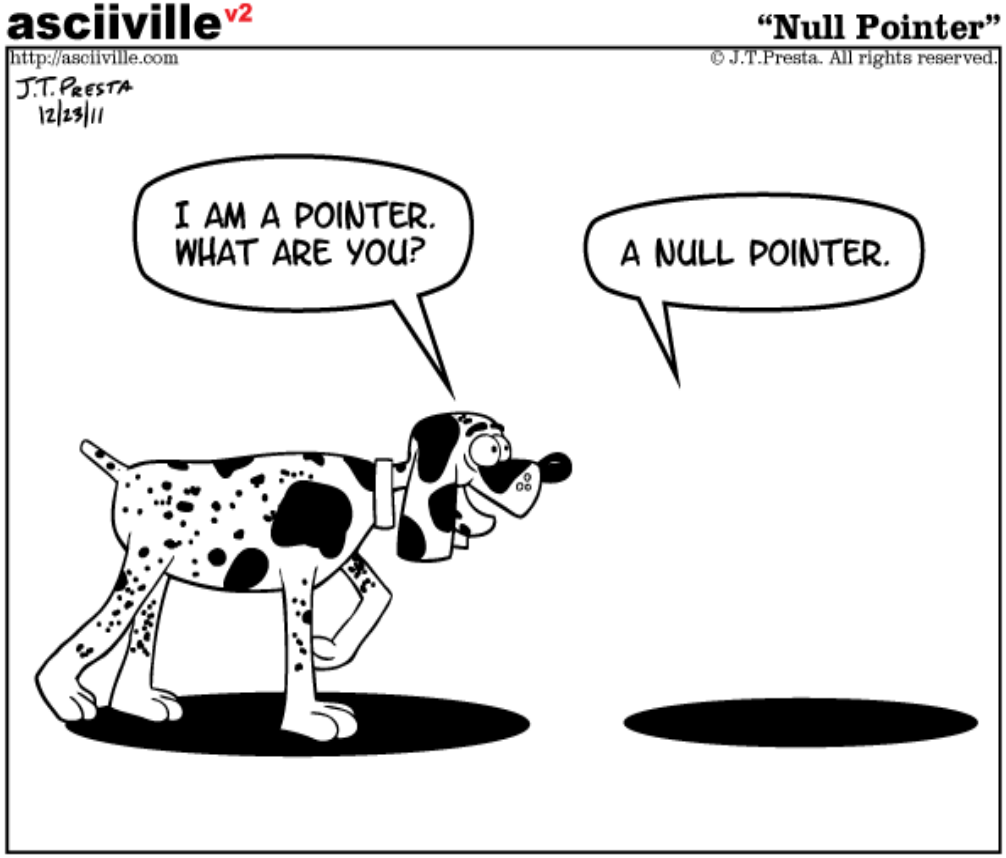
Antes de Começarmos I

- 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

Onde estamos ...

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



Motivação aos Ponteiros I

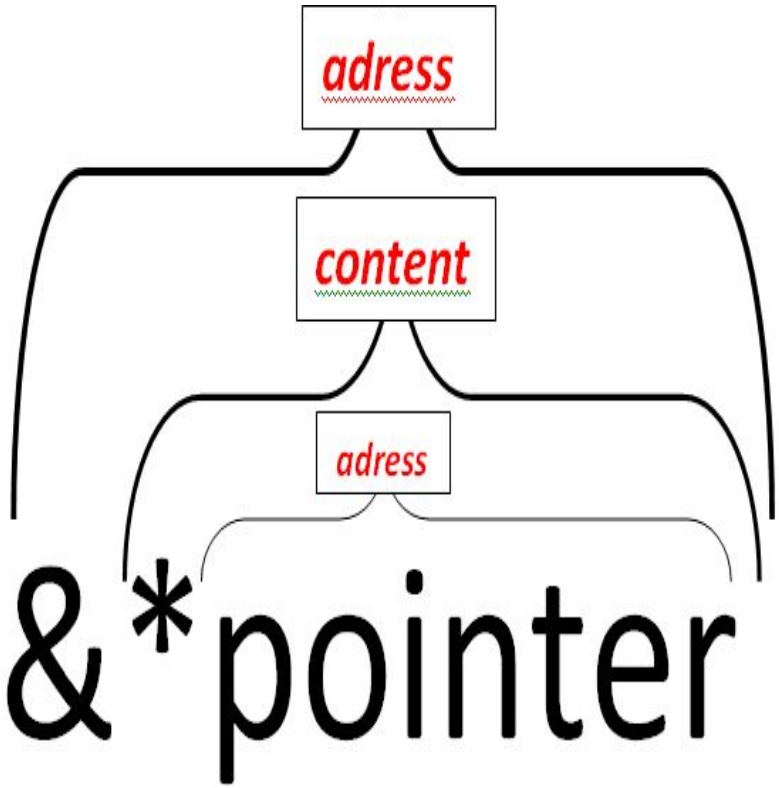


Figura 1: A história está por vir ...

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.

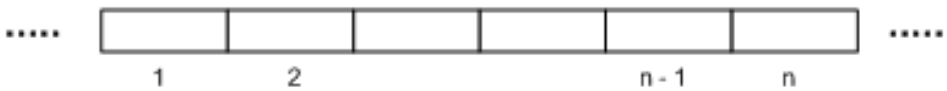


Figura 2: Representação da memória de uma máquina típica

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.

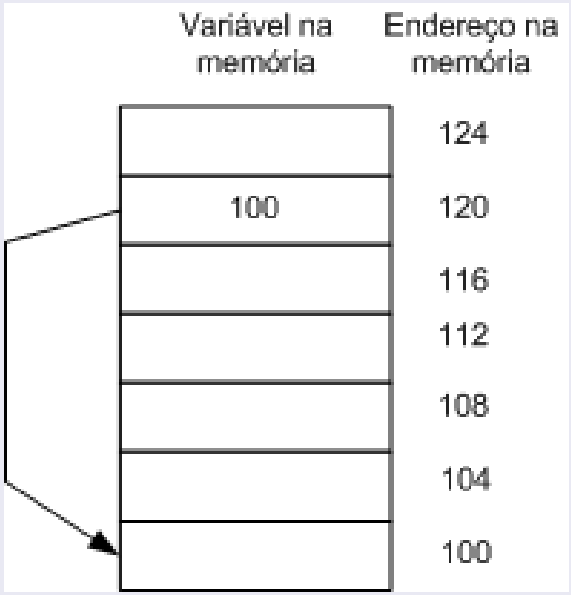


Figura 3: Representação de ponteiro

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

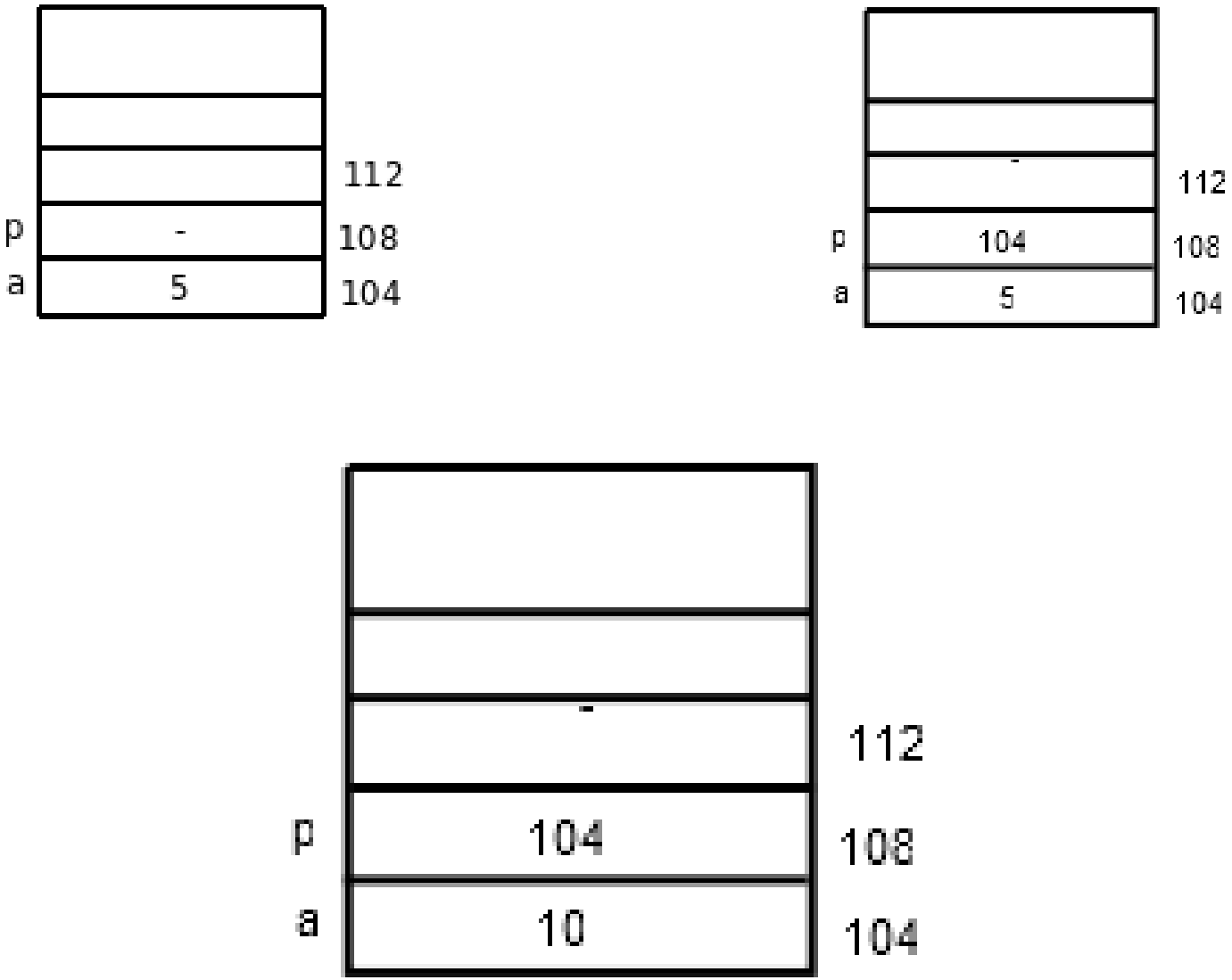
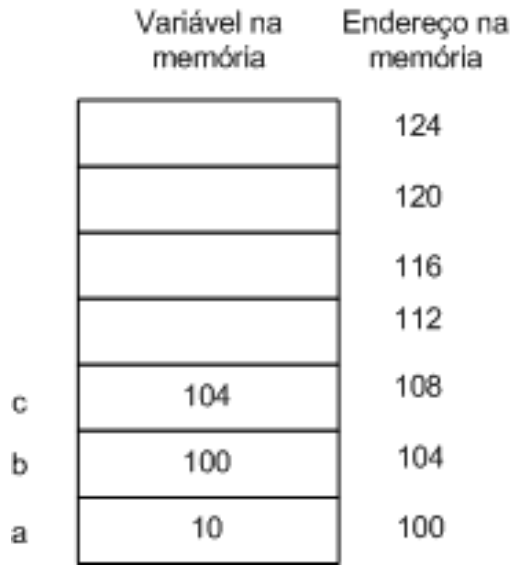


Figura 4: Efeito da atribuição de variáveis na pilha de execução

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 implica 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 implica na movimentação do mesmo para o próximo tipo apontado.
  - Se **pa** é um ponteiro para inteiro tem no seu conteúdo o valor 200 (um endereço), depois de executada a instrução, **pa++**, o valor de **pa** será o endereço 204 (compiladores onde um inteiro é 4 bytes) e não 201.

Operações com Ponteiros

- **Incremento/Decremento**:
  - Podemos incrementar um ponteiro através da adição regular ou pelo operador de incremento (++).
  - Incrementar um ponteiro implica na movimentação do mesmo para o próximo tipo apontado.
  - Se **pa** é um ponteiro para inteiro tem no seu conteúdo o valor 200 (um endereço), depois de executada a instrução, **pa++**, o valor de **pa** será o endereço 204 (compiladores onde um inteiro é 4 bytes) e não 201.
  - Logo, 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 implica na movimentação do mesmo para o próximo tipo apontado.
  - Se **pa** é um ponteiro para inteiro tem no seu conteúdo o valor 200 (um endereço), depois de executada a instrução, **pa++**, o valor de **pa** será o endereço 204 (compiladores onde um inteiro é 4 bytes) e não 201.
  - Logo, 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 implica na movimentação do mesmo para o próximo tipo apontado.
  - Se **pa** é um ponteiro para inteiro tem no seu conteúdo o valor 200 (um endereço), depois de executada a instrução, **pa++**, o valor de **pa** será o endereço 204 (compiladores onde um inteiro é 4 bytes) e não 201.
  - Logo, 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 implica na movimentação do mesmo para o próximo tipo apontado.
  - Se **pa** é um ponteiro para inteiro tem no seu conteúdo o valor 200 (um endereço), depois de executada a instrução, **pa++**, o valor de **pa** será o endereço 204 (compiladores onde um inteiro é 4 bytes) e não 201.
  - Logo, 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 × 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

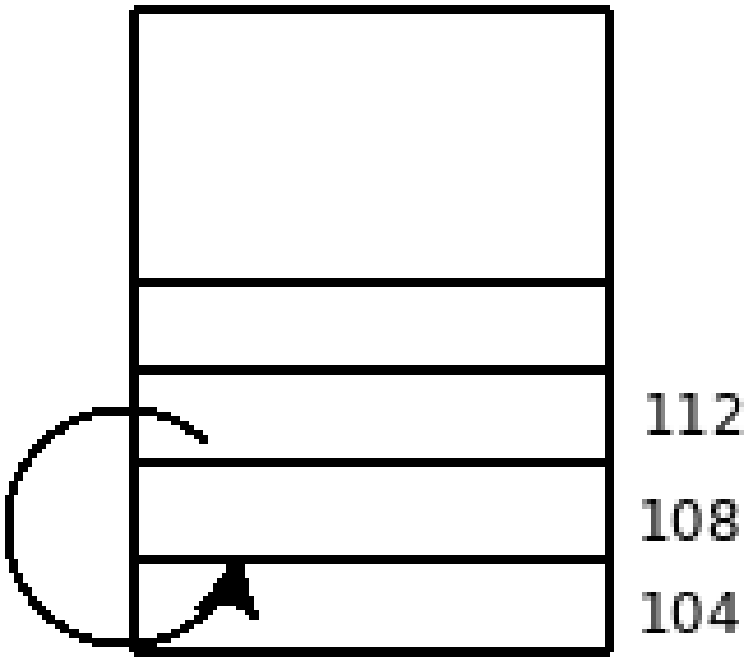


Figura 5: Representação gráfica do valor de um ponteiro

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

```
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 /*Permuta x e y*/
12 void troca(int *x, int *y) {
13   int tmp = *x;
14   *x = *y;
15   *y = tmp;
16 }
```

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.

Onde estamos ...

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];
    // valido para char apenas
    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];
    // valido para char apenas
    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 main(void)
{
        float matriz [50][50];
    float *pt_float;
    int count;
    pt_float = matriz[0]; // OU &matriz[0];
    // mas nao pt_float = matriz;
    for (count=0;count < 2500 ; count++)
    {
     *pt_float = 0.0;
      pt_float ++;
    }
   printf("\n ... Acabou ....\n");
   return 1;
}
```

Exemplos de programas com matrizes

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

int main(void)
{
        float matriz [50][50];
    float *pt_float;
    int count;
    pt_float = matriz[0]; // OU &matriz[0];
    // mas nao pt_float = matriz;
    for (count=0;count < 2500 ; count++)
    {
     *pt_float = 0.0;
      pt_float ++;
    }
   printf("\n ... Acabou ....\n");
   return 1;
}
```

Pergunta de aluno do laboratório

Exemplos de programas com matrizes

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

int main(void)
{       int matriz [][3] =  {{9, 8, 7},
                            {99, 88, 77},
                            {1, 2, 3}};
    int *pt_int;
    int count;
    pt_int = &matriz [0][0]; // tem que indicar
                             // a celula
    for (count=0;count<9;count++)
    {
     printf("%d : ", *pt_int );
     pt_int ++;
    }
   printf("\n ... Acabou ....\n");
   return 1;
}
```

Dúvida de aula ... qual a saída do código acima?

Exemplos de programas com matrizes

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

int main(void)
{       int matriz [][3] =  {{9, 8, 7},
                            {99, 88, 77},
                            {1, 2, 3}};
    int *pt_int;
    int count;
    pt_int = &matriz [0][0]; // tem que indicar
                             // a celula
    for (count=0;count<9;count++)
    {
     printf("%d : ", *pt_int );
     pt_int ++;
    }
   printf("\n ... Acabou ....\n");
   return 1;
}
```

Dúvida de aula ... qual a saída do código acima?

```
[ccs@gerzat ponteiros]$ ./a.out
9 8 7 99 88 77 1 2 3
 ... Acabou ....
```

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.

Cuidados

```
int vetor[10];
int *ponteiro, i;
ponteiro = &i;

/* as operacoes a seguir sao invalidas */
vetor = vetor + 2; /* ERRADO: vetor nao eh variavel */
vetor++;           /* ERRADO: vetor nao eh variavel */
vetor = ponteiro;  /* ERRADO: vetor nao eh variavel */

/* as operacoes abaixo sao validas */

ponteiro = vetor;   /* CERTO: ponteiro eh variavel */
ponteiro = vetor+2; /* CERTO: ponteiro eh variavel */
```

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

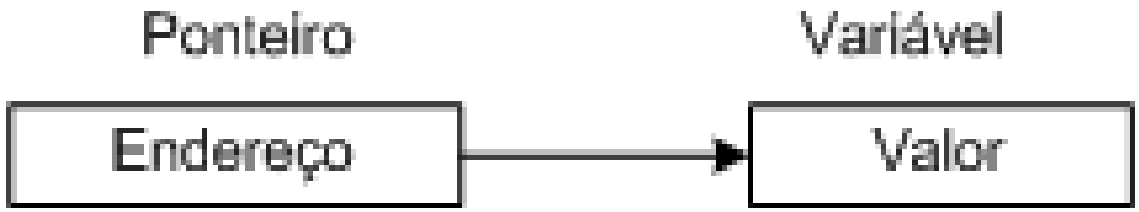


Figura 6: Indireção simples

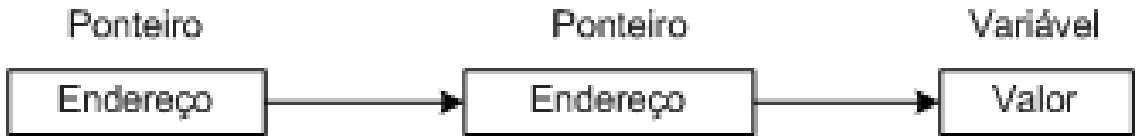


Figura 7: Indireção múltipla

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 }
```

Onde estamos ...

Ponteiros no Retorno de Funções

- Este caso está presente em quase todas funções da linguagem C

Ponteiros no Retorno de Funções

- Este caso está presente em quase todas funções da linguagem C
- Como ponteiros e vetores são praticamente a mesma coisa, a existência de funções que retornam um ponteiro, é presente na linguagem C

Ponteiros no Retorno de Funções

- Este caso está presente em quase todas funções da linguagem C
- Como ponteiros e vetores são praticamente a mesma coisa, a existência de funções que retornam um ponteiro, é presente na linguagem C
- Sim, pois ai não a cópia de valor no retorno, e sim o ponteiro que indica tal endereço de início

Ponteiros no Retorno de Funções

- Este caso está presente em quase todas funções da linguagem C
- Como ponteiros e vetores são praticamente a mesma coisa, a existência de funções que retornam um ponteiro, é presente na linguagem C
- Sim, pois ai não a cópia de valor no retorno, e sim o ponteiro que indica tal endereço de início
- Funções de vetores de carácteres: `strcmp`, etc

Exemplo I

```c
#include <stdlib.h>
#include <stdio.h>
// funcao que  retorna via ponteiro
float * f_media(int , int);
int main(void)
{
    float resultado;
    float *pt_float;

    pt_float = f_media(3,4);
    resultado = * pt_float; // apenas lendo
    // onde o ponteiro esta apontando : a funcao
    // system("clear"); ... cuidar ... causou danos ...
    printf("VALORES: %6.2f : %6.2f", *pt_float, resultado );
    printf("\nENDERECOS: %x : %x : %x", pt_float, &pt_float,
                                        &resultado );
    printf("\nENDERECOS: %x ", & f_media );
    printf("\n ... Acabou ....\n");
    return 1;
}
```

Exemplo II

```
21
22 //uma atribuicao de conteudo entre ponteiros: float
23 float * f_media(int a,  int b)
24  {
25    float res = (a+b)*0.5; // media 2 valores
26    float *pt_aux = & res;
27    printf("\nNA FUNCAO: %x : %x\n", pt_aux, &pt_aux );
28    return pt_aux;
29  }
```

Onde estamos ...

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:

```
1 int main(){
2   int (*func)(const char*, ...);
3   func = printf;
4   (*func)("%d\n", 1);
5 }
```

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:

```
1 int main(){
2   int (*func)(const char*, ...);
3   func = printf;
4   (*func)("%d\n", 1);
5 }
```

- No código acima, a instrução

```
1  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:

```
1  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.
- Resumindo: `func` pode funcionar como um sinônimo para `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.
- 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.
- Resumindo: `func` pode funcionar como um sinônimo para `printf`
- 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

- 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.
- Um primeiro passo para o conceito de *funções genéricas* (ora *anônimas*)

Ponteiro para Funções

```
1 /**
2  * Ordena o vetor v de tamanho n, utilizando
3  * o algoritmo de ordenacao implementado pela
4  * funcao: algOrdenacao.
5  */
6 void ordenar(int v[], int n,
7    void (*algOrdenacao)(int v[], int n)){
8    (*algOrdenacao)(v,n);
9 }
```

- **O exemplo acima é um caso clássico discutido na literatura, leia sobre ele!**
- Vamos há um exemplo original e completo para fixarmos o aprendizado

Exemplo completo: ponteiro para funções I

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

// funcoes a serem apontada via ponteiros
float  f_div(int a,  int b);
int    f_dist(char a,  char b);

int main(void)
{
     char a = 'a' , z = 'z';
     int  x = 3 , y = 4 ;
     // cria ponteiros para funcoes ... veja tipagem
     float (* pt_divisao) (int, int);
     int   (* pt_distancia) (char, char) ;
     /* inicializa ponteiros as funcoes */
     pt_divisao = f_div ;
     pt_distancia = f_dist; /* inicializa ponteiro */

     // chamando as funcoes funcao ... retornando algo
     float R_1 = (*pt_divisao) (x, y);   /* invoca funcao */
```

Exemplo completo: ponteiro para funções II

```
21      int R_2 = (*pt_distancia) (a, z);  /* invoca funcao */
22
23      system("clear"); // OK ... mas cuidar
24
25     printf("SAIDAS: %6.2f : %d", R_1 , R_2 );
26     printf("\nENDERECOS: %x = %x ", pt_divisao, f_div );
27     printf("\nENDERECOS: %x : %x ", &pt_divisao, &f_div );
28     printf("\nENDERECOS: %x = %x ", pt_distancia, f_dist );
29     printf("\nENDERECOS: %x : %x ", &pt_distancia, &f_dist );
30     printf("\n ... Acabou ....\n");
31    return 1;
32 }
33 // AS FUNCOES
34 float f_div(int a,  int b)
35  { //printf("\n %d .. %d \n", a , b);
36    float resp = ((float)a) / b;
37    return ( resp );
38  }
39
40 int f_dist(char a,  char b)
41  {
```

Exemplo completo: ponteiro para funções III

```
42    return (b - a);
43  }
```

Quanto há uma saída:

Uma saída:

SAIDAS: 0.75 : 25
ENDERECOS: 632178e3 = 632178e3
ENDERECOS: 9f06eac8 : 632178e3
ENDERECOS: 6321792e = 6321792e
ENDERECOS: 9f06ead0 : 6321792e
... Acabou ....

Ponteiro para Funções

- Em resumo podemos:
  1. Passar endereços de variáveis como parâmetros
  2. Passar ponteiros como parâmetros
  3. Retornar um ponteiro de uma função
  4. Declarar um ponteiro para uma função
  5. Atribuir o endereço de uma função a um ponteiro
  6. 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.

Onde estamos ...

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?

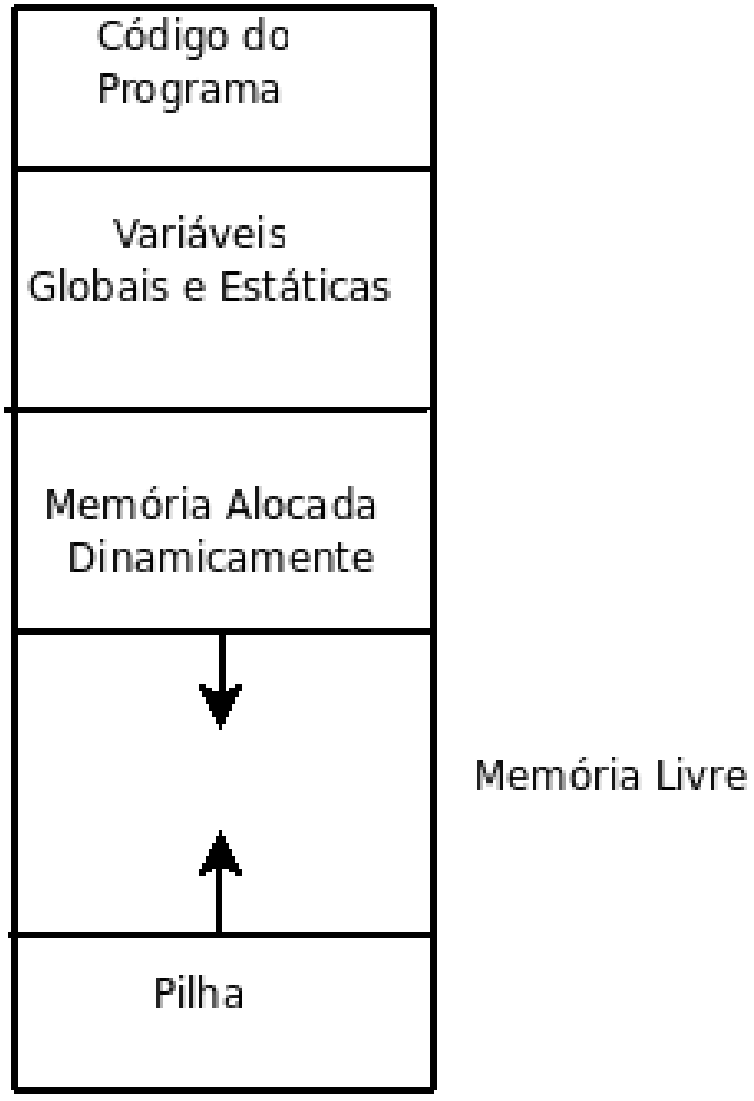
Alocação Dinâmica

- 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.

Onde estamos ...

Exemplo 01 completo: AD – *malloc* I

```
1  /*
2  malloc() : Allocates requested size of bytes and returns a pointer fir
3  calloc() : Allocates space for an array elements, initializes to zero
4             (the block initializes the allocates memory to all bits zer
5  free(): deallocate the previously allocated space
6  realloc() : Change the size of previously allocated space
7   */
8  #include <stdio.h>
9  #include <stdlib.h>
10
11 int main()
12 {
13     int num, i, *ptr, sum = 0;
14
15     printf("Entre numero de elementos: ");
16     scanf("%d", &num);
17
18     ptr = (int*) malloc(num * sizeof(int));
19     //memoria alocada usando malloc
20     if(ptr == NULL)
```

Exemplo 01 completo: AD – *malloc* II

```
21      {
22          printf("\n Erro! Memoria NAO alocada.\n");
23          exit(0);
24      }
25
26      printf("Lendo o vetor de elementos: ");
27      for(i = 0; i < num; ++i)
28      {
29          scanf("%d", ptr + i);
30          sum += *(ptr + i);
31      }
32
33      printf("\nSOMA FINAL = %d\n", sum);
34      free(ptr);
35      return 0;
36  }
```

Uma saída:

- Laboratório ⇒ voce
- Para o exemplo acima, gere um arquivo texto com 1000 inteiros, por exemplo
- Modifique o exemplo do professor, digamos, calcule a média no lugar de uma simples soma
- Crie um ponteiro para função média
- Confira onde foram alocadas estas variáveis
- Verifique os resultados
- Valide o seu aprendizado com o texto acima

Exemplo 01 completo: AD – *calloc* I

```c
/*
calloc() : Allocates space for an array elements, initializes to zero
*/

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

int main()
{
    int num, i, *ptr, sum = 0;

    printf("Entre numero de elementos: ");
    scanf("%d", &num);
    // AQUI EH UM VETOR DE N POSICOES  ... pequena diferenca
    ptr = (int*) calloc(num , sizeof(int));

    if(ptr == NULL)
    {
        printf("\n Erro! Memoria NAO alocada.\n");
        exit(0);
```

Exemplo 01 completo: AD – *calloc* II

```
21      }
22
23      printf("Lendo o vetor de elementos: ");
24      for(i = 0; i < num; ++i)
25      {
26          scanf("%d", ptr + i);
27          sum += *(ptr + i);
28      }
29
30      printf("\nSOMA FINAL = %d\n", sum);
31      free(ptr);
32      return 0;
33  }
```

Uma saída:

- Laboratório ⇒ voce
- Para o exemplo acima, gere um arquivo texto com 1000 inteiros, por exemplo
- Modifique o exemplo do professor, digamos, calcule a média no lugar de uma simples soma
- Crie um ponteiro para função média
- Confira onde foram alocadas estas variáveis
- Verifique os resultados
- Valide o seu aprendizado com o texto acima

Exemplo 01 completo: AD – *realloc* I

```c
/*
realloc() : Change the size of previously allocated space
 */
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int *ptr, i , n1, n2;
    printf("\n Entre TAM do array -> N1: ");
    scanf("%d", &n1);
    // Aloca sequencialmente ....
    ptr = (int*) malloc(n1 * sizeof(int));

    printf("Enderecos alocados na memoria: ");
    for(i = 0; i < n1; ++i)
        printf("\n %u\t %d ",ptr + i, *(ptr + i));

    printf("\nUM NOVO (N2) TAMANHO DE ARRAY (>,=,<): ");
    scanf("%d", &n2);
```

Exemplo 01 completo: AD – *realloc* II

```
21      // Aloca, reaproveitando o inicio do anterior n1
22      ptr = (int*) realloc(ptr, n2); // cuidar com cast (int*)
23      for(i = 0; i < n2; ++i)
24         printf("\n %u\t %d ",ptr + i, *(ptr + i));
25
26      printf("\n N1: %d\t N2: %d\n", n1, n2);
27
28      return 0;
29  }
```

Uma saída:

- Laboratório ⇒ voce
- Para o exemplo acima, gere um arquivo texto com 1000 inteiros, por exemplo
- Modifique o exemplo do professor, digamos, calcule a média no lugar de uma simples soma
- Crie um ponteiro para função média
- Confira onde foram alocadas estas variáveis
- Verifique os resultados
- Valide o seu aprendizado com o texto acima

Onde estamos ...

Capítulo 02 – Pilha

Pontos fundamentais a serem cobertos:

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

Introdução

- Uma das estruturas de dados mais simples
- Embora seja uma das estrutura de dados mais utilizadas em programação
- A pilha se *fortalece* quando combinada dentro de outras estruturas
  - Uso de pilhas na sequência de visita a nós de uma árvore
  - Dentro de outras estruturas como filas (depois)
- Há uma metáfora emprestada do mundo real, que a computação utiliza pilhas para resolver muitos problemas de forma simplificada.

Aplicações

Alguns exercícios são clássicos (e devemos implementá-los):

- Balanceamento de símbolos. Exemplo: `([aaa])`
- Conversão da notação infixa para pós-fixa
- Conversão da notação infixa para in-fixa
- Avaliação de uma expressão pós-fixa. Exemplo: `2 3 +`
- Implementações de chamadas de funções (inclusive as chamadas recursivas de funções)
- Armazenamento de páginas visitadas no navegador em uma dada janela (botão **back**)
- Sequência de comandos de um editor de texto, e depois aplique o *undo*, ou `crtl-z`
- Casamento de *tags* in HTML e XML
- As teclas ↑ e ↓ na console ou terminal do Linux, duas pilhas neste caso!

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 (*clássico*)
  - 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.)

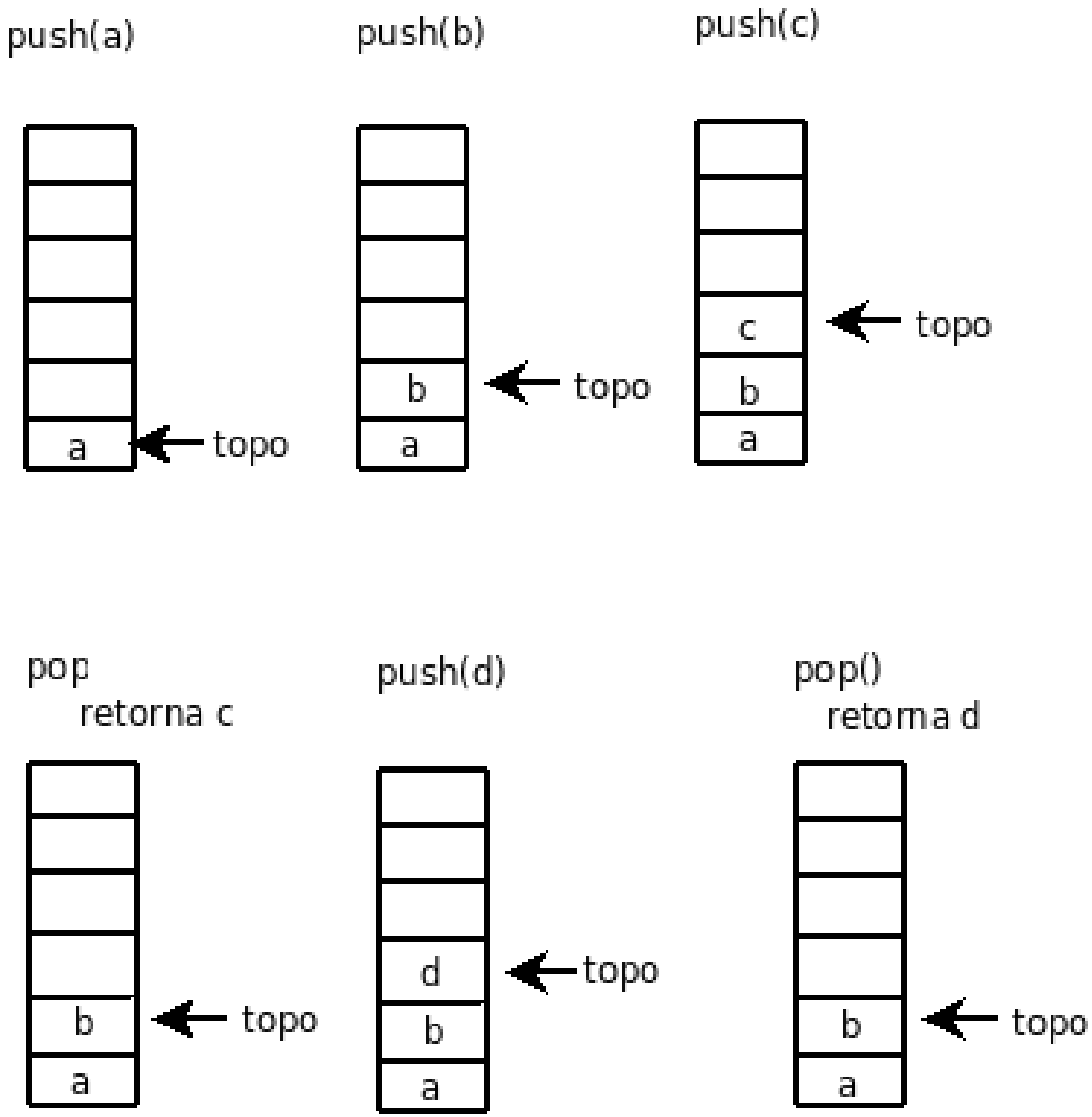
Operações básicas

- 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 1: 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()
retorna 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 2: Operações auxiliares da estrutura de dados pilha.

Interface do Tipo Pilha – Típica

```
1  /* Definicao da estrutura */
2  typedef struct { DEFINA O SEU MODELO AQUI }  Pilha;
3
4  /*Aloca dinamicamente ou estaticamente a estrutura pilha,
5    inicializando seus campos e retorna seu ponteiro.*/
6  Pilha * create(void);
7
8  /*Insere o elemento e na pilha p.*/
9  void push(Pilha *p, int e);
10
11 /*Retira e retorna o elemento do topo da pilha p*/
12 int pop(Pilha *p);
13
14 /*Informa se a pilha p esta ou nao vazia.*/
15 int empty(Pilha *p);
```

Implementações

1. Baseada em um simples vetor
2. Baseada em um vetor dinâmico
3. Baseada em lista encadeada

Implementações

1. Baseada em um simples vetor
2. Baseada em um vetor dinâmico
3. Baseada em lista encadeada
4. mas todas usam ponteiros!

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

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

(recupera valor do topo e depois decrementa)

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

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

(soma antes e depois insere)

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 }
```

Exemplos 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.

Exercícios

1. Os exercícios propostos no início deste capítulo (slide inicial)
2. Escreva uma função que inverta a ordem das letras de cada palavra de uma sentença, preservando a ordem das palavras. Suponha que as palavras da sentença são separadas por espaços. A aplicação da operação à sentença **AMU MEGASNEM ATERCES**, por exemplo, deve produzir **UMA MENSAGEM SECRETA**.
3. Implemente uma função que receba uma pilha como parâmetro e retorne o valor armazenado em seu topo, restaurando o conteúdo da pilha. Essa função deve obedecer ao protótipo:

```
char topo(Pilha* p);
```

4. Implemente uma função que receba duas pilhas, p_1, p_2, e passe todos os elementos da pilha p_2 para o topo da pilha p_1. Essa função deve obedecer ao protótipo:

```
void concatena(Pilha* p1, Pilha* p2);
```

Onde estamos ...

Capítulo 03 – Filas

Pontos fundamentais a serem cobertos:

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

Introdução

- Assim como a estrutura de dados Pilha, Fila é outra estrutura de dados bastante utilizada em computação.
- Um exemplo é a implementação de uma fila de impressão.
- Se uma impressora é compartilhada por várias máquinas, normalmente adota-se uma estratégia para determinar a ordem de impressão dos documentos.
- A maneira mais simples é tratar todas as requisições com a mesma prioridade e imprimir os documentos na ordem em que foram submetidos – o primeiro submetido é o primeiro a ser impresso.

Fila

Aplicações

1. Escalonamento de processos na CPU (processos com a mesma prioridade) os quais são executados em ordem de chegada
2. Simulação de filas no mundo real tais como: filas em banco, compra de tickets, etc
3. Multi-programação
4. Transferência assíncrona de dados (IO de arquivos, pipe, sockets)
5. Fila de espera em um *call-center*
6. Encontrar número de atendentes em supermercados e caixas bancários dado uma demanda de pessoas na sala de espera

Aplicações Indiretas

1. Estrutura de dados auxiliares em algoritmos
2. Componente de outras estruturas

Fila

Definição

Um conjunto ordenado de itens a partir do qual podem-se eliminar itens numa extremidade (chamada de início da fila) e no qual podem-se inserir itens na outra extremidade (chamada final da fila).

Representação

- Os nós de uma fila são armazenados em endereços contínuos.
- A Figura 8 ilustra uma fila com três elementos.

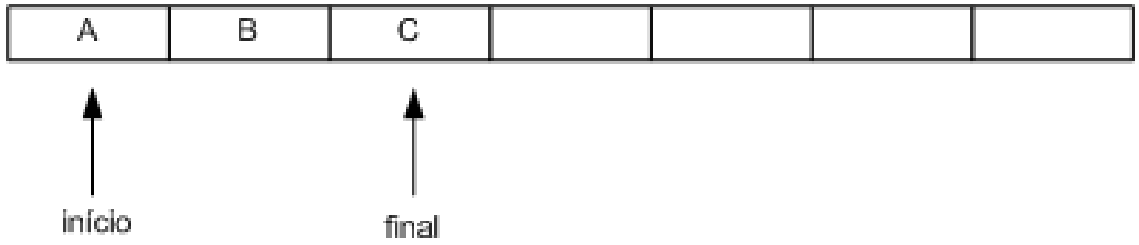


Figura 8: Exemplo de representação de fila.

- Após a retirada de um elemento (*primeiro*) temos:

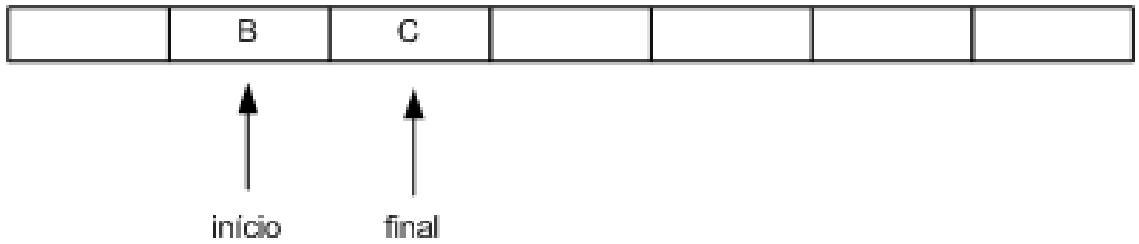


Figura 9: Representação de uma fila após a remoção do elemento "A".

Implementações

Quais as estruturas usadas por filas?

- Baseada num vetor simples – limitada
- Baseada num vetor circular simples – limitada
- Baseada num vetor circular dinâmico – **i**limitada
- Baseada numa lista encadeada (depois de listas)

Representação

- Após a inclusão de dois elementos temos:

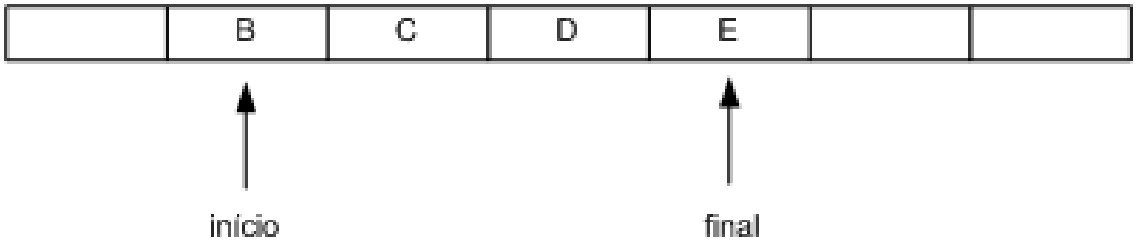


Figura 10: Representação de uma fila após a inclusão de dois elementos "D" e "E".

- Como podemos observar, a operação de inclusão e retirada de um item da fila incorre na mudança do endereço do ponteiro que informa onde é o início e o término da fila.

Representação

- Em uma fila, o primeiro elemento inserido é o primeiro a ser removido.
- Por essa razão, uma fila é chamada **fifo(*first-in first-out*)** – primeiro que entra é o primeiro a sair – ao contrário de uma pilha que é lifo (*last-in, first-out*)
- Para exemplificar a implementação em C, vamos considerar que o conteúdo armazenado na fila é do tipo inteiro.
- A estrutura de fila possui a seguinte representação:

```
1  struct fila{
2    int elemento[N];
3    int ini, n;
4  }
5  typedef struct fila Fila;
```

- Trata-se de uma estrutura heterogênea constituída de membros distintos entre si. Os membros são as variáveis *ini* e *fim*, que serve para armazenar respectivamente, o início e o fim da fila e o vetor *elemento* de inteiros que armazena os itens da fila.

Operações Primitivas

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

Operação	**Descrição**
criar()	aloca dinamicamente a estrutura da fila.
insere(f,e)	adiciona um novo elemento *(e)*, no final da fila *f*.
retira(f)	remove o elemento do início da fila *f*.

Tabela 3: Operações básicas da estrutura de dados fila.

Operações auxiliares

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

Operação	**Descrição**
vazia(f)	informa se a fila está ou não vazia.
libera(f)	destrói a estrutura, e assim libera toda a memória alocada.

Tabela 4: Operações auxiliares da estrutura de dados fila.

Interface do Tipo Fila

```c
typedef struct fila Fila;
/* Aloca dinamicamente a estrutura Fila, inicializando seus
 * campos e retorna seu ponteiro. A fila depois de criada
 * estarah vazia.*/
Fila* criar(void);

/* Insere o elemento e no final da fila f, desde que,
 * a fila nao esteja cheia.*/
void insere(Fila* f, int e);

/* Retira o elemento do inicio da fila, e fornece o
 * valor do elemento retirado como retorno, desde que a fila
 * nao esteja vazia*/
int retira(Fila* f);

/*Verifica se a fila f estah vazia*/
int vazia(Fila* f);

/*Libera a memoria alocada pela fila f*/
void libera(Fila* f);
```

Implementação de Fila com Vetor

- Assim como nos casos da pilha e lista, a implementação de fila será feita usando um vetor para armazenar os elementos.
- Isso implica, que devemos fixar o número máximo de elementos na fila.
- O processo de inserção e remoção em extremidades opostas fará a fila "andar" no vetor.
- Por exemplo, se inserirmos os elementos 8, 7, 4, 3 e depois retiramos dois elementos, a fila não estará mais nas posições iniciais do vetor.

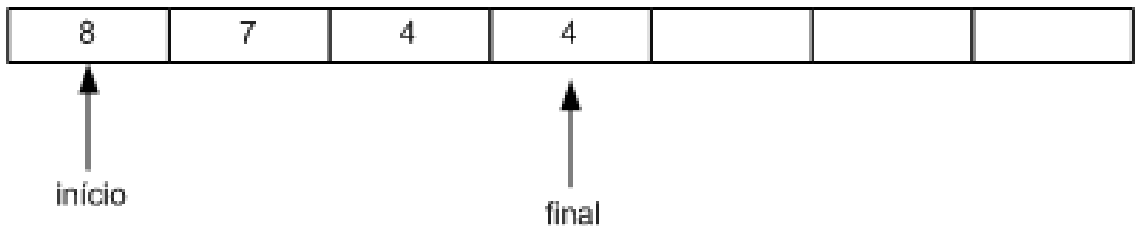


Figura 11: Fila após inserção de quatro elementos.

Implementação de Fila com Vetor

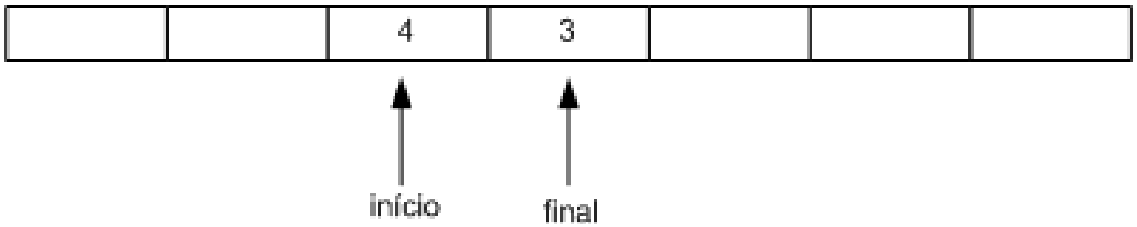


Figura 12: Fila após retirar dois elementos.

- Com essa estratégia, é fácil observar que, em um dado instante, a parte ocupada pelo vetor pode chegar a última posição.
- Uma solução seria ao remover um elemento da fila, deslocar a fila inteira no sentido do início do vetor.
- Entretanto, essa método é bastante ineficiente, pois cada retirada implica em deslocar cada elemento restante da fila. Se uma fila tiver 500 ou 1000 elementos, evidentemente esse seria um preço muito alto a pagar.

Implementação de Fila com Vetor

- Para reaproveitar as primeira posições do vetor sem implementar uma "re-arrumação" dos elementos, podemos incrementar as posições do vetor de forma "circular".
- Para essa implementação, os índices do vetor são incrementados de maneira que seus valores progridam "circularmente".
- Dessa forma, se temos 100 posições no vetor, os índices assumem os seguintes valores:

$$0, 1, 2, 3, \cdots, 98, 99, 0, 1, 2, 3, \cdots, 98, 99, \cdots$$

Porquê um *vetor circular*?

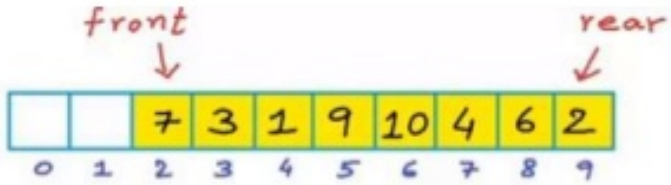
Reflexões

- Tamanho fixo – muito interessante para maioria dos problemas
- Velocidade
- Fácil de manipular
- *Circular* é a manipulação!

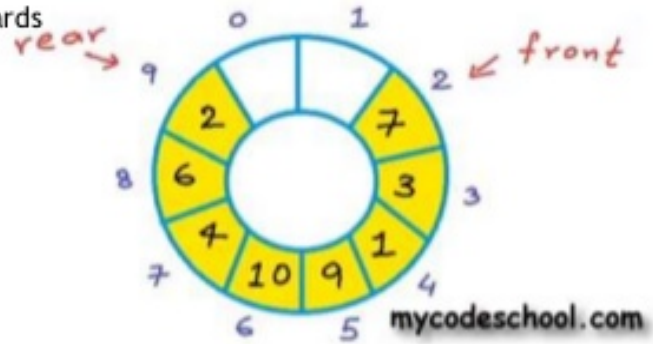
Vetor Circular: Explicações

Queue: implementation Cyclic Array

no end on array
wrapping around
ring buffer / circular buffer
push/pop - O(1)



when item inserted to rair, tails's pointer moves upwards
when item deleted, head's pointer moves downwards



current_position = i
next_position = (i + 1) % N
prev_position = (N + i - 1) % N

19

Vetor Circular: Explicações

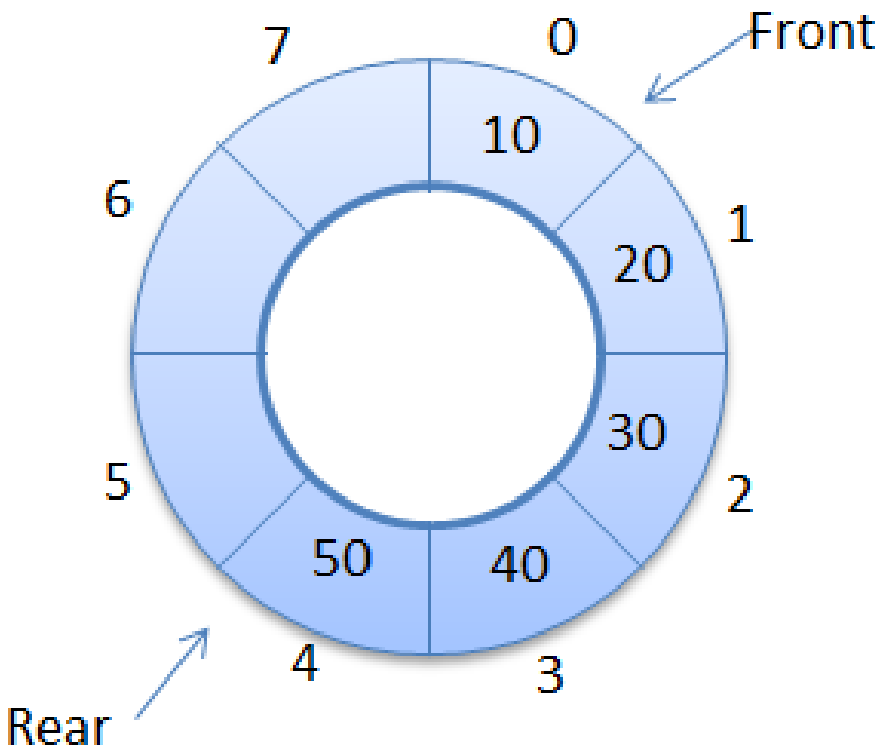
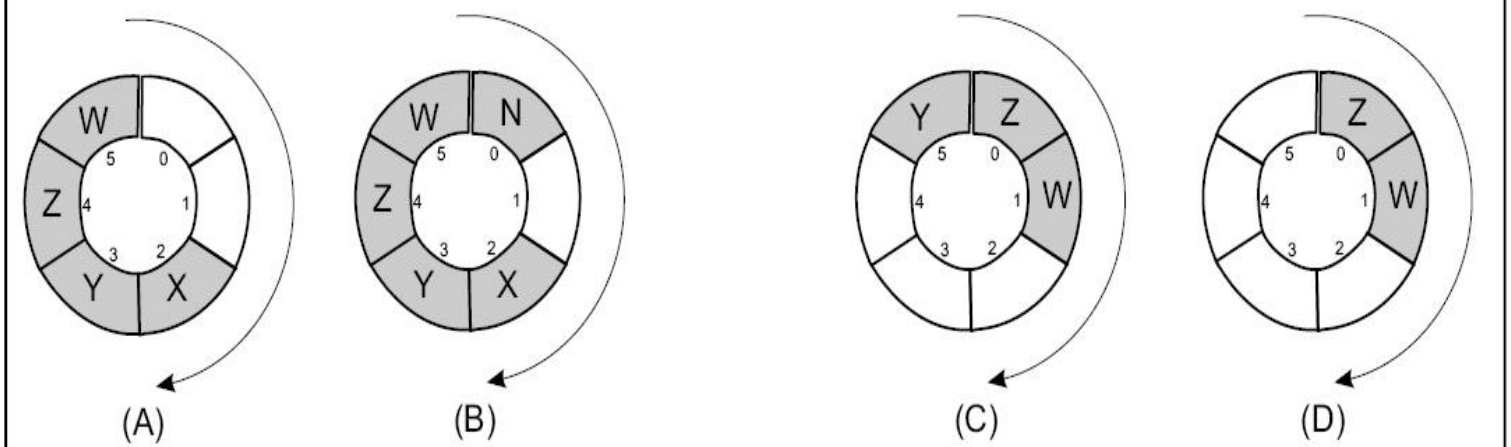


Figura 13: Notação aqui empregada

Vetor Circular: Explicações

	Frente	Cauda	Vetor 0	1	2	3	4	5	Operação
A)	2	5			X	Y	Z	W	Insere N
B)	2	0	N		X	Y	Z	W	

	Frente	Cauda	Vetor 0	1	2	3	4	5	Operação
C)	5	1	Z	W				Y	Retira da Fila
D)	0	1	Z	W					



Função de Criação

- A função que cria uma fila, deve criar e retornar o ponteiro de uma fila vazia
- A função deve informar onde é o início da fila, ou seja, fazer `f->ini = 0`, como podemos ver no código abaixo
- A complexidade de tempo para criar a fila é constante, ou seja, $O(1)$

```
1  /* Aloca dinamicamente a estrutura Fila, inicializando seus
2   * campos e retorna seu ponteiro. A fila depois de criada
3   * estarah vazia.
4   */
5  Fila* criar(void)
6  {
7      Fila* f = malloc(sizeof(Fila));
8      f->n = 0; // quantidade CORRENTE elementos
9      f->ini = 0; // inicio
10     return f;
11 }
```

Função de Inserção

- Para inserir um elemento na fila, usamos a próxima posição livre do vetor, indicada por n.
- Devemos assegurar que há espaço para inserção do novo elemento no vetor, haja vista se tratar de um vetor com capacidade limitada.
- A complexidade de tempo para inserir um elemento na fila é constante, ou seja, $O(1)$.

```
1  /* Insere o elemento e no final da fila f.*/
2  void insere(Fila* f, int e)
3  {
4    int fim;
5    if (f->n == N){
6       printf("Fila cheia!\n"); }
7    else{
8       fim = (f->ini + f->n) % N;
9       f->elementos[fim] = e;
10      f->n++;
11   }
12 }
```

Função de Remoção

- A função para retirar o elemento do início da fila fornece o valor do elemento retirado como retorno.
- Para remover um elemento, devemos verificar se a fila está ou não vazia.
- A complexidade de tempo para remover um elemento da fila é constante, ou seja, $O(1)$.

```
1  int retira(Fila* f)
2  {
3    int e;
4    if ( vazia(f) )
5      printf("Fila vazia!\n");
6    else{
7      e = f->elementos[f->ini];
8      f->ini = (f->ini + 1) % N;
9      f->n--;
10   }
11   return e;
12 }
```

Exemplo de Uso da Fila

```c
#define N 10
#include <stdio.h>
#include "fila.h"  // TDA.h

int main(void)
{
    Fila * f = criar();

    int i;
    for (i = 0; i < N; i++)
      insere(f, i * 2);

    printf("\nElementos removidos: ");

    for (i = 0; i < N/2; i++)
     printf("%d ", retira(f));

}
```

Exercícios

1.
2.
3.

Referências

1. Tenenbaum, A. M., Langsam, Y., and Augestein, M. J. (1995). *Estruturas de Dados Usando C*. MAKRON Books, pp. 207-250.
2. Wirth, N. (1989). *Algoritmos e Estrutura de Dados*. LTC, pp. 151-165.