

Estrutura de Dados

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

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

12 de agosto de 2017

Sumário I

- 1 O Curso
 - Ferramentas
 - Metodologia e avaliação
 - Dinâmica
 - Referências
- 2 Ponteiros
 - Motivação aos Ponteiros
- 3 Ponteiros e Matrizes
 - Indireção Múltipla
- 4 Ponteiro para Funções
- 5 Alocação dinâmica

- 6 Pilha
 - Introdução

Agradecimentos

Vários autores e colaboradores ...

- Ao Google Images ...

Onde estamos ...

1 O Curso

- Ferramentas
- Metodologia e avaliação
- Dinâmica
- Referências

2 Ponteiros

- Motivação aos Ponteiros

3 Ponteiros e Matrizes

- Indireção Múltipla

4 Ponteiro para Funções

5 Alocação dinâmica

6 Pilha

- Introdução

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

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

Objetivos I

- *Geral:*

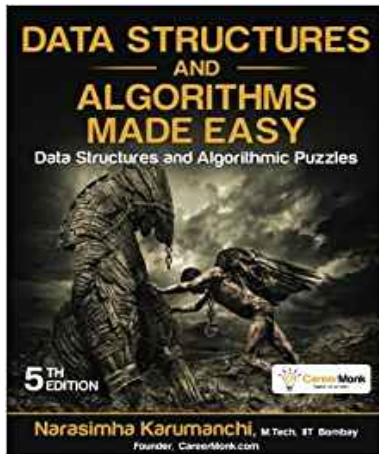
Há um documento específico sobre isto = Plano de Ensino

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

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:

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

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 I

- 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 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!

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 eletrônicos, web, smartphones, etc

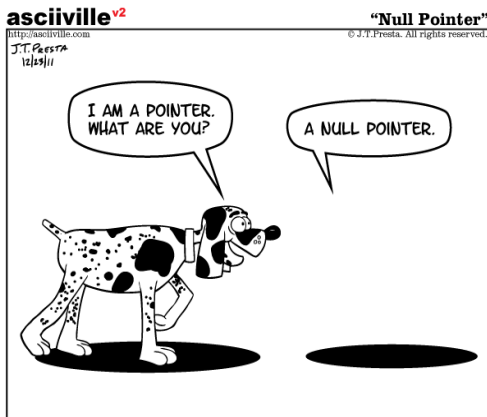
Onde estamos ...

- 1 O Curso
 - Ferramentas
 - Metodologia e avaliação
 - Dinâmica
 - Referências
- 2 **Ponteiros**
 - Motivação aos Ponteiros
- 3 Ponteiros e Matrizes
 - Indireção Múltipla
- 4 Ponteiro para Funções
- 5 Alocação dinâmica
- 6 Pilha
 - Introdução

Capítulo 01 – Ponteiros I

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

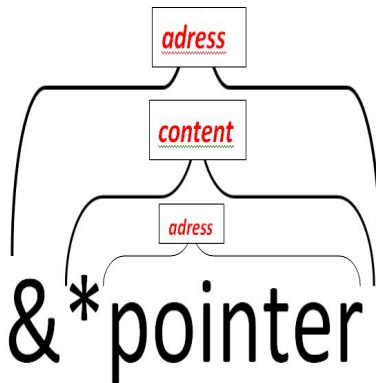


Figura 1: A história está por vir ...

Exemplos Lúdicos

```
1 #include <stdio.h>
2 int main()
3 {
4     int a;
5     int *ptr;    // declara um ponteiro -- ptr-- para um inteiro
6                 // um ponteiro para uma variavel do tipo inteiro
7     a = 90;
8     ptr = &a;
9     printf("Valor de A: %d\n", a);
10    printf("Valor de ptr: %d \t Conteudo via ptr: %d\n", ptr, *ptr);
11    return 1;
12 }
```

Exemplos Lúdicos

```
1 #include <stdio.h>
2 int main()
3 {
4     int a;
5     int *ptr;    // declara um ponteiro -- ptr-- para um inteiro
6                 // um ponteiro para uma variavel do tipo inteiro
7     a = 90;
8     ptr = &a;
9     printf("Valor de A: %d\n", a);
10    printf("Valor de ptr: %d \t Conteudo via ptr: %d\n", ptr, *ptr);
11    return 1;
12 }
```

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

Exemplos e agora?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5     int a;
6     int *ptr;    // declara um ponteiro -- ptr-- para um inteiro
7                 // um ponteiro para uma variavel do tipo INTEIRO
8     a = 2017;
9     ptr = &a;
10    system("clear");
11    printf("Valor de A: %d\t Endereco de A: %x\n", a, &a);
12    printf("Valor de ptr: %d \t Conteudo via ptr: %d\n", ptr, *ptr);
13    printf("Endereco de PTR: %x\n", & ptr);
14    return 1;
15 }
```

Exemplos e agora?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5     int a;
6     int *ptr;    // declara um ponteiro -- ptr-- para um inteiro
7                 // um ponteiro para uma variavel do tipo INTEIRO
8     a = 2017;
9     ptr = &a;
10    system("clear");
11    printf("Valor de A: %d\t Endereco de A: %x\n", a, &a);
12    printf("Valor de ptr: %d \t Conteudo via ptr: %d\n", ptr, *ptr);
13    printf("Endereco de PTR: %x\n", & ptr);
14    return 1;
15 }
```

Exemplos e agora?

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main()
4 {
5     int a;
6     int *ptr;    // declara um ponteiro -- ptr-- para um inteiro
7                 // um ponteiro para uma variavel do tipo INTEIRO
8     a = 2017;
9     ptr = &a;
10    system("clear");
11    printf("Valor de A: %d\t Endereco de A: %x\n", a, &a);
12    printf("Valor de ptr: %d \t Conteudo via ptr: %d\n", ptr, *ptr);
13    printf("Endereco de PTR: %x\n", & ptr);
14    return 1;
15 }
```

- 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

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

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

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

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

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

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

- Basicamente há três razões para utilizar ponteiros:

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

- 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;

- 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.

- 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 uma 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 uma 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 uma 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.

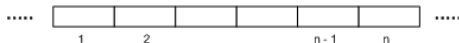


Figura 2: Representação da memória de uma máquina típica

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.

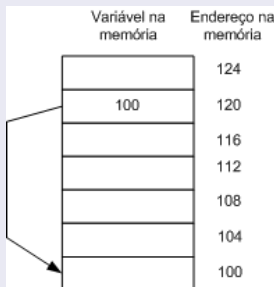


Figura 3: Representação de ponteiro

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

- 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.

- ❶ A linguagem C permite o armazenamento e a manipulação de valores de endereço de memória.
- ❷ 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.
- ❸ 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:

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

- 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 endereço de a (p aponta para a). */  
5  p = &a;  
6  
7  /* conteúdo de p recebe o valor 10 */  
8  *p = 10;  
9
```

Atribuição e acessos de endereço

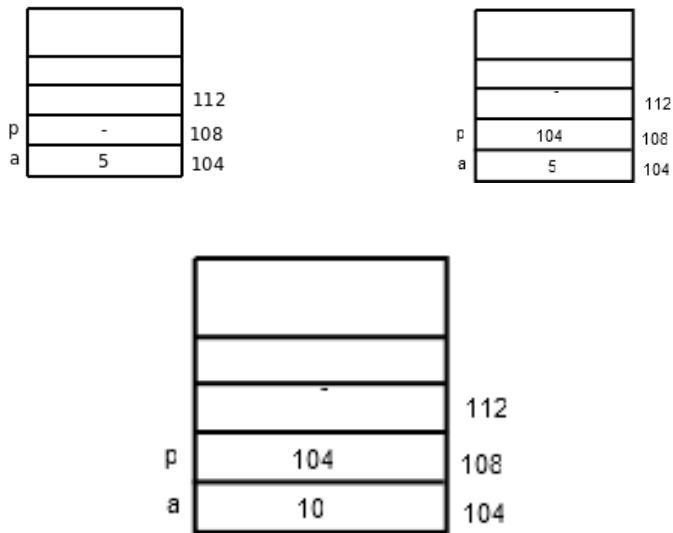


Figura 4: Efeito da atribuição de variáveis na pilha de execução

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
```

	Variável na memória	Endereço na memória
		124
		120
		116
		112
c	104	108
b	100	104
a	10	100

Operações com Ponteiros

- **Incremento/Decremento:**

- **Incremento/Decremento:**

- Podemos incrementar um ponteiro através da adição regular ou pelo operador de incremento (++).

- **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.

- **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 (- -)

- **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

- **Comparações entre Ponteiros:**

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

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

- **Comparações entre Ponteiros:**

- Ponteiros podem ser comparados:

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

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

- **Comparações entre Ponteiros:**

- Ponteiros podem ser comparados:

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

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

- **Comparações entre Ponteiros:**

- Ponteiros podem ser comparados:

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

- Testes relacionais com \geq , \leq , $>$, $<$ 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:
 - ❶ 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:
 - ❶ O nome do ponteiro retorna o endereço para o qual ele aponta.
 - ❷ 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, aí precisamos da chamada por referência.
- Uso de ponteiros

Atribuição e acessos de endereço

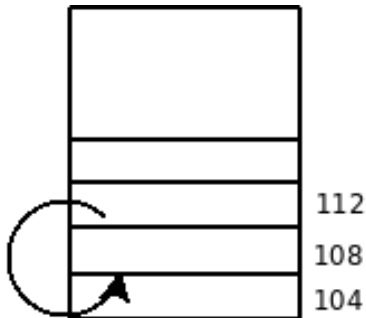


Figura 5: Representação gráfica do valor de um ponteiro

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 }
```

- 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 ...

- 1 O Curso
 - Ferramentas
 - Metodologia e avaliação
 - Dinâmica
 - Referências
- 2 Ponteiros
 - Motivação aos Ponteiros
- 3 Ponteiros e Matrizes
 - Indireção Múltipla
- 4 Ponteiro para Funções
- 5 Alocação dinâmica
- 6 Pilha
 - Introdução

- Em C, há um estreito relacionamento entre 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.

- 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

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int strtam(char * );
5
6 int main(void)
7 {   char vetor[] = "ABCDEFGH" ;
8     // char *lista = "Ola como vai?"; ... HUM ...
9     char *pt;
10    pt = vetor; //mais saudavel => pt = &vetor[0];
11    // valido para char apenas
12    system("clear");
13    printf("O tamanho de \"%s\" e %d caracteres.\n", vetor, strtam( pt
14    printf("\n ... Acabou ....");
15    return 1;
16 }
17
18 int strtam(char *s){
19     int tam=0;
20     //while(*(s + tam++) != '\0');
21     while(*s != '\0')
22     {   tam++;
23         s++;
24     }
25     return tam; //tam-1; --> \0
```


Exemplos de programas com matrizes

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int strtam(char * );
5
6 int main(void)
7 {   char vetor[] = "ABCDEFGH" ;
8     // char *lista = "Ola como vai?"; ... HUM ...
9     char *pt;
10    pt = vetor; //mais saudavel => pt = &vetor[0];
11    // valido para char apenas
12    system("clear");
13    printf("O tamanho de \"%s\" e %d caracteres.\n", vetor, strtam( pt
14    printf("\n ... Acabou ....");
15    return 1;
16 }
17
18 int strtam(char *s){
19     int tam=0;
20     //while(*(s + tam++) != '\0');
21     while(*s != '\0')
22     {   tam++;
23         s++;
24     }
25     return tam; //tam-1; --> \0
```

Exemplos de programas com matrizes

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     float matriz [50][50];
7     float *pt_float;
8     int count;
9     pt_float = matriz[0]; // OU &matriz[0];
10    // mas nao pt_float = matriz;
11    for (count=0; count < 2500 ; count++)
12    {
13        *pt_float = 0.0;
14        pt_float ++;
15    }
16    printf("\n ... Acabou ....\n");
17    return 1;
18 }
```

Exemplos de programas com matrizes

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     float matriz [50][50];
7     float *pt_float;
8     int count;
9     pt_float = matriz[0]; // OU &matriz[0];
10    // mas nao pt_float = matriz;
11    for (count=0; count < 2500 ; count++)
12    {
13        *pt_float = 0.0;
14        pt_float ++;
15    }
16    printf("\n ... Acabou ....\n");
17    return 1;
18 }
```

Pergunta de aluno do laboratório

Exemplos de programas com matrizes

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     int matriz [][3] = {{9, 8, 7},
7                          {99, 88, 77},
8                          {1, 2, 3}};
9
10    int *pt_int;
11    int count;
12    pt_int = &matriz [0][0]; // tem que indicar
13                                // a celula
14    for (count=0; count<9; count++)
15    {
16        printf("%d : ", *pt_int );
17        pt_int ++;
18    }
19    printf("\n ... Acabou ....\n");
20    return 1;
21 }
```

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

Exemplos de programas com matrizes

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int main(void)
5 {
6     int matriz [][3] = {{9, 8, 7},
7                          {99, 88, 77},
8                          {1, 2, 3}};
9
10    int *pt_int;
11    int count;
12    pt_int = &matriz [0][0]; // tem que indicar
13                                // a celula
14    for (count=0; count<9; count++)
15    {
16        printf("%d : ", *pt_int );
17        pt_int ++;
18    }
19    printf("\n ... Acabou ....\n");
20    return 1;
21 }
```

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

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

Exemplo de programas com matrizes

```
1 /*imprime os valores da matriz*/
2 main(){
3     int nums [5] = {100,200,90,20,10};
4     int d;
5     for(d = 0; d < 5; d++)
6         printf("%d\n", nums[d]);
7 }
8
9 /*usa ponteiros para imprimir os valores da matriz*/
10 main(){
11     int nums [5] = {100,200,90,20,10};
12     int d;
13     for(d = 0; d < 5; d++)
14         printf("%d\n", *(nums + d));
15 }
```

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 */
```

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 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.

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.

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

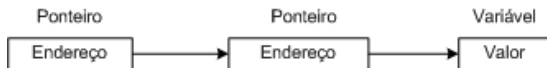


Figura 7: Indireção múltipla

Declaração

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

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;
```

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**).

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 ...

- 1 O Curso
 - Ferramentas
 - Metodologia e avaliação
 - Dinâmica
 - Referências
- 2 Ponteiros
 - Motivação aos Ponteiros
- 3 Ponteiros e Matrizes
 - Indireção Múltipla
- 4 Ponteiro para Funções
- 5 Alocação dinâmica
- 6 Pilha
 - Introdução

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;
```

- 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.

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

- 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 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.

- 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.

- 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

```
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 }
```

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.

Onde estamos ...

- 1 O Curso
 - Ferramentas
 - Metodologia e avaliação
 - Dinâmica
 - Referências
- 2 Ponteiros
 - Motivação aos Ponteiros
- 3 Ponteiros e Matrizes
 - Indireção Múltipla
- 4 Ponteiro para Funções
- 5 Alocação dinâmica**
- 6 Pilha
 - Introdução

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

- 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.

- 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.

- 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.

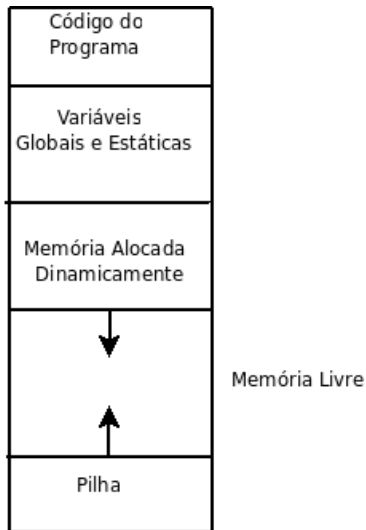
- Na declaração de um vetor visto até agora é necessário conhecer e informar o número de elementos do vetor.
- Esse pré-dimensionamento é um fator limitante, pois, nos obriga a conhecer de antemão a quantidade de elementos do vetor.
- Uma solução é dimensionar o vetor com um número muito alto, para não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

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

Onde estamos ...

- 1 O Curso
 - Ferramentas
 - Metodologia e avaliação
 - Dinâmica
 - Referências
- 2 Ponteiros
 - Motivação aos Ponteiros
- 3 Ponteiros e Matrizes
 - Indireção Múltipla
- 4 Ponteiro para Funções
- 5 Alocação dinâmica
- 6 Pilha
 - Introdução

Pontos fundamentais a serem cobertos:

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



- 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**).

- 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.)

- 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 .

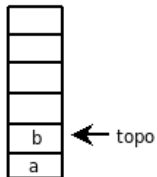
Tabela 1: Operações básicas da estrutura de dados pilha.

Exemplo

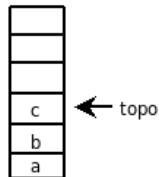
push(a)



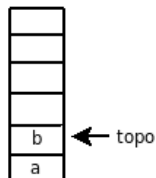
push(b)



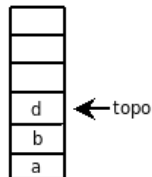
push(c)



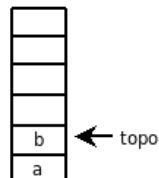
pop
retorna c



push(d)



pop()
retorna d



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

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