Estrutura de Dados

Diógenes Carvalho Matias

Informativos

Avaliação

Prova 01: 08 à 13 de abril

Prova 02: 17 à 22 de junho

2 Chamada: 25 à 29 de junho

Avaliação Final: 01 à 04 de julho.

Introdução

Tipo de dados abstrato

As propriedades lógicas de um tipo de dado é o tipo de dado abstrato, ou TDA. Fundamentalmente, um tipo de dado significa um conjunto de valores e uma seqüência de operações sobre estes valores.

Introdução

Tipo de dados abstrato

Este conjunto e estas operações formam uma construção matemática que pode ser implementada usando determinada estrutura de dados do hardware ou do software.

A expressão "tipo de dado abstrato" refere-se ao conceito matemático básico que define o tipo de dado.

Introdução

Tipo de dados abstrato

Como podemos difeinir novos tipos de dados mas antes disso temos que pensar em
A estrutura da informacao propriamentedita.

Na linguagem de programacao C representamos estrutura da seguinte forma:

```
struct coordenada {
        int x;
        int y;
}
```

Introdução

Tipo de dados abstrato

A palavra reservada para isso é a *struct.*

```
struct coordenada {
int x;
int y;
}
struct coordenada coord;
coord.x = 3;
coord.y = 5;
```

Introdução

Tipo de dados abstrato

Com base no exemplo acima represente novas estrutura de informações,
E imprima na tela do console esse dados, lembrando que tem que ser
15 tipos novos.

Introdução

Tipo de dados abstrato

Como visto nos slide passado vamos agora criar novas estruras abstratas de dados, para Isto utilizamos a palavra reservada *TYPEDEF:*

```
typedef struct coordenada {
int x;
int y;
} Coordenada;


Coordenada c;
c.x = 10;
```

Introdução

Ponteiros

Para começar temos que analisar três propriedades que um programa deve manter quando armazena dados:

– onde a informação é armazenada;
– que valor é mantido lá;
– que tipo de informação é armazenada.

Introdução

Ponteiros

A definição de uma variável simples obedece a estes três pontos. A declaração provê o tipo e um nome simbólico para o valor. Também faz com que o programa aloque memória para o valor e mantenha o local internamente.

Introdução

Ponteiros operador de endereço: &

Segunda estratégia baseada em ponteiros, que são variáveis que armazenam endereços ao invés dos próprios valores.

Mas antes de discutir ponteiros, vejamos como achar endereços explicitamente para variáveis comuns.

Aplique o operador de endereço, &, a uma variável para pegar sua posição; por exemplo, se *notas* é uma variável, *&notas* é seu endereço.

Introdução

Ponteiros operador de endereço: &

Exemplo:

```
#include <stdio.h>
void main()
{
        int idade = 6;
        double salario = 4.5;
                printf("Valor de sua idade = %d", idade);
                printf(" e enderco das idade = %d\n", &idade);
                printf("Valor dos salario = %g", salario);
                printf(" e endereco dos salario = %d\n",&salario);
}
```

Introdução

Ponteiros Operador de dereferenciação: *

O uso de variáveis comuns trata o valor como uma quantidade nomeada e a posição como uma quantidade derivada. A nova estratégia, usando ponteiros, trata a posição como a quantidade nomeada e o valor como uma quantidade derivada.

Introdução

Ponteiros Operador de dereferenciação: *

Este tipo especial de variável – o ponteiro – armazena o endereço de um valor. Então, o nome do ponteiro representa a posição. Aplicando o operador *, chamado de operador de valor indireto ou de dereferenciação, fornece o valor da posição.

Suponha por exemplo, que ordem é um ponteiro. Então, ordem representa um endereço, e *ordem representa o valor naquele endereço. *ordem torna-se equivalente a um tipo comum.

Introdução

Ponteiros Operador de dereferenciação: *

```
#include <stdio.h>
void main()
{
        int atualiza = 6;
        // declara uma variável
        int * p_atualiza;
        // declara ponteiro para um int
                p_atualiza = &atualiza;
        // atribui endereço do int para o
                ponteiro
        // expressa valores de duas formas
                printf("Valores: atualiza = %d", atualiza);
                printf(", *p_atualiza = %d\n", *p_atualiza);
        // expressa endereço de duas formas
                printf("Enderecos: &atualiza = %d", &atualiza);
                printf(", p_atualiza = %d\n", p_atualiza);
        // usa ponteiro para mudar valor
                *p_atualiza = *p_atualiza + 1;
        printf("Agora atualiza = %d\n", atualiza);
        }
```

Introdução

Ponteiros Operador de dereferenciação: *

Imagine o seguinte exemplo:

```
int jumbo = 23;
int *pe = &jumbo;
```

O que acontece na pratica na memoria?

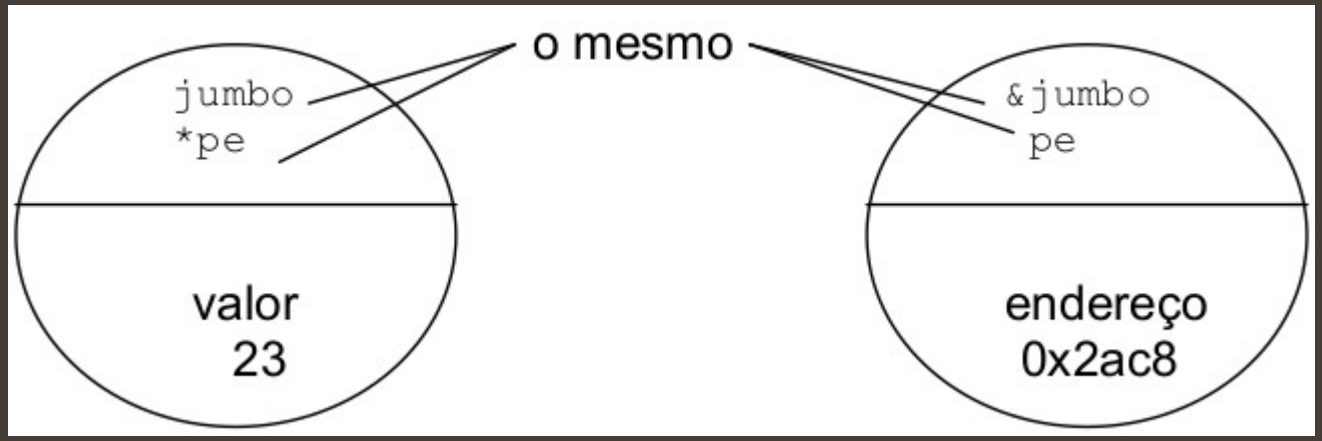
Introdução

Ponteiros Operador de dereferenciação: *

Imagine o seguinte exemplo:

```
int jumbo = 23;
int *pe = &jumbo;
```

O que acontece na pratica na memoria?



Introdução

Ponteiros Declarando e iniciando ponteiros.

```
#include <stdio.h>
void main()
{

	int totalAlunos = 5;
	int * alunos = &higgens;
	printf("Valor de totalAlunos = %d, Endereco de totalAlunos = %d\n", totalAlunos,
&totalAlunos);
	printf("Valor de *alunos = %d; Valor de alunos = %d\n", *alunos, alunos);

}
```

Introdução

Ponteiros Declarando e iniciando ponteiros.

Qual o resultado na tela???

Introdução

Ponteiros Declarando e iniciando ponteiros.

OBS: perigo quando se cria um ponteiro: o computador aloca memória para armazenar um endereço, mas ele não aloca memória para armazenar o dado para o qual o ponteiro aponta. Criar espaço para dados envolve um passo separado. Sem esse passo, pode ocorrer um desastre:.

```
long *perigo;
*perigo = 223323;
```

Introdução

Ponteiros Declarando e iniciando ponteiros.

OBS: perigo quando se cria um ponteiro: o computador aloca memória para armazenar um endereço, mas ele não aloca memória para armazenar o dado para o qual o ponteiro aponta. Criar espaço para dados envolve um passo separado. Sem esse passo, pode ocorrer um desastre:.

```
long *perigo;
*perigo = 223323;
```

Onde o valor 223323 é colocado? Não se sabe. Ou seja, SEMPRE inicie um ponteiro a um endereço apropriado e definido antes de aplicar o operador de dereferenciação (*) nele.

Mas tem como resolver onde veremos mais adiante.

Introdução

Alocação Estática de Memória

Na alocação estática o espaço de memória, que as variáveis irão utilizar durante a execução do programa, é definido no processo de compilação.

Não sendo possível alterar o tamanho desse espaço durante a execução do programa.

Introdução

Alocação Estática de Memória

Exemplos:

/*Espaço reservado para um valor do tipo char. O char ocupa 1 byte na memória.*/

```
char sexo;
```

/*Espaço reservado para dez valores do tipo int. O int ocupa 4 bytes na memória, portanto 4x10=40 bytes.*/

```
int notas[10];
```

Introdução

Alocação Estática de Memória

Exemplos:

/*Espaço reservado para nove(3x3) valores do tipo double. O double ocupa 8 bytes na memória, portanto 3x3x8 = 72 bytes.*/

```
double matriz[3][3];
```

Introdução

Alocação Dinâmica de Memória

Na alocação dinâmica o espaço de memória, que as variáveis irão utilizar durante a execução do programa, é definido enquanto o programa está em execução.

Introdução

Alocação Dinâmica de Memória

Ou seja, quando não se sabe ao certo quanto de memória será necessário para o armazenamento das informações, podendo ser determinadas, sob demanda, em tempo de execução conforme a necessidade do programa.

Introdução

Alocação Dinâmica de Memória

No padrão C ANSI existem quatro funções para alocação dinâmica de memória:

1-malloc()
2-calloc()
3-realloc()
4-free()

Introdução

Alocação Dinâmica de Memória

OBS: Todas elas pertencem a biblioteca <stdlib.h>.

Estrutura de Dados

Sintaxe da função malloc():

```
void *malloc(size_t num_bytes);
```

Esta função recebe como parâmetro "num_bytes" que é o número de bytes de memória que se deseja alocar.
O tipo size_t é definido em stdlib.h como sendo um inteiro sem sinal.

O interessante é que esta função retorna um ponteiro do tipo void podendo assim ser atribuído a qualquer tipo de ponteiro.

Estrutura de Dados

Sintaxe da função malloc():

Exmeplo:

```
int *vetor;

vetor = malloc(400);
```

No exemplo acima foram alocados, dinamicamente, quatrocentos bytes da memória Heap. Quando os programas alocam memória dinamicamente, a biblioteca de execução C recebe memória a partir de um reservatório de memória livre (não utilizado) chamado Heap.

Estrutura de Dados

Sintaxe da função malloc():

A função malloc() devolve um ponteiro do tipo void,desta forma pode-se atribuí-lo a qualquer tipo de ponteiro.

Portanto, precisamos fazer uma conversão (cast) para o tipo desejado e também alocar um espaço compatível com o tipo de destino.

Exemplo:

vetor = (int *) malloc (400*sizeof(int));

Onde (int *) → Conversão para o tipo int *

E 100*sizeof(int) → Serão alocados 1600 bytes no total: 400 x 4.

Estrutura de Dados

Sintaxe da função malloc():

Contudo, não há garantias de que a Heap possua memória disponível para o seu programa.

Portanto, é necessário verificar se existe espaço suficiente na Heap para alocar a memória desejada. Para atender a este requisito, devemos fazer um teste para verificar se a memória foi realmente alocada.

Exemplo:

```
vetor = (int *) malloc (400*sizeof(int));

if (vetor == NULL){
        printf ("Não há memória suficiente para alocação");
        exit(1);
}
```

Estrutura de Dados

Sintaxe da função malloc():

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

void main()
{
        int * pi;
        double * pd;
        pi = (int *)malloc(sizeof(int));
        *pi = 1001;
                // armazena um valor lá
        printf("int ");
        printf("valor = %d: posicao = %d\n", *pi, pi);
        pd = (double *)malloc(sizeof(double));
                // aloca espaço para um double
        *pd = 10000001.0;
                // armazena um double lá
        printf("double ");
```

Estrutura de Dados

Sintaxe da função malloc():

```
        printf("valor = %g: posicao = %d\n", *pd, pd);
        printf("tamanho de pi = %d", sizeof pi);
        printf(": tamanho de *pi = %d\n", sizeof *pi);
        printf("tamanho de pd = %d", sizeof pd);
        printf(": tamanho de *pd = %d\n", sizeof *pd);
        free(pi);
        free(pd);
}
```

Estrutura de Dados

Sintaxe da função free():

void free (void *plivre);

Quando o programa não precisa mais da memória que foi alocada pela função *malloc*, ele deve liberar esta memória,ou seja, informar ao Sistema Operacional que aquela região de memória não será mais utilizada. Para liberar memória alocada, devemos utilizar a função free.

Estrutura de Dados

Sintaxe da função free():

```
void free (void *plivre);
```

Na sintaxe da função free, o parâmetro *plivre* é um ponteiro para o início da região de memória que se quer liberar.

Estrutura de Dados

Sintaxe da função calloc():

```
void *calloc(size_t  numero_itens, size_t  tamanho_item);
```

O parâmetro numero_itens especifica quantos elementos calloc precisa alocar. Já o parâmetro tamanho_item especifica o tamanho, em bytes, de cada um desses elementos.

Estrutura de Dados

Sintaxe da função realloc():

A função *realloc*, que é responsável por alterar o tamanho de um bloco de memória previamente alocado.

void *realloc(void *ptr, size_t tamanhoA);

O parâmetro ptr representa a região de memória que se deseja alterar. Já o parâmetro *tamanhoA* representa o novo tamanho da memória apontada por ptr. Este valor de *tamanhoA* pode ser maior ou menor que o original.

Estrutura de Dados

Diferenças X Vantagens X Desvantagens

1-Na alocação estática, o espaço de memória é definido durante o processo de compilação, já na alocação dinâmica o espaço de memória e reservado durante a execução do programa.

2- Na alocação estática não é possível alterar o tamanho do espaço de memória que foi definido durante a compilação, já na alocação dinâmica este espaço pode ser alterado dinamicamente durante a execução.

Estrutura de Dados

Diferenças X Vantagens X Desvantagens

3- alocação estática tem a vantagem de manter os dados organizados na memória, dispostos um ao lado do outro de forma linear e sequencial. Isto facilita a sua localização e manipulação, em contrapartida, precisamos estabelecer previamente a quantidade máxima necessária de memória para armazenar uma determinada estrutura de dados.

Exemplo:

```
int vetor[5] = {13, 30, 35, 55, 70};
```

Estrutura de Dados

Diferenças X Vantagens X Desvantagens

A quantidade de memória que se deve alocar estaticamente é definida durante a compilação, portanto corre-se o risco de sub ou superestimar a quantidade de memória alocada.

Isso não acontece na alocação dinâmica, pois como a alocação é feita durante a execução, sabe-se exatamente a quantidade necessária. Isso permite otimizar o uso da memória.

Estrutura de Dados

Diferenças X Vantagens X Desvantagens

A alocação dinâmica é feita por meio de funções de alocação e liberação de memória e é de responsabilidade do programador usar essas funções de forma coerente, pois o seu uso incorreto pode causar efeitos colaterais indesejados no programa como vazamento de memória.

Estrutura de Dados

Diferenças X Vantagens X Desvantagens

A na alocação dinâmica podemos redimensionar uma área previamente alocada, já na alocação estática isso não é possível, pois o tamanho foi definido durante a compilação.

Estrutura de Dados

Exercicio para ser feliz XD

A Faça uma pesquisa sobre Ponteiros e Strings para ser entregue na próxima aula por e-mail: prof.dcm.web@hotmail.com