Tipos Abstratos de Dados

Tipo Abstrato de Dados ou TAD

- Idéia principal: desvincular o tipo de dado (valores e operações) de sua implementação: *O "que" o tipo faz e não "como" ele faz!*
- Vantagens da desvinculação:
  1. Integridade dos dados
  2. Facilidade de manutenção
  3. Reuso

- Tipo abstrato de dados: definido matematicamente pelo par (V,O) onde V é um conjunto de valores e O um conjunto de operações sobre estes valores.

Tipo Abstrato de Dados

- Com TAD's, a programação tem 2 fases:
  1. Aplicação: apenas as operações definidas de maneira abstrata são utilizadas; não é permitido o acesso direto aos dados (o usuário só tem acesso as operações)
  2. Implementação: codificação das operações em uma linguagem de programação que dê suporte ao TAD definido
- Formalmente, um TAD consiste de 2 partes:
  - Definição dos valores: em geral, consiste de uma cláusula de definição e outra de condição
  - Definição dos operadores: cabeçalho (parâmetros e resultado), pré-condições (opcionais) e pós-condições

APONTADORES ou PONTEIROS

Apontadores ou Ponteiros (1)

- Ponteiro ou Apontador: variável que contém o endereço de memória de uma outra variável ou estrutura de dados
- A variável ponteiro armazena o endereço do espaço de memória que é “alocado” durante a execução real do programa
- Um apontador fornece então uma maneira indireta de acessar o valor de uma variável do programa

Apontadores ou Ponteiros (2)

- Alocação estática: previsão no pior caso da quantidade de memória a ser usada; reserva de memória em tempo de compilação/tradução
- Alocação dinâmica: alocação de memória para o componente quando ele começa a existir durante a execução do programa
- Apontadores permitem:
  - Representar estruturas de dados complexas;
  - Passagem de valores como argumentos de funções
  - Trabalhar com memória alocada dinamicamente

Implementando Apontadores

- Um apontador pode “apontar” para uma área de memória associada a qualquer tipo de dado, incluindo os registros (*records*)
- Um dos usos mais comuns dos apontadores é exatamente para referenciar tipos estruturados como registros.
- A sintaxe básica em C é:

```
TipoDado  *NomePonteiro;
```

Exemplo 1: como os apontadores funcionam? (C)

```
#include <stdio.h>
void main(){
 int *iptr;
 iptr = malloc(sizeof *iptr);
 *iptr = 10;
  printf("the value is %d ",
  *iptr);
  free(iptr);  /* free((char *)
  iptr */
}
```

Análise do Programa (1)

- A declaração em C:

```
int *iptr;
```

  declara um tipo de variável chamado *iptr*, que é um apontador (denotado por *) para um *integer*.

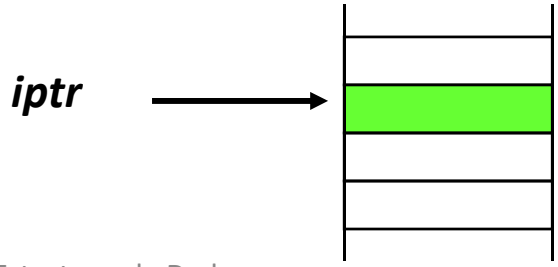
- A variável *iptr* não contém um valor numérico; ela irá armazenar endereços de memória de uma variável criada dinamicamente (pelo uso da declaração *malloc* vista adiante...)

Análise do Programa (2)

- **Note que neste instante, AINDA NÃO EXISTE ESPAÇO DE MEMÓRIA ALOCADO com *iptr*, i.e. não existe nenhum espaço de memória associado ao apontador!**

- O comando `malloc(sizeof *iptr);`

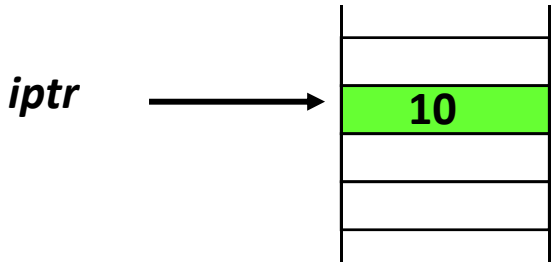
  cria um novo espaço dinâmico "apontado" por *iptr* (i.e, ele é criado quando o programa "rodar" na máquina)
- A variável do tipo apontador *iptr* "aponta" para o endereço do espaço de memória usado para "guardar" um valor inteiro



Análise do Programa (3)

- O comando `*iptr = 10;`

  escreve no espaço de memória apontado (ou associado a) por *iptr*, um valor inteiro 10



- **O comando `free (iptr);`**

  "desaloca" (libera) o espaço de memória apontado (ou associado a) por *iptr*, retornando esse espaço para o sistema. O apontador *iptr* só poderá ser usado novamente se for associado a outra declaração *malloc()*

Exemplo 1 modificado... (C)

```
#include <stdio.h>
void main(){
 int *iptr;
 iptr = malloc(sizeof *iptr);
 *iptr = 10;
  printf("o valor e %d ", *iptr);
  free(iptr);
  iptr = NULL;
  if (iptr == NULL)
    printf("iptr não referencia nada");
  else
    printf("O valor da referência de iptr é %d", *iptr);
}
```

Valor *NULL*

- Se o apontador não faz referência a nenhuma posição de memória, ele deve receber o valor *NULL* como atribuição
- A declaração `iptr = NULL;` faz exatamente isso.

  Isto é, o apontador é válido, existe, mas não está apontando para nenhuma posição de memória ou variável dinâmica
- A declaração `if (iptr == NULL)` é um exemplo de teste muito comum com apontadores;

Valores e apontadores (C)

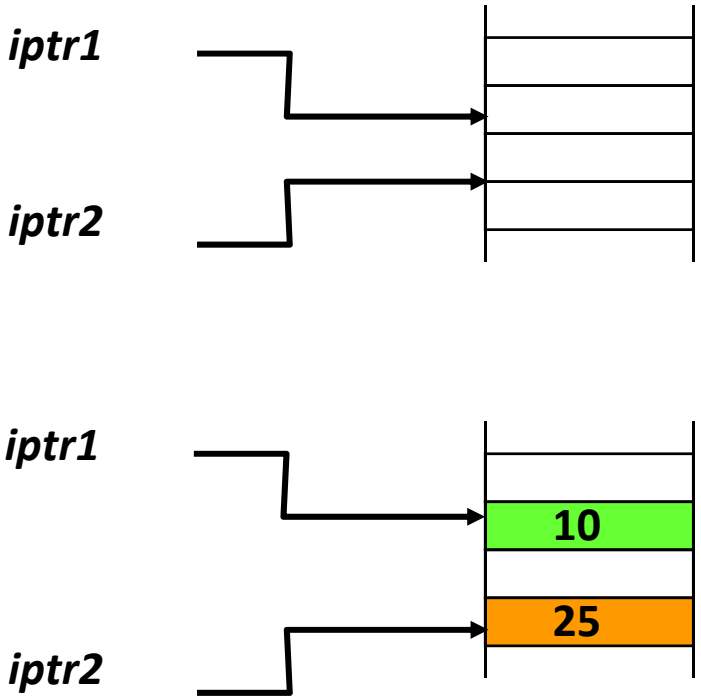
```
#include <stdio.h>

void main(){
 int *iptr1, *iptr2;
 iptr1 = malloc(sizeof *iptr1);
 iptr2 = malloc(sizeof *iptr2);
 *iptr1 = 10;      *iptr2 = 25;
  printf("Valor de iptr1: %d", *iptr1);
  printf("Valor de iptr2: %d", *iptr2);
  free( iptr1 );
   iptr1 = iptr2;
   *iptr1 = 3;
   printf("Valor de iptr1: %d", *iptr1);
   printf("Valor de iptr2: %d", *iptr2);
   free( iptr2 );
}
```

Analisando o programa...

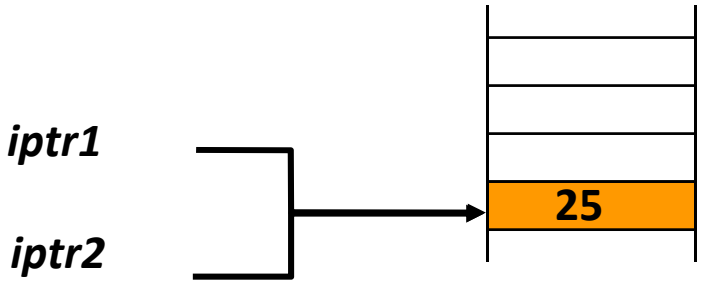
```
malloc(sizeof
   *iptr1);
malloc(sizeof
   *iptr2);

*iptr1  = 10;
*iptr2  = 25;
```

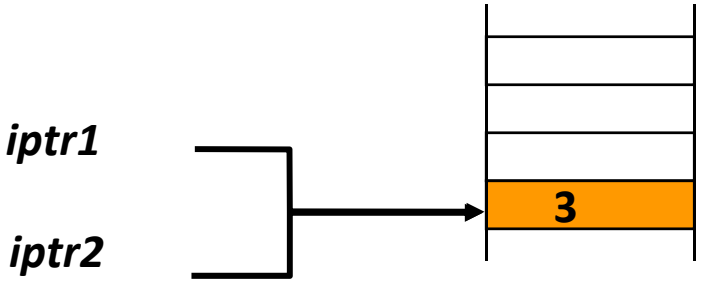


Analisando o programa...

```
free(iptr1);
iptr1 = iptr2;
```




```
*iptr1 = 3;
```



Resultado da execução

```
Valor de iptr1: 10
Valor de iptr2: 25
Valor de iptr1: 3
Valor de iptr2: 3
```

Resumo Ponteiros

- Um ponteiro pode referenciar qualquer tipo de dado na memória, mesmo estruturas mais complexas;
- Em C , a instrução *malloc* aloca espaço de memória para ser usado pelo ponteiro *p*
- A instrução *free(p)* desaloca o espaço de memória associado ao ponteiro *p*

Resumo Ponteiros

- Um ponteiro pode ser associado a um espaço de memória usando *malloc*, ou através da atribuição de um valor por um ponteiro de mesmo tipo (ex, `iprt1 = iptr2;`)
- Pode-se atribuir um valor NULL a um ponteiro para indicar que ele não aponta para nenhum espaço de memória
- O valor do espaço de memória associado ao ponteiro pode ser lido ou alterado através da sintaxe `*iprt1`
- Um ponteiro pode ser declarado e referenciar