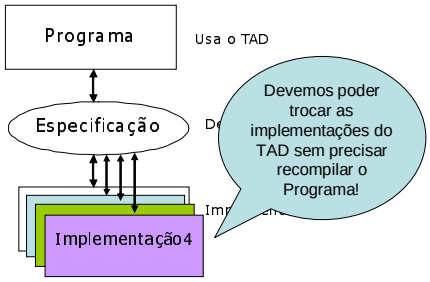
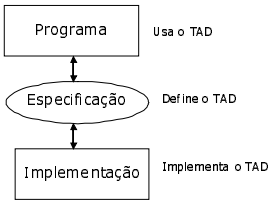
Estruturas de Informação
Aula 3: Tipos Abstratos de Dados & Uso da Memória

13/03/2008

## Tipo Abstrato de Dado

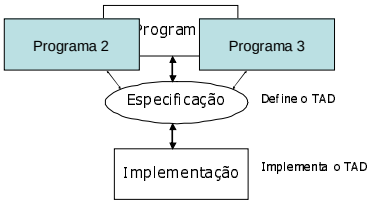



## Software em Camadas




- As camadas de software são independentes
- Modificações na implementação do TAD não geram (grandes) mudanças no programa

## Software em Camadas (2)




- Essa abordagem também permite o reuso de códico
- A mesma implementação pode ser usada por vários programas

## TAD's em C

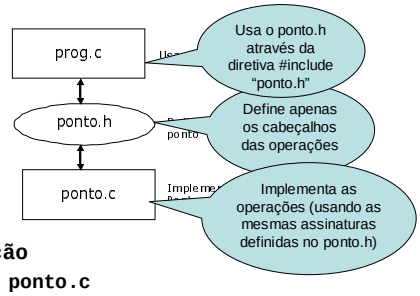
- A linguagem C oferece mecanismos para especificação e uso de TAD's:
  - O uso é possível pois C permite modularização de programas
  - A especificação é possível com o arquivo cabeçalho (.h)
    - O arquivo .h possui apenas os protótipos das operações
    - Usar a #include para incluir o arquivo .h. Inclui o arquivo antes da compilação
  - Os diferentes módulos são incluídos em um único programa executável na "linkagem"

## TAD's em C (2)

Exemplo:

- TAD Ponto no arquivo *ponto.h*
- Implementação do tipo ponto no arquivo *ponto.c*
- Módulo que usa a implementação do ponto é *prog.c*
  - `#include "ponto.h"`
  - **Inclui o cabeçalho na pré-compilação (chamado pré-processamento)**

## TAD's em C (2)




- **Compilação**
  - **gcc –c ponto.c**
  - **gcc –c prog.c**
- **Linkagem**
  - **gcc –o prog.exe ponto.o prog.o**

## Abstração

- "É a habilidade de concentrar nos aspectos essenciais de um contexto qualquer, ignorando características menos importantes ou acidentais"
- Quando definimos um TAD (Tipo *Abstrato* de Dados), nos concentramos nos aspectos essencias do tipo de dado (operações) e nos abstraímos de como ele foi implementado

## Encapsulamento

- "Consiste na separação de aspectos internos e externos de um objeto".
- O TAD provê um mecanismo de encapsulamento de um tipo de dado, no qual separamos a especificação (aspecto externo) de sua implementação (aspecto interno)

## Memória

- Ponteiros
- Uso de memória
- Alocação estática e dinâmica

## Recap: Ponteiros

- Permite o armazenamento e manipulação de endereços de memória
- *Forma geral*
  - ***tipo_do_ponteiro *nome_da_variável***
  - Símbolo * indica ao compilador que a variável guardará o endereço da memória
  - Neste endereço da memória haverá um valor do tipo especificado (tipo_do_ponteiro)
  - `char *p;` (p pode armazenar endereço de memória em que existe um caracter armazenado)
  - `int *v;` (v pode armazenar endereço de memória em que existe um inteiro armazenado)

## Recap: Ponteiros (2)

- Exemplo

```
/*variável inteiro*/
int a;

/*variavel ponteiro para inteiro */
int* p;

/* a recebe o valor 5*/
a = 5;

/* p recebe o endereço de a */
p = &a;

/*conteúdo de p recebe o valor 6 */
*p = 6;
```

## Recap: Ponteiros (3)

- Exemplo

```
int main (void)
{
  int a;
  int *p;
  p = &a;
  *p = 2;
  printf (" %d ", a)
  return 0;
}
```

## Recap: Ponteiros (4)

- Exemplo

```
int main (void)
{
  int a, b, *p;
  a = 2;
  *p = 3;
  b = a + (*p);
  printf (" %d ", b);
  return 0;
}
```

## Uso da memória

- Existem 3 maneiras de reservar o espaço da memória:
  - Variáveis globais (estáticas)
    - Espaço existe enquanto programa estiver executando
  - Variáveis locais
    - Espaço existe enquanto a função que declarou estiver executando
  - Espaços dinâmicos (alocação dinâmica)
    - Espação existe até ser explicitamente liberado

## Alocação estática da memória

- Estratégia de alocação de memória na qual toda a memória que um tipo de dados pode vir a necessitar (como especificado pelo usuário) é alocada toda de uma vez sem considerar a quantidade que seria realmente necessária na execução do programa
- O máximo de alocação possível é ditado pelo hardware (tamanho da memória "endereçável")

## Alocação estática da memória (2)

- int v[1000]
  - Espaço contíguo na memória para 1000 valores inteiros
  - Se cada int ocupa 4 bytes, 4000 bytes, ~4KB
- char v[50]
  - Espaço contíguo na memória para 50 valores do tipo char
  - Se cada char ocupa 1 byte, 50 bytes

## Alocação estática X Alocação dinâmica

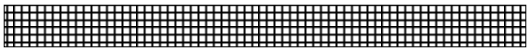
- Exemplo: Alocar nome e sobrenome dos alunos do curso
  - 3000 espaços de memória
  - Vetor de string (alocação estática)
  - 100 caracteres (Tamanho máximo do nome inteiro)
  - Podemos então definir 30 pessoas
  - Não é o ideal pois a maioria dos nomes não usam os 100 caracteres
  - Alocação dinâmica não é necessário definir de ante-mão o tamanho máximo para os nomes.

## Alocação dinâmica da memória

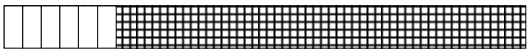
- Oposto a alocação estática
- Técnica que aloca a memória sob demanda
- Os endereços podem ser alocados, liberados e realocados para diferentes propósitos, durante a execução do programa
- Em C usamos **`malloc(`*`n`*`)`** para alocar um bloco de memória de tamanho n bytes.
- Responsabilidade do programador de liberar a memória após seu uso

## Alocação dinâmica da memória (2)

- Espaço endereçável (3000) ainda livre:

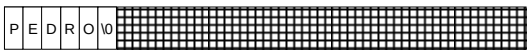


  - Alocar espaço para o nome PEDRO
  - 5 bytes para o nome e um byte para o caracter NUL (\0). Total 6 bytes
    - **`Malloc (6)`**

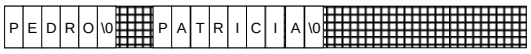


## Alocação dinâmica da memória (3)

- Escrevemos PEDRO no espaço

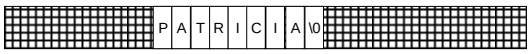



- Alocamos e escrevos PATRICIA



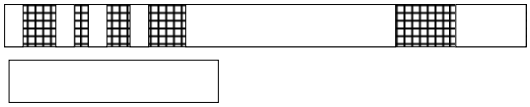

## Alocação dinâmica da memória (4)

- Endereços não necessariamente contíguos
- Alocador de memória do SO aloca blocos de memória que estão livres
- Alocador de memória gerencia espaços ocupados e livres
- Memória alocada contém lixo. Temos que inicializar
- Em C, liberamos a memória usando `free(p)`




## Alocação dinâminca (problemas)

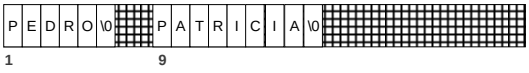
- Liberar memória é responsabilidade do usuário
  - "memory violation"
  - acesso errado à memória, usada para outro propósito
- Fragmentação
  - Blocos livres de memória não contíguos



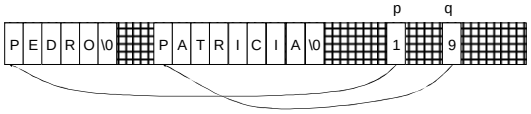
- Estruturas encadeadas fazem melhor uso da memória fragmentada

## Endereçamento em alocação dinâminca

- Precisamos saber os endereços dos espaços de memória usados




- Ponteiros são variáveis que armazenam o endereço na própria memória
  - char *p; char *q;



## Alocação dinâminca em C

- Funções disponíveis na stdlib
  - malloc
    - void *malloc (unsigned int num);
  - calloc
    - void *calloc (unsigned int num, unsigned int size);
  - Realoc
    - void *realloc (void *ptr, unsigned int num);
  - free
    - void free (void *p);

## malloc

```
#include <stdio.h>
#include <stdlib.h>
main (void)
{
int *p;
int a;
... /* Determina o valor de a em algum lugar */
p=(int *)malloc(a*sizeof(int));
if (!p)
        {
        printf ("** Erro: Memoria Insuficiente **");
        exit;
        }
...
return 0;
}
```

**Alocada memória suficiente para se colocar *a* números inteiros**

## calloc

```
#include <stdio.h>
#include <stdlib.h>
main (void)
{
int *p;
int a;
...
p=(int *)calloc(a, sizeof(int));
if (!p)
        {
        printf ("** Erro: Memoria Insuficiente **");
        exit;
        }
...
return 0;
}
```

**Alocada memória suficiente para se colocar *a* números inteiros**

## free

```
#include <stdio.h>
#include <alloc.h>
main (void)
{
int *p;
int a;
...
p=(int *)malloc(a*sizeof(int));
if (!p)
        {
        printf ("** Erro: Memoria Insuficiente **");
        exit;
        }
...
free(p);
...
return 0;
}
```