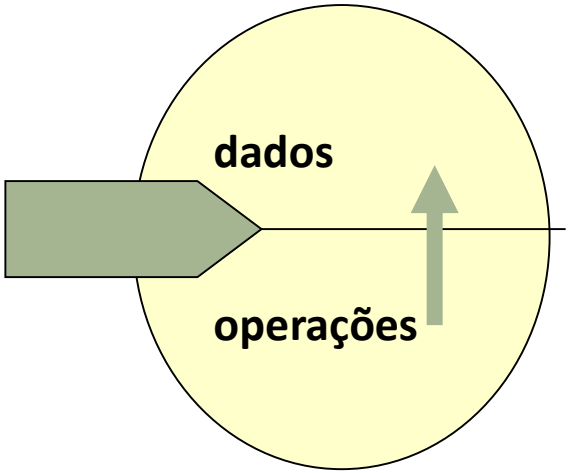
Tipo Abstrato de Dados

Introdução:



- Programas consistem em 2 coisas:
  1. Algoritmos e
  2. Estruturas de dados (ED)

- Um bom programa é uma combinação de ambos;

- A escolha e a implementação de uma ED são tão importantes quanto as rotinas que manipulam os dados.
  - A forma como a informação é organizada e acessada é normalmente determinada pela natureza do problema de programação

Introdução:

- Por essa razão, para cada tipo de representação de dados, devemos ter o método correto para a manipulação da informação

- Um “Tipo Abstrato de Dado” é a ferramenta certa para garantir que dados sejam manipulados da forma certa

Tipos Abstratos de Dados (TADs):

- Quando a linguagem não oferece um *tipo nativo*, adequado à representação de uma entidade do domínio do problema, podemos recorrer a um TAD
  - Contexto Inicial: Programação Estruturada

- "TAD é um encapsulamento que inclui somente a *representação das informações* de um tipo específico de dados, e os *subprogramas* que fornecem as operação para esse tipo" [Sebesta99]

- TAD = encapsulamento(estrutura+operações)$_{\textbf{TIPO}}$

Definição:

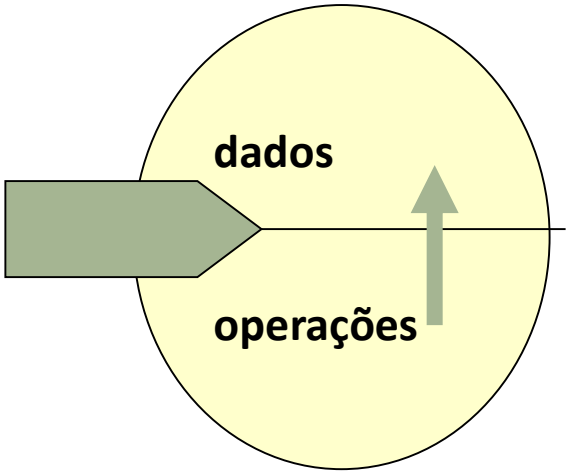
- Tipo Abstrato de Dados é uma especificação de tipo contendo um conjunto de dados e operações que podem ser executadas sobre esses dados.
  - Descreve quais dados podem ser armazenados (características) e como eles podem ser manipulados (operações)
  - Mas não descreve como isso é implementado no programa

- TADs são geralmente implementados através de tipos compostos heterogêneos (Registros), associados a um conjunto de funções que operam sobre essa estrutura.

- Exemplo:

```
struct Estudante {
     char nome [64]; int idade; char matricula[10]
 }
 int maiorDeIdade(Estudante estudante); //retorna 1 se verdadeiro
 void validaMatricula(Estudante estudante); // efetua matricula
```

Definição:



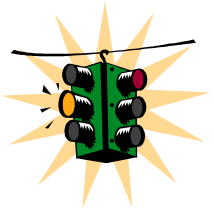
- Um TAD pode ser visto como um modelo matemático, acompanhado das operações definidas sobre o modelo

- Por exemplo: o conjunto dos inteiros acompanhado das operações de adição, subtração e multiplicação

- TADs podem ser considerados generalizações de tipos primitivos de dados (como tipo inteiro, real), da mesma forma que funções são generalizações de operações primitivas tais como adição, subtração e multiplicação

Encapsulamento:

- Tipos Abstratos de Dados utilizam o conceito de Encapsulamento, para reduzir dependências entre as estruturas e garantir o comportamento correto das operações sobre os dados.
  - Encapsular: esconder aquilo que não deve ser manipulado diretamente.



- Importante: em um TAD, os dados armazenados são acessados sempre através das funções definidas para ele, e nunca diretamente
  - Independência: é possível alterar a estrutura interna de um TAD sem afetar código cliente
  - Consistência: as funções definidas para um TAD garantem que os dados serão sempre acessados na ordem e da forma corretas

Encapsulamento:

> "***Encapsulamento*** *é o processo de* ***esconder todos os detalhes*** *de um objeto* ***que não contribuem*** *para suas características essenciais."*
>
> [Grady Booch]

Exemplo de Encapsulamento:

Consideração:

"***Nenhuma parte*** *de um sistema complexo* ***deve depender*** *dos* ***detalhes internos*** *das outras partes*"

"*Para uma* ***abstração dar certo*** *sua* ***implementação*** *deve estar* ***encapsulada***."

[Grady Booch]

Características:

- Encapsulamento / Abstração
  - A abstração representa um conceito
  - O encapsulamento impede os clientes de verem como este conceito foi construído
- Encapsulamento:
  - Esconde os detalhes internos de uma abstração
  - Uma vez selecionada a implementação de uma abstração, ela deve ser tratada como um segredo da abstração e escondida dos seus clientes.
  - Este conceito é anterior à POO

Exemplo de TADs em C (Pilha):

- TAD “pilha” implementado com arranjo:

```
typedef struct st_pilha {
  int topo;
  float dados[MAX];
} pilha;

void cria_pilha (pilha *){...}
int empilha (pilha *, float) {...}
int desempilha (pilha *, float *) {...}
int ta_vazia (pilha) {...}
int topo (pilha, float*) {...}
int limpa (pilha *) {...}
```

Representação ou Estrutura do TAD “pilha”

Operações do TAD “pilha”

Exemplo de TADs em C (Pilha):

- Um cliente para o TAD “pilha”:

```
int main() {
    pilha p;
    float x, y;
    cria_pilha(&p);
    if (ta_vazia(p)) {
      p.dados[p.topo] = 1.2f; p.topo++;
      p.dados[p.topo] = 3.0f; p.topo++;
    }
    topo(&p, &x);
    if (x == 3.0f) {
      desempilha (&p, &y);
    }
    limpe (&p);
}
```

Exemplo de TADs em C (Pilha):

- Um cliente para o TAD “pilha”:

```
int main() {
    pilha p;
    float x, y;
    cria_pilha(&p);
    if (ta_vazia(p)) {
      p.dados[p.topo] = 1.2f; p.topo++;
      p.dados[p.topo] = 3.0f; p.topo++;
    }
    topo(&p, &x);
    if (x == 3.0f) {
      desempilha (&p, &y);
    }
    limpe (&p);
}
```

Isto pode ser problemático!

Exemplo de TADs em C (Pilha):

- A implementação muda para lista encadeada:

```
typedef struct st_pilha {
  float dado;
  struct st_pilha *prox;
} nopilha;
typedef nopilha* pilha;

void cria_pilha (pilha *){...}
int empilha (pilha *, float) {...}
int desempilha (pilha *, float *) {...}
int ta_vazia (pilha) {...}
int topo (pilha, float*) {...}
int limpa (pilha *) {...}
```

Modifica-se a representação do TAD e as implementações das operações

Exemplo de TADs em C (Pilha):

- O cliente é *afetado*: **Violação do Encapsulamento!**

```
int main() {
    pilha p;
    float x, y;
    cria_pilha(&p);
    if (ta_vazia(p)) {
      p.dados[p.topo] = 1.2f; p.topo++;
      p.dados[p.topo] = 3.0f; p.topo++;
    }
    topo(&p, &x);
    if (x == 3.0f) {
      desempilha (&p, &y);
    }
    limpe (&p);
}
```

Gera acoplamento entre TAD e cliente

Exemplo de TADs em C (Pilha):

- Cliente comprometido com o encapsulamento:

```
int main() {
    pilha p;
    float x, y;
    cria_pilha(&p);
    if (ta_vazia(p)) {
      empilha(&p, 1.2f);
      empilha(&p, 3.0f);
    }
    topo(&p, &x);
    if (x == 3.0f) {
      desempilha (&p, &y);
    }
    limpe (&p);
}
```

TADs e Encapsulamento:

"*A habilidade de **mudar a representação** de uma abstração **sem perturbar quaisquer de seus clientes** é o principal benefício do **encapsulamento***"

"*O **encapsulamento não impede** o programador de fazer **coisas estúpidas***"

[Grady Booch]

"*O **encapsulamento previne acidentes**, não **fraudes***"

[Bjarne Stroustroup]

TADs e Encapsulamento:

- O conjunto de operações públicas oferecidas por um TAD é chamado de *Interface do TAD*

- Quando é que uma mudança num TAD exige mudanças nos clientes?
  - Quando há mudanças em sua interface.

  - Conclusão Importante: se a interface permanecer a mesma, os clientes não se alteram.

TADs e Encapsulamento:

- Regras para um bom encapsulamento:

```
typedef struct st_pilha {
   float dado;
   struct st_pilha prox;
} nopilha;
typedef nopilha* pilha;

void cria_pilha (pilha *){...}
int empilha (pilha *, float) {...}
int desempilha (pilha *, float *) {...}
int ta_vazia (pilha) {...}
int topo (pilha, float*) {...}
int limpa (pilha *) {...}
```

Permite

Impede

- **Permitir** acesso ao tipo definido para o TAD e ao cabeçalho das operações.
- **Impedir** o acesso à representação do TAD e à implementação das operações.

TADs: considerações

- Suporte para TADs em C:
  - C permite que se especifiquem TADs em bibliotecas compiladas em separado.
  - A definição do tipo e os protótipos das funções que implementam as operações ficam num arquivo de cabeçalhos (com extensão .h).
  - A implementação das operações ficam num arquivo de implementação (extensão .c)
  - O encapsulamento de C não é perfeito: os clientes podem acessar os campos da estrutura do tipo.

TADs: considerações

- Suporte para TADs em Java:
  - Possui recurso mais poderoso para representação de TADs: classes (arquivo .java).
  - A classe é a unidade sintática para encapsulamento de TADs e também a unidade de compilação.
  - Pode-se controlar o nível de encapsulamento de cada elemento de uma classe.
  - Java não tem construções de tipo como struct do C: cada classe em si é um novo tipo, com propriedades e operações.
  - Valores de classes são chamados de objetos.

TAD: Vantagens

- Mais fácil programar (sem se preocupar com os detalhes de implantação);
- Mais seguro programar (apenas as operações do próprio TAD alteram as estruturas locais);
- Maior independência e portabilidade de código (alterações na implementações de um TAD não precisam alterar funcionalidades do sistema);
- Maior potencial de reutilização de código (pode-se alterar a lógica de um programa sem a necessidade de reconstruir um TAD);
- Consequência: Custo Menor no Desenvolvimento do Software

TAD Exemplo:

- Sabemos que a linguagem C não possui o tipo *String* definido. Representamos textos como vetores de char e usamos funções de biblioteca para manipulá-los

- Vamos construir um TAD "*String*", que usa um vetor de char para representar o texto (como normalmente fazemos)
  - O vetor deve ser dinâmico, com tamanho múltiplo de 64 caracteres
  - Deve ser redimensionado quando necessário (sempre em blocos de 64)

- Vamos também definir algumas operações para esse TAD:

1. `String *createEmptyStr()`: cria uma string vazia ('\0')
2. `String *createStr(char c[])`: cria string contendo c[]
3. `void append(String *s1, String *s2)`: apenda s2 em s1
4. `void addChar(String *s1, char c)`: adiciona c no final de s1
5. `String *substr(String *s1, int ini, int final)`: retorna a substring delimitada entre `ini` e `final`, como uma nova String