# Tipos Abstratos de Dados

**Baseado em**
**"Concepts of Programming Languages- 4rd Ed."**
**Robert W. Sebesta**
**(Addison-Wesley, 1998)**

**Elaborado por: Alberto Castro**
**DCC/UA, 1999**

## Abstração

**- O conceito de abstração é fundamental em programação**

**- Quase todas as linguagens suportam abstração de processos, através de subprogramas**

**- Quase todas as linguagens de programação projetadas desde 1980 tem suporte à abstração de dados com algum tipo de módulo**

## Encapsulamento

**- *Motivação original*: Grandes programas tem duas necessidades especiais:**

**1. Algum tipo de organização, além da simples divisão em subprogramas**

**2. Algum tipo de compilação parcial (unidades de compilação são menores que o programa inteiro)**

**- *Solução óbvia*:**

**Um agrupamento de subprogramas que são logicamente relacionados em uma unidade que pode ser compilada separadamente**
**- São chamados *encapsulamentos***

## Exemplos de Mecanismos de Encapsulamento

**1. Subprogramas aninhados em alguma linguagem similar ao ALGOL (e.g., Pascal)**

**2. FORTRAN 77 e C - Arquivos contendo um ou mais subprogramas podem ser compilados independentemente**

**3. FORTRAN 90, C++, Ada (e outras linguagens contemporâneas) - separadamente em módulos compiláveis**

**Def: Um *tipo abstrato de dados* é um tipo de dados definido pelo usuário que satisfaz a duas condições:**

**1. A representação e as operações em objetos do tipo são definidos em uma unidade sintática simples; outras unidades podem também criar objetos daquele tipo**

**2. A representação de objetos do tipo é ocultado das unidades do programa que usam esses objetos, assim as únicas operações possíveis são aquelas que provêm da definição do tipo.**

***Vantagens da Restrição n. 1:***

**- As mesmas referentes à encapsulamento: organização do programa, alterabilidade (tudo que estiver associado com uma estrutura de dados está junto), e compilação em separado**

***Vantagem da Restrição n. 2:***

**- *Confiabilidade* - ao esconder as representações dos dados, o código do usuário não pode acessar diretamente objetos do tipo. O código do usuário não pode depender da representação dos dados, permitindo que tal representação seja alterada sem afetar o código do usuário.**

***Tipos primitivos (built-in) são tipos abstratos de dados***

**e.g. tipo `int` no C**
- **A representação é oculta**
- **Todas as operações são *built-in***
- **Programas do usuário podem definir objetos do tipo `int`**

**- Tipos de dados definidos pelo usuário devem ter as mesmas características dos TAD *built-in***

## Requisitos da Linguagem para Abstração de Dados:

**1. Uma unidade sintática na qual a definição de tipo seja encapsulada.**

**2. Um método de fazer nomes de tipos e cabeçalhos de subprogramas visíveis a clientes, ao mesmo tempo que esconde as verdadeiras definições.**

**3. Algumas operações primitivas devem fazer parte do processador da linguagem (usualmente apenas atribuições e comparações para igualdade e desigualdade)**
- **Algumas operações são normalmente necessárias, mas devem ser definidas por quem especifica o tipo**
- **e.g., iteradores, construtores, destrutores**

## Questões do Projeto das LP:

**1. Encapsulamento é um tipo simples, ou algo mais?**
**2. Quais tipos podem ser abstratos?**
**3. Os tipos abstratos podem ser parametrizados?**
**4. Quais controles de acesso são permitidos/disponibilizados?**

## Exemplos de Linguagens:

***1. Simula 67***

- **Permite encapsulamento, mas não ocultamento da informação**

***2. Ada***
- **O composto de encapsulamento é o pacote**

- **Pacotes usualmente possuem duas partes:**

  1. **Especificação do pacote (interface)**
  2. **Corpo do pacote (implementação das entidades enumeradas na especificação)**

- **Qualquer tipo pode ser exportado**

- ***Ocultamento da Informação***

- **Tipos ocultos são nomeados na especificação do pacote, como em:**

```
type NODE_TYPE is private;
```

- **Representação de um tipo oculto exportado é especificado numa parte da especificação do pacote (a cláusula `private`), que é invisível a clientes, como em:**

```
package … is
   type NODE_TYPE is private;
   …
   type NODE_TYPE is
     record
     …
     end record;
   …
```

- **Uma especificação de pacote também pode definir tipos não ocultos simplesmente dispondo sua representação fora de uma cláusula privada**

- ***Os motivos para a definição em duas partes são:***

  1. **O compilador deve estar apto a ver a representação após ver somente a especificação do pacote (o compilador pode ver as cláusulas privadas)**

  2. **Clientes devem ver o nome do tipo, mas não a representação (clientes não podem ver cláusulas privadas)**

- ***Tipos privados* possuem operações *built-in* para atribuição e comparação com = e `/=`**

- ***Tipos privados limitados* não possuem operações *built-in***

***Avaliação dos TAD do Ada***

1. **Falta de restrição a ponteiros (comparando-o com Modula-2) é melhor**
   - **Custo é a recompilação de clientes quando a representação é modificada**

2. **Não pode importar entidades específicas de outros pacotes**

*4. C++*

**- Baseado no tipo `struct` do C e nas classes do Simula 67**
**- A classe é o dispositivo de encapsulamento**
**- Todas as instâncias de uma classes compartilham um só cópia das funções-membro**
**- Cada instância de uma classe possui sua própria cópia dos membros de dados**
**- Instâncias podem ser estáticas, dinâmicas na pilha ou dinâmicas no heap**

***- Ocultamento da Informação:***
**- *Cláusula private* para entidades ocultas**
**- *Cláusula public* para entidades da interface**
**- *Cláusula protected* para herança**

***- Construtores:***
**- Funções para inicializar os membros de dados das instâncias (não criam os objetos)**
**- Podem também fazer alocação de armazenamento se parte do objeto é dinâmico no heap**
**- Podem incluir parâmetros para fornecer parametrização de objetos**
**- Implicitamente chamados quando uma instância é criada**
**- Podem ser chamados explicitamente**
**- O nome é o mesmo do nome da classe**

***- Destrutores***
**- Funções para limpeza após uma instância ser destruída; usualmente apenas uma retomada de espaço de armazenamento no heap**
**- Implicitamente chamados quando o tempo de vida do objeto acaba**
**- Podem ser chamados explicitamente**
**- Nome é o mesmo do nome da classe, precedido de um til (~)**

***- Funções ou classes friend* - provêm acesso a membros privados a algumas unidades ou funções não relacionadas (*necessário* em C++)**

***Avaliação do suporte do C++ a TAD***

**- Classes são similares a pacotes do Ada ao prover TAD**

**- Differença: pacotes são encapsulamentos, enquanto que classes *são* tipos**

## Java

**- Similar ao C++, exceto que:**

**- Todos os tipos definidos pelo usuário são classes**
**- Todos os objetos são alocados no heap e acessados através de variáveis de referência**

**- Entidades individuais nas classes tem modificadores do controle de acesso (*private* ou *public*), ao invés de cláusulas**

**- Java possui um segundo mecanismo de *scoping*, o *package scope*, que pode ser usado no lugar de *friends***

**- Todas as entidades em todas as classes num pacote que não tem acesso aos modificadores do controle de acesso, são visíveis através do pacote**

## TAD Parametrizados

***1. Pacotes genéricos no Ada***

**- Fazem o tipo pilha mais flexível ao tornarem o tipo do elemento e o tamanho da pilha genérica**

**2. Classes de template no C++**

**- Classes podem ser genéricas, quando se escreve funções construtoras parametrizadas**

**e.g.**

```
stack (int size) {
  stk_ptr = new int [size];
  max_len = size - 1;
  top = -1;
 }
stack (100) stk;
```

**- O elemento da pilha pode ser parametrizado fazendo-se a classe um tipo de classe por template**

**- Java não suporta TAD genéricos**