# *Estruturas de Linguagem*

# *Tipos Abstratos de Dados*

**Francisco Sant'Anna**

**francisco@ime.uerj.br**

**http://github.com/fsantanna/EDL**

# O Conceito de Abstração

An **abstraction** is a view or representation of an entity that includes only the most significant attributes.

- Classes de abstrações
  - processos

```
sortInt(list, listLen)
```

  - dados

```
float d;
```

- Uma "arma" contra complexidade
  - facilita no gerenciamento de programas

# Abstração de Dados
## (definida pelo programador)

- Tipos de dados definidos pelo programador
- Invólucro que inclui
  - representação do tipo de dado
  - implementação das operações sobre o tipo de dado
- Detalhes desnecessários são escondidos
  - "information hiding"
- Interface de acceso
  - construtores (instâncias) + operações

# Exemplo: Pilha

| | |
|---|---|
| create(stack) | Creates and possibly initializes a stack object |
| destroy(stack) | Deallocates the storage for the stack |
| empty(stack) | A predicate (or Boolean) function that returns true if the specified stack is empty and false otherwise |
| push(stack, element) | Pushes the specified element on the specified stack |
| pop(stack) | Removes the top element from the specified stack |
| top(stack) | Returns a copy of the top element from the specified stack |

```
. . .
create(stk1);
push(stk1, color1);
push(stk1, color2);
temp = top(stk1);
. . .
```

```
Stack stk;  //** Cre
stk.push(42);
stk.push(17);
topOne = stk.top();
stk.pop();
. . .
```

- Vetor?
- Lista?

# Considerações

- Como expor a interface sem expor a representação / implementação?
  - modificadores
    - private / protected
  - módulos
  - unidades de compilação

# Pilha em C++ e C

```
class Stack {
  public:
    Stack();
    ~Stack();
    int  empty (void);
    void push  (int);
    void pop   (void);
    int  top   (void);
  private:
    int* stackPtr;
    int  maxLen;
    int  topSub;
}
```

```
typedef struct Stack Stack;
Stack* create (void);
int    empty  (Stack*);
void   push   (Stack*, int);
void   pop    (Stack*);
int    top    (Stack*);
```

define

# Abstração de Dados

We now formally define an abstract data type in the context of user-defined types. An **abstract data type** is a data type that satisfies the following conditions:

- The representation of objects of the type is hidden from the program units that use the type, so the only direct operations possible on those objects are those provided in the type's definition.
- The declarations of the type and the protocols of the operations on objects of the type, which provide the type's interface, are contained in a single syntactic unit. The type's interface does not depend on the representation of the objects or the implementation of the operations. Also, other program units are allowed to create variables of the defined type.

# Abstração de Dados

- Vantagens?
  - confiabilidade (reliability)
    - clientes não podem manipular as representações internas (conscientemente ou não)
  - legibilidade/redigibilidade (readability/writability)
    - menos código, mudanças centralizadas
- Getters and Setters?

> 1. Read-only access can be provided by having a getter method but no corresponding setter method.
> 2. Constraints can be included in setters. For example, if the data value should be restricted to a particular range, the setter can enforce that.
> 3. The actual implementation of the data member can be changed without affecting the clients if getters and setters are the only access.

# Pilha em C++ e C

```
class Stack {
  public:
    Stack();
    ~Stack();
    int  empty (void);
    void push  (int);
    void pop   (void);
    int  top   (void);
  private:
    int* stackPtr;
    int  maxLen;
    int  topSub;
}
```

```
typedef struct Stack Stack;
Stack* create (void);
int    empty  (Stack*);
void   push   (Stack*, int);
void   pop    (Stack*);
int    top    (Stack*);
```

define

# Pilha em Elm

```
module Stack exposing (create, empty, top, push, pop)

type alias Stack = List Int

create : Stack
create = []

empty : Stack -> Bool
empty s = List.isEmpty s

top : Stack -> Maybe Int
top s = List.head s

push : Int -> Stack -> Stack
push x s = (x :: s)

pop : Stack -> Maybe Stack
pop s = case s of
            [] ->         Nothing
            _ :: rest -> Just rest
```