

# CS421 Lecture 14: Data Abstraction<sup>1</sup>

Mark Hills

`mhills@cs.uiuc.edu`

University of Illinois at Urbana-Champaign

July 8, 2008

---

<sup>1</sup>Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, and Elsa Gunter

Basic Data Abstraction

Abstract Data Types

Parameterized ADTs

# Objectives

By the end of this lecture, you should

- ▶ be familiar with differences in basic data abstraction constructs (tuples, records, etc)
- ▶ understand the programming language concept of *abstract data type*
- ▶ know, at a high level, how several languages handle data abstraction
- ▶ understand parameterized abstract data types

# Basic Data Abstraction Facilities

Most languages include at least one form of *user-defined type*. We've already seen several examples in OCaml:

- ▶ tuples
- ▶ records
- ▶ variant records

Another is the array, available in OCaml but much more common in imperative languages.

# Arrays

Arrays are simply ordered sequences of objects (in the non-OO sense) all of the same type.

```
A[3] := 15;
```

- ▶ often of fixed size
- ▶ generally indexed by integers; occasionally by subrange or enumeration types
- ▶ can be multidimensional, with one index per dimension
- ▶ l-value of any element much be calculated using formula based on index ranges and element sizes
- ▶ goal: l-value calculation should be constant-time
- ▶ heavily used in numeric computation; many languages provide extended support for arrays (slices, array reductions, etc)

# Tuples

Like most arrays, tuples are fixed size. Unlike arrays, tuples allow different types in each position.

$(1, \text{'Bob'}, 3.17) : (Int \times String \times Float)$

- ▶ provide an easy way to group information
- ▶ do not provide a way to label individual items
- ▶ tuples are order dependent – flipping items in a tuple changes the type

# Records

Records provide functionality similar to tuples – we can group multiple items of different types together. However, records additionally provide a way to name each item, making them order independent.

- ▶ generally referred to as records or structures
- ▶ lookup usually as `var-name.field-name`
- ▶ with ability to name fields, provide a way to model data in the problem domain
- ▶ generally not recursive (at least without pointers)

# Variant Records

Variant records are like records, but allow different forms of data to be stored in records of the same type, based (usually) on a constructor or type tag.

```
type 'a mylist = Empty | Cons of 'a * 'a mylist
```

- ▶ provide different groups of data items in the same type
- ▶ uses type constructors in most functional languages
- ▶ variant records, disjoint unions, or just unions in most non-functional languages (imperative, OO)
- ▶ valid items usually determined with either pattern matching (functional) or tag/discriminator (imperative)

# Variant Records Example: Modula-2

```
1 TYPE IntOrReal =  
2   RECORD  
3     CASE IsInt: BOOLEAN OF  
4       TRUE: i : INTEGER |  
5       FALSE: r : REAL  
6     END;  
7   END;  
8   ...  
9   x.IsInt := TRUE;  
10  x.i := 0;
```

# Limitations

While these different abstraction mechanisms are useful, they provide no method for *information hiding* or *encapsulation*.

Consider the C++ code:

```
1 struct Rational {  
2     int numerator;  
3     int denominator;  
4 };  
5 Rational mk_rat (int n,int d) { ... }  
6 Rational add_rat (Rational x, Rational y) { ... }
```

- ▶ Can we use Rationals without knowing the implementation details?
- ▶ Can we access the implementation directly without going through the interface?

# Limitations

The problem with the last example was raised in the second point – structures give us the ability to group data, but the implementation is not hidden.

- ▶ Users can create Rationals without using `mk_rat`
- ▶ Users can directly alter the contents of a rational
- ▶ Because the implementation is not hidden, changes may break user code that relies on it

# Abstract Data Types

How do we fix these limitations? Ideally:

- ▶ we could specify an interface through which the data is manipulated;
- ▶ we could then prevent unsupported changes to the data representation, hiding the representation completely.

To do this, we need **abstract data types**.

## Abstract Data Types: A Simple Example

We actually use ADTs all the time. For instance, look at floating point numbers.

- ▶ We can create variables of floating point type;
- ▶ we can then use system-provided operations on floating point numbers, but we cannot provide our own operations not based on these;
- ▶ we can hide the binary representation of the numbers.

Ideally, our user-defined ADTs would give us the same capabilities.

## ADTs, More Formally

Formally, we want ADTs to have the following two properties:

- ▶ The declarations of the type and the type interface are available in a single syntactic unit, and do not rely on the underlying data representation. Implementations of the operations and users of the ADT (programs declaring variables of the ADT type) may be in separate syntactic units.
- ▶ The underlying data representation is hidden, so only those operations in the interface can be used to alter values of the ADT.

## A Running Example

We will take stacks as our running example. We want a stack to have the following capabilities:

- ▶ `create(stack)`: creates and initializes a stack
- ▶ `destroy(stack)`: destroys a stack, performing any required cleanup
- ▶ `empty(stack)`: returns true if the stack is empty, false otherwise
- ▶ `push(stack, element)`: pushes `element` onto the top of the stack
- ▶ `pop(stack)`: discards the top element, if any, from the stack
- ▶ `top(stack)`: returns a copy of the top element of the stack (but does *not* remove it)

# Modules

For many languages, *modules* provide the method to build an ADT.

- ▶ modules provide a construct for grouping related types, data structures, and operations
- ▶ typically allow for encapsulation (hiding), making them suitable for ADTs
- ▶ act as a unit of scope for names
- ▶ generally include an interface section saying what is exported and, sometimes, what is imported
- ▶ act as units of compilation and code reuse

# ADTs in Ada

The Ada language allows for ADTs, and was one of the earliest languages to include extensive support for them. Several important points:

- ▶ The module mechanism in Ada is called a **package**
- ▶ An ADT is made up of a **specification package**, which provides the interface, and an optional **body package**, which provides the implementation
- ▶ Hidden portions of a package are declared in **private** sections

# Stack ADT in Ada – Specification Package

```
1 package Stack_Pack is
2   type Stack_Type is limited private;
3   Max_Size : constant := 100;
4   function Empty (Stk : in Stack_Type) return Boolean;
5   procedure Push (Stk : in out Stack_Type;
6                 Element : in Integer);
7   procedure Pop (Stk : in out Stack_Type);
8   function Top (Stk : in Stack_Type) return Integer;
9   private
10    type List_Type is array (1..Max_Size) of Integer;
11    type Stack_Type is
12    record
13      List : List_Type;
14      Topsub : Integer range 0..Max_Size := 0;
15    end record;
16 end Stack_Pack;
```

## Several Quick Points

- ▶ The maximum size of the stack has been predefined with `Max_Size`
- ▶ The items in the stack are predefined as integers – we would need another similar type for a stack of strings, or floats, for instance
- ▶ Note the keywords `limited private` – this means that we don't have support for predefined operations, such as equality testing or assignment, which may not make sense anyway

# Stack ADT in Ada – Body Package

```
1 with Ada.Text_IO; use Ada.Text_IO;
2 package body Stack_Pack is
3   function Empty (Stk : in Stack_Type) return Boolean is
4   begin
5     return Stk.Topsub = 0
6   end Empty;
7
8   procedure Push (Stk : in out Stack_Type;
9                 Element : in Integer) is
10  begin
11    if Stk.Topsub >= Max_Size then
12      Put_Line("ERROR - Stack overflow");
13    else
14      Stk.Topsub := Stk.Topsub + 1;
15      Stk.List(Topsub) := Element;
16    end if
17  end Push;
```

# Stack ADT in Ada – Body Package

```
1  procedure Pop(Stk : in out Stack_Type) is
2  begin
3      if Stk.Topsub = 0
4          then Put_Line("ERROR - Stack underflow");
5          else Stk.Topsub := Stk.Topsub - 1;
6      end if;
7  end Pop;
8
9  function Top(Stk : in Stack_Type) return Integer is
10 begin
11     if Stk.Topsub = 0
12         then Put_Line("ERROR - Stack is empty");
13         else return Stk.List(Stk.Topsub);
14     end if;
15 end Top;
16 end Stack_Pack;
```

# Using the Stack

```
1 with Stack_Pack, Ada.Text_IO;  
2 use Stack_Pack, Ada.Text_IO;  
3 procedure Use_Stacks is  
4   Topone : Integer;  
5   Stack : Stack_Type;  
6   begin  
7     Push(Stack,42);  
8     Push(Stack,17);  
9     Topone := Top(Stack);  
10    Pop(Stack);  
11    ...  
12  end Use_Stacks;
```

# ADTs in OO Languages

Abstract Data Types are now core to many imperative languages. However, object-oriented languages also provide support for ADTs, through the *class* mechanism. ADTs can be seen as a simplified version of classes, without the extra features that make a language object-oriented.

# Stack ADT in C++

```
1 #include<iostream.h>
2 class stack {
3 private:
4     int *stackPtr;
5     int maxLen;
6     int topPtr;
7 public:
8     stack() {
9         stackPtr = new int[100];
10        maxLen = 99;
11        topPtr = -1;
12    }
13    ~stack() { delete stackPtr; }
14    int top() { return (stackPtr[topPtr]); }
15    int empty() { return (topPtr == -1); }
```

# Stack ADT in C++

```
1 void push(int number) {
2     if (topPtr == max_len)
3         cerr << "Error in push-stack is full" << endl;
4     else
5         stackPtr[++topPtr] = number;
6 }
7 void pop() {
8     if (topPtr == -1)
9         cerr << "Error in pop-stack is empty" << endl;
10    else
11        topPtr--;
12 }
13};
```

## Using the C++ ADT

```
1 void main() {  
2     int topOne;  
3     stack stk;  
4     stk.push(42);  
5     stk.push(17);  
6     topOne = stk.top();  
7     stk.pop();  
8     ...  
9 }
```

## A Key Difference

The Ada and C++ examples are very similar. However, there is a key difference:

- ▶ In Ada, the package includes the declaration of a type for the ADT
- ▶ In C++, the class *is* the type of the ADT
- ▶ This means, in the C++ example, the type can initialize itself – we don't run the risk of using an uninitialized stack

## But what if I want a float stack?

One problem, noted earlier, is that the type of the stack is declared in the ADT. What if I want a float stack instead? Also, what if I want 101 elements?

- ▶ The Solution: Parameterized ADTs
- ▶ Allows data and type parameters to be specified to make the ADT generic

# Parameterized Stack ADT in Ada – Specification Package

```
1 generic
2   Max_Size : Positive;
3   type Element_Type is private;
4 package Generic_Stack is
5   type Stack_Type is limited private;
6   function Empty (Stk : in Stack_Type) return Boolean;
7   procedure Push (Stk : in out Stack_Type;
8                 Element : in Element_Type);
9   procedure Pop (Stk : in out Stack_Type);
10  function Top (Stk : in Stack_Type) return Element_Type;
```

# Parameterized Stack ADT in Ada – Specification Package

```
1 private
2   type List_Type is array (1..Max_Size) of Element_Type;
3   type Stack_Type is
4     record
5       List : List_Type;
6       Topsub : Integer range 0..Max_Size := 0;
7     end record;
8 end Generic_Stack;
```

# Using the Parameterized ADT

To use the parameterized ADT, the size of the stack, as well as the type of the elements in the stack, both need to be provided:

```
1 package Integer_Stack is new Generic_Stack(100, Integer);  
2 package Float_Stack is new Generic_Stack(500, Float);
```

Note that here we create new types explicitly – we would then use `Integer_Stack` and `Float_Stack` in declarations.

## Parameterized C++ ADTs

C++, along with many other OO languages, provides a generics capability. To duplicate the parameterization we showed above in Ada, we would use a combination of a class constructor with parameters and a *template*.

# Parameterized C++ ADTs

```
1  template <class Type>
2  class Stack {
3  private:
4      Type *stackPtr;
5      ...
6  public:
7      ...
8      stack(int size) {
9          stackPtr = new Type[size];
10         maxLen = size - 1;
11         topPtr = -1;
12     }
13     ...
14 };
```

## Using the Parameterized C++ ADT

Unlike with Ada, in C++ we don't need to declare a new class as an instance of the parameterized class. We can instead just provide the type parameter at the time of variable creation.

```
1 void main() {  
2     int topInt;  
3     double topDouble;  
4     stack<int> intstk(100);  
5     stack<double> doublestk(500);  
6  
7     intstk.push(42);  
8     doublestk.push(3.14);  
9     ...  
10    topInt = intstk.top();  
11    topDouble = doublestk.top();  
12    ...  
13 }
```

# Problems with ADTs

ADTs solve quite a few problems, but can still be misused.

- ▶ ADTs that export types cannot adequately control operations on the variables of those types. For instance, we have no way of requiring that initialization occur before otherwise trying to use a stack variable.
- ▶ ADTs (and modules or classes in general) give us no way to specify the semantics of provided operations. For instance, we cannot specify something like:

```
push(stack,5); pop(stack) = stack
```