Topics

- The Concept of Abstraction
- Introduction to Data Abstraction
- Design Issues for Abstract Data Types
- Language Examples

- Office hours today changed to 1:00–2:00pm

The Concept of Abstraction

- An *abstraction* is a view or representation of an entity that includes only the most significant attributes
- The concept of abstraction is fundamental in programming (and computer science)
- Nearly all programming languages support process abstraction with subprograms
- Nearly all programming languages designed since 1980 support *data abstraction*

Introduction to Data Abstraction

- An *abstract data type* is a user-defined data type that satisfies the following two conditions:
  - The representation of objects of the type is hidden from the program units that use these objects, so the only operations possible are those provided in the type's definition
  - The declarations of the type and the protocols of the operations on objects of the type are contained in a single syntactic unit. Other program units are allowed to create variables of the defined type.

Advantages of Data Abstraction

- Advantages the first condition
  - Reliability--by hiding the data representations, user code cannot directly access the underlying representations of objects, allowing the representation to be changed without affecting user code
  - Reduces the range of code and variables of which the programmer must be aware
  - Name conflicts are less likely
- Advantages of the second condition
  - Provides a method of program organization
  - Aids modifiability (everything associated with a data structure is together)
  - Separate compilation

Language Requirements for ADTs

- A syntactic unit in which to encapsulate the type definition
- A method of making type names and subprogram headers visible to clients, while hiding actual definitions
- Some primitive operations must be built into the language processor

Design Issues

- Can abstract types be parameterized?
- What access controls are provided?
- Is the specification of the type physically separate from its implementation?

Language Examples: C++

- Based on C `struct` type and Simula 67 classes
- The class is the encapsulation device
- A class is a type
- All of the class instances of a class share a single copy of the member functions
- Each instance of a class has its own copy of the class data members
- Instances can be static, stack dynamic, or heap dynamic

Language Examples: C++ (continued)

- Information Hiding
  - *Private* clause for hidden entities
  - *Public* clause for interface entities
  - *Protected* clause for inheritance (Chapter 12)

Language Examples: C++ (continued)

- Constructors:
  - Functions to initialize the data members of instances (they *do not* create the objects)
  - May also allocate storage if part of the object is heap-dynamic
  - Can include parameters to provide parameterization of the objects
  - Implicitly called when an instance is created
  - Name is the same as the class name

Language Examples: C++ (continued)

- Destructors
  - Functions to cleanup after an instance is destroyed; usually just to reclaim heap storage
  - Implicitly called when the object's lifetime ends
  - Name is the class name, preceded by a tilde (~)

An Example in C++

```
class Stack {
   private:
        int *stackPtr, maxLen, topPtr;
   public:
        Stack() { // a constructor
               stackPtr = new int [100];
               maxLen = 99;
               topPtr = -1;
        };
        ~Stack () {delete [] stackPtr;};
        void push (int number) {
           if (topSub == maxLen)
             cerr << "Error in push - stack is full\n";
           else stackPtr[++topSub] = number;
        };
        void pop () {…};
        int top () {…};
        int empty () {…};
}
```

A `Stack` class header file

```
// Stack.h - the header file for the Stack class
#include <iostream.h>
class Stack {
private: //** These members are visible only to other
//** members and friends (see Section 11.6.4)
  int *stackPtr;
  int maxLen;
  int topPtr;
public: //** These members are visible to clients
  Stack(); //** A constructor
  ~Stack(); //** A destructor
  void push(int);
  void pop();
  int top();
  int empty();
}
```

The code file for `Stack`

```
// Stack.cpp - the implementation file for the Stack class
#include <iostream.h>
#include "Stack.h"
using std::cout;
Stack::Stack() { //** A constructor
  stackPtr = new int [100];
  maxLen = 99;
  topPtr = -1;
}
Stack::~Stack() {delete [] stackPtr;}; //** A destructor
void Stack::push(int number) {
  if (topPtr == maxLen)
  cerr << "Error in push--stack is full\n";
  else stackPtr[++topPtr] = number;
}
...
```

Language Examples: Java

- Similar to C++, except:
  - All user-defined types are classes
  - All objects are allocated from the heap and accessed through reference variables
  - Individual entities in classes have access control modifiers (private or public), rather than clauses
  - All entities in all classes in a package that do not have access control modifiers are visible throughout the package
  - Declared and defined in a single syntactic unit
  - Implicit garbage collection of all objects

An Example in Java

```
class StackClass {
  private:
      private int [] stackRef;
      private int maxLen, topIndex;
      public StackClass() { // a constructor
            stackRef = new int [100];
            maxLen = 99;
            topPtr = -1;
      };
      public void push (int num) {…};
      public void pop () {…};
      public int top () {…};
      public boolean empty () {…};
}
```

Language Examples: C#

- Based on C++ and Java
- Adds two access modifiers, *internal* and *protected internal*
- All class instances are heap dynamic
- Default constructors are available for all classes
- Garbage collection is used for most heap objects, so destructors are rarely used
- **`structs`** are lightweight classes that do not support inheritance

Language Examples: C# (continued)

- Common solution for access to data members: accessor methods (getter and setter)
- C# provides *properties* as a way of implementing getters and setters without requiring explicit method calls

C# Property Example

```
public class Weather {
   public int DegreeDays { //** DegreeDays is a property
      get {return degreeDays;}
      set {
        if (value < 0 || value > 30)
          Console.WriteLine(
              "Value is out of range: {0}", value);
        else degreeDays = value;}
   }
   private int degreeDays;
   ...
   }
...
Weather w = new Weather();
int degreeDaysToday, oldDegreeDays;
...
w.DegreeDays = degreeDaysToday;
...
oldDegreeDays = w.DegreeDays;
```

Abstract Data Types in Ruby

- Encapsulation construct is the class
- Local variables have “normal” names
- Instance variable names begin with “at” signs (`@`)
- Class variable names begin with two “at” signs (`@@`)
- Instance methods have the syntax of Ruby functions (`def … end`)
- Constructors are named `initialize` (only one per class)—implicitly called when `new` is called
- Class members can be marked private or public, with public being the default
- Classes are dynamic

Abstract Data Types in Ruby (continued)

```
class StackClass
   def initialize
    @stackRef = Array.new
    @maxLen = 100
    @topIndex = -1
   end

   def push(number)
     if @topIndex == @maxLen
       puts "Error in push - stack is full"
     else
       @topIndex = @topIndex + 1
       @stackRef[@topIndex] = number
     end
   end
   def pop … end
   def top … end
   def empty … end
end
```