Chapter 11

Abstract Data Types and Encapsulation Concepts
Chapter 11 Topics

- The Concept of Abstraction
- Introduction to Data Abstraction
- Language Examples
- Parameterized Abstract Data Types
- Encapsulation Constructs
- Naming Encapsulations
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, and operations on, objects of the type are defined in a single syntactic unit
  - 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
Advantages of Data Abstraction

- **Advantage of the first condition**
  - Program organization, modifiability (everything associated with a data structure is together), and separate compilation

- **Advantage the second condition**
  - Reliability--by hiding the data representations, user code cannot directly access objects of the type or depend on the representation, allowing the representation to be changed without affecting user code
Language Examples: C++

- The class is the encapsulation device
  - **Data member**: the data defined in a class
  - **Member function**: functions defined in a class
    - Both header and body are in the class -> inline (in caller’s code)
    - header appears in class, and the definition is outside class
  - 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 stack dynamic or heap dynamic**
  - Stack dynamic: reference using value variables
    created by elaborating an object declaration
  - Heap dynamic: reference through pointer
    **new and delete** operators to manage heap
Language Examples: C++ (continued)

- Information Hiding
  - *Private* clause for hidden entities
    - visible only to members within this class, or friends.
    - *not* by inheritors,
    - *not* by other classes in the package.
  - *Protected* clause for inheritance
    - Visible to classes that inherit the class.
    - Visible to all classes in the package (Java)
  - *Public* clause for interface entities
    - Visible to all classes
Friend functions or classes - to provide access to private members to some other units or functions

Example

```cpp
class Matrix;

class Vector {
    friend Vector multiply( Matrix&, Vector& );
    ...
}

Class Matrix {
    friend vector multiply( Matrix&, Vector& );
    . . .
}

Vector multiply( Matrix& m, Vector& v) {...}
```
Language Examples: C++ (continued)

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

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

- **Both have no return type and no return is used**
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 num) {...};
    void pop () {...};
    int top () {...};
    int empty () {...};
}

stack stk;
int topOne;
stk.push(43);
stk.push(7);
topOne = stk.top();
stk.pop();
...

Copyright © 2006 Addison-Wesley. All rights reserved.
Language Examples: Java

- Similar to C++, except:
  - All user-defined types are classes (no `struct` in Java)
  - All objects are allocated from the heap and accessed through reference variables
  - Methods must be defined in classes (C++ allows them to be declared inside class, but defined outside)
  - Individual entities in classes have access control modifiers (private or public), rather than clauses
  - No destructor because of implicit garbage collection
  - Java has a second scoping mechanism, package scope, which can be used in place of friends
    - All entities in all classes in a package that do not have access control modifiers are visible throughout the package
An Example in Java

class StackClass {
    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
- private, public, protected, *internal* and *protected internal*
  - *Internal*: visible to all the classes within the same assembly
  - *Protected internal*: only derived classes or classes within the same assembly can access it
- All class instances are heap dynamic (same to java)
- Default constructors are available for all classes
  - Any instance that is not initialized in user defined constructor is assigned a value by default constructor.
  - it provides typical initial values: 0 for *int*, false for *boolean*
- Garbage collection is used for most heap objects, so destructors are rarely used
- *struct* are lightweight classes that do not support inheritance
Language Examples: C# (continued)

- **Accessor methods – getter and setter**
  - The way to access data members but not making them public

- **Why not making data public for access directly?**
  - Read-only can be provided: if no setter is defined
  - Constraints can be included in setters (e.g., if value should be restricted to a particular range, setter can enforce that.)
  - The actual implementation of data member can be changed without affecting the clients if use getters and setters.

- **C# provides properties as a way of implementing getters and setters without requiring explicit method calls**
C# Property Example

```csharp
public class Weather {  
    public int DegreeDays { // it is a property
        get {return degreeDays;}  
        set {degreeDays = value;}  
    }  
    private int degreeDays;  
    ...
}  
...

Weather w = new Weather();
int degreeDaysToday, oldDegreeDays;
...

w.DegreeDays = degreeDaysToday;
oldDegreeDays = w.DegreeDays;
```

In client code, the degreeDays is treated as if it was a public data member, although access to it is only through the property -- DegreeDays
Parameterized Abstract Data Types

- Parameterized ADTs allow designing an ADT that can store any type elements
- Also known as generic classes
- C++ and Ada provide support for parameterized ADTs
- Java 5.0 provides a restricted form of parameterized ADTs
- C# does not currently support parameterized classes
Parameterized ADTs in C++

- The element type of the stack can be made generic by making the class a `template` class.
- The stack can be generic in size by writing `parameterized` constructor functions.

```cpp
template <class type>
class stack {
  private:
    type *stk_ptr;
    int max_len, top;
  public:
    stack (int size) {
      stk_ptr = new int [size];
      max_len = size - 1;
      top = -1;
    }
    ...
} stack stk(100);
```
Encapsulation Constructs

- Large programs have two special needs:
  - Some means of organization, other than simply division into subprograms
  - Some means of partial compilation (compilation units that are smaller than the whole program)
- Obvious solution: a grouping of subprograms that are logically related into a unit that can be separately compiled (compilation units)
- Such collections are called encapsulation
Encapsulation in C

- Files containing one or more subprograms can be independently compiled
  - The interface is placed in a header file
  - The header file, in source form, and the compiled version of the implementation file (called library) are furnished to clients.
  - `#include preprocessor specification`

Encapsulation in C++

- Similar to C
Assembly: it is a collection of files that appear to be a single dynamic link library or an executable

A DLL is a collection of classes and methods that are individually linked to an executing program

- Only the parts that are actually used are ever loaded and linked to the program

C# has an access modifier called internal; an internal member of a class is visible to all classes in the assembly in which it appears

In .NET world, the assembly is the basic unit of deployment of software.

Assemblies can be private (they are available to just one application), or public (any application can use them).
Naming Encapsulations

- Large programs define many global names; need a way to divide into logical groupings
- A *naming encapsulation* is used to create a new scope for names
- **C++ Namespaces**
  - Each library can create its own namespace to prevent its names from conflicting with the names defined outside this namespace.
  - E.g., `namespace MyStack{
    ...
  }
  MyStack::topPtr // use scope operator ::
  using MyStack::topPtr; // to qualify individual name
  using namespace MyStack; // to qualify all names from a namespace

- **C# also includes namespaces**
Naming Encapsulations (continued)

- **Java Packages**
  - Packages can contain more than one class definition
    - *public, protected* entities or entities *without access modifier* are visible throughout the same package
  - Java has no friend functions or friend classes.
  - Clients of a package can use fully qualified name or use the *import* declaration
    - e.g. `package myStack;`  
      `topPtr = ...`  
      `x = myStack.topPtr;`  
      `import myStack.*;`  
      `x = topPtr`

Clients of a package can use fully qualified name or use the *import* declaration.
Summary

- The concept of ADTs and their use in program design was a milestone in the development of languages.
- Two primary features of ADTs are the packaging of data with their associated operations and information hiding.
- Ada provides packages that simulate ADTs.
- C++ data abstraction is provided by classes.
- Java’s data abstraction is similar to C++.
- Ada and C++ allow parameterized ADTs.
- C++, C#, Java, and Ada provide naming encapsulation.