Chapter 11

Abstract Data Types and Encapsulation Concepts

Chapter 11 Topics

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

Stack implementation as data type

```
#include <stdio.h>
#include<ctype.h>
# define MAXSIZE 200

int stack[MAXSIZE];
int top;//index pointing to the
   top of stack
void main()
{
    void push(int);
    int pop();
    int topOne;
    push(5);
    push(10);
    topOne=pop();
    printf("%d",topOne);
}
```

```
void push(int y)  {
     if(top>MAXSIZE)  {
         printf("STACK FULL");
    }
    else  {
     top++;
     stack[top]=y;   }
}
int pop()  {
    int a;
    if(top<=0) {
     printf("STACK EMPTY");
     return 0; }
    else  {
     a=stack[top];
     top--;   }
    return(a);
}
```

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

Stack implementation as abstract data type

```
class stack {
   public:
        stack();
        ~stack(void);
        void push(int num);    // Adds Item to the top
        int pop(void);         // Returns Item from the top
  protected:
        int data[MaxSize];     // The actual Data array
}
void main() {
        int topOne;
        stack stk;
        stk.push(5);
        stk.push(10);
        topOne = stk.pop();
}
```

Design Issues

- A syntactic unit to define an ADT
- Built-in operations
  - Assignment
  - Comparison
- Common operations
  - Iterators
  - Accessors
  - Constructors
  - Destructors
- Parameterized ADTs

Language Examples: C++

- Based on C `struct` type and Simula 67 classes
- The class is the encapsulation device
- Data members are the data defined in a class
- Member functions are the functions (methods) defined in a class.
- A member function can be defined in 2 ways
  - the complete definition: header and body ->inlined
  - only its header -> seperately compiled

Language Examples: C++ (continued)

- 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
- Information Hiding
  - *Private* clause for hidden entities
  - *Public* clause for interface entities
  - *Protected* clause for inheritance

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
  - Can be explicitly called
  - 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
  - Can be explicitly called
  - 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 num) {…};
     void pop () {…};
     int top () {…};
     int empty () {…};
}
```

```
void main( )
{
     int topOne;
     stack stk;
     stk.push(42);
     stk.push(17);
     topOne = stk.top();
     stk.pop();
     …
}
```

Language Examples: C++ (continued)

- Friend functions or classes – to provide access to private members to some unrelated units or functions
  - Necessary in C++

```
void PrintWeather(Temp &cTemp, Humid &cHumid)
{
  std::cout << "The temperature is " <<
  cTemp.m_nTemp <<          " and the humidity is " <<
  cHumid.m_nHumid << std::endl;
}
```

Friend function in C++

```
class Humid;
class Temp {
    private:
        int m_nTemp;
   public:
        Temp(int nTemp) { m_nTemp = nTemp; }
        friend void PrintWeather(Temp &cTemp,  Humid &cHumid);
};
class Humid {
   private:
        int m_nHumid;
   public:
        Humid(int nHumid) { m_nHumid = nHumid; }
        friend void PrintWeather(Temp &cTemp, Humid &cHumid);};
```

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
  - 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:
      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 () {…};
}
```

An Example in Java

```
public class TestStack {
  public static void main(String[] args) {
     StackClass myStack = new StackClass();
     myStack.push(42);
     myStack.push(17);
     System.out.println("top of stack is:"+ myStack.top());
     myStack.pop();
     …
  }
}
```

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
- `struct`s are lightweight classes that do not support inheritance

Language Examples: C# (continued)

- Common solution to need 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 {degreeDays = value;}
   }
   private int degreeDays;
   ...
   }
...
Weather w = new Weather();
int degreeDaysToday, oldDegreeDays;
...
w.DegreeDays = degreeDaysToday;
...
oldDegreeDays = w.DegreeDays;
```

C# example

```
class CelsiusToFahrenhit
{
   private double celsius;
   public double celsius  {
         set { celsius = value; }
   }
   public double ToFahrenhit()  {
        return (celsius*9/5)+32;
   }
}
static void Main(string[] args)
 {
     CelsiusToFahrenhit myCTOF = new CelsiusToFahrenhit();
     myCTOF.Celsius = 37;
     Console.WriteLine(myCTOF.ToFahrenhit());
}
```

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++

- Classes can be somewhat generic by writing parameterized constructor functions

```
template <class Type>
class stack {
  private:
      Type *stackPtr;
      …
  public:
      stack (int size) {
         stk_ptr = new Type [size];
         max_len = size - 1;
         top = -1;
      }
 …
}
stack<int> 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*
- Problem: the linker does not check types between a header and associated implementation
- `#include` preprocessor specification

Encapsulation in C++

- Similar to C
- Addition of *friend* functions that have access to private members of the friend class

C# Assemblies

- A collection of files that appear to be a single dynamic link library or executable
- Each file contains a module that can be separately compiled
- A DLL is a collection of classes and methods that are individually linked to an executing 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

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
  - Can place each library in its own namespace and qualify names used outside with the namespace
  - C# also includes namespaces

Namespaces in C++

```
Syntax
namespace identifier
{
     entities
}
namespace::entities

Example:
namespace myNamespace
{
    int a, b;
}
namespace::a
namespace::b
```

```
#include <iostream>
using namespace std;
namespace first
{
     int var = 5;
}
namespace second
{
    double var = 3.1416;
}
int main () {
    cout << first::var << endl;
    cout << second::var <<
   endl;
    return 0;
}
```

C++ example

```
#include <iostream>


int main() {
    float celsius;
    float fahrenheit;
    std::cout << "Enter Celsius
    std::cin >> celsius;
    fahrenheit = 1.8 * celsius + 32;
    std::cout << "Fahrenheit = " << fahrenheit <<
    std::endl;
}
```

Naming Encapsulations (continued)

- Java Packages
  - Packages can contain more than one class definition; classes in a package are *partial* friends
  - 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
- C++ data abstraction is provided by classes
- Java's data abstraction is similar to C++
- C++ allow parameterized ADTs
- C++, C#, and Java provide naming encapsulation