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*.
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 objects of the type or depend on the representation, 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)
• 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
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 (~)
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 () {...};

}
// 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;
}
...
• Friend functions or classes – to provide access to private members to some unrelated units or functions
  – Necessary in C++
• **Interface containers**
  ```objective-c
@interface class-name: parent-class {
  instance variable declarations
}
  method prototypes
@end
• **Implementation containers**
  ```objective-c
@implemention class-name
  method definitions
@end
• **Classes are types**
• Method prototypes form
  (+ | −) (return-type) method-name [: (formal-parameters)];
  − Plus indicates a class method
  − Minus indicates an instance method
  − The colon and the parentheses are not included when there are no parameters
  − Parameter list format is different
    − If there is one parameter (name is meth1:)
      -(void) meth1: (int) x;
    − For two parameters
      -(int) meth2: (int) x second: (float) y;
    − The name of the method is meth2::
• Method call syntax

\[ \text{object-name method-name} ; \]

Examples:

\[ \text{myAdder add1: 7} ; \]
\[ \text{myAdder add1: 7: 5: 3} ; \]

- For the method:

\[-(\text{int)} \text{meth2: (int) x second: (float) y} ; \]

the call would be like the following:

\[ \text{myObject meth2: 7 second: 3.2} ; \]
• Constructors are called *initializers* – all they do is initialize variables
  – Initializers can have any name – they are always called explicitly
  – Initializers always return *self*

• Objects are created by calling *alloc* and the constructor

  ```
  Adder *myAdder = [[Adder alloc] init];
  ```

• All class instances are heap dynamic
Language Examples – Objective-C (continued)

• To import standard prototypes (e.g., i/o)
  
    #import <Foundation/Foundation.h>

• The first thing a program must do is allocate and initialize a pool of storage for its data (pool’s variable is pool in this case)

    NSAutoreleasePool * pool = 
    [[NSAutoreleasePool alloc] init];

• At the end of the program, the pool is released with:

    [pool drain];
• Information Hiding
  - The directives @private and @public are used to specify the access of instance variables.
  - The default access is protected (private in C++)
  - There is no way to restrict access to methods
  - The name of a getter method is always the name of the instance variable
  - The name of a setter method is always the word set with the capitalized variable’s name attached
  - If the getter and setter for a variable does not impose any constraints, they can be implicitly generated (called properties)
// stack.m – interface and implementation for a simple stack
#import <Foundation/Foundation.h>
@interface Stack: NSObject {
  int stackArray[100], stackPtr,maxLen, topSub;
}
-(void) push: (int) number;
-(void) pop;
-(int) top;
-(int) empty;
@end

@implementation Stack
-(Stack *) initWith {
  maxLen = 100;
  topSub = -1;
  stackPtr = stackArray;
  return self;
}

Copyright © 2015 Pearson. All rights reserved.
// stack.m - continued

-(void) push: (int) number {
    if (topSub == maxLen)
        NSLog(@"Error in push - stack is full");
    else
        stackPtr[++topSub] = number;
...
}
• **An example use of** `stack.m`

  - Placed in the `@implementation of stack.m`

```c
int main (int argc, char *argv[]) {
    int temp;
    NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];
    Stack *myStack = [[Stack alloc] initWith];
    [myStack push: 5];
    [myStack push: 3];
    temp = [myStack top];
    NSLog(@"Top element is: %i", temp);
    [myStack pop];
    temp = [myStack top];
    NSLog(@"Top element is: %i", temp);
    temp = [myStack top];
    myStack pop];
    [myStack release];
    [pool drain];
    return 0;
}
```
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
  - Implicit garbage collection of all objects
  - 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

```java
class StackClass {
    private:
        private int[] *stackRef;
        private int[] maxLength, topIndex;
    public StackClass() { // a constructor
        stackRef = new int[100];
        maxLength = 99;
        topIndex = -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 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

```csharp
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
  – If more constructors are needed, they must have different names and they must explicitly call new
• Class members can be marked private or public, with public being the default
• Classes are dynamic
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
Parameterized Abstract Data Types

- Parameterized ADTs allow designing an ADT that can store any type elements – only an issue for static typed languages
- Also known as generic classes
- C++, Java 5.0, and C# 2005 provide support for parameterized ADTs
Parameterized ADTs in C++

- Classes can be somewhat generic by writing parameterized constructor functions

```cpp
Stack (int size) {
    stk_ptr = new int [size];
    max_len = size - 1;
    top = -1;
};
```

A declaration of a stack object:

```cpp
Stack stk(150);
```
Parameterized ADTs in C++ (continued)

- The stack element type can be parameterized by making the class a templated class

```
template <class Type>
class Stack {
  private:
    Type *stackPtr;
    const int maxLen;
    int topPtr;
  public:
    Stack() {  // Constructor for 100 elements
      stackPtr = new Type[100];
      maxLen = 99;
      topPtr = -1;
    }
    Stack(int size) {  // Constructor for a given number
      stackPtr = new Type[size];
      maxLen = size - 1;
      topPtr = -1;
    }
...

  - Instantiation: Stack<int> myIntStack;
```
Parameterized Classes in Java 5.0

- Generic parameters must be classes
- Most common generic types are the collection types, such as `LinkedList` and `ArrayList`
- Eliminate the need to cast objects that are removed
- Eliminate the problem of having multiple types in a structure
- Users can define generic classes
- Generic collection classes cannot store primitives
- Indexing is not supported
- Example of the use of a predefined generic class:

```java
ArrayList<Integer> myArray = new ArrayList<Integer>();
myArray.add(0, 47); // Put an element with subscript 0 in it
```
Parameterized Classes in Java 5.0 (continued)

```java
import java.util.*;
public class Stack2<T> {
    private ArrayList<T> stackRef;
    private int maxLen;
    public Stack2() {
        stackRef = new ArrayList<T> ();
        maxLen = 99;
    }
    public void push(T newValue) {
        if (stackRef.size() == maxLen)
            System.out.println("Error in push - stack is full");
        else
            stackRef.add(newValue);
```

- **Instantiation:** `Stack2<String> myStack = new Stack2<String>() ;`
Parameterized Classes in C# 2005

- Similar to those of Java 5.0, except no wildcard classes
- Predefined for Array, List, Stack, Queue, and Dictionary
- Elements of parameterized structures can be accessed through indexing
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*
Nested Subprograms

- Organizing programs by nesting subprogram definitions inside the logically larger subprograms that use them
- Nested subprograms are supported in Python, JavaScript, and Ruby
Encapsulation in C

- Files containing one or more subprograms can be independently compiled
- The interface is placed in a header file
- Problem 1: the linker does not check types between a header and associated implementation
- Problem 2: the inherent problems with pointers
- \#include preprocessor specification – used to include header files in applications
Encapsulation in C++

• Can define header and code files, similar to those of C
• Or, classes can be used for encapsulation
  – The class is used as the interface (prototypes)
  – The member definitions are defined in a separate file
• *Friends* provide a way to grant access to private members of a class
C# Assemblies

- A collection of files that appears to application programs 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
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.
• *Ruby Modules:*
  – Ruby classes are name encapsulations, but Ruby also has modules
  – Typically encapsulate collections of constants and methods
  – Modules cannot be instantiated or subclassed, and they cannot define variables
  – Methods defined in a module must include the module’s name
  – Access to the contents of a module is requested with the `require` method
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++.
• C++, Java 5.0, and C# 2005 support parameterized ADTs.
• C++, C#, Java, and Ruby provide naming encapsulations.