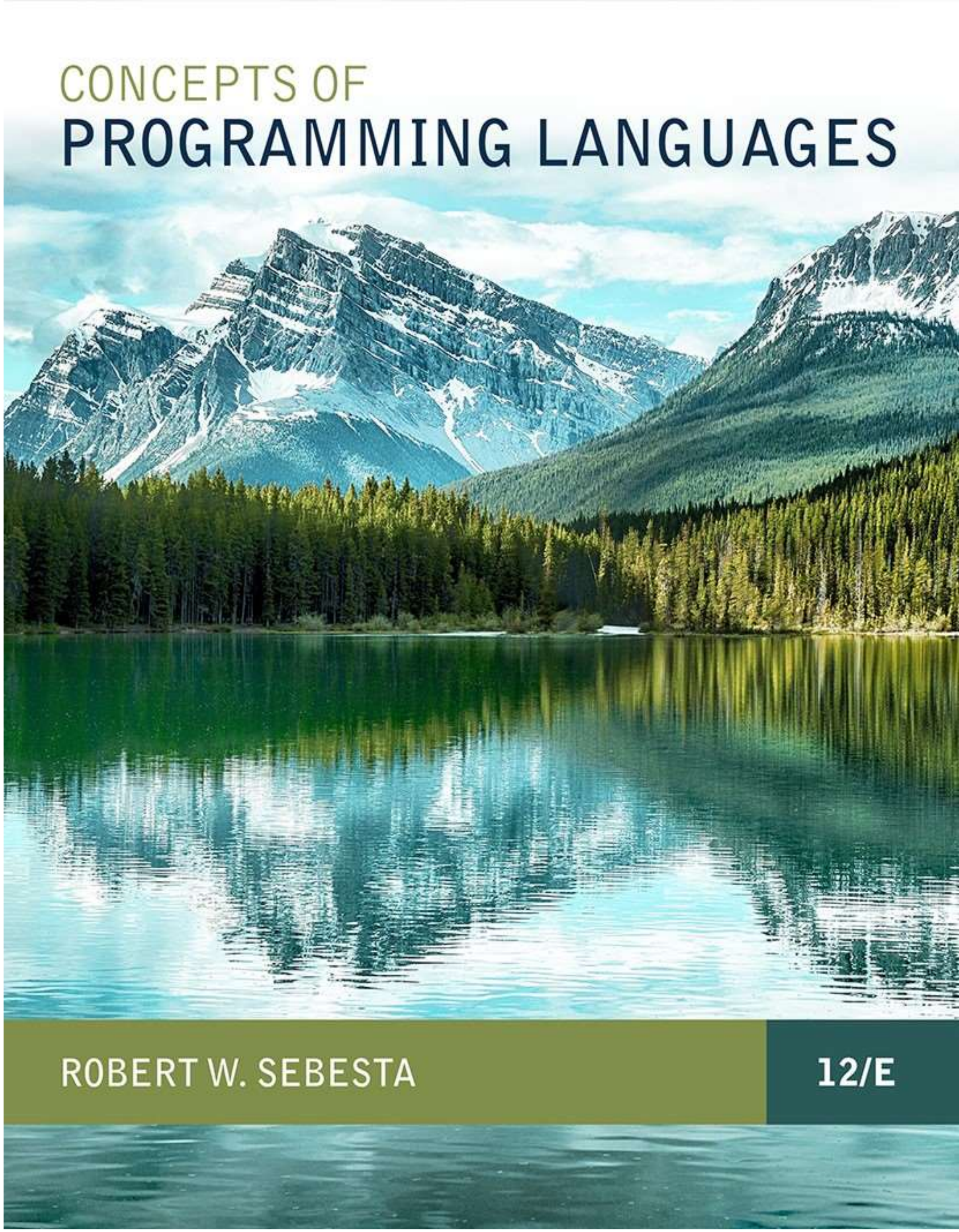
Lecture 8
(Chapter 11)

Abstract Data Types and Encapsulation Concepts



ISBN 0-321-49362-1

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*
  - *Ada is the early advocator of*

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 on objects 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.
    - Single syntactic unit: e.g. class, unit, module, …
    - Other program units are allowed to create variables or objects 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++

- The *class* is the encapsulation device
- A class is a type
  - All of the 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
- Information Hiding
  - *private* clause for hidden entities
  - *public* clause for interface entities
  - *protected* clause for inheritance (Lecture 12)
  - *friend* functions or classes – to provide access to private members to some unrelated units or functions

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
  - Name is the same as the class name
  - Can include parameters to provide parameterization of the objects
  - Implicitly called when an instance is created
  - Can be explicitly called
- Destructors
  - Functions to cleanup after an instance is destroyed
    - usually just to reclaim heap storage
  - Name is the class name, preceded by a tilde (~)
  - Implicitly called when the object's lifetime ends
  - Can be explicitly called

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;};    //destructor
        void push (int number) {                    //member function
                if (topSub == maxLen)
                        cerr << "Error in push - stack is full\n";
                else stackPtr[++topSub] = number;
      };
        void pop () {…};      //more member functions
        int top () {…};
        int empty () {…};
} //end of class Stack
```

The class definition could also be written into a header file and a code file as shown in next two slides.

A Stack class header file

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

```
Stack s;  //a stack-dynamic object
Stack *ps = new Stack();
          //a heap-dynamic object
```

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() {                          //implementation of constructor
  stackPtr = new int [100];               //allocation of heap-dynamic storage
  maxLen = 99;
  topPtr = -1;
}
Stack::~Stack() {                         //implementation of destructor
     delete [] stackPtr;                  //deallocation of heap-dynamic stroage
};
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

```
public void push (…) { … }
public void pop() { …}
```

  - 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

```
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 () {…};
}
```

```
StackClass myS
   = new StackClass();

//myS is an object reference.
//The StackClass object is
//explicitly (via new)
//allocated on heap
```

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 (since Java 5.0), C# etc. provide support for parameterized ADTs

Generic classes in C++

- Classes can be somewhat generic by writing parameterized constructor functions

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

A declaration of stack objects:

```
Stack smallStk (10), largeStk (150);
```

// The above Stack definition provided limited “generic” feature
// i.e. “generic” in terms of stack size

- Demand for more powerful “generic” definitions
  - What about a stack of integer, a stack of double, a stack of student objects?
  - Do we have to define each of them separately, or could have a generic definition of stack?

C++: template classes

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

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

- Instantiation:

```
Stack<int> myIntStack;
Stack<double> myDblStack;
```

Parameterized Classes in Java 5.0

- Generic parameters must be classes
- Most common generic types are the collection types, such as `LinkedList` and `ArrayList`
- Users can define generic classes
- Generic collection classes cannot store primitives
- Indexing is not supported
- Example of the use of a predefined generic class:

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

Java: user-defined Parameterized Classes

```
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> ();`

Encapsulation

- In OOP, encapsulation refers to the bundling of data with the methods that operate on that data, or the restricting of direct access to some of an object's components.
- ADTs can be used as encapsulation constructs
- However, some languages provide additional encapsulation constructs to support
  - Better program organization
  - Partial/separate compilation
  - …

Abstraction vs. Encapsulation

Abstraction — Implementation hiding.

Hide unwanted details

Encapsulation — Information hiding.

Hide the data to protect from outside

private, public, protected

ADT: both abstraction and encapsulation

Encapsulation Constructs

- Large programs have two special needs:
  - Some means of organization, other than simply division into subprograms
    - nested subprograms is one of the organization means
  - 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*
  - e.g. C# assembly

Encapsulation in C

- Files containing one or more subprograms can be independently compiled
- The interface is placed in a *header file*
- Problems
  - the linker does not check types between a header and associated implementation
  - the inherent problems with pointers
- `#include` preprocessor specification – used to include header files in applications

Example in C

```
//factorial.h – the header file
    int factorial (int n) {
        int result=1, iter;
        for (iter=1; iter<=n; iter++)
                result *= iter;
        return result;
    }
//myProg.c   – client program that uses factorial
        #include "factorial.h"
        int main () {
                int f5 = factorial (5);
                …
        }
```

A header file may include declarations only, as commonly used.

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
  - Example: stack.h and stack.cpp (next slide)

Reference: [CPP Forum]

Here is stack.h

```cpp
#include    <iostream>
#include    <iomanip>
using namespace std;


class Stack
{
public:
        Stack(int);              // constructor
        Stack(const Stack &);  // copy constructor
        ~Stack();                 // destructor

        void push(int);          // push an int into a Stack
        int  pop();               // pop an int from a Stack

        bool empty() const;      // is the Stack empty?
        bool full() const;       // is the Stack full?

        int capacity() const;    // capacity of the stack
        int size() const;        // current size of the stack

        friend ostream &operator <<(ostream &, const Stack &);

private:
        int *stack;                // pointer to local stack of ints

        int top;                   // top of stack (next avail. location)
        int maxsize;               // max size of the stack
};
```

Stack.cpp

```
#include    <iostream>
#include    <iomanip>
using namespace std;

#include    "Stack.h"

Stack::Stack(const Stack &s)
{
        maxsize = s.maxsize;
        // allocate stack for left side object
        stack = new int[maxsize];
        // now copy right side object to left side object
        for (top = 0; top < maxsize; ++top)
        {
                stack[top] = s.stack[top];
        }
}


void Stack::push(int i)   // push an int into a Stack
{
        if (!full())
        {
                cout << "push( " << i << " )\t at location "
                        << top << '\n';
                stack[top] = i;
                ++top;              // advance to the next empty location
        }
}

int Stack::pop()            // pop an int from a Stack
{
        if (empty())
        {
                return -1;          // stack is empty; return -1
        }
        else
        {
                --top;
                cout << "pop( ) " << stack[top]
                        << " at location " << top << '\n';
                // return item at top of the stack
                return stack[top];
        }
}

ostream &operator <<(ostream &out, const Stack &s)
{
        for (int i = s.size() - 1; i >= 0; --i)
        {
                out << setw(3) << i << setw(5) << s.stack[i] << '\n';
        }
```

Here is the main.cpp

```cpp
#include    "Stack.h"

int main()
{
        Stack s(2);             // create stack with space for 3 ints
        s.push(10);
        s.push(20);
        s.push(30);             // pushing one int too many

        cout << "stack s size     = " << s.size() << '\n';
        cout << "stack s capacity = " << s.capacity() << '\n';
        cout << "stack s:\n" << s << '\n';

        {                           // start a new block
                Stack t(s);         // t contains a copy of s
                cout << "\nstack t:\n" << t << '\n';
                cout << "pop one element: " << t.pop() << '\n';

                cout << "stack t size     = " << t.size() << '\n';
                cout << "stack t capacity = " << t.capacity() << '\
                cout << "\nstack t:\n" << t << '\n';
        }                           // end the new block

        cout << "\nstack s:\n" << s << '\n';
        cout << "pop one element: " << s.pop() << '\n';
        cout << "pop one element: " << s.pop() << '\n';
        // poping one element too many
        cout << "pop one element too many: " << s.pop() << '\n';
        cout << "stack s size     = " << s.size() << '\n';
        cout << "stack s capacity = " << s.capacity() << '\n';
        // print an empty stack s
        cout << "\nstack s:\n" << s << '\n';

        return 0;
}
```

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++ and C# Namespaces
  - Can place each library in its own namespace and qualify names used outside with the namespace
- 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 and so on …

Example: C++ Namespace

```
#include <iostream>
using namespace std;

namespace first  {
  int x = 5;
  int y = 10;
}

namespace second  {
  double x = 3.1416;
  double y = 2.7183;
}
```

```
int main () {

  using first::x;
  using second::y;

  cout << x << endl;
  cout << y << endl;
  cout << first::y << endl;
  cout << second::x << endl;

return 0;
}
```

Summary

- The concept of ADTs and their use in program design was a milestone in the development of languages
  - ADTs also serve as an essential component of OOP
- Two primary features of ADTs are the packaging of data with their associated operations and information hiding
- Many different ways of supporting ADTs
  - C++ data abstraction is provided by classes
  - Java's data abstraction is similar to C++
  - C++, Java, and C# support parameterized ADTs
- Encapsulations support larger problem development
  - C++, C#, Java, and Ruby provide naming encapsulations