Object Oriented Programming

Week 1 Part 1
Data Abstraction and Encapsulation
Lecture

- Object Orient Languages and Paradigm
- Abstraction
- Design Issues
- Language Examples
- Parameterized Abstract Data Types
- Encapsulation
The Object Oriented Languages and Paradigm
Object Oriented Languages

• Object Oriented Programming is a Programming Paradigm
  – Java is one example of an Object Oriented Language

• Other Object Oriented Programming Languages
  – Smalltalk: the first OO language
  – JavaScript: Web oriented OO language
  – CLOS: OO extension for Lisp
  – C++: OO extension for C
  – Many more
Object Oriented Paradigm

- Think of real-world things and their interactions.
  - Objects contain
    - Properties: i.e., data
    - Behaviors: i.e., methods (similar to functions)

- Relationships between things
  - A thing may have distinct parts
    - E.g., A airplane has wings and a motor
  - A thing may be a kind of thing
    - An airplane is a kind of vehicle
    - A jet airplane is a kind of airplane
Why OOP

- You can use intuitions about the real world.
  - E.g., an airplane flies, a train does not.
- You can make object to take action
- The same action may be different for different objects
  - e.g. airplane.board(); train.board()
- Objects inherit traits from more abstract objects
  - E.g., vehicle.num_seats → train.num_seats; airplane.num_seats
- Information can be hidden
Abstraction
Data vs Process Abstraction

• Process abstraction
  – Sub-programs and functions

• Data abstraction
  – Types (e.g. $1 + 2 \neq 1.0 + 2.0$)
  – Data abstractions hides implementation differences between integer addition and floating point addition

• OOP lets programmers do the same
  – Plane.board() $\neq$ Train.board()
Data Abstraction Advantages

• Interface is independent of implementation
  – You ask an object to do something; you need not tell it how to do it

• Implementation is hidden from user
  – The implementation of an object's behavior can change as long as the behavior remains the same
Abstract Data Types

• An abstract data type defines behavior
  – E.g. integer division drops remainder; floating point division encode remainder as decimal part

• OOP lets you define new types
  – E.g. Stack s;
    • Allows: s.empty(); s.push(); s.pop(), s.top()
  – Implementation is hidden
    • Stack may be implemented as array, linked list, ...
Abstract Data Type in C++

- Based on C structs and typedef
- Called a class
  - C++ classes contain functions as well as data
  - Function in classes are called member functions
  - Each instance of a class has its own data members
- Information hiding
  - Members may be
    - Private: only object functions may access them
    - Public: other functions may access them
    - Protected: only sub-class may access them
The Object Oriented Languages and Paradigm
Member Functions Defined in Class

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() {...};
    }

Implicitly inlined → code placed in caller’s code
Language Examples: C++ (cont.)

- 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++ (cont.)

• Destructors
  – Functions to clean up 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 (~)

• Friend functions or classes: to allow access to private members to some unrelated units or functions (see Section 11.6.4)
  – Necessary in C++
Uses of the Stack Class

```java
void main()
{
    int topOne;
    Stack stk; // create an instance of the Stack class
    stk.push(42); // c.f., stk += 42
    stk.push(17);
    topOne = stk.top(); // c.f., &stk
    stk.pop();
    ...
}
```
// Stack.h - header file for Stack class
class Stack {

private:
    int *stackPtr, maxLen, topPtr;

public:
    Stack();  //** A constructor
    ~Stack();  //** A destructor
    void push(int);
    void pop();
    int top();
    int empty();
};
// Stack.cpp - implementation for Stack
#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;};
void Stack::push(int number) {
    if (topPtr == maxLen) 
        cerr << "Error in push--stack is full\n";
    else stackPtr[++topPtr] = number;}
...
Abstract Data Types in Java

• Similar to C++, except:
  − All user-defined types are classes
    • All objects are allocated from the heap and accessed through reference variables
  − Methods must be defined completely in a class
    → an abstract data type in Java is defined and declared in a single syntactic unit
  − Individual entities in classes have access control modifiers (private or public), rather than clauses
  − No destructor → implicit garbage collection
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 () {...};
}
An Example in Java

```java
public class TstStack {
    public static void main(String[] args) {
        StackClass myStack = new StackClass();
        myStack.push(42);
        myStack.push(29);
        System.out.println("::"+myStack.top());
        myStack.pop();
        myStack.empty();
    }
}
```
“Hello World!” Compared

C
#include <stdio.h>
int main(void){
    printf("Hello world!");
}

C++
#include <iostream>
using namespace std;
int main(){
    cout<<"Hello World!"<<endl;
}

Java
public class HelloWorld {
    public static void
        main(String[] args){
        System.out.println
            ("Hello world!");
    }
}

Ruby
puts 'Hello, world!' or
class String
    def say
        puts self
    end
end
'Hello, world!'.say

(http://en.wikibooks.org/wiki/Hello_world_program)
Outline

● The Concept of Abstraction (Sec. 11.1)
● Introduction to Data Abstraction (Sec. 11.2)
● Design Issues (Sec. 11.3)
● Language Examples (Sec. 11.4)
● Parameterized Abstract Data Types (Sec. 11.5)
● Encapsulation Constructs (Sec. 11.6)
● Naming Encapsulations (Sec. 11.7)
Parameterized ADTs

- Parameterized abstract data types allow designing an ADT that can store any type elements (among other things): only an issue for static typed languages
- Also known as *generic classes*
- C++, Ada, Java 5.0, and C# 2005 provide support for parameterized ADTs
Parameterized ADTs in C++

• Make Stack class generic in stack size by writing parameterized constructor function

```c++
class Stack {

    ...

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

    ...

} 

Stack stk(150);
```
Parameterized ADTs in C++ (cont.)

- Parameterize element type by *templated* class
  
  ```cpp
  template <class Type>
  class Stack {
    private:
      Type *stackPtr;
      int maxLen, topPtr;
    public:
      Stack(int size) {
        stackPtr = new Type[size];
        maxLen = size-1;  topPtr = -1; }
      ...
  }
  Stack<double> stk(150);
  ```

- Instantiated by compiler
Outline

• The Concept of Abstraction (Sec. 11.1)
• Introduction to Data Abstraction (Sec. 11.2)
• Design Issues (Sec. 11.3)
• Language Examples (Sec. 11.4)
• Parameterized Abstract Data Types (Sec. 11.5)
• Encapsulation Constructs (Sec. 11.6)
• Naming Encapsulations (Sec. 11.7)
Generalized Encapsulation

• Enclosure for an abstract data type defines a SINGLE data type and its operations
• How about defining a more generalized encapsulation construct that can define any number of entries/types, any of which can be selectively specified to be visible outside the enclosing unit
  – Abstract data type is thus a special case
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 logically related code and data into a unit that can be separately compiled (compilation units)
• Such collections are called encapsulation
  – Example: libraries
Means of Encapsulation: Nested Subprograms

- Organizing programs by nesting subprogram definitions inside the logically larger subprograms that use them
- Nested subprograms are supported in Ada, Fortran 95, Python, and Ruby
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:
  - Used to include header files in client programs to reference to compiled version of implementation file, which is linked as libraries
Encapsulation in C++

• Can define header and code files, similar to those of C
• Or, classes can be used for encapsulation
  – The class header file has only the prototypes of the member functions
  – The member definitions are defined in a separate file
    → Separate interface from implementation
• *Friends* provide a way to grant access to private members of a class
  – Example: vector object multiplied by matrix object
Friend Functions in C++

class Matrix;

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

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

Vector multiply(const Matrix& ml,
                 const Vector& vl) {
    ...  }
Naming Encapsulations

• Encapsulation discussed so far is to provide a way to organize programs into logical units for separate compilation
• On the other hand, large programs define many global names; need a way to avoid name conflicts in libraries and client programs developed by different programmers
• A naming encapsulation is used to create a new scope for names
Naming Encapsulations (cont.)

- **C++ namespaces**
  - Can place each library in its own namespace and qualify names used outside with the namespace

```cpp
namespace MyStack {
  ...
  // stack declarations
}
```
  - Can be referenced in three ways:
    ```cpp
    MyStack::topPtr
    using MyStack::topPtr; p = topPtr;
    using namespace MyStack; p = topPtr;
    ```
  - C# also includes namespaces
Naming Encapsulations (cont.)

- **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, e.g., `myStack.topPtr`, or use `import` declaration, e.g., `import myStack.*`;

- **Ada Packages**
  - Packages are defined in hierarchies which correspond to file hierarchies
  - Visibility from a program unit is gained with the `with` clause
Naming Encapsulations (cont.)

• Ruby classes are name encapsulations, but Ruby also has *modules*

• Module:
  − Encapsulate libraries of related constants and methods, whose names in a separate namespace
  − Unlike classes → 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
Ruby Modules

module MyStuff
  PI = 3.1415
  def MyStuff.mymethod1(p1)
    ...
  end
  def MyStuff.mymethod(p2)
    ...
  end
end

Require ‘myStuffMod’
myStuff.mymethod1(x)
Summary

- Concept of ADTs and the use in program design was a milestone in languages development
  - Two primary features are 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++
- Ada, C++, Java 5.0, and C# 2005 support parameterized ADTs
- C++, C#, Java, Ada, and Ruby provide naming encapsulations