Language Examples of ADT: C++

- Based on C **`struct`** type and Simula 67 classes
- 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

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

- Constructors:
  - Functions to initialize the data members of instances
  - Can include parameters to provide parameterization of the objects
  - Name is the same as the class name

- Destructors
  - Functions to cleanup after an instance is destroyed;
  - Name is the class name, preceded by a tilde (~)

An Example in C++

```
class Classic_Example {
public:
// Data and methods accessible to any user of the class
protected:
// Data and methods accessible to class methods,
// derived classes, and friends only
private:
// Data and methods accessible to class
// methods and friends only
};
```

```
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 () {…};
}
```

- Friend functions or classes – provide access to private members of some unrelated units or functions

Friend Classes

Inside a class, you can indicate that other classes (or simply functions) will have direct access to protected and private members of the class.

```
friend class aClass;
```

Example:

```
class Node
{
   private:
        int data;
        int key;
   // ...

   friend class BinaryTree; // class BinaryTree can now access data
                            // directly
};
```

In the BinaryTree class, you can treat the key and data fields as though they were public:

```
class BinaryTree
{
   private:
      Node *root;

      int find(int key);
};

int BinaryTree::find(int key)
{
   // check root for NULL...
   if(root->key == key)
   {
      // no need to go through an accessor function
      return root->data;
   }
   // perform rest of find
```

Friend Functions

A class can grant access to its internal variables on a more selective basis--restricting access to only a single function.

```
friend return_type class_name::function(args);
```

Example:

```
class Node
{
   private:
   int data;
   int key;
   // ...

   friend int BinaryTree::find(); // Only BinaryTree's find function has access
};
```

Class Data Members

- Data members may be objects of built-in types, as well as user-defined ADT.

```
class Node
   {
   private:
      void *pData;
      Node *pNext;

   public:
      Node( void *_pData )
      {
         pData = _pData;
         pNext = NULL;
      } // end constructor

      Node *Next() { return( pNext ); }
      void SetNext( Node *pNode ) { pNext = pNode; }
      void *Data() { return( pData ); }

   }; // end Node
```

Language Examples: Java

- Similar to C++, except:
  - –All user-defined types are classes
  - – Rather than having private and public clauses in its class definition, in Java access control modifiers can be attached to methods and variable definitions.

```
Import java.io.*;
class StackClass {
 private int [] stackRef;
 private int [] maxLen, topIndex;

public StackClass() {
        stackRef = new int [100];
        maxLen = 99;
        topPtr = -1;        };
public void push (int num) {…};
public void pop () {…};
    public int top () {…};
    public boolean empty () {…};
}
```

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

```
#include <iostream>

 template <class ElemType>
 class Stack {
 private:
    ElemType *stackPtr;
    int topElem; // invariant: -1 <= topElem <= size - 1
 public:
  Stack() {
     stackPtr = new ElemType[100];

     topElem = -1;       // no elements in stack yet

   }

   ~Stack() { delete stackPtr;}

    void push(ElemType e) {
     if (topElem >= size - 1) {
        cout « "Error: stack is full.\n" ; }
        else {
           topElem++;
           tr[topElem] = e;}
    void pop() {
     if (topElem == -1) {
        cout « "Error: stack is empty.\n" ;
        } else {
          topElem--;}}
    ElemType top() {
     // pre: topElem > -1
     return stackPtr[topElem];
     }
}
```

- Java 5.0 provides a restricted form of parameterized ADTs

```
ArrayList <Integer> myArray = new ArrayList <Integer>();
```

or

```
Public void drawAll(ArrayList<? Extends Shape> things)
```

- C# does not currently support parameterized classes

Encapsulation Constructs

- Large programs have two special needs:
  - Some means of organization, other than simply division into subprograms
  - Some means of recompilation (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 to such a file, including data, type, and function declaration, is placed in a *header file*
- `#include` preprocessor specification

Encapsulation in C++

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

Naming Encapsulations

A large program may create a naming problem:
How can independently working developers create names for their variables, methods, and classes without accidentally using names already in use by some other programmer developing a different part of the same software system?

- Large programs define many global names; need a way to divide into logical groupings.

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

```
namespace MyList {
class ListNode {
   // define class here
class LinkedList {
   // define class here
```

Code outside of the namespace can refer to names defined inside the namespace using scope resolution, i.e.:

```
MyList::LinkedList *1 = new MyList::LinkedList;
```

So a large program can have multiple classes called LinkedList.

  - C# also includes namespaces

• Java Packages

In Java, a package is a collection of classes.

- Every class is part of some *package*.
- You can specify the package using a *package declaration*:

  package *name* ;

- Multiple files can specify the same package name.
- You can access public classes in another (named) package using:

  package-name.class-name

  You can access the public fields and methods of such classes using:

  package-name.class-name.field-or-method-name

  You can avoid having to include the *package-name* using:

  import *package-name*.*; or

  import *package-name.class-name*;

- Java packages are also useful for avoiding name clashes.

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++ and Ada allow parameterized ADTs
- C++, C#, and Java provide naming encapsulation