Lecture 08: Support for Abstract Data Types

Support for abstract data types (ADTs) is critical in contemporary programming. This lecture discusses the topic under the following subheadings:

- Introduction
- Design Issues
- Other ADTs

8.1 Introduction

An *abstract data type* (ADT) is a programming construct with a defined set of data items, and a set of possible operations on those data items. In contemporary programming, ADTs are implemented as *classes* and/or *packages*. Because of the nature of ADTs in contemporary programming, we typically look for their support in object-oriented programming languages (OOPLs).

Common ADTs as discussed in your course in Data Structures and Algorithms:
- Dynamic Lists
- Linked Lists
- Stacks
- Queues
- Binary Trees
- Binary Search Trees (BSTs)
- Heaps
- B-trees
- Hash tables
- Graphs

Common sort algorithms include the following:
- Straight Selection-sort
- Exchange Selection-sort
- Insertion-sort
- Bubble –sort
- Quick-sort
- Merge-sort
- Tree-sort (as in BST)
- Heap-sort

In your course in Data Structures and Algorithms, you would have learned how to construct, implement, and test these ADTs and algorithms in at least one programming language (but preferably multiple programming environments). In that course, you would have also learned how to analyze these algorithms for efficiency.

In this lecture, we shall look at implementation of these ADTs but from a much broader perspective. Here, our concern is not implementation details in any given language, but rather, how different programming languages provide support for these ADTs. As you look at a new programming language, this broadened focus should prepare you nicely for learning any language within a short timeframe (which by the way is an important objective of the course). The lecture gets you started but as usual, I will not be doing all the work; rather, you will be given important guidelines in your language exploration.

8.2 Design Comparison

The obvious starting point is the construction of a class. How is this done in the new language that you are probing? Let's start with the familiar before launching into the unknown. Figure 8.1 shows the basic Java class anatomy, while figure 8.2 shows the C++ class anatomy. Take some time to familiarize yourself, or refresh your memory on class definition in both languages. Notice the similarities as well as the differences.

Figure 8.1: Anatomy of a Java Class

```
JavaClass ::=
public | private | protected class <ClassName>
[extends<ClassName>][implements<InterfaceName>[,…<InterfaceN>] ]
{
  // Data Item(s)
  …
  //Member Methods
  <Modifier> <ClassName> (<Parameter(s)>) // the constructor
  {
   // …
  }
  // … Additional methods
}
```

```
Method ::=
<Modifier><ReturnType><MethodName>(<Parameters>)
{
  … // Body of Method
}

Modifier ::=
[final] public | private | protected [static] [abstract]
```

Each keyword is important and therefore needs some clarification:

- The **public** keyword means that the method is available from anywhere in the program or from another class.
- The **private** keyword means that only instances of the class has access to this method.
- The **protected** keyword means that the method is protected within the class hierarchy only. It will be further clarified later in the course (lecture 6).
- The **static** keyword means that the method can be (and is of often) used without an instance of the class being created. In such case the class-name takes the place of the instance name.
- Keyword **abstract** means that the class or method is abstract. An abstract method has no statement (s). An abstract class is one for which instances cannot be created. It consists of at least one abstract method.
- The keyword **class** simply indicates to the Java compiler that a class is being defined.
- The **final** keyword means that the class or method cannot be inherited.

Figure 8.2: Syntax for Defining a C++ Class

```
<ClassDeclaration> :: =
class <ClassName>  [: <Modifier> <Base-class1> [… , <Modifier> <Base-class1>] ]
{
        [private:]
    <Private Members>       /* data items and function prototypes*/
        [public:
    <Public Members>]       /* data items and function prototypes */
    [protected:
    <Protected Members>]    /* data items and function prototypes */
};

/* Actual function definitions follow */
// …
```

```
<FunctionDefinition>::=
<ReturnType> <FunctionName> ([<Parameter>] [*…,<Parameter>*])
{
        <FunctionBody>
}
```

```
<FunctionPrototype>::=
<ReturnType> <FunctionName> ([<Parameter>] [*…,<Parameter>*])
```

```
<MemberFunctionSpecification> ::=
<Return Type> <Class Name> :: <MemberFunctionName> ([<Parms>])
{
…
}
```

One obvious difference you will notice between the C++ and the Java class definitions is that the syntax rules are different; this is expected. Another more far-reaching difference has to do with the principle of inheritance. From the definitions you can clearly see that Java supports inheritance from a single super-class, while C++ supports multiple inheritances from different super-classes.

Another not-so-obvious but very significant difference (but one you are no doubt familiar with) has to do with the specification of component methods of a class: Java prefers the term *method(s),* and insists that these component methods be part of the definitional structure of the class; in other words, they must be specified when the class is being defined. C++ prefers the term *member function(s)*. Moreover, C++ provides a mechanism for declaring the member functions at class definition (via function prototyping), but not specifying the details of their internal code. The language gives the programmer the flexibility of specifying detail codes for these member functions at a subsequent time, and such detailed code may or may not be part of the file with the class definition. To tie the member functions back to their host class, the scope resolution operator (::) is used.

Figure 8.3: Creating & Manipulating Student Objects in C++

```
// *********************************************************************************
// Program: StudentObjectDemo: Defines a Student class and manipulates it
// Author:  E. Foster
// *********************************************************************************
#include <cstdlib>
#include <iostream>
#include<ctype.h>
#include<string.h>
using namespace std;

typedef char* String;
// struct DateType {int Year; int Month; int Day;};
class StudentC
{
 float GPA;     /* known only to class members */
 public:
 int ID_Number;
 char SurName[16];
 char FirstName[16];
 int DateOfBirth;
 char Major[31];

 // member function prototypes
 public:
 StudentC( ); // Constructor
 void Modify (StudentC aStudent);
 void InputData();
 void PrintMe();
 void DetermineGPA (float Score [ ]);
 int GetNumber();
}; // End of Student class declaration

// The Member Functions of Student
// Constructor of Student
StudentC :: StudentC( )
{
 int y;
 ID_Number = 0;
 DateOfBirth = 19000101;
 for (y = 1; y <= 15; y++) SurName[y-1] = FirstName[y-1] = ' '; // 15 spaces
 for (y = 1; y <= 30; y++) Major[y-1] = ' '; // 30 spaces
 GPA = 0.0;
}

// … Continued on next page
```

Figure 8.3: Creating & Manipulating Student Objects in C++ (continued)

```
// Member function Modify of Student
void StudentC :: Modify (StudentC aStudent)
{
 ID_Number = aStudent.ID_Number;
 strcpy(SurName, aStudent.SurName); // SurName = aStudent.SurName;
 strcpy(FirstName, aStudent.FirstName); // FirstName = aStudent.FirstName
 strcpy(Major, aStudent.Major); // Major = aStudent.Major;
 DateOfBirth = aStudent.DateOfBirth;
}

// Member function InputData of Student
void StudentC :: InputData ( )
{
 int DoB;
 cout << "\nStudent Information Entry \n\n";
 cout << "Please enter the following: \n";
 cout << "ID Number:   " ; cin >> ID_Number; getchar();
 cout << "\nSurname:  " ; gets(SurName);
 cout << "\nFirst Name:  "; gets(FirstName);
 cout << "\nMajor:  "; gets(Major);
 cout << "Date of Birth (YYMMDD):  ";  cin >> DoB; getchar();
 DateOfBirth = DoB;
}

// Member function PrintMe of Student
void StudentC :: PrintMe ( )
{
 cout<< "\nStudent Information: \n";
 cout<< "ID Number:   " << ID_Number << endl;
 cout<< "Name:  " << FirstName << " " << SurName << endl;
 cout<< "Date of Birth:  " << DateOfBirth << endl;
 cout<< "Major:  " << Major << endl;
 cout<< "GPA:  " << GPA << endl << endl;
}

// Member function DetermineGPA of Student
void StudentC :: DetermineGPA (float Score [ ])
{
        // Calculate GPA from input scores
        // …
}

// Member function GetNumber of Student
int StudentC :: GetNumber ( )
{ return ID_Number; }

// … Continued on next page
```

Figure 8.3: Creating & Manipulating Student Objects in C++ (continued)

```
// Main function
int main(int argc, char *argv[])
{
 // Declarations
 String FullName1 = new char[31], FullName2 = new char[31];
 bool ExitTime = false;
 char ExitKey;

 while (!ExitTime) // While user wishes to continue
 {
  // Initialize
  for (int x=1; x <= 31; x++) FullName1[x-1] = FullName2[x-1] = ' ';

  // Create New student objects and assign initial values
  StudentC NewStud = StudentC(); // Instantiates NewStud
  NewStud.InputData (); // The InputData() function is invoked
  //       …
  // Create another Student object
  StudentC OtherStud = StudentC(); // Instantiates OtherStud
  OtherStud.InputData(); // The InputData() function is invoked

  // To print the Student objects
  NewStud.PrintMe( );
  OtherStud.PrintMe( );
  // …
  // Print welcome message to each student
  FullName1 = strcat (NewStud.SurName, strcat(" ", NewStud.FirstName));
  FullName2 = strcat (OtherStud.SurName, strcat(" ", OtherStud.FirstName));
  cout<< "\n\nWelcome" << FullName1 << " and " << FullName2 << "!" << endl;
  //…
  // To create a new student object with default initial values
  StudentC Dummy = StudentC(); // or simply, Student Dummy;

  // Switch the values of the two Student objects and redisplay
  Dummy.Modify(NewStud);
  NewStud.Modify(OtherStud);
  OtherStud.Modify(Dummy);

  NewStud.PrintMe( );
  OtherStud.PrintMe( );

  //Check whether user wishes to continue
  cout << "\n Press any key to continue or X to exit ";
  ExitKey = getchar();
  if (toupper(ExitKey) == 'X')ExitTime = true;
  else {NewStud.Modify(Dummy); OtherStud.Modify(Dummy);}
 } // End of While-user -wishes-to-continue

 system("PAUSE");
 return EXIT_SUCCESS;
} // End of main function
```

8.2 Design Comparison (continued)

Figure 8.4 provides a summary of the main differences between C++ class implementation and Java class implementation. When learning a new language, it is a good idea to construct a comparison list like this, where on each programming principle of concern, you compare the implementation detail of the language you are learning with that of a language which you are very familiar with. This technique helps you to relate the new language to something that you are familiar with.

Figure 8.4: Comparison of Java Class Implementation with C++ Class Implementation

Java	**C++**
Each class must be defined in a separate file with the same name as the class-name	A program file may or may not contain zero or more class definitions. Moreover, the program-file may carry a different name from the class name.
A class is made up of data items and/or methods.	A class is made up of data items and/or member functions.
The methods are defined as part of the class definition.	The member functions may or may not be defined in the class declaration. The class declaration may contain member function prototypes instead of detailed function definitions (this is the preferred approach). The detailed function definitions typically follow the class declaration.
Multiple inheritance is not supported	Multiple inheritance is supported
Interfaces are supported	Interfaces are not supported.

Exercise 1: Try finding out how classes are supported in languages such as C#, Ada, Python, and Ruby. Set up your comparison grids for various areas of concern to you.

8.3 Other ADTs

Having looked at how classes are constructed in different programming languages, the next logical step is to conduct a similar study for each of the ADTs mentioned in the introduction. Obviously, you should easily see where this is going: we could have one section per ADT, and this lecture would extend for several pages. While this exercise is very tempting and would no doubt be quite enlightening, that bate will not be taken here. Rather, here is an exercise for you:

Exercise 2: Identify a language (an OOPL) that you would like to learn. Examine how the various ADTs mentioned in section 8.1 are supported in the language, and set up comparison grids comparing the new language with one that you are familiar with.

8.4 Summary and Concluding Remarks

Here are the salient points of this lecture:

- An ADT is a programming construct with a defined set of data items, and a set of possible operations on those data items.
- Common ADTs that appear in contemporary programming are dynamic lists, linked lists, stacks, queues, binary trees, binary search trees, heaps, B-trees, hash tables, and graphs.
- Common sort algorithms that are discussed in contemporary programming are straight-selection-sort, exchange-selection-sort, insertion-sort, bubble –sort, quick-sort, merge-sort, tree-sort (as in binary search tree), and heap-sort.
- The starting point for probing how a language supports ADTs is to look at how the language supports classes. From here, additional probe may be made into various ADTs.

The next lecture will build on the discussion here by focusing on support for OOP in contemporary programming languages.

8.5 Recommended Readings

[Pratt & Zelkowitz 2001] Pratt, Terrence W. and Marvin V. Zelkowits. 2001. *Programming Languages: Design and Implementation* 4th Edition. Upper Saddle River, NJ: Prentice Hall. See chapters 6 & 10.

[Sebesta 2012] Sebesta, Robert W. 2012. *Concepts of Programming Languages* 10th Edition. Colorado Springs, Colorado: Pearson. See chapter 11.

[Webber 2003] Webber, Adam B. 2003. *Modern Programming Languages: A Practical Introduction*. Wilsonville, Oregon: Franklin, Beedle & Associates. See chapters 12, 14, 15 & 16.