Lecture 06: Introduction to Object Oriented Programming

So far, all our programming has been procedural. In procedural programming, the programmer breaks down a complex algorithm into component tasks which are repeatedly refined until a step-by-step solution to the problem is defined. This process is referred to as stepwise refinement, or modular programming.

C++ is known as a hybrid language — it supports both procedural programming and object-oriented programming (OOP). As you are aware, C++ was developed from (and by) the C programming language, and still supports all features of that language. Although technically speaking, C++ is best described as a hybrid language, it is often loosely categorized with object-oriented programming languages (OOPLs), because it supports OOP. This and the next few lectures explore the object-oriented (OO) features of C++. The lecture proceeds under the following captions:

- Rationale for Object-Oriented Programming
- Basic Concepts
- Defining and Representing a Class
- Specifying Member Functions of a Class
- Accessing Class Members
- Structures and Unions versus Classes
- Arrays of Objects
- Pointers to Objects
- Dynamic List of Objects
- Friend Functions
- Assigning Objects
- Copy Constructor and its Alternative
- The this Keyword
- The **string** Class
- Summary and Concluding Remarks

6.1 Rationale for Object Oriented Programming

The salient features of procedural programming are summarized below:

- The programmer concentrates on the component activities in terms of what they are and how they are to be carried out.
- The order in which instructions are issued is of paramount importance.
- The computer scientist develops algorithms composed of variables, subroutines and control structures such as sequential control, selection control, iterative control and recursive control.
- Algorithms (hence programs) for complex problems tend to be voluminous and detail specific.

Object-oriented programming (OOP) is an enhancement of procedural programming, adding another layer of abstraction. It is an effort to allow software developers to think like human beings, and have the compiler adjust to them. This philosophy runs counter to the procedural programming philosophy, where the software engineer is forced to think (procedurally) like the computer.

The object-oriented (OO) approach to software construction may be summarized as follows:

- The software engineer (or programmer) identifies and analyses objects (more precisely object types) with which work is required.
- For each object type, the operations to be performed on or with the object are identified.
- The operations are implemented via methods (or functions). These methods may work differently, depending on the context in which they are used.
- Objects communicate with each other by sending messages.
- Objects can inherit properties (structure and/or operations) from other previously defined objects.

The OO approach forces the software developer to concentrate on object types first and procedural details last, at each level of abstraction. With this paradigm shift comes a number of advantages. The salient ones are summarized in figure 6-1. Some of them relate directly to the use of OOPLs; others are indirect benefits due to the use of OOPLs in constructing OO-RAD tools and OO-CASE tools. These tools bring additional benefits to the software engineering arena. The use of OOPLs, OO-RAD tools, OO-CASE tools and other OO approaches to software engineering is called object-oriented methodologies (OOM), or object-oriented technology (OOT), or simply object technology (OT).

You are likely to find software construction via OOM a very fascinating experience and rewarding. You are therefore strongly encouraged to pursue such a course. In this course, we introduce the basic concepts and concentrate on OOP using C++.

Figure 6-1: Benefits of OOP

Benefit	Clarification
Reusability of Code	A tested system component can be reused in the design of another component.
Stability and Reliability	Software can be constructed from tested components. Organizations can be assured of guaranteed performance of software.
Sophistication	More complex systems can be constructed.
Understandability	Designers and users think in terms of objects and behaviors rather than low-level functional details. This results in better communication among stakeholders of a software system, and more realistic modeling that is easier to learn and communicate.
Faster Design	Most contemporary RAD (rapid application development) tools and CASE (computer aided software engineering) tools are object-oriented to some degree. Also, code reusability enhances faster development.
Higher Quality Design	New software can be constructed by using tested and proven components.
Easier Maintenance	Since systems are broken down into manageable component objects, isolation of system faults is easy.
Dynamic Lifecycle	Integrated objected-oriented CASE (IOO-CASE) tools integrate all stages of the software development life cycle (SDLC).
Interoperability	Classes may come from different vendors, and used in different scenarios without excessive customization.
Design Independence	Classes may be designed to operate and/or communicate across different platforms.
Better CASE Tools	OOP leads to better CASE and RAD tools, which in turn leads to better software development.

6.2 Basic Concepts

Object oriented technology is predicated on some fundamental concepts that must be fully understood from the outset. We shall briefly discuss these concepts here.

Object Type and Object: An *object type* is a concept or thing about which data is stored. An object is an instance of an object type. For example, if *Student* is an object type, then *Bruce Jones* is an instance of the object type *Student*. The simplest form of an object type is a primitive data type. The variables that you define on that data type are instances of that type.

Operation: An *operation* is a task that can be performed on an object. In an OOPL environment, operations are typically implemented as methods or functions (Java favors the term methods). However, you could have a complex operation that is implemented as a class that consists of various methods.

Example 6-1:

For the **Student** object type, valid operations may be **Add, Modify, Remove, Search, Display,** and **Print**.

Member Function: A *member function* is a set of instructions for carrying out an operation. In some programming environments, member functions are referred to as methods, or procedures. Also, because a member function merely implements an operation, the two terms are often used interchangeably. An older (pseudo-code) term which a method epitomizes is the subroutine. However, C++ favors the term member function, so for the rest of this course, we'll stick with that.

6.2 Basic Concepts (continued)

Class and Encapsulation: In the simplest form, encapsulation is the act of hiding detail, of an object (type) until it is required. A *class* is the *encapsulation* of an object's structure with its operations. The object can be accessed only through its class. At the highest level, therefore, one is not focusing on one an object's (encapsulated) methods, but on some aspect of the object's structure, some of its operations, or both.

Classes and Methods versus Object Types & Operations: The convention in software engineering is to use terms such as classes and methods at the implementation level, and object types and operations at the design level. Since programming is predominantly an implementation issue (but you still have to design your programs), we will stick to the classes & methods.

Structure and Properties: The structure of a class refers to the data items and member functions that have been defined within the class. The data items and member functions are also referred to as the *properties* of the class.

Amalgamation: An object may be composed of other objects. In OOP, we refer to such an object as an *aggregate* or *composite* object. The act of incorporating other component objects into an object is called *aggregation* or *composition*. Since this is done through the object's class, the class is also called an aggregation (or composition) class. Throughout this course, we shall use the term amalgamation to mean an aggregation and/or composition.

Inheritance: An object inherits all the *properties* (structure and/or methods) of its parent class**.** Additionally, a class may inherit from a super-class in its hierarchy. The inheriting class is referred to as the sub-class.

Example 6-2:

Computer-Science-Student could be defined as a sub-class of the super-class **Student**; **Rectangle** could be defined as a sub-class of the super-class, **Polygon**. If **Bruce** is a **Student** object, **Bruce** inherits all the inheritable properties of **Studen**t. If **Karen is Computer-Science-Student** object, **Karen** inherits all the inheritable properties of **Computer-Science-Student** and **Studen**t. The **Student** object may be composed of a **StudentPersonal** object and a **StudentAcademic** object. If so, then **Student** is an amalgamation.

Dot Operator: In OOP, whenever we want to refer to a component of a package, or class, we use the *dot operator* (which is simply a period). From this point onwards, you will see the dot operator used a lot throughout this course, and will no doubt use it in your own programs.

Example 6-3:

Suppose that **Student** is a class with data items **Name** and **DateOfBirth**. Suppose that it also has methods **AddMe() and PrintMe()**. If **Bruce** is an instance of **Student**, then we can refer to its components as **Bruce.Name**, **Bruce.DateOfBirth**, **Bruce.AddMe()** and **Bruce.PrintMe()**.

6.2 Basic Concepts (continued)

Polymorphism: An operation, object, or operator may be required to take on a different form, depending on the context of its usage. This phenomenon is called *polymorphism*. In C++, polymorphism is implemented via *function overloading*, *function default arguments*, *operator overloading*, *object casting*, *function overriding*, and *templates*. You are already familiar with casting, function default arguments, and function overloading (review lecture 4). We will revisit the other terms later in the course.

Example 6-4:

Suppose that **CollegeMember**is a class with data items **ID_Number** and **Name**, and methods **InputData()**, **Modify(CollegeMember Current)**, and **PrintMe()**. Suppose further that **Student** is a subclass of **CollegeMember** with additional data items **Department** and **Major** and methods **InputData()**, **Modify(CollegeMember Current)**, and **PrintMe()**.

The methods **InputData()**, **Modify(CollegeMember Current)**, and **PrintMe()** are polymorphic methods. The ones belonging to the subclass (**Student**) will override the corresponding methods belonging to the super-class (**CollegeMember**). Thus, if **Bruce** is an instance of **Student** and **Lambert** is an instance of **CollegeMember**, then we can reasonably expect that **Bruce.InputData()** will behave slightly differently from **Lambert.InputData()** — it will likely obtain values for **ID_Number** and **Name** (as **Lambert.InputData()** would), but would also obtain values for **Department** and **Major**.

Referencing Static Properties: The concept of static resources (data items or methods) was mentioned earlier (section 2.2.1). The convention for referencing static data items from a class is slightly different (due to the definition of a static resource): To reference a static data item from a class, instead of specifying the instance name (there is no instance), you specify the class name where that static data item resides.

Example 6-5:

Suppose that **College** is a class with a static data item called **Name**. Then we would reference this data item as **College.Name** from outside of the class, and simple **Name**, if we are referencing from within the class.

6.3 Defining and Representing a Class

A class defines a new data type, which can be used for subsequent creation of objects. It is different from (superior to) other data types in that in addition to holding data elements, it may contain methods (implemented by C++ functions).This section discusses how you would define and represent a class.

6.3.1 Defining the Class

In defining a class, the programmer must address three fundamental characteristics of classes:

- Every class must have a unique name.
- A class may have zero of more data members (attributes); typically there are several.
- A class may have zero or several defined operations (loosely called methods in C++) implemented as member functions.

6.3.1 Defining the Class (continued)

Note: A class without operations is equivalent to a structure. Class elements and operations are often referred to as members or properties.

The required syntax for class declaration is shown in figure 6-2.

Figure 6-2: Syntax for Defining a Class

```
ClassDeclaration :: =
class <ClassName>
{
  [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 */
// ...
```

From the definition a number of points are worth noting:

1. By default, class members are private; therefore, the **private** keyword may be omitted. Private members are known only to the class; they can be accessed only by other members of the class. This is one way of enforcing encapsulation (information hiding).

2. Class members that are public are accessible to other parts of your program. Moreover, each member function has access to all (private and public) properties of the class.

3. Actual function definitions for member functions are typically specified after the class declaration. However, if a member function is very simple, it may be specified within the class declaration (referred to as an *inline function*). Except for overloaded functions, function names must be unique.

4. Data members are declared in the same manner as variable declarations, but initialization is not allowed.

5. A special member function called a *constructor*, facilitates object initialization. A constructor is a special function, the same name as the class, and without a return type (not even **void**). It is automatically invoked after object allocation, to perform object initialization. The constructor is usually declared ahead of the member function prototypes.

6. Once the class is defined, you can declare instances (objects) of that class (remember a class is an advanced data type) similar to the declaration of variables of a structure. Created instances are objects that have their own copies of the class members i.e. they inherit the class properties.

6.3.1 Defining the Class (continued)

The class may be defined in a program file containing other program components such as the **main** function and other functions. Alternately, the class may be defined in a special class file that is used exclusive for that purpose. When this approach is employed, the class is normally defined in a file with the same name; the file is stored as a header file.

Example 6-6: Figure 6-3 shows some skeleton C++ code for a class called **StudentC**.

Figure 6-3: C++ Code for class StudentC

```
class StudentC
{
        float studGPA;          // known only to class members
        public:
        int studID_Number;
        char studLastName[15];
        char studFirstName[15];
        int studDateofBirth;
        char studMajor[30];

        // member function prototypes
        StudentC (int sNumber, char sLname[15], sFname[15], sMajor[30], int sDoB, float sGPA); // constructor
        StudentC ( ); // overloaded constructor
        void modify (int sNumber, char sLname[15], sFname[15], sMajor[30], int sDoB, float sGPA);
        void modify (Student aStudent); // Overloaded
        void inputData();
        void printMe ( );
        void determineGPA (float Score [ ]);
        int getNumber ( );
} prevStudent, thisStudent, nextStudent;  // creates 3 instances of StudClass
// ...
/* Alternate declaration of objects would be
StudentC PrevStudent, Student, NextStudent;  */
// ...
```

6.3.2 Representing the Class

As you can well appreciate, a class is a very complex data structure that could involve several lines of C++ code. Remembering all the properties of the class, and their purpose, can be quite challenging. To assist with this, the class is often represented with a UML (Unified Modeling Language) diagram. Figure 6-4 illustrates a UML diagram for the class of figure 6-3. The UML diagram has three compartments for the class-name, the data items, and the member functions respectively.

Figure 6-4: The UML Diagram of the Student Class of Figure 6-3

StudentC // The name of the class

// The data items of the class
private float studGPA; // known only to class members
public int studID_Number
public char studLastName[15]
public char studFirstName[15]
public int studDateofBirth
public char studMajor[30]

// Member function signatures for the publicly accessible functions of the class
StudentC (int sNumber, char sLname[15], sFname[15], sMajor[30], int sDoB, float sGPA) // constructor
StudentC ( ) // overloaded constructor
void modify int sNumber, char sLname[15], sFname[15], sMajor[30], int sDoB, float sGPA)
void modify (Student aStudent) // Overloaded
void inputData()
void printMe ( )
void determineGPA (float Score [ ])
int getNumber ( )

Note: In the original UML notation, the data type is specified after the property (data item or method) in a manner consistent with variable declaration in the Pascal programming language. However, since the C++ language specifies data type first in variable and method declaration, in the interest of clarity, this convention is also used for the UML notation.

6.4 Specifying Member Functions for a Class

Unless member functions are very simple, they are not usually specified within the body of the class. Rather, the class declaration contains prototypes of member functions as illustrated in the previous example. These member functions are declared afterwards. If the program file contains other components apart from the class, then it is recommended that specifications of member functions be kept together — either after the **main** function, or immediately after the class declaration, ahead of the **main** function.

One of these member functions is the class constructor. Note that a class may have more than one overloaded constructors.

In specifying member functions, you must inform the C++ compiler as to which class the function belongs to. To do this, simply qualify the function name with the class name in the following way:

```
<ReturnType> <Class Name> :: <MemberFunctionName> ([<Parms>])
{
  // ...
}
```

6.4 Specifying Member Functions for a Class (continued)

Note the following related points:

1. The symbol **::** is required here (it is not a BNF notation as described in lecture 1 and used throughout the course up until this point). It is called the **scope resolution operator**. It associates a particular member function to a specific class.

2. Member function names may be the same for different classes. C++ combines the class name with the function name to determine uniqueness.

3. Overloaded member functions for a given class are allowed.

4. The constructor, if used, is a special case of function overloading where the constructor overlays the class itself.

5. The term *member function* is analogous to *method* in other object-oriented languages such as Java and C#. We will stick to the preferred C++ terminology in this course, but bear in mind that both terms are essentially equivalent.

Figure 6-5 shows member functions of the **StudentC** class of figure 6-3. These functions would follow the class declaration.

Figure 6-5: Member Functions of the StudentC Class

```
// … Following the class declaration

// Constructor of Student
StudentC :: StudentC (int sNumber, char sLname[15], sFname[15], sMajor[30], int sDoB float sGPA)
{
   studID_Number = Number;   strcpy(studLastName, sLname);  strcpy(studFirstName, sFname);
   strcpy(studMajor, sMajor);    studDateOfBirth = sDoB;  studGPA = sGPA
}
StudentC :: StudentC ( )          // overloaded constructor
{
     studDateOfBirth = 19000101; studID_Number = 0; studGPA = 0.0;
     for (y = 1; y <= 15; y++) studLastName[y-1] = studFirstName[y-1] = ' '; // 15 spaces
     for (y = 1; y <= 30; y++) studMajor[y-1] = ' '; // 30 spaces
}
// Member function modify Student
void StudentC :: modify int sNumber, char sLname[15], sFname[15], sMajor[30], int sDoB, float sGPA)
{
   studID_Number = sNumber; strcpy(studLastName, sLname);  strcpy(studFirstName, sFname);
   strcpy(studMajor, sMajor); studDateofBirth = sDoB; studGPA = sGPA;
}
// Overloaded member function modify of StudentC
void StudentC :: modify (StudentC aStudent)
{
   studID_Number = aStudent. studID_Number;
    strcpy(studLastName, aStudent. studLastName); // studLastName = aStudent. studLastName;
    strcpy(studFirstName, aStudent. studFirstName); // studFirstName = aStudent. studFirstName
    strcpy(studMajor, aStudent. studMajor); // studMajor = aStudent.studMajor;
    studDateOfBirth = aStudent.studDateOfBirth; studGPA = aStudent.studGPA;
}
```

Figure 6-5: Member Functions of the StudentC Class (continued)

```
// Member function inputData of StudentC
void StudentC :: inputData ( )
{
   int sDoB; float sGPA;
   cout << "Student Information Entry \n\n";
   cout << "Please enter the following: \n"
   cout << "ID Number:   " ; cin >> studID_Number; char dummy = getchar();
   cout << "\nSurname:  " ; gets(studLastNamer);
   cout << "\nFirst Name:  "; gets(studFirstName);
   cout << "\nstudMajor:  "; gets(studMajor);
   cout << "Date of Birth (YYMMDD):  ";  cin >> DoB; dummy = getchar();  studDateOfBirth = sDoB;
   cout << "GPA:  ";  cin >> sGPA; dummy = getchar(); studGPA = sGPA;
}

// Member function printMe of StudentC
void StudentC :: printMe ( )
{
   cout<< "Student Information: \n"
   cout<< "ID Number:   " << studID_Number << endl;
   cout<< "Surname:  " << studLastNamer << endl;
   cout<< "First Name:  " << studFirstName << endl;
   cout<< "Major:  " << studMajor << endl;
   cout<< "Date of Birth:  " << studDateOfBirth << endl;
   cout<< "GPA:  " << studGPA << endl;
}

// Member function determineGPA of StudentC
void StudentC :: determineGPA (float inScore [ ])
{
   // Calculate GPA from input scores
   // ...
}

// Member function getNumber of Student
int StudentC :: getNumber ( )
{ return studID_Number; }

// ...
```

The complement of the constructor is the *destructor* — a function that prescribes actions to be taken when an object is destroyed. Global objects are created when the program begins and destroyed when the program is terminated; block objects are created when the block is entered and destroyed when the block is left. The destructor has the same name as the constructor, but is preceded by a tilde (~). Like the constructor, a destructor has no return type.

A destructor is useful is a situation where the constructor allocates memory via the **new** operator. In such instances, the destructor will de-allocate memory via the **delete** operator. For the purpose of illustration, the next example shows how a destructor is declared and specified. Figure 6-6 illustrates how you could include a destructor for the **StudentC** class or earlier discussion.

Figure 6-6: Illustrating Specification of a Destructor

```
class StudentC
{
  float studGPA;        // known only to class members
  public:
  int studID_Number;
  char studLastName[15];
  char studFirstName[15];
  int studDateofBirth;
  char studMajor[30];

  // member function prototypes
   StudentC (int sNumber, char sLname[15], sFname[15], sMajor[30], int sDoB, float sGPA); // constructor
   StudentC ( ); // overloaded constructor
    ~StudentC( ); // destructor
   void modify (int sNumber, char sLname[15], sFname[15], sMajor[30], int sDoB, float sGPA);
   void modify (StudentC aStudent); // Overloaded
   void inputData();
   void printMe ( );
   void determineGPA (float Score [ ]);
   int getNumber ( );
} prevStudent, thisStudent, nextStudent;          // creates 3 instances of StudentC
// ...

StudentC :: ~StudentC ( )  // Destructor
{ delete this; }
```

6.5 Accessing Class Members

Once the class is defined and objects (variables) of it have been created, class members can be accessed through these objects. Remember, each object is an instance of the class. Data members as well as member functions can be accessed via the dot-operator. Declarations must specify constructor arguments if the constructor requires arguments. This is so because when you declare an instance of a class, the C++ compiler actually instantiates that object by automatically invoking (calling) its constructor.

Remember, block objects are created when the block is entered and destroyed when the block is left; global objects are created when the program begins and destroyed when the program is terminated.

Example 6-7: Assuming the class declarations of the previous discussion, figure 6-7 illustrates how instances of the class may be manipulated.

Figure 6-7: Manipulating Instances of a Class

```
// … Assume the class definition of figures 6-3, 6-4, and 6-5.
// …
// The main function
int main(int argc, char *argv[])
{
    char fullName[31];
    char newLName[15], newFName[15], newMajor[30];
    int newNumber; int newDoB;float newGPA = 0.0;
    cin >> newNumber; char dummyChar = getchar();
    gets (newLName);
    gets (newFName);
    gets (newMajor);
    cin >> newDoB; dummyChar = getchar();

        // Create new student objects and assign initial values via its constructor
    StudentC newStud = StudentC (newNumber, newLName, newFName, newMajor, newDOB, newGPA);
    // …
    // To modify newStud's first name and major, call its member function Modify
    newStud.modify (newNumber, newLName, "Bruce", "Computer Science", bewDOB, newGPA);
        …
    // Below is an alternate way to create a Student instance, and accept data for it
    StudentC otherStud = StudentC(); // otherStud has default values via the overloaded constructor
    otherStud.inputData(); // The inputData() function is invoked

    // To print the Student objects
    newStud.printMe( );
    otherStud.printMe( );
    …
    // Accessing data members
    fullName = strcat (newStud.studLastName, strcat(" ", newStud.studFirstName));
    cout << "Welcome" << fullName << endl;
    …
    // To create a new student object with default initial values
    StudentC dummy = StudentC( ); // or simply, StudentC dummy;

    // Switch the values of the two Student objects and redisplay
    dummy.modify(newStud);
    newStud.modify(otherStud);
    otherStud.modify(dummy);

    newStud.printMe( );
    otherStud.printMe( );

    // ...
}
```

Note: Object instantiation is facilitated via calling the constructor. However, unlike Java, calling the C++ constructor does not require use of the new operator.

Figure 6-8 provides a program listing of a simple program that implements the concepts we have been discussing. The upper half of the program defines the **StudentC** class; the lower half contains a main function that manipulates instances of the **StudentC** class.

Figure 6-8a: Creating and Manipulating Student Objects — the SudentC Class

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

typedef char* cString;

class StudentC
{
  float studGPA;       // known only to class members
  public:
  int studID_Number;
  char studLastName[16];
  char studFirstName[16];
  int studDateofBirth;
  char studMajor[31];

  // member function prototypes
  public:
  StudentC( ); // Constructor
  void modify (StudentC aStudent);
  void inputData();
  void printMe();
  void determineGPA (float inScore [ ]);
  int getNumber();
}; // End of Student class declaration

// The Member Functions of Student
// Constructor of Student
StudentC :: StudentC( )
{
  int y;
  studID_Number = 0;
  studDateOfBirth = 19000101;
  for (y = 1; y <= 15; y++) studLastName[y-1] = studFirstName[y-1] = ' '; // 15 spaces
  for (y = 1; y <= 30; y++) studMajor[y-1] = ' '; // 30 spaces
  studGPA = 0.0;
}

// … Continued on next page
```

Figure 6-8a: Creating and Manipulating Student Objects — the SudentC Class (continued)

```
// Member function modify of StudentC
void StudentC :: modify (StudentC aStudent)
{
    studID_Number = aStudent. studID_Number;
     strcpy(studLastName, aStudent. studLastName); // studLastName = aStudent. studLastName;
     strcpy(studFirstName, aStudent. studFirstName); // studFirstName = aStudent. studFirstName
     strcpy(studMajor, aStudent. studMajor); // studMajor = aStudent.studMajor;
     studDateOfBirth = aStudent.studDateOfBirth; studGPA = aStudent.studGPA;
}

// Member function inputData of StudentC
void StudentC :: inputData ( )
{
    int sDoB; float sGPA;
    cout << "Student Information Entry \n\n";
    cout << "Please enter the following: \n"
    cout << "ID Number:   " ; cin >> studID_Number; char dummy = getchar();
    cout << "\nSurname:  " ; gets(studLastNamer);
    cout << "\nFirst Name:  "; gets(studFirstName);
    cout << "\nstudMajor:  "; gets(studMajor);
    cout << "Date of Birth (YYMMDD):  ";  cin >> DoB; dummy = getchar();  studDateOfBirth = sDoB;
    cout << "GPA:  ";  cin >> sGPA; dummy = getchar(); studGPA = sGPA;
}

// Member function printMe of StudentC
void StudentC :: printMe ( )
{
    cout<< "Student Information: \n"
    cout<< "ID Number:   " << studID_Number << endl;
    cout<< "Surname:  " << studLastNamer << endl;
    cout<< "First Name:  " << studFirstName << endl;
    cout<< "Major:  " << studMajor << endl;
    cout<< "Date of Birth:  " << studDateOfBirth << endl;
    cout<< "GPA:  " << studGPA << endl;
}

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

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

// … Continued on next page
```

Figure 6-8b: Creating and Manipulating Student Objects — the Main Program

```
// ********************************************************************************
// 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* cString;

// Main function
int main(int argc, char *argv[])
{
 // Declarations
 cString 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
  oherStud.IiputData(); // The InputData() function is invoked

  // To print the Student objects
  newStud.printMe( );   otherStud.printMe( );
  // …
  // Print welcome message to each student
  fullName1 = strcat (newStud.studLastName, strcat(" ", newStud.studFirstName));
  fullName2 = strcat (otherStud.studLastName, strcat(" ", otherStud.studFirstName));
  cout<< "\n\nWelcome" << fullName1 << " and " << fullName2 << "!" << endl;
  //…
  // To create a new student object with default initial values
  StudentC dummyS = StudentC(); // or simply, Student dummyS;

  // Switch the values of the two Student objects and redisplay
  dummyS.modify(newStud);    newStud.modify(otherStud);   otherStud.modify(dummyS);
  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(dummyS); otherStud.modify(dummyS);}
 } // End of While-user -wishes-to-continue

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

6.6 Structures and Unions versus Classes

All the forgoing discussion about classes can be applied to structures. In fact, formal C++ syntax states that a structure actually creates a class type. You can define member functions for a structure type in the same way you do for a class.

There is one distinction between a structure and a class: a structure's members are by default public while a class's members are by default private. In order to avoid confusion between structures and classes, this course makes the following recommendation:

- If you need to implement a record consisting of several fields, but do not intend to define operations on the record, use a structure.
- If you need to implement a record consisting of several fields, upon which you intend to define specific operations, use a class.

A union also defines a class type (where all elements are stored in the same location) a union can also have member functions, constructor(s) and destructor(s). Like structures, the members are by default, public. Again it is advised that you avoid confusion and use the appropriate types for their original intentions.

Note: These apparent redundancies exist in order to facilitate upward integration from C to C++.

6.7 Arrays of Objects

You can create arrays of objects in the same way that arrays of other base types are declared and managed. The one additional fact you must be cognizant of is that each item in the array is an instance of the base type (class) and therefore inherits all the properties defined in the class.

Example 6-8: Assuming the declaration of the previous section, figure 6-9 illustrates how we may use load an array of **StudentC** objects.

Figure 6-9: Loading an Array of StudencC Objects

```
// This illustration assumes the previous code as illustrated in figures 6-3, 6-4, and 6-8a

int main(int argc, char *argv[])
{
        int x, sLim = 20;
    StudentC studList [sLim];  // Array of Lim student objects
    // ...
    // Prompt for information on each student
    for (x = 1; x <=Lim; x++ )
    {   studList [x-1] = StudentC( ); // Instantiate the item
        studList [x-1].iputData( );   // Obtain  information for the item
    }

    // ...
}
```

6.7 Arrays of Objects (continued)

If the constructor requires arguments, you must specify constructor arguments for each entry of the array by specifying the constructor name, followed by its arguments in parenthesis. All this must be done within curly braces.

Example 6-9: Assuming declarations of sections 6.3 and 6.4, the following code (figure 6-10) creates an array of three **StudentC** objects.

Figure 6-10: Array Loading When the Constructor Requires Arguments

```
int main(int argc, char *argv[])
{
    // ...
    // Use the overloaded constructor that required arguments to initialize an array of StudentC objects
        StudentC student [3] = {
        StudentC ( 1994001, “Foster”, “Bruce”, “Computer Science”, 19641231, 0.0),
        StudentC ( 1994123, “Jones”, “Bruce”, “Computer Science”, 19650127, 0.0),
        StudentC (1994133, “Harrigan”, “Pamela”, “Mathematics”, 19660830, 0.0)};
    // ...
}
```

As you can see, using parameterized constructors in this way could result in rather inelegant coding. It is therefore prudent to use constructor parameters with default values, or avoid constructors which require multiple arguments, when arrays of objects will be required. Unless the array contains only a few elements (less than six), this approach is not recommended (even with less than six items, it remains cumbersome).

Let us now put together an example that implements an array of **StudentC** objects where **StudentC** is as defined in the previous sections. The program listing is shown in figure 6-11.

Figure 6-11a: Loading a List of StudentC Objects via Arrays — the StudentC class

```
// ***********************************************************************************************************************************************
// Program: StudentList02B: Builds a list of StudentC objects
// Class: StudentC
// Author:  E. Foster
// ***********************************************************************************************************************************************
#include <cstdlib>      #include <iostream>     #include<ctype.h>      #include<string.h>
using namespace std;

typedef char* cString;
class StudentC
{
 float studGPA;  // known only to class members
 protected:
 int studID_Number;
 char studLastName[16];
 char studFirstName[16];
 int studDateOfBirth;
 char studMajor[31];

// member function prototypes
 public:
 StudentC(); // Constructor
 void modify (StudentC aStudent);
 void inputData(int x);
 void printMe();
 void determineGPA (float inScore [ ]);
 int getNumber();
 String getName();
}; // End of Student class declaration

// The Member Functions of StudentC
// Constructor of StudentC
StudentC :: StudentC( )
{
 int y;    studID_Number = 0;     studDateOfBirth = 19000101;    studGPA = 0.0;
 for (y = 1; y <= 15; y++) studLastName[y-1] = studFirstName[y-1] = ' '; // 15 spaces
 for (y = 1; y <= 30; y++) studMajor[y-1] = ' '; // 30 spaces
 }

// Member function modify of StudentC
void StudentC :: modify (StudentC aStudent)
{
 studID_Number = aStudent. studID_Number;
 strcpy(studLastName, aStudent. studLastName); // studLastName = aStudent. studLastName;
 strcpy(studFirstName, aStudent. studFirstName); // studFirstName = aStudent. studFirstName
 strcpy(studMajor, aStudent. studMajor); // studMajor = aStudent. studMajor;
 studDateOfBirth = aStudent. studDateOfBirth;
}

// … Continued on next page
```

Figure 6-11a: Loading a List of StudentC Objects via Arrays — the StudentC class (continued)

```
// Member function inputData of StudentC
void StudentC :: inputData (int x)
{
 int sDoB;
 cout << "\nStudent Information Entry for student " << x << "\n\n";
 cout << "Please enter the following: \n";
 cout << "studID Number:   " ; cin >> studID_Number; getchar();
 cout << "\nSurname:  " ; gets(studLastName);
 cout << "\nFirst Name:  "; gets(studFirstName);
 cout << "\nMajor:  "; gets(studMajor);
 cout << "\nDate of Birth (YYMMDD):  ";  cin >> sDoB; getchar();
 studDateOfBirth = sDoB;
}

// Member function printMe of StudentC
void StudentC :: printMe ( )
{
 // cout<< "\nStudent Information: \n";
 cout<< "ID Number:   " << studID_Number << endl;
 cout<< "Name:  " << studFirstName << " " << studLastName << endl;
 cout<< "Date of Birth:  " << studDateOfBirth << endl;
 cout<< "Major:  " << studMajor << endl;
 cout<< "GPA:  " << studGPA << endl << endl;
}

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

// Member function GetNumber of StudentC
int StudentC :: getNumber ( )  { return studID_Number; }

// Member function getName of StudentC
String StudentC :: getName ( )
{ cString myName = new char[31];
  for (int x=1; x <= 31; x++) myName[x-1] = ' ';
  strcpy(myName, studFirstName);        strcat(myName, " ");      strcat(myName, studLastName);
  return myName;
}

// … Continued on the next page
```

Figure 6-11b: Loading a List of StudentC Objects via Arrays — the Driver Class

```
// ********************************************************************************************************************************************
// Program: StudentList02B: Builds a list of StudentC objects
// Class/File: StudentMain
// Author:  E. Foster
// ********************************************************************************************************************************************
#include <cstdlib>      #include <iostream>    #include<ctype.h>      #include<string.h>      #include<"StudentC.h">
using namespace std;

// Global Variables and function prototypes
typedef char* cString;
const cString HEADING = "List of Student Objects";
void printList(StudentC studL [ ], int sLimit);

// Main function
int main(int argc, char *argv[])
{
 // Declarations
 bool exitTime = false;   char exitKey;     int x, sLim;

 while (!exitTime) // While user wishes to continue
 {
  // Prompt the user for the number of StudentC objects and create the array
  cout << HEADING; cout << "\n\nPlease enter the number of students required: ";   cin >> sLim; getchar();
  StudentC studList [sLim];  // Array of Lim StudentC objects
  // …
  // Prompt for information on each student
  for (x = 1; x <=sLim; x++)
  {        studList [x-1] = StudentC(); // Instantiate the item
           studList [x-1].inputData(x);   // Obtain  information for the item
  }

  // Print the list, then check whether user wishes to continue
  printList(studList, sLim);
  cout << "\n Press any key to continue or X to exit ";   exitKey = getchar();
  if (toupper(exitKey) == 'X') exitTime = true;

 } // End of While-user -wishes-to-continue

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

// Function PrintList
void printList (StudentC studL [ ], int sLimit)
{
 int x;
 cout << "\nYou have entered information for " << sLimit << " " << "Student(s)";
 cout << " as follows: \n";
 for (x = 1; x <= sLimit; x++) studL [x-1].printMe( );
} // End of printList function
```

6.8 Pointers to Objects

Like other variables, you can access a class object directly or indirectly via a pointer. To access a specific member of the referenced object, through the pointer, you use the arrow (→) operator instead of the indirection (*) and dot (.) operators.

Example 6-10: Assuming declarations of sections 6.3 and 6.4, the following code (figure 6-12) creates and updates pointers to twenty **StudentC** objects.

Figure 6-12: Illustrating Pointers to StudentC Objects

```
int main(int argc, char *argv[])
{
   StudentC* studPtr;    // Pointer to StudentC
   StudentC      stud, dummy;    // Instances of StudentC
   int x, sLim = 20;
   // ...
   StudentC* sTracker[sLim];
   for (x = 1; x <=Lim; x++) // For-each StudentC object
   {
          stud = StudentC(); // Instantiate Stud
          stud.inputData(x); // Obtain  information for the item
          studPtr = new StudentC;
          studPtr->modify(stud); // Store the value
          // (*studPtr).modify(stud); // (*studPtr) = stud; These are alternate ways of storing the value
          sTracker [x-1] = studPtr; // Store the pointer
  }       // End of For-each StudentC object
   // ...
}
```

As done in the previous section, we can now put together an example that implements pointers to several **StudentC** objects where **StudentC** is as defined in the previous sections. The program listing is shown in figure 6-13.

Figure 6-13a: Implementing a List of StudentC Objects via Pointers — the StudentC Class

```
// ********************************************************************************************************************************************
// Program: StudentList02D: Builds a list of StudentC objects
// Class: StudentC
// Author:  E. Foster
// ********************************************************************************************************************************************
#include <cstdlib>      #include <iostream>    #include<ctype.h>      #include<string.h>
using namespace std;

typedef char* cString;
class StudentC
{
 float studGPA;  // known only to class members
 protected:
 int studID_Number;
 char studLastName[16];
 char studFirstName[16];
 int studDateOfBirth;
 char studMajor[31];

// member function prototypes
 public:
 StudentC(); // Constructor
 void modify (StudentC aStudent);
 void inputData(int x);
 void printMe();
 void determineGPA (float inScore [ ]);
 int getNumber();
 String getName();
}; // End of Student class declaration

// The Member Functions of StudentC
// Constructor of StudentC
StudentC :: StudentC( )
{
 int y;    studID_Number = 0;     studDateOfBirth = 19000101;    studGPA = 0.0;
 for (y = 1; y <= 15; y++) studLastName[y-1] = studFirstName[y-1] = ' '; // 15 spaces
 for (y = 1; y <= 30; y++) studMajor[y-1] = ' '; // 30 spaces
 }

// Member function modify of StudentC
void StudentC :: modify (StudentC aStudent)
{
 studID_Number = aStudent. studID_Number;
 strcpy(studLastName, aStudent. studLastName); // studLastName = aStudent. studLastName;
 strcpy(studFirstName, aStudent. studFirstName); // studFirstName = aStudent. studFirstName
 strcpy(studMajor, aStudent. studMajor); // studMajor = aStudent. studMajor;
 studDateOfBirth = aStudent. studDateOfBirth;
}

// … Continued on next page
```

Figure 6-13a: Loading a List of StudentC Objects via Pointers — the StudentC class (continued)

```
// Member function inputData of StudentC
void StudentC :: inputData (int x)
{
 int sDoB;
 cout << "\nStudent Information Entry for student " << x << "\n\n";
 cout << "Please enter the following: \n";
 cout << "studID Number:   " ; cin >> studID_Number; getchar();
 cout << "\nSurname:  " ; gets(studLastName);
 cout << "\nFirst Name:  "; gets(studFirstName);
 cout << "\nMajor:  "; gets(studMajor);
 cout << "\nDate of Birth (YYMMDD):  ";  cin >> sDoB; getchar();
 studDateOfBirth = sDoB;
}

// Member function printMe of StudentC
void StudentC :: printMe ( )
{
 // cout<< "\nStudent Information: \n";
 cout<< "ID Number:   " << studID_Number << endl;
 cout<< "Name:  " << studFirstName << " " << studLastName << endl;
 cout<< "Date of Birth:  " << studDateOfBirth << endl;
 cout<< "Major:  " << studMajor << endl;
 cout<< "GPA:  " << studGPA << endl << endl;
}

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

// Member function GetNumber of StudentC
int StudentC :: getNumber ( )  { return studID_Number; }

// Member function getName of StudentC
String StudentC :: getName ( )
{ cString myName = new char[31];
  for (int x=1; x <= 31; x++) myName[x-1] = ' ';
  strcpy(myName, studFirstName);
  strcat(myName, " ");
  strcat(myName, studLastName);
  return myName;
}

// … Continued on the next page
```

Figure 6-13b: Implementing a List of StudentC Objects via Pointers — the Driver Class

```
// ***********************************************************************************************************************************************
// Program: StudentList02D: Builds a list of StudentC objects
// Class/File: StudentMain
// Author:  E. Foster
// ***********************************************************************************************************************************************
#include <cstdlib>      #include <iostream>    #include<ctype.h>      #include<string.h>      #include<"StudentC.h">
using namespace std;

// Global Variables and function prototypes
typedef char* cString;
const cString HEADING = "List of Student Objects";
void printList (StudentC* sTracker [ ], int sLimit)

// Main function
int main(int argc, char *argv[])
{
 // Declarations
 bool exitTime = false;  char exitKey;  int x, sLim;
 StudentC stud, dummy;

 while (!exitTime) // While user wishes to continue
 {
  // Prompt the user for the number of StudentC objects and create the array
  cout << HEADING;  cout << "\n\nPlease enter the number of students required: ";
  cin >> sLim; getchar();
  // …
  // Prompt for information on each student
   StudentC* studPtr; // Pointer to StudentC objects
   StudentC* sTracker [sLim]; // Storage area for each pointer
   for (x = 1; x <=sLim; x++)
   {
    stud = StudentC(); // Instantiates Stud
    stud.inputData(x); // Obtain  information for the item
    studPtr = new StudentC;
    studPtr->modify(stud); // Store the value. Alternately, (*studPtr).modify(stud) or (*studPtr) = stud
    sTracker[x-1] = studPtr; // Store the pointer
   }

   // Print the list, then check whether user wishes to continue
   printList(sTracker, sLim);
   cout << "\n Press any key to continue or X to exit ";  exitKey = getchar();
   if (toupper(ExitKey) == 'X') exitTime = true;

 } // End of While-user -wishes-to-continue

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

// Function printList
void printList (StudentC* sTracker [ ], int sLimit)
{
 int x; cout << "\nYou have entered information for " << Limit << " " << "Student(s)";
 cout << " as follows: \n";
 for (x = 1; x <= sLimit; x++) sTracker[x-1]->printMe( );
} // End of PrintList function
```

6.9 Dynamic List of Objects

In section 5.9 of the previous lecture, it was mentioned that you can construct a dynamic list of structure instances by declaring the list as a pointer and manipulating it as an array. The truth is, you can do this with any C++ object, so here is a more general statement:

> You can construct a dynamic list of objects by declaring the list as a pointer and manipulating it as an array.

The guidelines from section 5.9 are repeated here for emphasis:

- Create a pointer of the base-type of interest.
- Determine the number of items to be added to the list at a specific point in time.
- Use the **new** operator to allocate memory for items to be added to the list (possibly in addition to items already in the list).
- Construct an iterative loop for adding the new items to the list, treating the list as an array.

Here again is the syntax for constructing a dynamic list (figure 6-14):

Figure 6-14: Syntax for Defining a Dynamic List, and Allocating Memory for it

```
DynamicListDefinition ::=
<BaseType>* <TargetList>; // You must specify the base-type and the name of the target-list
// …
<TargetList> = new <BaseType>"["<Length>"]"; // You must specify the amount of items
```

Example 6-11: Assuming declarations of figures 6.3 and 6.4, the following code (figure 6-15) creates a dynamic list of **StudentC** objects and manipulates it via the use of an array.

Figure 6-15: Manipulating a Dynamic List of StudentC Objects

```
// ...
typedef StudentC* StudentList;
// ...
// Obtain list of students
int sLim = 0;
 StudentList  sList = inputStudents(sLim);
// ...
StudentList inputStudents(int &lSize)
{
  cout << "\n\nPlease enter the number of students required: ";
  cin >> lSize; getchar();
  StudentList rList = new StudentC[lSize]; // Allocate space for lSize StudentC objects

  // Prompt for information on each student
  for (int x = 1; x <=lSize; x++)
  {       rList[x-1] = StudentC(); // Instantiate the item
          rList[x-1].inputData(x);   // Obtain  information for the item
  }
  return rList;
} // End of inputStudents Function
```

6.9 Dynamic List of Objects (continued)

Example 6-12: Now let us put this all together in another sample program that manipulates a dynamic list of **StudentC** objects. As the theory prescribes, the list will be declared as a pointer but manipulated via arrays. This is illustrated in figure 6-16. The **StudentC** class is exactly as depicted in figure 6-11a and 6-13a, so focus your attention on the second portion of the figure.

Figure 6-16: Program Listing for Manipulating a Dynamic List of StudentC Objects

```
// ********************************************************************************************************************************************
// Program: StudentList02E: Builds a list of StudentC objects
// Class: StudentC
// Author:  E. Foster
// ********************************************************************************************************************************************
#include <cstdlib>      #include <iostream>    #include<ctype.h>      #include<string.h>
using namespace std;

typedef char* cString;
class StudentC
{
 float studGPA;  // known only to class members
 protected:
 int studID_Number;
 char studLastName[16];
 char studFirstName[16];
 int studDateOfBirth;
 char studMajor[31];

// member function prototypes
 public:
 StudentC(); // Constructor
 void modify (StudentC aStudent);
 void inputData(int x);
 void printMe();
 void determineGPA (float inScore [ ]);
 int getNumber();
 String getName();
}; // End of Student class declaration

// The Member Functions of StudentC are as specified in figure 6-13a. They will not be duplicated here; see figure 6-13a for such details
```

```
// ********************************************************************************************************************************************
// Program: StudentList02E
// Class/File: StudentMain
// Author:  E. Foster
// Purpose: Builds a dynamic list of StudentC objects - Pointer & array implementation
// ********************************************************************************************************************************************
// Note: This is an improvement of the program EFStudList02B of EFConProject7B of the resource library
// ********************************************************************************************************************************************
#include <cstdlib>      #include <iostream>    #include<ctype.h>      #include<string.h>      #include "StudentC.h"
using namespace std;

// Continued on the next page
```

Figure 6-16: Program Listing for Manipulating a Dynamic List of StudentC Objects (continued)

```
// Global Variables
typedef char* cString;
typedef StudentC* StudentList;
const cString HEADING = "Process of Student Objects";

//Other Function Prototypes
StudentList inputStudents(int &lSize);
void printList(StudentList sList, int sLimit);

// Main function
int main(int argc, char *argv[])
{
 // Declarations
bool exitTime = false;   char exitKey;
 StudentList studList; int studLim;

 while (!exitTime) // While user wishes to continue
 {
  // Obtain list of students
  studLim = 0;   studList = inputStudents(studLim);

  // Print the list, then check whether user wishes to continue
  printList(studList, studLim);
  cout << "\n Press any key to continue or X to exit ";    exitKey = getchar();
  if (toupper(exitKey) == 'X') exitTime = true;

 } // End of While-user -wishes-to-continue

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

// InputStudents Function
StudentList inputStudents(int &lSize)
{
  cout << HEADING;   cout << "\n\nPlease enter the number of students required: ";
  cin >> lSize; getchar();
  StudentList rList = new StudentC [lSize]; // Allocate space for lSize StudentC objects

  // Prompt for information on each student
  for (int x = 1; x <=lSize; x++)
  {        rList [x-1] = StudentC(); // Instantiate the item
           rList [x-1].inputData(x);   // Obtain  information for the item
  }
  return rList;
} // End of inputStudents Function

// Function printList
void printList (StudentList sList, int sLimit)
{
 int x;  cout << "\nYou have entered information for " << sLimit << " " << "Student(s)";
 cout << " as follows: \n";
 for (x = 1; x <= sLimit; x++) sList [x-1].printMe( );
} // End of printList function
```

6.10 Friend Functions

You can allow a non-member function to have access to the data members of a class by declaring the function as a *friend* of the class.

To declare a friend function of a class, include a prototype of the function, preceded by the keyword, **friend**, in the public section of the class. In specifying the friend function, no scope resolution operator is required, since the friend function is not a member function.

Example 6-13: In the following code (figure 6-17), the friend function **bestWorst** has access to the **StudentC** class.

Figure 6-17: Illustrating Friend Class

```
typedef char* cString;
class StudentC
{
 float studGPA;  // known only to class members
 protected:
 int studID_Number;  char studLastName[16];  char studFirstName[16];
 int studDateOfBirth;  char studMajor[31];

// member function prototypes
 public:
 StudentC(); // Constructor
 void modify (StudentC aStudent);
 void inputData(int x);
 void printMe();
 void determineGPA (float inScore [ ]);
 int getNumber();
 String getName();
 friend void bestWorst (StudentC thisStud);  // The friend function
}; // End of Student class declaration

// Detailed member function specifications follow
// ...
// Global declarations
StudentC bestStud, worstStud;
// ...
// main function
int main(int argc, char *argv[ ])
{       // ...
        StudentC currentStud;  currentStud.inputData(x);  // Assume that x was defined and is assigned a value
        // ...
        bestWorst (currentStud);
        // ...
        return EXIT_SUCCESS;
}
// ...
void bestWorst (StudentC thisStud)
//Assume that bestStud and worstStud are global objects
{  if  (thisStud.studGPA > bestStud.studGPA) bestStud.modify(thisStud); // bestStud = thisStud;
   if  (thisStud.GPA < worstStud.studGPA) worstStud.modify(thisStud); // worstStud = thisStud;
}
```

6.10.1 Several Classes May Share a Common Friend

Friend functions are useful in the following situations:

- Operator overloading (to be discussed later)
- Reusability of code — where different classes need to employ the services of a given function, the function is declared as a friend function to each class that utilizes it. Alternately, a class may use a member function of another class as its friend function. In this situation, the scope resolution operator must be used when declaring the prototype of the friend function.

Example 6-14: In the following example (figure 6-18), the overloaded function called **below( )** compares the numeric portion of an instance of the **Sample1** class or the **Sample2** class with a numeric value received as an argument, and returns **true** if the numeric value is greater than the numeric portion of the object.

Figure 6-18: Illustrating Overloaded Friend Functions

```
// ...
class Sample1
{
   int number;
   public:
   void setValue (int newVal)
   { number = newVal; } // In-line specification of the setValue function
   friend bool below (Sample1 sam1, int thisval);
}; // End of Sample1 class declaration

// ...
class Sample2
{
   int number;
   char status;
   public:
   void setValue (int newVal, char newStatus)
   { number = newVal; Status newStatus;} // In-line specification of the setValue function
   friend bool below (Sample2 sam2, int thisval); // prototype
 } // End of Sample2 class declaration
// ...
// The main function
int main(int argc, char *argv[ ])
{
Sample1 s1;
Sample2 s2;
// ...
s1.setValue(24);
s2.setValue(30, 'A');
if ((below (s1, 50)) || (below (s2, 50)) ) cout << "Low value";
…
 return EXIT_SUCCESS;
} // End of main function

// Other functions follow
bool below (Sample1 sam1, int thisval)
{ if Sam1.number < thisval return true; else return false; }
bool below (Sample2 sam2, int thisval)
{ if sam2.number < thisval return true; else return false; } // overloaded
```

6.10.2 Making a Friend of a Member of Another Class

In making a friend of a member of another class, three steps are typically required:

- Forward declare the class that is granting friendship (a forward declaration simply has the keyword **class**, the name of the class and a semicolon).
- Declare the class that has the member function that will receive the friendship.
- Declare the class that is granting the friendship.

Example 6-15: The following example (figure 6-19) is similar to example 6-14, except that here, the function called **below( )** is defined as a member function of the **Sample1** class.

Figure 6-19: Illustrated Friend Function that Belongs to a Class

```
class Sample2;  //forward declaration for the class granting friendship
class Sample1 // this class has the member function that will receive the friendship
{
  int number;
  public:
  void setValue (int newVal)
  { number = newVal; } // In-line specification of the setValue function
  bool below (Sample1 sam1, int thisVal);
  bool below (Sample2 sam2, int thisval);
};
bool Sample1 :: below (Sample1 sam1, int thisval)
{ if sam1.number < thisval return true; else return false; }
bool Sample1 :: below (Sample2 sam2, int thisval)
{ if sam2.number < thisval return true; else return false; } // overloaded
// ...
class Sample2 // This class is granting the friendship
{
  int number;
  char status;
  public:
  void setValue (int newVal, char newStatus)
  { number = newVal; satus = newStatus;} // In-line specification of the setValue function
  friend bool Sample1 ::  below (Sample1 sam1, int thisval); // prototype
  friend bool Sample1 ::  below (Sample2 sam2, int thisval); // prototype
};
// ...
int main(int argc, char *argv[ ])
{
  Sample1 s1;
  Sample2 s2;
  // ...
  s1.setValue(24);
  s2.setValue(30, 'A');
  if ((s1.below (s1, 50)) || (s1.below (s2, 50))) cout << "Low Value";
  // ...
  return EXIT_SUCCESS;
}
```

6.10.3 Making a Friend of Another Class

You can bestow friendship on several member functions of another class. You may also bestow friendship on all member functions of another class, either by making each member function a friend, or by making the entire class, a friend. To make friend of another class, simply specify

```
friend class <className>;
```

within the class granting the friendship, along with any other member function prototypes of the class.

6.11 Assigning Objects

As you might have figured by now, C++ allows you to assign one object to another. Object assignment may occur in any of the following four ways:

- Using the assignment statement
- Specifying the object as an argument to a function defined with a matching reference parameter
- Returning the object from a function
- Using the member function defined for that purpose (typically called **modify(. . .)** in this course)

6.11.1 Assigning Objects and Passing Objects as Arguments

If two objects are created (declared) of the same class, then one may be assigned to the other in a typical assignment statement, as illustrated in figure 6-17 of the previous section. If the objects are not of the same type, then casting must be done. To reemphasize, if **stud1** and **stud2** are instances of the class **StudentC** of earlier discussions (for instance figures 6-12 and 6-13), then we may assign **stud1** the value of **stud2** or by working through the object modification member function. Thus, the following two statements would achieve the same end:

`stud1 = stud2;` is an alternative to `stud1.modify(stud2);`

You can also pass/return an object (or list of objects) as an argument to/from a function; figures 6-11, 6-13, 6-15, and 6-16 provide examples of these concepts. In passing objects to and from functions, please note the following:

1. When an object is passed as parameter to a function, a bit-wise copy of the argument is made to the function's parameter. The object's constructor is not called.

2. On return from the function, the object's destructor is automatically called, since the object copy is deleted once the scope changes.

6.11.1 Assigning Objects and Passing Objects as Arguments (continued)

3. If the destructor frees up allocated memory, this act will damage the original object on return from the function. This is due to the fact that the copy's destructor will free the space occupied by the original object. To avoid this dilemma, do not pass as arguments to a function, an object whose destructor de-allocates memory space. If the constructor allocates memory via the **new** operator, and the destructor de-allocates memory via **delete**, and you must pass the object as an argument, either make the object global (so the function sees it), or pass either a pointer to or a reference to the object rather than the object itself.

4. Any constructor that uses the **new** operator to allocate memory for instance(s) of that class should have a check for the null pointer. Since occurrence of a returned null pointer by **new** is a fatal error, once detected, you should immediately terminate the program normally. One way to do this is via the **exit()** function (of header file **<stdlib.h>** or **<cstdlib.h>**). The following example illustrates.

Example 6-16: The following code (figure 6-20) illustrates the use of **new** and **delete** keywords, as well as the **exit( )** function as they relate to object passing between functions.

Figure 6-20: Illustrating use of new and delete Operators

```
// Illustrating the use of new, delete, and exit ( ) in constructors and destructors.
// …
class SampleClass
{
  int* someNumner;
  public:
  SampleClass (int inNumber);  // constructor prototype
  ~SampleClass ( ) {delete someNumner;}
  void display ( ) {cout << * someNumner;}
};
SampleClass :: SampleClass (int Number) // constructor
{
  someNumner = new int;
  if (!someNumner) {cout << "Allocation failure"; exit(1)}; // If null pointer, exit
  * someNumner = inNumber;
}

int main(int argc, char *argv[ ])
{
  // …
  SampleClass thisNumber (50);
  // …
  thisNumber.display ( );
  // …
  return EXIT_SUCCESS;
}
```

6.11.2 Returning Objects from a Function

As you might have thought, you can return an object from a function. Similar to the problem cited in the previous section, if the object returned has a destructor which de-allocates memory space, returning the object could be problematic.

To avoid this situation try one of the following alternatives:
- Return a pointer to the object
- Use a global object
- Return the object only if its destructor does not involve the use of the **delete** operator on the object
- Use a *copy-constructor* or its equivalent (discussed next)

6.12 Copy Constructor and its Alternative

As we have seen in the previous section, it can be problematic to pass to, or return from a function, an object that has a destructor that de-allocates memory space (allocated by the constructor).

The problem arises when a copy of an object is made (as in the passing of function argument(s)). Because the copy is bit-wise, if the object contains pointers to memory locations, the copy also contains the pointers to the exact locations. When the copy is deleted (due to change of scope) and the destructor is called, these pointers are destroyed, thus affecting the original objects.

Another way to avoid this messy situation is to use a *copy-constructor*. The copy constructor avoids this situation by specifying exactly what action(s) must take place when a copy of an object is made. The copy constructor applies to object initialization only, not assignment. Once defined, the copy constructor takes precedence over the default bit-wise copy at object initialization.

The most common format of a copy constructor is as follows (note that this format facilitates referencing an object that is being used to initialize another object class):

```
<ClassName> (const <ClassName> &<ThisObject>)
{
… //   Body of the constructor
}
```

Example 6-17: The following code (figure 6-21) illustrates the use of the copy constructor

Figure 6-21: Illustrating us of Copy Constructor

```
class    SampleClass
{
   int *someNumber;
   public:
   SampleClass (int inNumber); // constructor prototype
   SampleClass (const SampleClass &copySample); // copy constructor prototype
   ~SampleClass ( ) { delete someNumber; } // destructor
};
// ...
SampleClass:: SampleClass (int inNumber)                  // Normal constructor
{
  someNumber = new int;
  if (!someNumber) {cout << "Allocation failure"; exit(1)};
  *someNumber = inNumber;
}
SampleClass:: SampleClass (const SampleClass, &copySample)  // Copy constructor
{
  someNumber = new int;
  if (!someNumber) {cout<< "Allocation failure"; exit (1)};
  *someNumber = *CopySample.someNumber;  // Copy value
}
// ...
int main(int argc, char *argv[ ])
{
  SampleClass testSample (20);
  display (testSample);
  // ...
  return EXIT_SUCCESS;
}

void display (SampleClass current)
{
 cout << *current.someNumber;
}

/* Explanation of operation:
At creation testSample's normal constructor is called with argument 20. When display(...) is called, the copy constructor copies
testSample to copySample, which is implicitly linked to current. On the call of display(...), current is displayed which means that
copySample is displayed. On exit from display(...), current's (i.e. copySample's) destructor is called to delete current (i.e.
copySample); testSample remains in tact until the program terminates. */
```

As an alternative to the use of copy-constructors, you can make use of the **modify(…)** method as defined and recommended in sections 6.4 – 6.9. Whenever an object is to be passed as an argument to a function, simply create a temporaryinstance of the class, set its data values to those of the original argument via the **modify(…)** method, and pass the temporary instance as the argument.

Example 6-18: The following (figure 6-22) code illustrates the use of an alternative to copy constructor.

Figure 6-22: Illustrating an Alternative to Using Copy-Constructor

```
class    SampleClass
{
   int *someNumber;
   public:
   SampleClass (int inNumber); // constructor prototype
   modify (SampleClass aSample);
   ~SampleClass ( ) { delete someNumber; } // destructor
};
// ...
SampleClass:: SampleClass (int inNumber)  // Normal constructor
{
  someNumber = new int;
  if (!someNumber) {cout << "Allocation failure"; exit(1)};
  *someNumber = inNumber;
}
SampleClass:: modify (SampleClass aSample)
{
  *someNumber = aSample.someNumber;
 }
// ...
int main(int argc, char *argv[ ])
{
  SampleClass testSample (20);
  SampleClass tempSample(0); tempSample.modify(testSample);
  display (tempSample);
  // ...
  return EXIT_SUCCESS;
} // End of main function

void display (SampleClass current)
{
 cout << *current.someNumber;
}

/* Explanation of operation:
Prior to the call of display(…), tempSample is instantiated and its value(s) modified via the modify(…) function. On exit from
display(…), current and tempSample are deleted, but testSample remains in tact. */
```

6.13 The *this* Keyword

Each invocation of a class's member function automatically triggers the passing of a pointer called **this** to the member function. The **this** pointer points to the host (current) object that invoked the member function.

As you will see in upcoming lecture(s), it is often desirable to return the current object for further processing, from a function. This is quite nicely facilitated via use of the **this** keyword (more on this later in the course).

By using structures and classes, the C++ programmer can significantly simplify his/her program and complexities of internal working within member functions of classes.

6.14 The string Class

So far, the approach to string manipulation that has been discussed is based on the traditional null-terminated strings (also called C-strings). Contemporary versions of C++ provide a **string** *class* in the header file **<string>**. This class provides a constructor that allows you to define a string instance in the normal way that you would define any variable on a primitive data type. It also provides a number of member functions that facilitate easy manipulation of strings. Figure 6-23 provides a UML class diagram showing some of the commonly used member functions of the class.

Figure 6-23: Abbreviated UML Diagram of the string Class

string
// Data items of the class are not included here

// Member function signatures for the publicly accessible functions of the class
 	
// Constructors
string() // Basic constructor
string(const string& s2) // Copy constructor
string(const char* nts); // convert from C-string
string(const char* buf, size_type bufsize); // From creating a sting of finite size
 	
// Assignment Operators — overloaded to work with strings
string& operator=(const string& s2) // normal assignment
string& operator=(const char* nts) // convert from C-string
string& operator=(char c) // assign a single character
 	
// Assignment Functions
string& assign(const string& s2)
string& assign(const string& s2, size_type pos2, size_type len2) // Takes specific number of characters from s2
string& assign(const char* nts)
string& assign(const char* buf, size_type buflen) // Takes specific number of characters from s2
void swap(string& s2) // Switches the values of the host string and s2
 	
// Continues on next page

Figure 6-23: Abbreviated UML Diagram of the string Class (continued)

```
// Iteration Functions
iterator  begin() // Return iterator to beginning
iterator  end() // Return iterator to end
iterator  rbegin() // Return reverse iterator to beginning
iterator  rend() // Return reverse iterator to end

// Size and Capacity Functions
size_type size() // Returns the size
size_type length()// Same as size()
size_type max_size() // Returns the maximum size
void resize(size_type size, char c = '\0') // Re-sizes the string
void clear() // Clears the string
bool empty() // Determines whether the string is empty

// Appendage Operators — overloaded to work with strings
string& operator+=(const string& s2)
string& operator+=(const char* nts)
string& operator+=(char c)

// Appendage Functions — overloaded to work with strings
string& append(const string& s2)
string& append(const string& s2, size_type pos2, size_type len2) // Appends specific number of characters from s2
string& append(const char* nts)
string& append(const char* buf, size_type buflen) // Appends specific number of characters from s2

// Insertion Functions
string& insert(size_type pos1, const string& s2) // Inserts s2, starting at pos1 in the host string
string& insert(size_type pos1, const string& s2, size_type pos2, size_type len2)
  // Copies len2 characters from s2, starting at pos2, and inserts them in the host string, starting at pos1
string& insert(size_type pos1, const char* nts)
  // Inserts null-terminated string (C-string) nts, starting at pos1 in the host string
string& insert(size_type pos1, const char* buf, size_type buflen)
  // Inserts buflen characters of buf into the host string, starting at pos1 in the host
string& insert(size_type pos1, size_type repetitions, char c)
  // Repeat the character stored in c for the specified times, and insert in the host string, starting at pos1

//Erase Function
string& erase(size_type pos = 0, size_type len = size()) // Eraces characters from the string

// Replacement Functions
string& replace(size_type pos1, size_type len1, const string& s2)
string& replace(size_type pos1, size_type len1, const string& s2,  size_type pos2, size_type len2)
string& replace(size_type pos1, size_type len1, const char* nts)
string& replace(size_type pos1, size_type len1, const char* buf, size_type buflen)
string& replace(size_type pos1, size_type len1, size_type repetitions, char c)

// Comparison Functions
// compare function returns -1 if *this < s2, 0 if *this == s2, and +1 if *this > s2.
int compare(const string& s2)
int compare(size_type pos1, size_type len1, const string& s2)
int compare(size_type pos1, size_type len1, const string& s2, size_type pos2, size_type len2)
int compare(const char* nts)
int compare(size_type pos1, size_type len1, const char* nts)
int compare(size_type pos1, size_type len1, const char* buf, size_type buflen)
```

Figure 6-23: Abbreviated UML Diagram of the string Class (continued)

```
// Substring Function and Swap Function
string substr(pos = 0, len = size()) // Returns a substring from the host string
void swap(string s2)

// Search Functions
size_type find(const string& s2, size_type pos1) // Attempts to find s2 in the host string, starting at pos1
size_type find(const char* nts, size_type pos1)
size_type find(const char* buf, size_type pos1, size_type bufsize)
size_type find(char c, size_type pos1)
size_type rfind(const string& s2, size_type pos1) // Attempts to find the last occurrence of s2 in the host string
size_type rfind(const char* nts, size_type pos1)
size_type rfind(const char* buf, size_type pos1, size_type bufsize)
size_type rfind(char c, size_type pos1)
size_type find_first_of(const string& s2, size_type pos1) // Attempts to find first occurrence of s2 in the host string
size_type find_first_of(const char* nts, size_type pos1)
size_type find_first_of(const char* buf, size_type pos1, size_type bufsize)
size_type find_first_of(char c, size_type pos1)
size_type find_last_of(const string& s2, size_type pos1) // Attempts to find first occurrence of s2 in the host string
size_type find_last_of(const char* nts, size_type pos1)
size_type find_last_of(const char* buf, size_type pos1, size_type bufsize)
size_type find_last_of(char c, size_type pos1)
size_type find_first_not_of(const string& s2, size_type pos1)
  // Attempts to find the first occurrence a pattern not in the search argument
size_type find_first_not_of(const char* nts, size_type pos1)
size_type find_first_not_of(const char* buf, size_type pos1, size_type bufsize)
size_type find_first_not_of(char c, size_type pos1)
size_type find_last_not_of(const string& s2, size_type pos1)
    // Attempts to find the last occurrence a pattern not in the search argument
size_type find_last_not_of(const char* nts, size_type pos1)
size_type find_last_not_of(const char* buf, size_type pos1, size_type bufsize)
size_type find_last_not_of(char c, size_type pos1)

// Conversion to C-string
char* c_str() // Returns an equivalent null-terminated string for the current string
```

A careful scrutiny of figure 6-23 will confirm that the **string** class allows you to define strings and manipulate them as you would with other objects (variables) defined on primitive data types. Specifically, here is a list of some operations you can carry out on strings:

- Assign values to strings via the assignment operator (=), or the **assign (…)** function
- Concatenate strings via the overloaded + operator (operator overloading will be more fully discussed in the next lecture), and/or the **append(…)** function
- Use shortcut += operator to append to strings
- Determine the length of strings
- Clear strings of non-blank data
- Check if a string is empty
- Insert strings into strings
- Replace portions of a string with another string
- Compare strings lexically
- Search for patterns in strings
- Extract substrings from strings

6.14 The string Class (continued)

In addition to the above, you should also know that C++ allows you to use the relational and shift operators (==, !=, <, <=, >, >=, <<, >>) on strings. One way this is conceivably achieved is through friend functions. Nonetheless, how this is achieved is not critical at this point. What is more important is to note that this provision makes absolute sense; it provides additional flexibility and convenience when manipulating strings.

Example 6-19: The following (figure 6-24) code provides an illustration of string manipulation via the **string** class.

Figure 6-24: Illustrating String Manipulation

```
#include <string.h>      #include <cstdlib>      #include <iostream>
using namespace std;

int main(int argc, char *argv[])
{
  bool exitTime  = false;
  while (!exitTime) // While user wishes to continue
  {
    char more, dummy;
    string firstName1 = " ";  string lastName1 = " ";  string midName1 = " ";  string message1 = " ";
    string message2 = "My name is Bruce Farnsworth Jones.\n";
    string initials1; string initialsMessage1 = "Your initials are ";

    //Prompt the user to enter a name
    cout << "Illustration of String Manipulation  via string Class\n";
    cout << "Please enter your first name: \n";
    cin >> firstName1;

    cout << "Please enter your middle name: \n";  cin >> midName1;
    cout << "Please enter your last  name: \n";     cin >> lastName1;

    // Construct a message for the user, and display it
    message1 = "Nice meeting you, " + firstName1 + " " + midName1 + " " + lastName1 + ".\n";
    initials1 = firstName1.substr(0, 1) + midName1.substr(0, 1) + lastName1.substr(0, 1);
    initialsMessage1 += initials1;
    cout << message1 << initialsMessage1 << endl;        cout << message2 << endl;

    // Now swap message1 and message2
    message1.swap(message2);
    cout << "\nI will now swap the first message containing your name with the second message containing my  +
          name.\n";
    cout << message1 << endl;   cout << message2 << endl;

    // Prompt for more
    cout << endl << "Press X to exit or any other letter to continue \n";
    more = getchar(); dummy = getchar();
    if ((More == 'X') || (More == 'x'))  ExitTime = true;
  } // End While user wishes to continue

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

6.15 Summary and Concluding Remarks

It is now time to summarize what has been covered in this lecture:

- OOP is an enhancement of procedural programming, providing a wide range of benefits. OOP supports concepts and abstractions such as object types, classes, objects, methods, encapsulation, inheritance, amalgamation, and polymorphism.
- A class is made up of data items and methods (also called member functions). Collectively, the data items and member functions are called properties (or members). Each property is defined with designation of public, private, or protected.
- The constructor of a class is a special method that is responsible for object instantiation. The destructor is an optional method that is responsible for de-allocation of any memory that might have been allocated by the constructor, and preparing for the object's destruction.
- UML class diagrams are used to summarize the properties of classes.
- You can create and manipulate an array of objects, or a pointer to objects.
- A friend function is a function that is not a property of a class, but upon which the privileges of a member function has been bestowed by the class.
- You can pass objects as arguments to a function, as well as return an object from a function. However, if the object's class has a destructor that de-allocates memory, you should proceed with caution by using either a copy constructor, or a **modify(…)** method as described in sections 6.4 – 6.9.

If you are becoming fascinated with how C++ implements classes, your fascination is warranted. There is much more to learn. The next lecture discusses *operator overloading*. You will see that when this principle is applied to C++ classes, the end result is a powerful programming resource.

6.16 Recommended Readings

[Friedman & Koffman 2011] Friedman, Frank L. & Elliot B. Koffman. 2011. *Problem Solving, Abstraction, and Design using C++*, 6th Edition. Boston: Addison-Wesley. See chapter 10.

[Gaddis, Walters & Muganda 2014] Gaddis, Tony, Judy Walters, & Godfrey Muganda. 2014. *Starting out with C++ Early Objects*, 8th Edition. Boston: Pearson. See chapters 7 & 12.

[Savitch 2013] Savitch, Walter. 2013. *Absolute C++*, 5th Edition. Boston: Pearson. See chapters 6 – 10.

[Savitch 2015] Savitch, Walter. 2015. *Problem Solving with C++*, 9th Edition. Boston: Pearson. See chapters 10, 11 & 8.

[Yang 2001] Yang, Daoqi. 2001. *C++ and Object-Oriented Numeric Computing for Scientists and Engineers*. New York, NY: Springer. See chapter 5.