Lecture 09: Inheritance and Amalgamation

Inheritance is one of the cornerstones of OOP. Like polymorphism, it promotes the reusability of code. This lecture focuses on inheritance in C++. Topics to be covered include:

- Introduction to Inheritance
- Creating an Inheritance Hierarchy
- Access Control in Inheritance
- Multiple Inheritance
- Inheritance Graphs
- Constructors and Destructors in Inheritance
- Virtual Base Classes
- Amalgamation
- Summary and Concluding Remarks

9.1 Introduction to Inheritance

Inheritance is at the heart of object-oriented programming. The concept of inheritance was introduced in lecture 6. In this lecture, we will reinforce the concept by providing additional clarifications and illustrations. In this section, we will clarify the salient concepts:

- Super-classes and Subclasses
- Modeling an Inheritance Hierarchy

9.1.1 Super-class and Subclass

Inheritance is the act of an object adopting the properties (data items and member functions) of the class on which it is defined. Additionally, a class may inherit from another class in its hierarchy. The inheriting class is referred to as the *sub-class* or the *derived class*, or the *child class*. The inherited class is called the *super-class*, or the *base class*, or the *parent class*.

Example 9-1: The following examples should clarify these concepts:

The classes **Student** and **Employee** could be defined as subclasses of the super-class **CollegeMember**.

If **Bruce** is a **Student** object, **Bruce** is an instance of Student and therefore inherits all the inheritable properties of **Studen**t and **CollegeMember**.

If **Karen** is an **Employee** object, **Karen** is an instance of Employee and therefore inherits all the inheritable properties of **Employee** and **CollegeMember**.

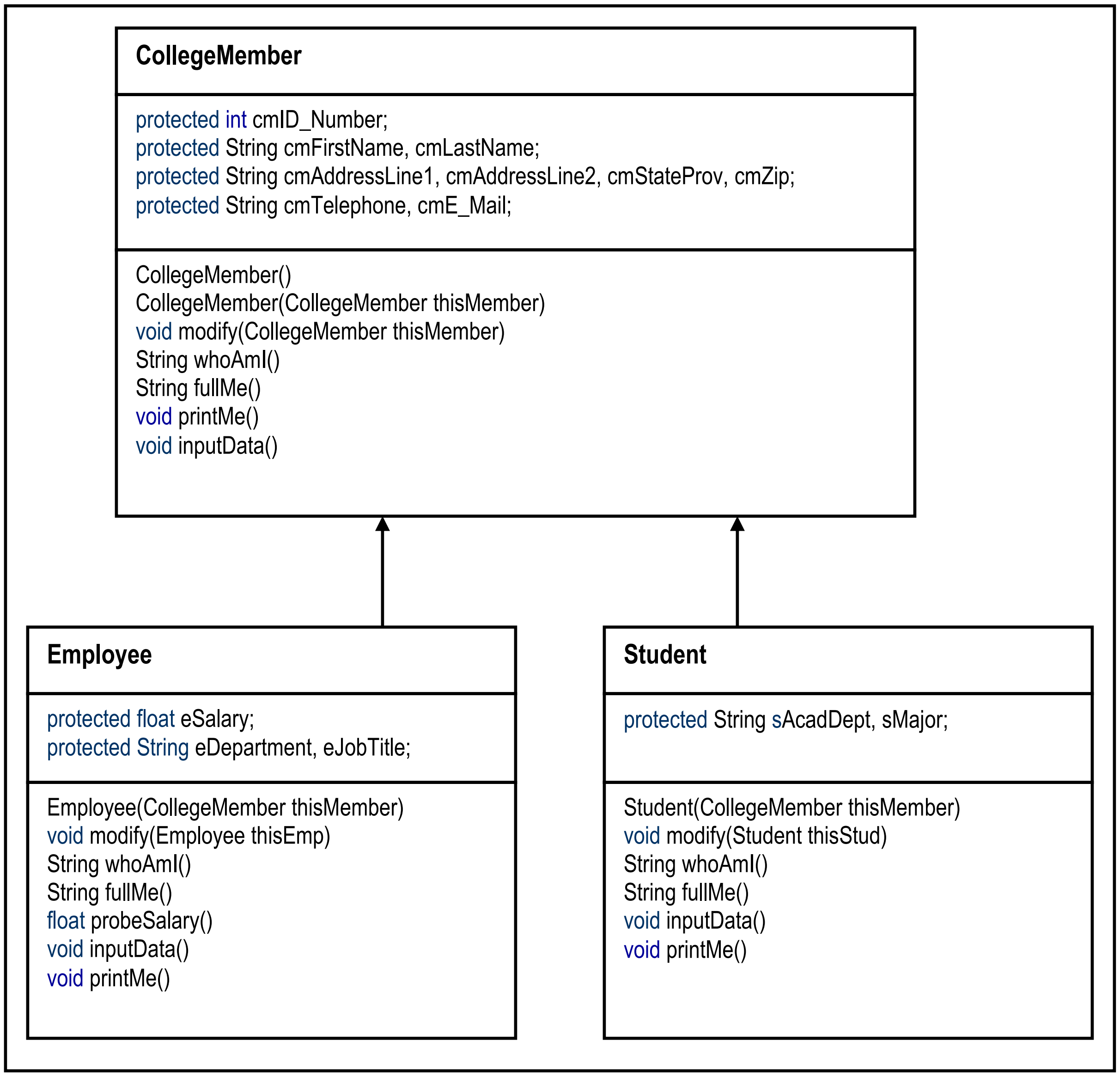
If classes **StudentCS** and **StudentMath** are defined as subclasses of **Student**, then **Student** is both a super-class and a subclass.

9.1.2 Modeling the Inheritance Hierarchy

Before constructing an inheritance hierarchy, we want to be able to represent it. As you have seen from lecture 6, the UML class diagram allows you to summarize the salient properties of a class. We can also use the UML notation to construct a representative model of a a class hierarchy. An inheritance relationship is represented by arrow leading from the subclass and pointing to the super-class. Figure 9-1 illustrates a UML diagram for the class hierarchy involving the classes **CollegeMember**, **Employee** and **Student**.

Notice from the figure, that a number of methods in each subclass have the same name as their counterparts in the superclass. This is deliberate. The methods in the subclass will override its counterpart in the superclass, and thereby achieve polymorphism. The methods **modify(…)**, **whoAmI()**, **fullMe()**, **printMe()**, and **inputData()** all fit this description; they are said to be polymorphic.

Figure 9-1: UML Diagram Representing a College Community


CollegeMember
protected int cmID_Number;
protected String cmFirstName, cmLastName;
protected String cmAddressLine1, cmAddressLine2, cmStateProv, cmZip;
protected String cmTelephone, cmE_Mail;
CollegeMember()
CollegeMember(CollegeMember thisMember)
void modify(CollegeMember thisMember)
String whoAmI()
String fullMe()
void printMe()
void inputData()
Employee
protected float eSalary;
protected String eDepartment, eJobTitle;
Employee(CollegeMember thisMember)
void modify(Employee thisEmp)
String whoAmI()
String fullMe()
float probeSalary()
void inputData()
void printMe()
Student
protected String sAcadDept, sMajor;
Student(CollegeMember thisMember)
void modify(Student thisStud)
String whoAmI()
String fullMe()
void inputData()
void printMe()

9.2 Creating an Inheritance Hierarchy

Due to the principle of inheritance, class hierarchies can be set up so that more specific sub-classes inherit common properties from more generic super classes. In C++, an inherited (super class) is (also) referred to as a *base class*, while the inheriting (sub-class) is (also) referred to as the *derived class*. A class may be inherited and inheriting at the same time.

Specifying within the inheriting class, which super-class is being inherited, sets up an inheritance hierarchy. This is done by including a colon, followed by the name of the inherited class, immediately after the class name of the inheriting class:

Figure 9-2: Revised Syntax for a Class with Inheritance

```
DerivedClassDefinition ::=
class <DerivedClassName> : [<Modifier>] <BaseClassName>
{
    //…body of derived class
}

Modifier ::= public | private | protected
```

The access keyword (**public/private/protected***)* determines how members of the base-class will be accessed.

Public Access:
- All public members of the base-class will be public members of the derived class.
- Protected members of the base-class are also protected in the derived class.
- Private members of the base-class are inaccessible to the derived class.

Protected Access:
- Public base-class members become protected in the derived class.
- Protected base-class members remain protected in the derived class.
- Private base-class members are inaccessible to the derived class.

Private Access:
- Public base-class members become private in the derived class.
- Protected base-class members become private in the derived class.
- Private base-class members are inaccessible to the derived class.

Note: If the access keyword is not used, the default is **private** if the derived type is a class, and **public** if the derived type is a structure. In all cases, private members of a base class are inaccessible to a derived class.

Let us now proceed to implementing the inheritance hierarchy of the previous section. Figure 9-3 shows a summarized C++ implementation. Please note that each subclass heading specifies the super-class being inherited. Also observe that within each subclass, each polymorphic method invokes its counterpart in the super-class (notice how this is done via the scope resolution operator), and then caries out additional activities. Constructors of each subclass are also polymorphic, but in an implicit manner — they invoke their super-class counterparts without programmmer intervention. Finally, note that C++ provides you with much flexibility as to how to implement the program: you may define and store each class in a separate header file, or keep them all in one program file as shown.

Figure 9-3: Implementing the College Hierarchy

```
class CollegeMember
{
    protected:
    int cmID_Number;
    char cmFirstName [16], cmLastName[16];
    char cmAddressLine1 [16], cmAddressLine2 [16], cmStateProv[7], Zip[7], cmTelephone[13];
    char cmE_Mail[31];

    // member functions
    public:
    CollegeMember ( ); // constructor
    void modify (CollegeMember thisMember);
    void inputData(int x);
    void printMe ( );
};

class Employee: protected CollegeMember
{
    float eSalary;
    protected:
    char eDepartment [31], eJobTitle [31];
    // Member Functions
    public:
    Employee ( );        // constructor
    void modify (Employee ThisEmp);
    void inputData(int x);
    void setSalary (float Sal) // Specified as an in-line function
    { eSalary = Sal;}
    float getSalary ( ) {return Salary;} // Specified as an in-line function
    void printMe ( );
};

class Student: protected CollegeMember
{
    public:
    char sAcadDept[31], sMajor [31];
    //Member Functions
    Student ( );      // constructor
    void modify (Student ThisStud);
    void inputData(int x);
    void printMe ( );
};

// ************************************************************************
// CollegeMember Member Functions
// ************************************************************************
CollegeMember :: CollegeMember ( )
{
    int x;
    cmID_Number = 0;
    for (x = 1; x <=15; x++) cmFirstName[x-1], cmLastName[x-1] = ‘  ’;
    for (x = 1; x <=15; x++) cmAddressLine1[x-1], cmAddressLine2[x-1] = ‘  ’;
    for (x = 1; x <= 6; x++) cmStateProv[x-1], cmZip[x-1] = ‘0’;
    for (x = 1; x <=30; x++) cmE_Mail[x-1] = ‘  ’;
}

// continued on next page
```

Figure 9-3: Implementing the College Hierarchy (continued)

```
void CollegeMember :: modify (CollegeMember thisMember)
{
      cmID_Number = thisMember. cmID_Number;
      strcpy (cmFirstName, thisMember.cmFirstName); strcpy (cmLastName, thisMember.cmLastName);
      strcpy (cmAddressLine1, thisMember.cmAddressLine1); strcpy (cmAddressLine2, thisMember.cmAddressLine2);
      strcpy (cmTelephone, thisMember.cmTelephone);
      strcpy (cmE_Mail, thisMember.cmE_Mail);
}

void CollegeMember :: inputData(int x)
{
      cout << "College Member Information Input for Member " << x << endl << endl;
      cout << "ID Number: "; cin >> cmID_Number; getchar();
      cout << "First Name: "; gets (cmFirstName); cout << "Last Name: "; gets (cmLastName);
      cout << "Address: "; gets (cmAddressLine1); gets (cmAddressLine2); gets (cmStateProv); gets (cmZip);
      cout << "Telephone: "; gets(cmTelephone);
      cout << "E-Mail: "; gets(cmE_Mail);
}

void CollegeMember :: printMe( )
{
      cout << "Member: " << cmID_Number << " – " << cmFirstName << " " << cmLastName << endl;
      cout<< "Address Line1: "  << cmAddressLine1 <<  endl;
      cout<< "Address Line2: "  << cmAddressLine2 <<  endl;
      cout<< "State: "  << cmStateProv <<  endl;
      cout<< "Zip: "  << cmZip <<  endl;
      cout<< "Telephone: "  << cmTelephone <<  endl;
      cout<< "E-Mail: "  << cmE_Mail <<  endl;
}
// ******************************************************************************************
// Employee Member Functions
// ******************************************************************************************
Employee :: Employee ( )
{
      // Invokes the CollegeMember constructor, then sets other values
      eSalary = 0;
      for ( int x = 1; x <= 30; x++) {eDepartment[x-1] = ' '; eJobTitle[x-1] = '  ';}
}

void Employee :: modify (Employee thisEmp)
{
      // Invoke CollegeMembers's Modify(), then modify additional data
      CollegeMember :: modify(thisEmp);
      eSalary = thisEmp.eSalary;
      strcpy (eDepartment, thisEmp.eDepartment);
      strcpy (eJobTitle, thisEmp.eJobTitle);
}

void Employee :: inputData (int x)
{
      // Invoke CollegeMembers's inputData(), then accept additional data
      cout << "Employee Information Input for Employee " << x << endl << endl;
      CollegeMember :: inputData(x);
      cout << "Department: " ; gets(eDepartment);
      cout << "Job Title: "; gets(eJobTitle);
      cout << "Salary: "; cin >> eSalary; getchar();
}
```

Figure 9-3: Implementing the College Hierarchy (continued)

```
void Employee :: printMe ( )
{
     // Invoke CollegeMembers's PrintMe(), then print additional data
     CollegeMember :: printMe( );
     cout << "Job Title: " << eJobTitle << endl;
     cout << "Department: " << eDepartment << endl;
     cout<< "Salary:  " << eSalary << endl;
}

// ***********************************************************************
// Student Member Functions
// ***********************************************************************
Student :: Student ( )
{
     // Invokes the CollegeMember constructor, then sets other values
     for (int x = 1; x <=30; x++) {sAcadDept[x-1] = '  '; sMajor[x-1] = ' ';}
}

void Student :: modify (Student thisStud)
{
     // Invoke CollegeMembers's modify(), then modify additional data
     CollegeMember :: modify(thisStud);
     strcpy(sAcadDept, thisStud.sAcadDept);
     strcpy(sMajor, thisStud.sMajor);
}

void Student :: inputData (int x)
{
     // Invoke CollegeMembers's inputData(), then accept additional data
     cout << "Student Information Input for Student " << x << endl << endl;
     CollegeMember :: inputData(x);
     cout << "Department: " ; gets(sAcadDept);
     cout << "Major: "; gets(sMajor);
}

void Student :: printMe ( )
{
     // Invoke CollegeMembers's printMe(), then print additional data
     CollegeMember :: printMe( );
     cout<< "Major:  " << sMajor << endl;
     cout << "Department:  " << sAcadDept << endl;
}

// continued on next page
```

Figure 9-3: Implementing the College Hierarchy (continued)

```
// ************************************************************************
// The rest of the program
// ************************************************************************
typedef CollegeMember * AffilList;
typedef Employee* EmpList;
typedef Student* StudList;

StudList inputStudents(int &sNum);  // prototype
EmpList inputEmployees(int &sNum); // prototype
AffilList inputAffiliates(int &aNum); // prototype

int main(int argc, char *argv[])
{
      int aLim, eLim, sLim;
      aLim = sLim = eLim = 0; // char ExitKey;
      AffilList aList;
       EmpList eList;
       StudList sList;
      // …
      aList = inputAffiliates(aLim)
      // …
      eList = inputEmployees(eLim);
      //…
      sList = inputStudents(sLim);
      // …
} // End of main

StudList inputStudents (int &sLim)
{
      StudList thisList;
      int x;
      /sLim = 0; // Reset the number of students
      cout << "Enter Number of Students to be Processed: "; cin >> sLim; getchar();
      thisList = new Student[sLim];
      //Student StudList[sLim];
      for (x = 1; x <= sLim; x++)
      { thisList[x-1] = Student();    thisList[x-1].inputData(x); }
      return thisList;
} // End of inputStudents Method

// The inputEmployees Function
EmpList inputEmployees(int &eLim)
{
      EmpList thisList;

      // Similar to inputStudents(…)
      return thisList;
} // End of inputEmployees Method

// The inputAffiliates Function
AffilList inputAffiliates(int &aLim)
{
 AffilList thisList;

 // Similar to inputStudents(…)
 return thisList;
} // End of inputAffiliates Method} // End of inputAffiliates Method
```

9.3 Access Control in Inheritance

As pointed out in the previous section, the access keywords are used to control access in an inheritance hierarchy. Figure 9-4 provides further clarity on this matter.

Figure 9-4: Clarifying the Access Keywords

```
class CollegeMember
{
    protected:
    int cmID_Number;
    char cmFirstName [16], cmLastName[16];
    char cmAddressLine1 [16], cmAddressLine2 [16], cmStateProv[7], Zip[7], cmTelephone[13];
    char cmE_Mail[31];

    public: // member functions
    CollegeMember ( ); // constructor
    void modify (CollegeMember thisMember);
    void inputData(int x);
    void printMe ( );
};
// …
class Employee: public CollegeMember
{
    // …
    // Public members of CollegeMember will be public members of Employee.
    // Private members are inaccessible.
    // Protected members remain protected
    // Attempts to directly access private members of CollegeMember will result in a compilation error
};

class Student: protected CollegeMember
{
    // …
    // Private members of CollegeMember are inaccessible.
    // Public members of CollegeMember are protected here.
    // Protected members of CollegeMember remain protected here.
};

class Alumnus: private CollegeMember
{
    // …
    // Private members of CollegeMember are inaccessible.
    // Public members of CollegeMember are private here.
    // Protected members of CollegeMember are private here.
};
```

Note: Class members that are protected can be inherited, but they remain protected or become private Within the class hierarchy. Class members that are private are not inherited. Class members that are public can be inherited to become members of a derived class.

9.4 Multiple Inheritance

C++ allows a derived class to inherit from multiple classes. This is done in the following way:

```
class <DerivedClass> : <Modifier> <Base-class1> [… , <Modifier> <Base-class1>]
{
 // … // body of derived class
};
```

Figure 9-5 provides an example of multiple inheritance. In this example, a **StudentWorker** class inherits from the **Student** class as well as the **Employee** class.

Figure 9-5: Illustrating Multiple Inheritance

```
class CollegeMember
{
    protected:
    int cmID_Number;
    char cmFirstName [16], cmLastName[16];
    char cmAddressLine1 [16], cmAddressLine2 [16], cmStateProv[7], Zip[7], cmTelephone[13];
    char cmE_Mail[31];

    public: // member functions
    CollegeMember ( ); // constructor
    void modify (CollegeMember thisMember);
    void inputData(int x);
    void printMe ( );
};

class Employee: protected CollegeMember
{
    float eSalary;
    protected:
    char eDepartment [31], eJobTitle [31];

    public: // Member Functions
    Employee ( );         // constructor
    void modify (Employee ThisEmp);
    void inputData(int x);
    void setSalary (float inSal) // Specified as an in-line function
    { eSalary = inSal;}
    float getSalary ( ) {return Salary;} // Specified as an in-line function
    void printMe ( );
};

class Student: protected CollegeMember
{
    protected:
    char sAcadDept[31], sMajor [31];

    public: //Member Functions
    Student ( );       // constructor
    void modify (Student thisStud);
    void inputData(int x);
    void printMe ( );
};
// Continued on next page
```

Figure 9-5: Illustrating Multiple Inheritance (continued)

```
class StudentWorker: protected Student, protected Employee
{
     protected:
     char swProject[31];
     float swHourlyRate;

     public: // Member Function Prototypes
     StudentWorker ( );        // constructor
     void modify (StudentWorker thisSW);
     void inputData(int x);
     void printMe ( );
};

// … Member functions of CollegeMember, Employee and Student as in figure 9-3

// Member functions for StudentWorker
StudentWorker :: StudentWorker ( )
{
     swHourly Rate = 8.5;
     for (int x = 1; x <= 30; x++) swProject[x-1] = ‘ ‘;
}

void StudentWorker :: modify (StudentWorker thisSW)
{
     Student :: modify(thisSW);
     Employee :: modify(thisSW);  // Alternately, issue the following three statements
     // Salary = thisEmp.eSalary;
     //strcpy (eDepartment, thisEmp.eDepartment);
     //strcpy (eJobTitle, thisEmp.JobTitle);

     stcopy (swProject, thisSW.swProject); swHourlyRate = thisSW.swHourlyRate;
}

void StudentWorker :: inputData (int x)
{
     // Invoke Student’s inputData(), then accept additional data
     cout << “Student Worker Information Input for Student “ << x << endl << endl;
     Student :: inputData(x);
     cout << “Department: “ ; gets(eDepartment);
     cout << “Job Title: “; gets(eJobTitle);
     cout << “Salary: “; cin >> eSalary; getchar();
     cout << “Project: “; gets(swProject);
     cout << “Hourly Rate: “; cin >> swHourlyRate;  getchar();
}

void StudentWorker :: printMe ( )
{
     Student :: printMe( ); // call print function of the base class
     cout << “Department: “ << eDepartment << endl;
     cout << “Job Title: “ << eJobTitle << endl;
     cout << “Salary:  ” << eSalary << endl;
     cout << “Project: ” << swProject << endl’
     cout << “HourlyRate: ” << swHourlyRate << endl;
}

// Continued on next page
```

Figure 9-5: Illustrating Multiple Inheritance (continued)

```
// Rest of the program …
void main ( )
{
      // …
      StudentWorker Alta, Leidy, Pierre;
      //…
      Alta.PrintMe ( ); // prints all information about the object Alta, including inherited information
      Leidy.PrintMe ( );
      Pierre.PrintMe ( );
      // …
}
```

Note: There is one problem with this example; the problem relates to ambiguity; this will be discussed in section 9.7.

9.5 Constructors and Destructors in Inheritance

The introduction of inheritances brings with it, two important questions that will be addressed:
- When are constructors and destructors called?
- How can parameters be passed to base constructors?

9.5.1 When Constructors and Destructor Functions are Invoked

Within a class hierarchy, it is quite possible for base class(es) and derived class(es) to have constructors and destructors. The order in which they are called is as follows:

Constructors are called top-down, left-right (from base class(es) to derived class(es)) and destructors are called bottom-top, right-left (from derived class(es) to base class(es).

The following examples should clarify this concept (called *constructor/destructor chaining*).

Figure 9-6a: Illustrating Constructor/Destructor Chaining

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

class Base {
public:
Base ( ) { cout << "Constructing Base\n"; }
~Base ( ) { cout<< "Destructing Base\n"; }
};
class Derived: public Base {
public:
Derived ( ) { cout << "Constructing Derived\n"; }
~Derived ( ) { cout<< "Destructing Derived\n"; }
};

void main ( )
{ Derived ob;  // do nothing but construct and destruct ob }
```

9.5.1 When Constructors and Destructor Functions are Invoked (Continued)

If you can run the program of figure 9-6a, the output displayed on the screen will be as follows.

```
Constructing Base
Constructing Derived
Destructing Derived
Destructing Base
```

Figure 9-6b: Illustrating Constructor/Destructor Chaining

```
# include <iostream>
class Base1 {
public:
Base1 ( ) { cout << "Constructing Base1\n"; }
~Base1 ( ) { cout<< "Destructing Base1\n"; }
};
class Base2 {
public:
Base2 ( ) { cout << "Constructing Base2\n"; }
~Base2 ( ) { cout<< "Destructing Base2\n"; }
};
class Derived: public Base1, public Base2 {
public:
Derived ( ) { cout << "Constructing Derived\n"; }
~Derived ( ) { cout<< "Destructing Derived\n"; }
};
```

```
void main ( )
{
 Derived ob;
// construct and destruct ob
}
```

If you run the program of figure 9-6b, the output displayed on the screen will be as follows:

```
Constructing Base1
Constructing Base2
Constructing Derived
Destructing Derived
Destructing Base2
Destructing Base1
```

9.5.2 Passing Parameters to Base Class Constructors

In situations where only the constructor of the derived class requires arguments, the standard method of declaring parameters (for the constructor) is used.

However, if a base constructor requires argument(s), we may need to pass this (these) argument(s) to it, from a derived class. To facilitate this, C++ provides an expanded form of the derived class's constructor declaration, which passes arguments to one or more base class constructors.

The general format for the prototype declaration is

```
<DerivedConstructor> (<ParameterList>): <Base1> (<Argumen-List1>),
                                        … <BaseN> (Argument-ListN>);
```

Of course, the actual function specification is similar, with the function body replacing the semicolon. In general, the constructor of a derived class must declare the parameter(s) that its base class requires, and pass it (them) to the base class as arguments. Figure 9-7 illustrates this concept by revisiting the college hierarchy of earlier discussions.

Figure 9-7: Revised College Hierarchy

```
class CollegeMember
{
    protected:
    int cmID_Number;
    char cmFirstName [16], cmLastName[16];
    char cmAddressLine1 [16], cmAddressLine2 [16], cmStateProv[7], Zip[7], cmTelephone[13];
    char cmE_Mail[31];

    public: // member functions
    CollegeMember ( ); // Constructor
    CollegeMember (int memberNbr );  // Overloaded constructor
    void modify (CollegeMember Member);
    void inputData(int x);
    void printMe ( );
};

class Employee: protected CollegeMember
{
    float eSalary;
    public:
    char eDepartment [31], eJobTitle [31];
    // Member Functions
    Employee ( );         // constructor
    Employee (int memberNumber ) : CollegeMember(memberNumber); // Overloaded Constructor
    void modify (Employee thisEmp);
    void inputData(int x);
    void setSalary (float inSal) // Specified as an in-line function
    { eSalary = inSal;}
    float getSalary ( ) {return eSalary;} // Specified as an in-line function
    void printMe ( );
};

// Continued on next page
```

Figure 9-6: Revised College Hierarchy (continued)

```
class Student: protected CollegeMember
{
      public:
      char sAcadDept[31], Major [31];
      //Member Functions
      Student ( );       // constructor
      Student (int memberNumber ) : CollegeMember(memberNumber) // Overloaded Constructor
      void modify (Student thisStud);
      void inputData(int x);
      void printMe ( );
};

// ***********************************************************************
// CollegeMember Member Functions
// ***********************************************************************
CollegeMember :: CollegeMember ( ) // Constructor
{
      int x;
      ID_Number = 0;
      for (x = 1; x <=15; x++) cmFirstName[x-1], cmLastName[x-1] = '  ';
      for (x = 1; x <=15; x++) cmAddressLine1[x-1], cmAddressLine2[x-1] = '  ';
      for (x = 1; x <= 6; x++) cmStateProv[x-1], cmZip[x-1] = '0';
      for (x = 1; x <=30; x++) cmE_Mail[x-1] = '  ';
}

CollegeMember :: CollegeMember (int thisID )  // Overloaded constructor
{
      int x;
      cmID_Number = thisID;
      for (x = 1; x <=15; x++) cmFirstName[x-1], cmLastName[x-1] = '  ';
      for (x = 1; x <=15; x++) cmAddressLine1[x-1], cmAddressLine2[x-1] = '  ';
      for (x = 1; x <= 6; x++) cmStateProv[x-1], cmZip[x-1] = '0';
      for (x = 1; x <=30; x++) cmE_Mail[x-1] = '  ';
}

void CollegeMember :: modify (CollegeMember Member)
{ // As in figure 9-3 }

void CollegeMember :: inputData(int x)
{ // As in figure 9-3 }

void CollegeMember :: printMe( )
{ // As in figure 9-3 }

// Continued on next page
```

Figure 9-6: Revised College Hierarchy (continued)

```
// ***********************************************************************
// Employee Member Functions
// ***********************************************************************
Employee :: Employee ( ) // Constructor
{ // As in figure 9-3 … }

Employee :: Employee (int memberNumber ) : CollegeMember(memberNumber) // Overloaded Constructor
{ // Must obtain a value for memberNumber so it can be passed up the parent's constructor
  // Code is as in figure 9-3 …
}

void Employee :: modify (Employee Member)
{ // As in figure 9-3 … }

void Employee :: inputData(int x)
{ // As in figure 9-3 … }

void Employee :: printMe( )
{ // As in figure 9-3 … }

// ***********************************************************************
// Student Member Functions
// ***********************************************************************
Student  :: Student ( ) // Constructor
{ // As in figure 9-3 … }

Student :: Student (int memberNumber ) : CollegeMember(memberNumber) // Overloaded Constructor
{ // Must obtain a value for memberNumber so it can be passed up the parent's constructor
  // Code is as in figure 9-3 …
}

void Student :: modify (Student thisMember)
{ // As in figure 9-3 … }

void Student :: inputData(int x)
{ // As in figure 9-3 … }

void Student :: printMe( )
{ // As in figure 9-3 … }

// ****************************************************************************************
// Rest of the program
// ****************************************************************************************
void main ( )
{
 Student Berns = Student(1996123);      // Creates a Student object with ID# 1996123
 Employee Bruce = Employee (2000145); // Creates an Employee object with ID# 2000145
 // …
 Bruce.PrintMe ( );  //  Prints Bruce's information
 Berns.PrintMe ( );  //  Prints Berns's information
 //   …
}
```

Note: Even if the constructor of a derived class does not require arguments, if its base class requires arguments, it must declare them (as parameters) and pass them to the base constructor.

9.5.2 Passing Parameters to Base Class Constructors (continued)

One elegant way to get around passing parameters to base constructors as discussed above, is as follows: For each class in the hierarchy, create an overloaded constructor that has as its parameter, an instance of the class — the way the **modify(…)** methods of figures 9-3 and 9-5 have been set up. Each constructor would be polymorphic, so that it would first call its corresponding parent constructor, and then perform additional instructions specified in its body. Figure 9-8 illustrates this approach.

Figure 9-8: College Hierarchy with Overloaded Constructors

```
class CollegeMember
{
    protected:
    int cmID_Number;
    char cmFirstName [16], cmLastName[16];
    char cmAddressLine1 [16], cmAddressLine2 [16], cmStateProv[7], Zip[7], cmTelephone[13];
    char E_Mail[31];

    public: // member functions
    CollegeMember ( ); // Constructor
    CollegeMember (const CollegeMember &thisMember);  // Overloaded constructor
    void modify (CollegeMember thisMember);
    void inputData(int x);
    void printMe ( );
};

class Employee: protected CollegeMember
{
    float Salary;
    public:
    char eDepartment [31], eJobTitle [31];
    // Member Functions
    Employee ( );          // constructor
    Employee (const Employee  &thisEmp ) // Overloaded Constructor
    void modify (Employee thisEmp);
    void inputData(int x);
    void setSalary (float inSal) // Specified as an in-line function
    { eSalary = inSal;}
    float getSalary ( ) {return Salary;} // Specified as an in-line function
    void printMe ( );
};

class Student: protected CollegeMember
{
    public:
    char sAcadDept[31],s Major [31];
    //Member Functions
    Student ( );       // constructor
    Student (const Student &thisStud) // Overloaded Constructor
    void modify (Student thisStud);
    void inputData(int x);
    void printMe ( );
};

// Continued on next page
```

Figure 9-8: College Hierarchy with Overloaded Constructors (continued)

```
// ***********************************************************************
// CollegeMember Member Functions
// ***********************************************************************
CollegeMember :: CollegeMember ( ) // Constructor
{
    int x;
    cmID_Number = 0;
    for (x = 1; x <=15; x++) cmFirstName[x-1], cmLastName[x-1] = ‘  ’;
    for (x = 1; x <=15; x++) cmAddressLine1[x-1], cmAddressLine2[x-1] = ‘  ’;
    for (x = 1; x <= 6; x++) cmStateProv[x-1], cmZip[x-1] = ‘0’;
    for (x = 1; x <=30; x++) cmE_Mail[x-1] = ‘  ’;
}

CollegeMember :: CollegeMember (const CollegeMember &thisMember)  // Overloaded constructor
{
    int x;
    cmID_Number = thisMember. cmID_Number;
    strcpy (cmFirstName, thisMember. cmFirstName); strcpy (cmLastName, thisMember. cmLastName);
    strcpy (cmAddressLine1, thisMember. cmAddressLine1); strcpy (cmAddressLine2, thisMember. cmAddressLine2);
    strcpy (cmE_Mail, thisMember. cmE_Mail);
    strcpy (cmTelephone, thisMember. cmTelephone);
    strcpy (cmStateProv, thisMember. cmStateProv);
}

void CollegeMember :: modify (CollegeMember thisMember)
{ // As in figure 9-3 }

void CollegeMember :: inputData(int x)
{ // As in figure 9-3 }

void CollegeMember :: printMe( )
{ // As in figure 9-3 }

// ***********************************************************************
// Employee Member Functions
// ***********************************************************************
Employee :: Employee ( ) // Constructor
{ // As in figure 9-3 … }

Employee :: Employee (const Employee &thisEmp ) // Overloaded Constructor
{
    // Invoke CollegeMembers’s overloaded constructor, and then modify additional data
    CollegeMember :: CollegeMember (thisEmp); // or CollegeMember :: modify (thisEmp);
    eSalary = thisEmp.eSalary;
    strcpy (eDepartment, thisEmp.eDepartment);
    strcpy (eJobTitle, thisEmp.eJobTitle);
}
void Employee :: modify (Employee thisMember)
{ // As in figure 9-3 … }

void Employee :: inputData(int x)
{ // As in figure 9-3 … }

void Employee :: printMe( )
{ // As in figure 9-3 … }

// Continued on next page
```

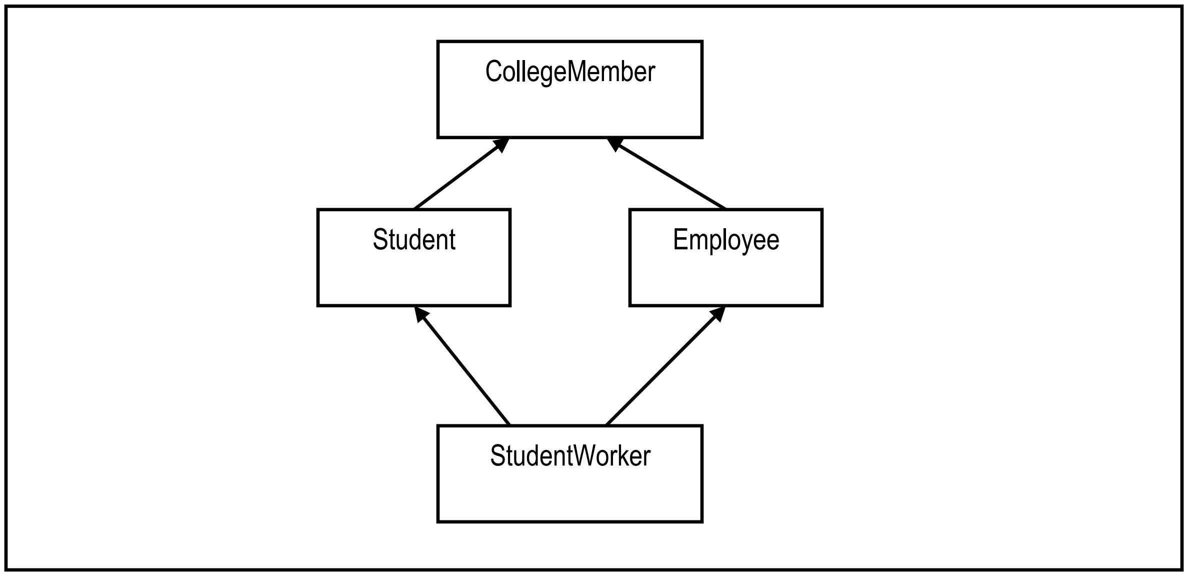
Figure 9-8: College Hierarchy with Overloaded Constructors (continued)

```
// ***********************************************************************
// Student Member Functions
// ***********************************************************************
Student  :: Student ( ) // Constructor
{ // As in figure 9-3 … }

Student :: Student (const Student &thisStud ) // Overloaded Constructor
{
     // Invoke CollegeMembers's overloaded constructor, and then modify additional data
     CollegeMember :: CollegeMember (thisStud); // or CollegeMember :: modify(thisStud);
     strcpy(sAcadDept, thisStud.sAcadDept);
     strcpy(sMajor, thisStud.sMajor);
 }

void Student :: modify (Student thisMember)
{ // As in figure 9-3 … }

void Student :: inputData(int x)
{ // As in figure 9-3 … }

void Student :: printMe( )
{ // As in figure 9-3 … }

// ******************************************************************************************
// Rest of the program
// ******************************************************************************************
void main ( )
{
 Student dummyS  = Student();
 Employee dummyE = Employee();
 // …
 Student Berns = Student(dummyS);     // Creates a Student object via dummyS
 Employee Bruce = Employee (dummyE); // Creates an Employee via dummyE
 // …
 Bruce.PrintMe ( );  //  Prints Bruce's information
 Berns.PrintMe ( );  //  Prints Berns's information
     …
}
```

9.6 Inheritance Graphs

It is sometimes useful to represent class hierarchies by using inheritance graphs. The C++ convention is that derived classes point to their base classes. An inheritance graph is simply a summarized UML diagram for a group of related classes that depicts only the class names. Figure 9-9 shows an inheritance graph for classes discussed in the previous section.

Figure 9-9: Inheritance Graph for a College Hierarchy



9.7 Virtual Base Classes

Multiple inheritance has the potential of causing confusion. For this reason it is avoided in some OOPLs, e.g. Java. C++ allows multiple inheritance, but arms the programmer with ways to avoid the confusion that may result.

Let us look at a situation when multiple inheritance could lead to ambiguity (hence confusion). The inheritance graph of figure 9-8 is in fact one such scenario: **StudentWorker** inherits **Student** as well as **Employee**. However both **Student** and **Employee** inherit **CollegeMember**. There will therefore be two copies of **CollegeMember**'s members in a **StudentWorker** object as illustrated in figure 9-10.

Figure 9-10: Illustrating Confusion in Multiple Inheritance

```
// Assuming the declarations of figure 9-4:
void main ( )
{
     // …
     StudentWorker Alta, Leidy;
     Alta.Telephone = 9783682415  // is ambiguous; it is not clear which copy of Telephone is to be used
     cout << Alta.Address                    // is also ambiguous
     // …
}
```

There are two ways to resolve this ambiguity:

Strategy 1*:* Apply the scope resolution operator to an expression when accessing the members about which there is ambiguity.

```
Example:
Alta.Student :: Telephone = 9783682415
```

This can be very inelegant and clumsy.

Strategy 2: Prevent two copies of a base class in a derived class by using *virtual classes*:
In the derived class, declare the base class as virtual when it is inherited; this is done by simply specifying the **virtual** keyword. Figure 9-11 illustrates this approach.

Note: When in doubt, use the *virtual* keyword as shown, since it does no harm in the situation where only one copy of the base class is present in an object of the derived class. Figure 9-11 illustrates.

Figure 9-11: Revised College Hierarchy Illustrating Virtual Base Class

```
class CollegeMember
{
    protected:
    int cmID_Number;
    char FirstName [16], LastName[16];
    char cmAddressLine1 [16], cmAddressLine2 [16], cmStateProv[7], Zip[7], cmTelephone[13]; char cmE_Mail[31];

    public: // member functions
    CollegeMember ( ); // Constructor
    CollegeMember (int memberNbr );  // Overloaded constructor
    void modify (CollegeMember thisMember);
    void inputData(int x);
    void printMe ( );
};

class Employee: virtual protected CollegeMember
{
    float eSalary;
    protected:
    char eDepartment [31], eJobTitle [31];
    public: // Member Functions
    Employee ( );          // constructor
    Employee (int memberNumber ) : CollegeMember(memberNumber) // Overloaded Constructor
    void modify (Employee thisEmp);
    void inputData(int x);
    void setSalary (float inSal) // Specified as an in-line function
    { eSalary = inSal;}
    float getSalary ( ) {return eSalary;} // Specified as an in-line function
    void printMe ( );
};

class Student: virtual protected CollegeMember
{
    protected:
    char sAcadDept[31], sMajor [31];
    public: //Member Functions
    Student ( );       // constructor
    Student (int memberNumber ) : CollegeMember(memberNumber) // Overloaded Constructor
    void modify (Student thisStud);
    void inputData(int x);
    void printMe ( );
};

class StudentWorker: protected Student, protected Employee
{
    protected:
    char swProject[31];
    float swHourlyRate;

    public: // Member Function Prototypes
    StudentWorker ( );        // constructor
    void modify (StudentWorker thisSW);
    void inputData(int x);
    void printMe ( );
};

// Continued on next page
```

Figure 9-11: Revised College Hierarchy Illustrating Virtual Base Class (continued)

```
// ***********************************************************************
// CollegeMember Member Functions
// ***********************************************************************

CollegeMember :: CollegeMember ( ) // Constructor
{ // As in figures 9-3  & 9-7}

CollegeMember :: CollegeMember (int thisID )  // Overloaded constructor
{ // As in figure 9-7}

void CollegeMember :: modify (CollegeMember thisMember)
{ // As in figure 9-3 }

void CollegeMember :: inputData(int x)
{ // As in figure 9-3 }

void CollegeMember :: printMe( )
{ // As in figure 9-3 }

// ***********************************************************************
// Employee Member Functions
// ***********************************************************************
Employee :: Employee ( ) // Constructor
{ // As in figure 9-3 … }

Employee :: Employee (int memberNumber ) : CollegeMember(memberNumber) // Overloaded Constructor
{ // As in figure 9-7 …}

void Employee :: modify (Employee thisMember)
{ // As in figure 9-3 … }

void Employee :: inputData(int x)
{ // As in figure 9-3 … }

void Employee :: printMe( )
{ // As in figure 9-3 … }

// ***********************************************************************
// Student Member Functions
// ***********************************************************************
Student  :: Student ( ) // Constructor
{ // As in figure 9-3 … }

Student :: Student (int memberNumber ) : CollegeMember(memberNumber) // Overloaded Constructor
{ // As in figure 9-7 …}

void Student :: modify (Student thisMember)
{ // As in figure 9-3 … }

void Student :: inputData(int x)
{ // As in figure 9-3 … }

void Student :: printMe( )
{ // As in figure 9-3 … }
```

Figure 9-11: Revised College Hierarchy Illustrating Virtual Base Class (continued)

```
// ***************************************************************************************
// … Member functions of StudentWorker
// ***************************************************************************************
StudentWorker :: StudentWorker ( )
{ //  Similar to code in figure 9-3 … }

void StudentWorker :: modify (StudentWorker thisSW)
{ //  Similar to code in figure 9-3 … }

void StudentWorker :: inputData (int x)
{ //  Similar to code in figure 9-3 … }

void StudentWorker :: printMe ( )
{ //  Similar to code in figure 9-3 … }

// *************************************************************************************
// Rest of the program
// *************************************************************************************
void main ( )
{
     Student Berns = Student(1996123);       // Creates a Student object with ID# 1996123
     Employee Bruce = Employee (2000145); // Creates an Employee object with ID# 2000145
     StudentWorker Alta;

     Alta.Telephone = 9783682415   // is unambiguous
     cout << Alta.Address                       // is also unambiguous

     // …
     Bruce.printMe ( );    //  Prints Bruce's information
     Berns.printMe ( );    //  Prints Berns's information
     // …
}
```

9.8 Amalgamation

You are now familiar with the concepts of inheritance, super-classes and subclasses. Another type of relationship that is very common in OOP is the *amalgamation* relationship. An amalgamation relationship is a special form of association that establishes an *owner* class and a set of one or more *constituent* classes. An object from the owner class is comprised of objects from the constituent classes.

9.8 Amalgamation (continued)

The UML notation identifies two kinds of amalgamation relationships — *component* relationship and an *aggregation* relationship:

- If the constituent objects are existence-dependent on the owner object (i.e. they do not exist unless the owner object exists), the relationship is called a component relationship. This is represented by a filled in diamond symbol next to the owner-class.
- If the constituent objects are existence-independent of the owner object (i.e. they can exist on their own without the owner object, or the owner object can exist without them), the relationship is called an aggregation relationship. This is represented by an open diamond next to the owner-class.

Example 9-2: A motorcar engine is an excellent illustration of a composition relationship. The engine consists of several components that do not exist on their own (example, crank shaft, pistons, engine block, etc.); they are all integral components of the engine. Note however, that this perspective is to some extent subjective, since it can be argued that engine parts do exist and are in fact marketed as independent components.

Example 9-3: An employee record presents an excellent illustration of composition as well as aggregation relationships. The employee record (object) itself may consist of several components — personal information, employment history, academic record, publications and extra curricular activities. The personal information may be an address (aggregate) and an identity (component). Figure 9-12 illustrates this within the context of a college community. Figure 9-13 shows a more elaborate UML diagram and pseudo-code with respect to the **Employee** class and the **EmpEmploymentHistory** class. Implementation of this is left as an exercise for you.

Figure 9-12: College Community with Aggregation and Component Relationships

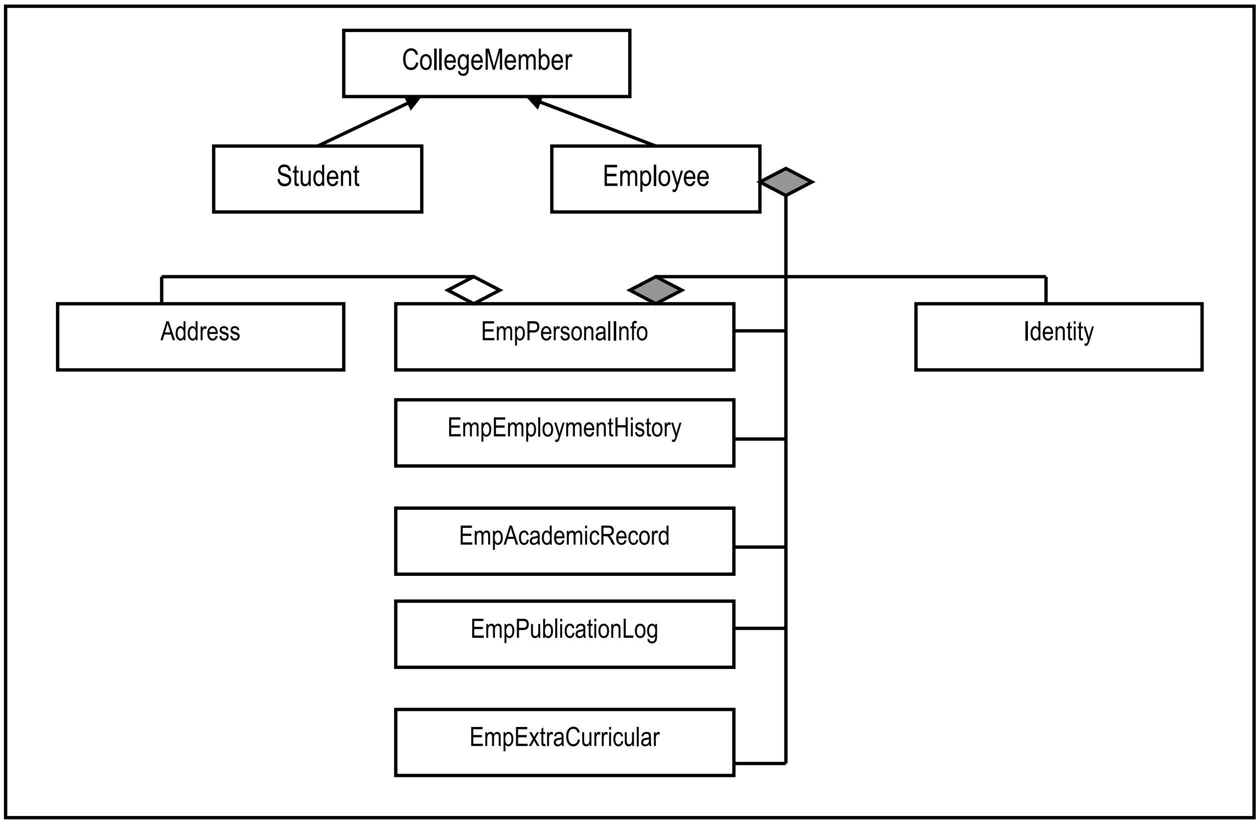
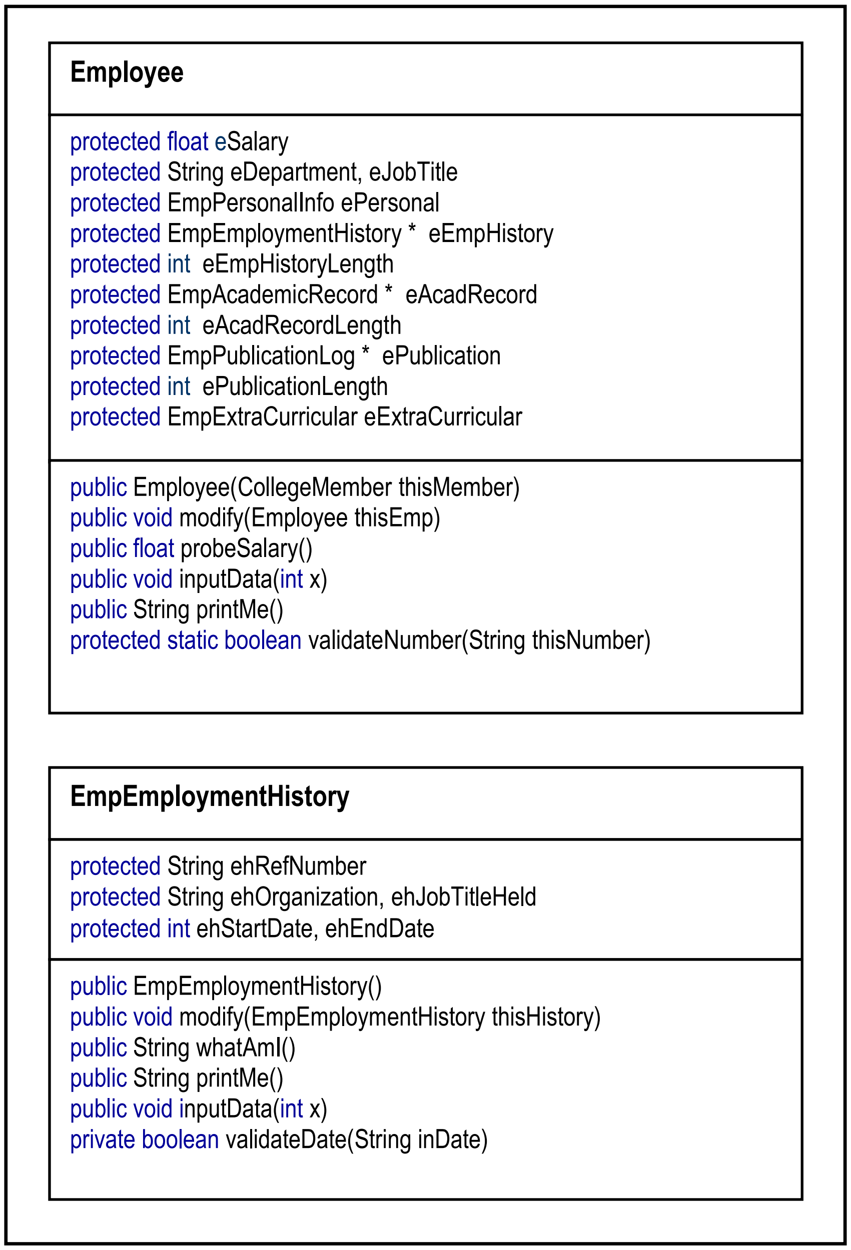


Figure 9-13a: Employee and EmpEmploymentHistory Classes



```
Employee

protected float eSalary
protected String eDepartment, eJobTitle
protected EmpPersonalInfo ePersonal
protected EmpEmploymentHistory *  eEmpHistory
protected int  eEmpHistoryLength
protected EmpAcademicRecord *  eAcadRecord
protected int  eAcadRecordLength
protected EmpPublicationLog *  ePublication
protected int  ePublicationLength
protected EmpExtraCurricular eExtraCurricular

public Employee(CollegeMember thisMember)
public void modify(Employee thisEmp)
public float probeSalary()
public void inputData(int x)
public String printMe()
protected static boolean validateNumber(String thisNumber)
```

```
EmpEmploymentHistory

protected String ehRefNumber
protected String ehOrganization, ehJobTitleHeld
protected int ehStartDate, ehEndDate

public EmpEmploymentHistory()
public void modify(EmpEmploymentHistory thisHistory)
public String whatAmI()
public String printMe()
public void inputData(int x)
private boolean validateDate(String inDate)
```

Figure 9-13b: Pseudo-code for Methods of Employee Class

Note: The keyword **super** is used to represent the corresponding method in the super-class.

```
The Employee(CollegeMember thisMember) Constructor
START
    super (thisMember); // Invoke CollegeMember's constructor with argument thisMember
    eSalary := 0; eDepartment, eJobTitle := " ";
    ePersonal, eEmpHistory, eAcadRecord, ePublication, eExtraCurricula := null;
STOP

The inputData(int x) Method
START
    Let y, Max be integers;
    super.inputData(x, "employee"); // Invoke CollegeMember's InputData(…)
    Prompt for and accept Department for employee x;
    Prompt for and accept Job Title for employee x;
    Prompt for and accept Salary for employee x;

    //  Accept data for aggregated fields
    /*  Note that eEmpHistory, eAcadRecord and ePublication are dynamic lists.
        eEmpHistoryLength, eAcadRecordLength, and ePublicationLength are the respective lengths of these lists.  */

    Personal.InputData(x);
    Prompt for eEmpHistoryLength; Create eEmpHistory for eEmpHistoryLength records;
    For (y := 1 to eEmpHistoryLength with increments of 1) do the following
        Instantiate eEmpHistory[x-1];
        eEmpHistory[x-1].inputData(x, y);
    End-For;
    Prompt for eAcadRecordLength; Create eAcadRecord for eAcadRecordLength records;
    For (y := 1 to eAcadRecordLength with increments of 1) do the following
        Instantiate eAcadRecord[x-1];
        eAcadRecord[x-1].inputData(x, y);
    End-For;
    Prompt for ePublicationLength; Create ePublication for ePublicationLength records;
    For (y := 1 to ePublicationLength with increments of 1) do the following
        Instantiate ePublication[x-1];
        ePublication[x-1].inputData(x, y);
    End-For;
     eExtraCurricular.inputData(x);
STOP

The probeSalary( ) Method: Returns float
START Return Salary;  STOP

The validateNumber(String thisID) Method: Returns Boolean
START
    Let isValid be Boolean, initialized to true;
    Let x be an integer;
    For  (x:= 1 to thisID.length() with increments of 1) do the following:
        If (thisID.CharacterAt(x-1) is not a digit) isValid := false; EndIf;
    End-For;
    Return isValid;
STOP
```

Figure 9-12b: Pseudo-code for Methods of Employee Class (continued)

```
The Modify(Employee thisEmp) Method
START
    Let x be an integer;
    super.modify(thisEmp); // Invoke CollegeMember's Modify (…)
    eSalary := thisEmp.Salary;
    eDepartment := thisEmp.Department;
    eJobTitle := thisEmp.JobTitle;

    //  Modify the aggregated fields based on the corresponding fields of ThisEmp
    /*  Note that EmpHistory, AcadRecord and Publication are dynamic lists.
        EmpHistoryLength, AcadRecordLength, and PublicationLength are the respective lengths of these lists.  */

    ePersonal.modify(thisEmp.ePersonal);
    eEmpHistory := null; Create eEmpHistory for thisEmp.eEmpHistoryLength records;
    For (x := 1 to thisEmp.eEmpHistoryLength with increments of 1) do the following
        Instantiate eEmpHistory[x-1];
        eEmpHistory[x-1].modify(thisEmp.eEmpHistory[x-1]);
    End-For;
    eAcadRecord := null; Create eAcadRecord for thisEmp.eAcadRecordLength records;
    For (x := 1 to thisEmp.eAcadRecordLength with increments of 1) do the following
        Instantiate eAcadRecord[x-1];
        eAcadRecord[x-1].modify(thisEmp.eAcadRecord[x-1]);
    End-For;
    ePublication := null; Create ePublication for thisEmp.ePublicationLength records;
    For (x := 1 to thisEmp.ePublicationLength with increments of 1) do the following
        Instantiate ePublication[x-1];
        ePublication[x-1].modify(thisEmp.ePublication[x-1]);
    End-For;
    eExtraCurricular.nodify(thisEmp.eExtraCurricular);
STOP

The printMe( ) Method: Returns a string
START
    Let printString be a string;
    Let x be an integer;
    printString := super.printMe() +  <NewLine> + "Salary:   " + Salary + <NewLine> +
        "Department: " + Department + <NewLine>  + "Job Title: " + JobTitle;

    Append ePersonal.printMe() to printString;
    For (x := 1 to eEmpHistoryLength with increments of 1) do the following
        Append eEmpHistory[x-1].printMe() to printString;
    End-For;
    For (x := 1 to eAcadRecordLength with increments of 1) do the following
        Append eAcadRecord[x-1].printMe() to PrintString;
    End-For;
    For (x := 1 to ePublicationLength with increments of 1) do the following
        Append ePublication[x-1].printMe() to PrintString;
    End-For;
    Append eExtraCurricular.printMe() to PrintString;

    Return printString;
STOP
```

Figure 9-12c: Pseudo-code for Methods of EmpEmploymentHistory Class

```
The EmpEmploymentHistory( ) Constructor
START
    ehRefNumer := "00000000000";
    ehOrganization, ehJobTitleHeld := " ";
    ehStartDate, ehEndDate := 0;
STOP

The modify(EmpEmploymentHistory thisHistory) Method
START
        ehRefNumber :=  thisHistory.ehRefNumber;
        ehOrganization := thisHistory.ehOrganization;
        ehJobTitleHeld := thisHistory.ehJobTitleHeld;
        ehStartDate := thisHistory.ehStartDate;
        ehEndDate := thisHistory.ehEndDate;
STOP

The whatAmI( ) Method: Returns a string
START
    Return ehRefNumber + ": " +  ehOrganization + ": " + ehJobTitleHeld + ": " + ehStartDate + ": " + ehEndDate;
STOP

The inputData(int x,y) Method
START
    Let inStartDate, inEndDate, and inNumber be strings;
    Prompt for and accept ehRefNumber for employee x, item y;
    Prompt for and accept ehOrganization for employee x, item y;
    Prompt for and accept ehJobTitleHeld for employee x, item y;

    Prompt for and accept inNumber for employee x, item y;
    While (NOT Employee.validateNumber(inNumber)) do the following:
        DisplayMessage("Invalid number.");
        Prompt for and accept inNumber for employee x, item y;
    End-While;

    Prompt for and accept inStartDate for employee x, item y;
    While (NOT validateDate(inStartDate)) do the following:
        DisplayMessage("Invalid date. Required format is YYYYMMDD");
        Prompt for and accept inStartDate for employee x, item y;
    End-While;

    Prompt for and accept inEndDate for employee x, item y;
    While (NOT validateDate(inEndDate)) do the following:
        DisplayMessage("Invalid date. Required format is YYYYMMDD");
        Prompt for and accept inEndDate for employee x, item y;
    End-While;

    Convert inStartDate to an integer and store it in ehStartDate;
    Convert inEndDate to an integer and store it in ehEndDate;
    ehRefNumber := inNumber;
STOP
```

Figure 9-12c: Pseudo-code for Methods of EmpEmploymentHistory Class (continued)

```
The printMe( ) Method: Returns a string
START
    Let printString be a string;
    printString := "Reference Number: " + ehRefNumber + <NewLine> + "Organization: " +
        ehOrganization + < NewLine> + "Job Title Held: " + ehJobTitleHeld + <NewLine> + "Start Date: "
        + ehStartDate + <NewLine> + "End Date: " + ehEndDate;
    Return printString;
STOP

The validateDate(String thisInput) Method: Returns Boolean
START
    Let isValid be Boolean, initialized to true;
    Let isLeapYear be Boolean;
    Let Year, Month, Day, MaxDays be integers;
    Let CurrentYear be a integer, initialized to the current year;
    Let x be an integer;
    /* Assume that you language provides a substring operation such that substring(s,x,y) returns the substring from
    string s, starting at x and ending at y. */

    If  (Employee.validateNumber(thisInput)) // If valid number
        year := Numeric value of Substring(thisInput,1,4);
        month := Numeric value of Substring(thisInput,5,6);
        day := Numeric value of Substring(thisInput,7,8);

        Let daysInMonth be an array of 12 integers initialized to {31, 28, 31,30, 31, 30, 31, 31, 30, 31, 30, 31};
        If  ((year mod 4 = 0) and (year mod 100 <> 0) || (year mod 400 <> 0))
            daysInMonth[1] := 29;
        End-If

         If ((month > 0 and Month < 13) and (day > 0 and day <= daysInMonth[Month – 1]))
             isValid := True;
        Else isValid := False;
        End-If'

    Else isValid := false;
    End-If; // If valid number

    Return isValid;
STOP
```

9.9 Summary and Concluding Remarks

Here is a summary of what we have covered:

- Inheritance is the act of an object adopting the properties (data items and member functions) of the class on which it is defined. Additionally, a class may inherit from another class in its hierarchy.
- In defining an inheritance hierarchy, three access keywords are applicable: **public**, **protected** and **private**.
- In an inheritance hierarchy, constructors are executed top-down, left-right; Destructors are executed bottom-up, right-left.
- In inheritance graph is an abbreviated form of a UML diagram for the class hierarchy.
- C++ supports multiple inheritance. It allows the programmer to avoid the ambiguities that multiple inheritance potentially leads to by specifying the inherited bas class as a virtual class; this is done from the inheriting (derived) class.
- Amalgamation is the act of defining an object as the aggregation or composition of other objects.

Inheritance and amalgamation are fundamental principles that are at the heart of OOP. Both principles promote code reuse; additionally, inheritance facilitates polymorphism, which is another fundamental principle of OOP. These principles lead to significant improvement in convenience and code efficiency — a rare case in computer science when these two desirables coincide.

9.10 Recommended Readings

[Gaddis, Walters & Muganda 2014] Gaddis, Tony, Judy Walters, & Godfrey Muganda. 2014. *Starting out with C++ Early Objects*, 8th Edition. Boston: Pearson. See chapter 15.

[Savitch 2013] Savitch, Walter. 2013. *Absolute C++*, 5th Edition. Boston: Pearson. See chapters 14 & 15.

[Savitch 2015] Savitch, Walter. 2015. *Problem Solving with C++*, 9th Edition. Boston: Pearson. See chapter 15.

[Yang 2001] Yang, Daoqi. 2001. *C++ and Object-Oriented Numeric Computing for Scientists and Engineers*. New York, NY: Springer. See chapter 8.