Lecture 13.1

Topics:

1. **protected** Specification – Revisited
2. Inheritance – Introduction
3. Inheritance Specifications
4. Inheritance – Constructors and Destructors
5. Inheritance Examples
6. Inheritance – **public** Specification

---

1. protected Specification

Recall that there are three access specifications given inside a class definition. The `public` and `private` specifications have been introduced and used in most of the classes presented in lectures so far; while the **protected** specification is mostly used in inheritance structure.

For stand-alone classes (that means classes to be used as they were defined without any kind of extension) just as the classes presented, the `public` and `private` specifications are sufficient.

When the ideas of code saving and reuse are brought into the design of classes, several things can be done such as function (being called again and again at different stages of logical execution) or making the objects as members of classes (e.g., a **Point** object has a member that is a **Fraction** object).

The next idea and concept is to extend the existing class (if possible) to produce a new class. This is called inheritance. This is different from using existing classes as members of some new class to be designed with.

In the inheritance structure, **the existing class is called the base class and the extended class is called the derived class**. The accesses to the members within the derived class are interesting and important; and these processes provide the venues for using the last access specification of **protected**.

1.1 Description

There will be times when we want to keep a member of a base class to be private but still allow a derived class to access it. This can be accomplished through `protected` specification.

The `protected` specification is almost equivalent to `private` specification with only one exception: `protected` members of a base class are accessible to members of any class derived from that base class. Outside the base or derived classes, `protected` members are not accessible (directly).

Writing code is all about making sure that the code works and its code development should be effective and efficient. No unnecessary time and effort would be needed to complete the task.

A part of these above considerations is the way to reuse existing code. Writing function is to reuse code, or creating new data types through classes is also to reuse code.

However, the next big topic of reusing code is through **careful design of the classes, their relationships, and solutions** after logically obtaining the designs (and algorithms) and confirming the correctness of the results – It is called inheritance.

2. Inheritance -- Introduction

Inheritance is an important property of any Object-Oriented (OO) language (and thus C++). What is inheritance in C++?

2.1 Inheritance: A C++ Class Derivation

In general, inheritance is the mechanism and process from which one can inherit from others. An inheritance may either be simple (single root) or multiple (multi-roots).

Figure 1 depicts the general hierarchies through classes.

In C++, inheritance exists through classes where one class inherits from other(s). The class that is inherited from is called the **base** (or **super**) class, while the inheriting class is called the **derived** class.

Note that there may be several derived classes from a single base class (case of single inheritance); and there may be several base classes involved in a hierarchy (case of multiple inheritance). There can be as many branches as in any class hierarchy, and they must be constructed and declared appropriately.

In brief,

- The base class defines all of the traits that are common to all of the derived classes. These traits are the most general and less specific.
- The derived class inherits all of these general traits and adds additional properties that only available and specific to this derived class.

2.2 Examples

Let's say that Food is the kind of thing that one can digest. Food can be meat, vegetable, carbo-products. Meat can be one of many different kinds, and so on.

In the above chain of food categorization, one can go from the general description to a more detailed groups and items. The following observation and statements are true:

- Meat is Food.
- Vegetable is also Food.
- Carbo-product is also kind of Food.
- Vegetable is not Meat but is Food.
- Not all Food would be Vegetable.
- Some Food may be Meat.

We will be working on the inheritance, its structures and syntax in the next few lectures.

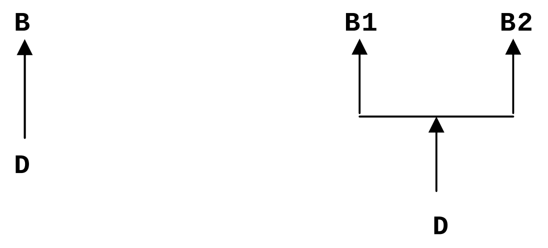

Figure 1a Direct base classes for single and multiple (base) inheritances

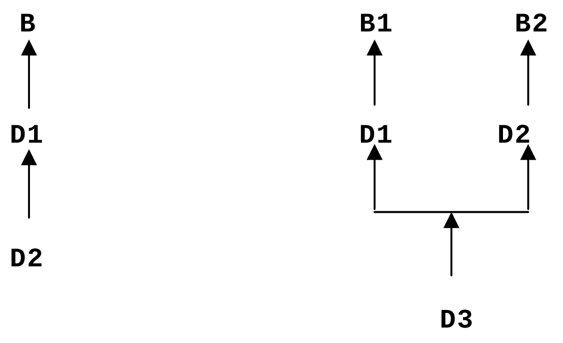

Figure 1b Indirect base class for single chaining (multi-level) inheritance and indirect multiple base (nonvirtual base) inheritance

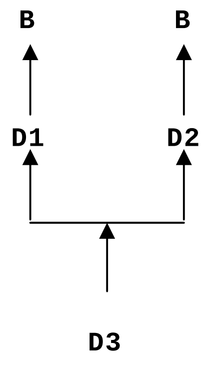

Figure 1c Indirect virtual base inheritance

3. Inheritance Specifications

The general form of a derived class is as follows,

```
class BaseClass;

class DerivedClass : accessSpecifier BaseClass
{
  // Added Members
};
```

where `accessSpecifier` is one of the three keywords: `public, private, or protected`.

There are three different specifications for inheritance as given above – `public`, `protected`, and `private`. Their class membership (access specification) within a class hierarchy will be as follows, `public` **Inheritance** (A popular type of access-specifier)

- Each public member in the base class is public in the derived class.
- Each protected member in the base class is protected in the derived class.
- Each private member in the base class remains private in the base class and so is visible only in the base class.

`protected` **Inheritance**

- Each public member in the base class is protected in the derived class.
- Each protected member in the base class is protected in the derived class.
- Each private member in the base class remains private in the base class and so is visible only in the base class.

`private` **Inheritance**

- Each public member in the base class is private in the derived class.
- Each protected member in the base class is private in the derived class.
- Each private member in the base class remains private in the base class and so is visible only in the base class.

By using any class hierarchy, objects will be created and removed (destroyed) by the systems accordingly.

Clearly, class constructors and destructors must be called into service for these objects. One must understand how the language's syntax would allow the system to perform these tasks with class constructors and destructors.

The following discussion will address the above issues of invoking constructors and destructors using the direct and single base inheritance structure as shown in Figure 1a.

4. Inheritance — Constructors & Destructors

As constructors and destructors are defined for the base and derived classes, their calls will depend on (class) derivation and object creation then destruction. Derivation refers to the formation of a chain of new classes. Object creation refers to specific values being assigned to objects at time of creation.

4.1 Order of Constructor and Destructor Calls

Class derivation directly involves constructors (plus its arguments) and destructors. The order of the constructor and destructor calls will be:

(i) If there are calls for constructors then they will be called starting from the base class to the derived class as specified in the inheritance chain.

(ii) The destructors, if any, are called in the reversed order of that of the constructors.

The reasons are easy to understand. During the creation, because the base class(es) has no knowledge of the derived class(es), the base constructor(s) must be executed first and separate from those of the derived one(s).

However, in destroying an object, the destructor for the derived class must be called first or else the base destructor(s) would destroy the object without any regard for data created using derived class(es).

4.2 Rules For Passing Arguments To Base Class

When only argument of derived class is needed then the argument should be passed in the usual way. If argument is needed in the base class (for base constructor) then additional rules must be observed as follows.

- First, definition of the derived class must specify the required argument(s).
- Second, object creation from derived class must have appropriate number of arguments for both derived class constructor and base class constructor(s). Otherwise, default constructors will take effects upon the creation of object.

4.3 Rules For Invoking Constructors

Let **B** be the base class and let **D** be the derived class from **B**. Then

1) If **D** has constructors but **B** has no constructors then base on the creation of an object of **D** a constructor in **D** will be invoked.
2) If **D** has no constructors but **B** has constructors then **B** must have a default constructor so that this default constructor will be invoked whenever an object of **D** is created.
3) If **D** has constructors and **B** has a default constructor then **B** 's default constructor is invoked automatically whenever an object of **D** is created unless a specific **B** 's constructor is requested.
4) If **D** and **B** have constructors but **B** has no default constructor then each **D** 's constructor must EXPLICITLY invoke a specific **B** 's constructor whenever an object of **D** is created.

4.4 Regarding Destructors

Technically, constructors and destructors can be of any functions. Furthermore, they will be invoked automatically and appropriately at the time of object creation and at the end of object lifetime.

Nonetheless, it is recommended to use constructors to perform appropriate initialization regarding the data members of class; and then to allow the destructor to clean up any possible trace or remnant of the object after destruction is called for.

One of the most frequent uses of the destructor is to release dynamic memory. This dynamic memory allocation is usually created by the constructors. Thus, proper invoking of constructor and destructor pairs is essential to avoid possible error during the compilation.

4.5 Example

The example given below will illustrate some of these rules. This example will show the order of calls for constructors and destructors.

Example 2

```
    // Program Name:  cis25L1311.cpp
```

```
// Discussion:    Inheritance -- Constructors & Destructors
#include <iostream>
using namespace std;

class B {
public:
  B() {
    cout << "\nBase Constructor!";
  }
  ~B() {
    cout << "\nBase Destructor!";
  }
};

class D : public B {
public:
  D() {
    cout << "\nDerived Constructor!";
  }
  ~D() {
    cout << "\nDerived Destructor!";
  }
};

int main() {
  D d1;

  char ch;
  cout << "\n\nPress Any Key + RETURN to continue : ";
  cin >> ch;
  cout << "\n";

  return 0;
}
```

OUTPUT

```
Base Constructor!
Derived Constructor!

Press Any Key + RETURN to continue : q


Derived Destructor!
```

What do we have here? Digression on the output will be given in class!

5. Inheritance Examples

Let's look at an example that would provide inheritance (that means "**is-a**" relationship) or hierarchical structure: **Financial Accounts**.

Most likely, a customer of any financial institution will have an account. This account will reflect the balance that the customer has at any time. As soon as the customer opens an account, there will be a minimum amount required to be deposited and this would give the initial balance. The account will have two operations of depositing and withdrawing that will affect the balance of the account.

5.1 Checking Account

Let's consider a regular checking account. A checking account will have a balance and the two operations as above. Any checking account will also have access to check-writing and there may be a charge per written check.

Clearly, the checking account has all members (data and operations) that any account may have and more. The additional members will reflect the check writing capability and check-fee. Check-fee is the charge for each check that was cashed; a free-checking account will have the check-fee set to zero (0).

The class definitions may be outlined in the following examples.

Example 3a

```
class Account {
  double dBalance;
private:
  //constructors
  //destructor
  //getter/setter

  //void deposit( double dAmount );
  //double withdraw( double dAmount );
};
```

```
//No inheritance
class ChkAccount {
  double dBalance;
  double dChkFee;
private:
  //Constructors
  //destructor

  //void deposit( double dAmount );
  //double withdraw( double dAmount );

  //void setChkFee( double dFee );
  //double writeCheck( double dAmount );
};
```

```
//With inheritance
//Base class of Account
//Derived class of CheckingAccount

class CheckingAccount : public Account {
  double dChkFee;
private:
  //Constructors
  //destructor

  //void setChkFee( double dFee);
  //double writeCheck( double dAmount );
};
```

For the case of no inheritance, the new class of **`ChkAccount`** will have all members defined and belonged to the class itself. There are no reuses of any existing code and **`ChkAccount`** is used just as any other classes.

5.2 Interest Checking Account

There is another kind of checking account called **`InterestChkAccount`**. This type of account will add a monthly interest to the account that maintains a minimum balance. Without inheritance, this account will have to be defined just as **`ChkAccount`** was; that means all members will have to be explicitly specified and defined.

With inheritance, one can certainly reuse the existing code and add a few members to form a new class of **`InterestCheckingAccount`**. This class should behave exactly the same as the **`InterestChkAccount`** mentioned above.

Example 3b

```
class InterestChkAccount {
  double dBalance;
  double dChkFee;
  double dInterest;
private:
  //Constructors
  //destructor
```

```
class InterestCheckingAccount : public
CheckingAccount {
  double dInterest;
private:
  //Constructors
  //destructor

  //void setInterest( double dAmount);
  //void deposit( double dAmount );
  //double withdraw( double dAmount );

  //void setChkFee( double dFee );
  //double writeCheck( double dAmount );

  //void setInterest( double dAmount );
  //double getInterest();
};
```

```
  //double getInterest( );
};
```

The above examples show a good promise of large amount of code to be reused through inheritance and its many structures.

Note that each of the above classes can be used to create objects of their own. They are “concrete” classes and their objects are creatable.

6. Inheritance – `public` Specification

The general form of a derived class with public inheritance is given as follows,

```
class DerivedClass : public BaseClass
{
  // Added Members
};
```

`public` **Inheritance**

- Each public member in the base class is public in the derived class.
- Each protected member in the base class is protected in the derived class.
- Each private member in the base class remains private in the base class and so is visible only in the base class.

The previous example shows a simple order of calls to constructors and destructors that have no arguments. The example below has both derived and base classes with the same argument required for constructors.

Example 4

```
// Program Name:  cis25L1312.cpp
// Discussion:    public Inheritance -- Passing Argument
#include <iostream>
using namespace std;

class B {
  int x;
public:
  B(int a) {
    cout << "\nBase Constructor!\n";
    x = a;
  }
  ~B()   {
    cout << "\nBase Destructor!\n";
  }
  int getX() {
    return x;
  }
};

class D : public B {
  int y;
public:
  D(int c) : B(c) {
    cout << "\nDerived Constructor!\n";
    y = c;
  }
  ~D() {
    cout << "\nDerived Destructor!\n";
  }
  int getY() {
    return y;
  }
};

int main() {
  D d1(10);

  cout << "\nValue of x in d1 is " << d1.getX();
  cout << "\nValue of y in d1 is " << d1.getY();

  char ch;
  cout << "\n\nPress Any Key + RETURN to continue : ";
  cin >> ch;

  return 0;
}
```

OUTPUT

```
Base Constructor!

Derived Constructor!

Value of x in d1 is 10
Value of y in d1 is 10

Press Any Key + RETURN to continue : c

Derived Destructor!

Base Destructor!
```

The constructors are designed to receive initial values during the creation of objects. Since, the value used in the base constructor is the same as the one used in the derived constructor, only one argument is declared within the derived constructor. Note that, on the heading of the derived constructor, a parameter is passed to the base constructor without type specification (implicit type).

In the next example, the base and derived constructors require different arguments. Thus, separate argument specifications should be given.

Example 5

```
// Program Name: cis25L1313.cpp
// Discussion:   public Inheritance -- Passing Argument
#include <iostream>
using namespace std;

class B {
  int x;
public:
  B(int a) {
    cout << "\nBase Constructor!\n";
    x = a;
  }
  ~B() {
    cout << "\nBase Destructor!\n";
  }
  int getX() {
    return x;
  }
};

class D : public B {
  int y;
public:
  D(int c, int d) : B(c) {
    cout << "\nDerived Constructor!\n";
    y = d;
  }
  ~D() {
    cout << "\nDerived Destructor!\n";
  }
  int getY() {
    return y;
  }
};

int main() {
  D d1( 10, 20 );

  cout << "\nValue of x in d1 is " << d1.getX();
  cout << "\nValue of y in d1 is " << d1.getY();

  return 0;
}
```

OUTPUT

```
Base Constructor!

Derived Constructor!

Value of x in d1 is 10
Value of y in d1 is 20
Derived Destructor!

Base Destructor!
```

In this case, appropriate argument is simply passed along to the base constructor. Note that argument names play important role in selecting which value(s) to be passed on. If the derived class does not need or use any argument then this argument can be passed to the base constructor with proper declaration as given in the following example.

Example 6

```
// Program Name: cis25L1314.cpp
// Discussion:   public Inheritance -- Passing Argument
#include <iostream>
using namespace std;
```

```
class B {
  int x;
public:
  B(int a) {
    cout << "\nBase Constructor!\n";
    x = a;
  }
  ~B() {
    cout << "\nBase Destructor!\n";
  }
  int getX() {
    return x;
  }
};

class D : public B {
  int y;
public:
  D(int c) : B(c) {
    cout << "\nDerived Constructor!\n";
    y = 1;
  }
  ~D() {
    cout << "\nDerived Destructor!\n";
  }
  int getY() { return y; }
};

int main() {
  D d1( 10 );

  cout  << "\nValue of x in d1 is " << d1.getX();
  cout << "\nValue of y in d1 is " << d1.getY();

  return 0;
}
```

OUTPUT

```
Base Constructor!

Derived Constructor!

Value of x in d1 is 10
Value of y in d1 is 1
Derived Destructor!

Base Destructor!
```