Module-4

Inheritance, Pointers, Virtual Functions, Polymorphism

Table of Contents

C++ Inheritance and Derived Classes.

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the **base** class, and the new class is referred to as the **derived** class.

The idea of inheritance implements the **is a** relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Base & Derived Classes:

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form:

```
class derived-class: access-specifier base-class
```

Where access-specifier is one of **public, protected,** or **private**, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class **Shape** and its derived class **Rectangle** as follows:

```
#include <iostream>

using namespace std;

// Base class
class Shape
```

```
{
  public:
    void setWidth(int w)
    {
      width = w;
    }
    void setHeight(int h)
    {
      height = h;
    }
  protected:
    int width;
    int height;
};

// Derived class
class Rectangle: public Shape
{
  public:
    int getArea()
    {
      return (width * height);
    }
};

int main(void)
{
  Rectangle Rect;

  Rect.setWidth(5);
```

```
   Rect.setHeight(7);

   // Print the area of the object.
   cout << "Total area: " << Rect.getArea() << endl;

   return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
Total area: 35
```

Access Control and Inheritance:

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to who can access them in the following way:

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions:

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.
-

Type of Inheritance:

When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied:

1. **Public Inheritance:** When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's**private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
2. **Protected Inheritance:** When deriving from a **protected** base class, **public** and**protected** members of the base class become **protected** members of the derived class.
3. **Private Inheritance:** When deriving from a **private** base class, **public** and **protected**members of the base class become **private** members of the derived class.

Multiple Inheritances:

A C++ class can inherit members from more than one class and here is the extended syntax:

```
class derived-class: access baseA, access baseB....
```

Where access is one of **public, protected,** or **private** and would be given for every base class and they will be separated by comma as shown above. Let us try the following example:

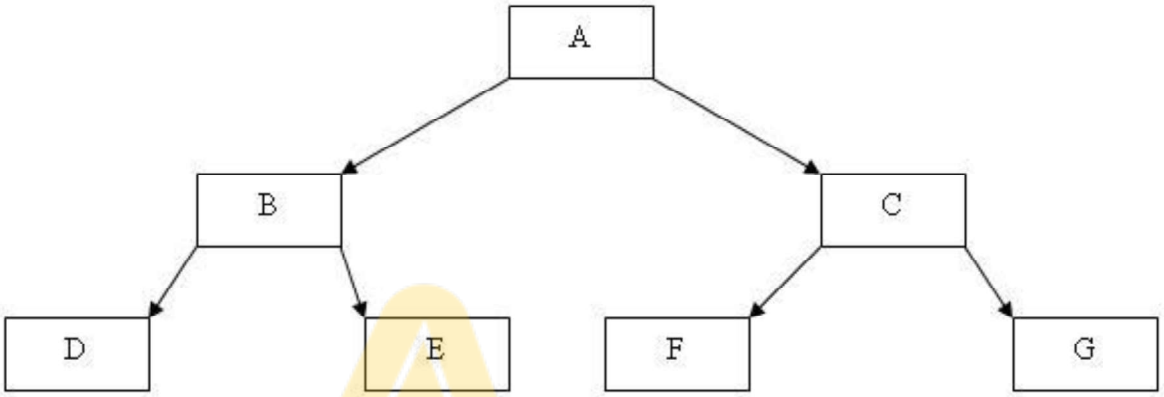
```
#include <iostream>
```

```
using namespace std;

// Base class Shape
class Shape
{
  public:
    void setWidth(int w)
    {
      width = w;
    }
    void setHeight(int h)
    {
      height = h;
    }
  protected:
    int width;
    int height;
};

// Base class PaintCost
class PaintCost
{
  public:
    int getCost(int area)
    {
      return area * 70;
    }
};

// Derived class
```

```
class Rectangle: public Shape, public PaintCost
{
  public:
    int getArea()
    {
      return (width * height);
    }
};

int main(void)
{
  Rectangle Rect;
  int area;

  Rect.setWidth(5);
  Rect.setHeight(7);

  area = Rect.getArea();

  // Print the area of the object.
  cout << "Total area: " << Rect.getArea() << endl;

  // Print the total cost of painting
  cout << "Total paint cost: $" << Rect.getCost(area) << endl;

  return 0;
}
```

Hierarchical Inheritance

It is the process of deriving two or more classes from single base class. And in turn each of the derived classes can further be inherited in the same way. Thus it forms hierarchy of classes or a tree of classes which is rooted at single



class.

Here A is the base class from which we have inherited two classes B and C. From class B we have inherited D and E and from C we have inherited F and G. Thus the whole arrangement forms a hierarchy or tree that is rooted at class A.

The syntax and example for hierarchical inheritance in case of C++ is given below:

Syntax:

```
class A
    {
            -------
            -------
    };

    class B : visibility_label A
    {
            -------
```

```
        -------
};

class C : visibility_label A
{
        -------
        -------
};



class D : visibility_label B
{
        -------
        -------
};

class E : visibility_label B
{
        -------
        -------
};

class F : visibility_label C
{
        -------
        -------
};

class G : visibility_label C
```

```
    {
            -------
            -------
    };
```

Here the visibility_label can be private, protected or public. If we do not specify any visibility_label then by default is private.

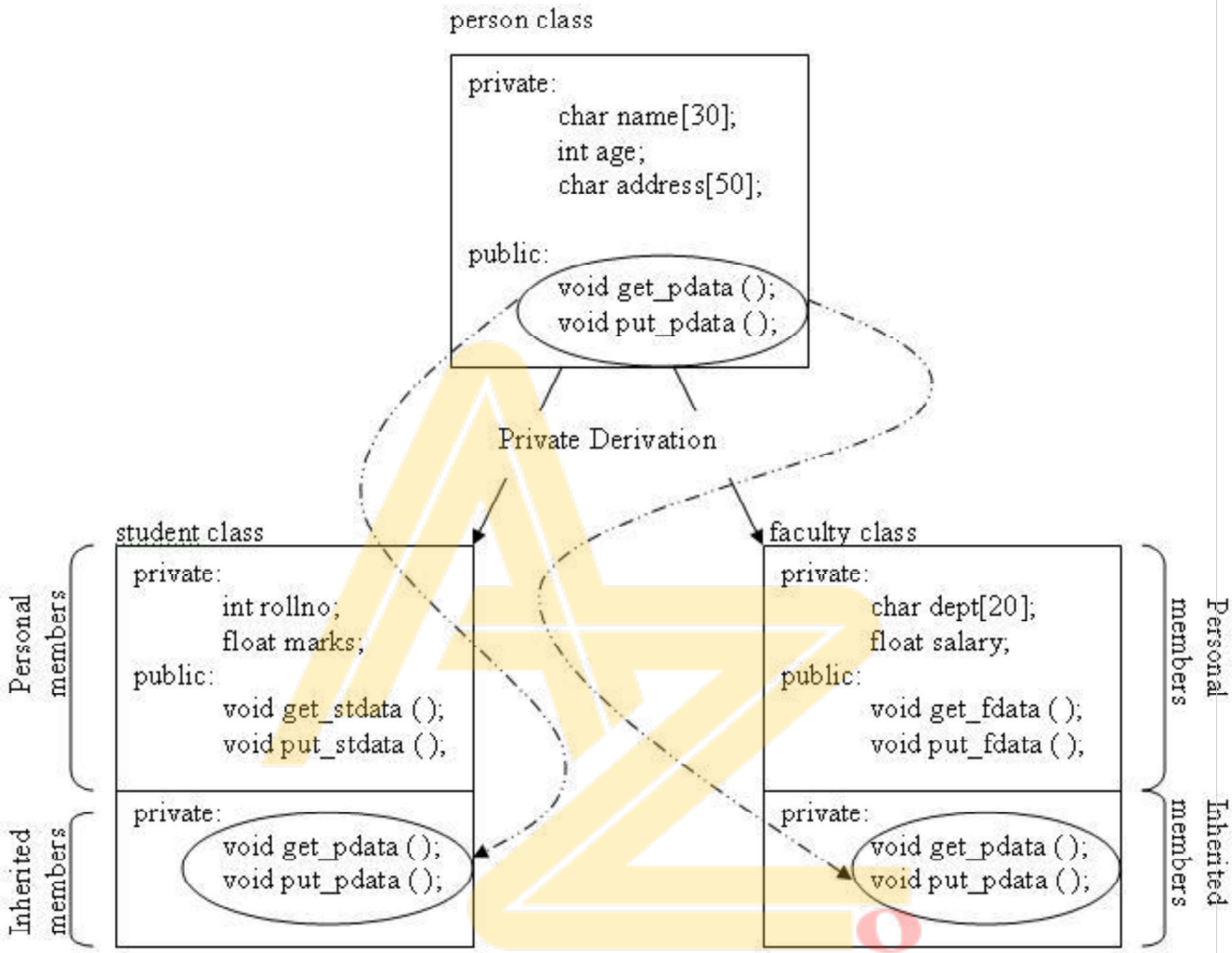
```
class person
{
        private:
                char name[30];
                int age;
                char address[50];
        public:
                void get_pdata ( )
                {
                        cout<<"Enter the name:";
                        cin>>name;
                        cout<<"\nEnter the address:";
                        cin>>address;
                }
                void put_pdata ( )
                {
                        cout<<"Name: "<<name;
                        cout<<"\nAddress: "<<address;
                }
};

class student : private person
{
```

```
        private:
                int rollno;
                float marks;
        public:
                void get_stdata ( )
                {
                        get_pdata ( );
                        cout<<"\nEnter roll number:";
                        cin>>rollno;
                        cout<<"\nEnter marks:";
                        cin>>marks;
                }
                void put_stdata ( )
                {
                        put_pdata ( );
                        cout<<"\nRoll Number: "<<rollno;
                        cout<<"\nMarks: "<<marks;
                }
};

class faculty : private person
{
        private:
                char dept[20];
                float salary;
        public:
                void get_fdata ( )
                {
                        get_pdata ( );
                        cout<<"\nEnter department:";
```

```
                        cin>>dept;
                        cout<<"\nEnter salary:";
                        cin>>salary;
                }
                void put_fdata ( )
                {
                        put_pdata ( );
                        cout<<"\nDepartment: "<<dept;
                        cout<<"\nSalary: "<<salary;
                }
};



void main ( )
{
        student stobj;
        faculty facobj;
        clrscr ( );
        cout<<"Enter the details of the student:\n";
        stobj.get_stdata ( );
        cout<<"\nEnter the details of the faculty member:\n";
        facobj.get_fdata ( );
        cout<<"\nStudent info is:\n";
        stobj.put_stdata ( );
        cout<<"\nFaculty Member info is:\n";
        facobj.put_fdata ( );
        getch ( );
}
```

Here ‘person’ is the base class from which ‘student’ and ‘faculty’ classes have been derived privately

so the public members of the 'person' class will become private members of the 'student' class and 'faculty' class. But the private members will not be inherited.



C++ Hybrid inheritance

Hybrid inheritance: The method of combining any two or more forms of inheritance in single form is called hybrid inheritance.

Program:

/*PROGRAM TO IMPLEMENT HYBRID INHERITANCE*/

#include<conio.h>

```
#include<iostream.h>

class base

{

int a;

public:

base()

{

cout<<"Enter a: ";

cin>>a;

}

int show()

{

cout<<"a = "<<a<<endl;

return(a);

}

~base()

{
```

```
cout<<"Destructor of base class is executed"<<endl;
}
};
class derived1:public base
{
int b;
public:
derived1():base()
{
cout<<"Enter b: ";
cin>>b;
}
int show1()
{
cout<<"b = "<<b<<endl;
return(b);
}
```

```
~derived1()
{
cout<<"Destructor of derived1 class is
executed"<<endl;
}
};
class derived2:public derived1
{
int a,b,c,sum;
public:
derived2():derived1()
{
cout<<"Enter c: ";
cin>>c;
}
void show2()
{
```

```
a=show();

b=show1();

cout<<"c = "<<c<<endl;

sum=a+b+c;

cout<<"Sum of given numbers: "<<sum<<endl;

}

~derived2()

{

cout<<"Destructor of derived2 class is

executed"<<endl;

}

};

class derived3:public derived1

{

int a,b,c,sum;

public:

derived3():derived1()
```

```
{
cout<<"Enter c: ";
cin>>c;
};
void show3()
{
cout<<"c = "<<c<<endl;
a=show();
b=show1();
sum=a+b+c;
cout<<"Sum of given numbers: "<<sum<<endl;
}
~derived3()
{
cout<<"Destructor of derived3 class is
executed"<<endl;
}
```

```
};
void main()
{
clrscr();
cout<<"Getting data for first object"<<endl;
{
derived3 d3;
d3.show3();
}
cout<<"Enter data for second object"<<endl;
{
derived2 d2;
d2.show2();
}
getch();
}
```

Virtual Base Classes

Because a class can be an indirect base class to a derived class more than once, C++ provides a way to optimize the way such base classes work. Virtual base classes offer a way to save space and avoid ambiguities in class hierarchies that use multiple inheritance.

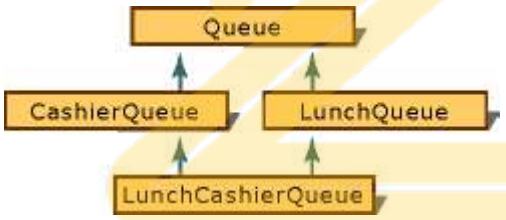
Each nonvirtual object contains a copy of the data members defined in the base class. This duplication wastes space and requires you to specify which copy of the base class members you want whenever you access them.

When a base class is specified as a virtual base, it can act as an indirect base more than once without duplication of its data members. A single copy of its data members is shared by all the base classes that use it as a virtual base.

When declaring a virtual base class, the **virtual** keyword appears in the base lists of the derived classes.

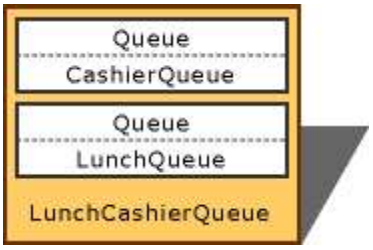
Consider the class hierarchy in the following figure, which illustrates a simulated lunch line.

Simulated Lunch-Line Graph



In the figure, `Queue` is the base class for both `CashierQueue` and `LunchQueue`. However, when both classes are combined to form `LunchCashierQueue`, the following problem arises: the new class contains two subobjects of type `Queue`, one from `CashierQueue` and the other from `LunchQueue`. The following figure shows the conceptual memory layout (the actual memory layout might be optimized).

Simulated Lunch-Line Object



Note that there are two `Queue` subobjects in the `LunchCashierQueue` object. The following code declares `Queue` to be a virtual base class:

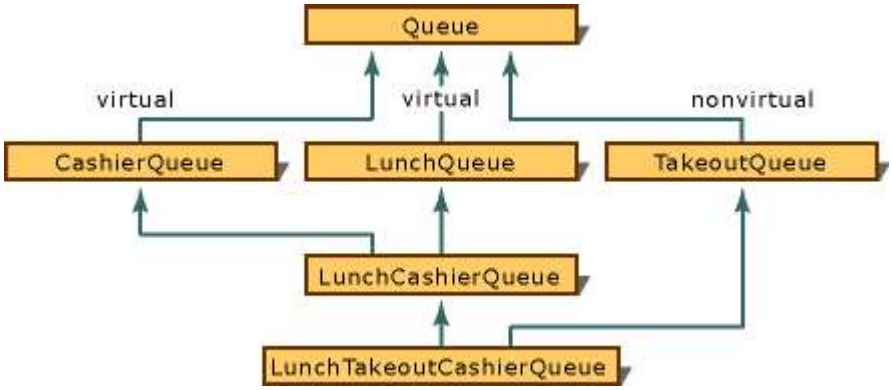
```
// deriv_VirtualBaseClasses.cpp
// compile with: /LD
class Queue {};
class CashierQueue : virtual public Queue {};
class LunchQueue : virtual public Queue {};
class LunchCashierQueue : public LunchQueue, public CashierQueue {};
```

The `virtual` keyword ensures that only one copy of the subobject `Queue` is included (see the following figure).

Simulated Lunch-Line Object with Virtual Base Classes

A class can have both a virtual component and a nonvirtual component of a given type. This happens in the conditions illustrated in the following figure.

Virtual and Nonvirtual Components of the Same Class



In the figure, `CashierQueue` and `LunchQueue` use `Queue` as a virtual base class. However, `TakeoutQueue` specifies `Queue` as a base class, not a virtual base class. Therefore,`LunchTakeoutCashierQueue` has two subobjects of type `Queue`: one from the inheritance path that includes `LunchCashierQueue` and one from the path that includes`TakeoutQueue`. This is illustrated in the following figure.

Object Layout with Virtual and Nonvirtual Inheritance

Note

Virtual inheritance provides significant size benefits when compared with nonvirtual inheritance. However, it can introduce extra processing overhead.

If a derived class overrides a virtual function that it inherits from a virtual base class, and if a constructor or a destructor for the derived base class calls that function using a pointer to the virtual base class, the compiler may introduce additional hidden "vtordisp" fields into the classes with virtual bases. The /vd0 compiler option suppresses the addition of the hidden vtordisp constructor/destructor displacement member. The /vd1 compiler option, the default, enables them where they are necessary.

Turn off vtordisps only if you are sure that all class constructors and destructors call virtual functions virtually.

The /vd compiler option affects an entire compilation module. Use the **vtordisp** pragma to suppress and then reenable vtordisp fields on a class-by-class basis:

#pragma vtordisp(off)

class GetReal : virtual public { ... };

#pragma vtordisp(on)

Abstract classes (C++ only)

An *abstract class* is a class that is designed to be specifically used as a base class. An abstract class contains at least one *pure virtual function*. You declare a pure virtual function by using a *pure specifier* (`= 0`) in the declaration of a virtual member function in the class declaration.

The following is an example of an abstract class:

Function `AB::f` is a pure virtual function. A function declaration cannot have both a pure specifier and a definition. For example, the compiler will not allow the following:

```
struct A {
  virtual void g() { } = 0;
};
```

You cannot use an abstract class as a parameter type, a function return type, or the type of an explicit conversion, nor can you declare an object of an abstract class. You can, however, declare pointers and references to an abstract class. The following example demonstrates this:

Class `A` is an abstract class. The compiler would not allow the function declarations `A g()` or `void h(A)`, declaration of object `a`, nor the static cast of `b` to type `A`.

Virtual member functions are inherited. A class derived from an abstract base class will also be abstract unless you override each pure virtual function in the derived class.

The compiler will not allow the declaration of object `d` because `D2` is an abstract class; it inherited the pure virtual function `f()` from `AB`. The compiler will allow the declaration of object `d` if you define function `D2::g()`.

Note that you can derive an abstract class from a nonabstract class, and you can override a non-pure virtual function with a pure virtual function.

You can call member functions from a constructor or destructor of an abstract class. However, the results of calling (directly or indirectly) a pure virtual function from its constructor are undefined. The following example demonstrates this:

```
struct A {
  A() {
    direct();
    indirect();
  }
  virtual void direct() = 0;
  virtual void indirect() { direct(); }
};
```

The default constructor of `A` calls the pure virtual function `direct()` both directly and indirectly (through `indirect()`).

The compiler issues a warning for the direct call to the pure virtual function, but not for the indirect call.

Pointers to objects and derived classes

Pointers are one of the most powerful and confusing aspects of the C language. A **pointer** is a variable that holds the address of another variable. To declare a pointer, we use an asterisk between the data type and the variable name:

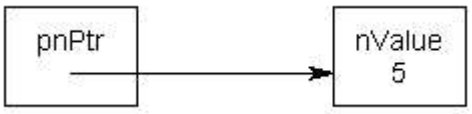
```
1  int *pnPtr; // a pointer to an integer value
2  double *pdPtr; // a pointer to a double value
3
4  int* pnPtr2; // also valid syntax
5  int * pnPtr3; // also valid syntax
```

Note that an asterisk placed between the data type and the variable name means the variable is being declared as a pointer. In this context, the asterisk is not a multiplication. It does not matter if the asterisk is placed next to the data type, the variable name, or in the middle — different programmers prefer different styles, and one is not inherently better than the other.

Since pointers only hold addresses, when we assign a value to a pointer, the value has to be an address. To get the address of a variable, we can use the **address-of operator (&)**:

```
1  int nValue = 5;
2  int *pnPtr = &nValue; // assign address of nValue to pnPtr
```

Conceptually, you can think of the above snippet like this:



It is also easy to see using code:

```
int nValue = 5;
int *pnPtr = &nValue; // assign address of nValue to pnPtr

cout << &nValue << endl; // print the address of variable nValue
```

```
cout << pnPtr << endl; // print the address that pnPtr is holding
```

On the author's machine, this printed:

0012FF7C
0012FF7C

The type of the pointer has to match the type of the variable being pointed to:

```
int nValue = 5;
double dValue = 7.0;

int *pnPtr = &nValue; // ok
double *pdPtr = &dValue; // ok
pnPtr = &dValue; // wrong -- int pointer can not point to double value
pdPtr = &nValue; // wrong -- double pointer can not point to int value
```

Dereferencing pointers

The other operator that is commonly used with pointers is the **dereference operator (*)**. A dereferenced pointer evaluates to the*contents* of the address it is pointing to.

```
int nValue = 5;
cout << &nValue; // prints address of nValue
cout << nValue; // prints contents of nValue

int *pnPtr = &nValue; // pnPtr points to nValue
cout << pnPtr; // prints address held in pnPtr, which is &nValue
```

```
cout << *pnPtr; // prints contents pointed to by pnPtr, which is contents of nValue
```

The above program prints:

```
0012FF7C
5
0012FF7C
5
```

In other words, when pnPtr is assigned to &nValue:
pnPtr is the same as &nValue
*pnPtr is the same as nValue

Because *pnPtr is the same as nValue, you can assign values to it just as if it were nValue! The following program prints 7:

```
int nValue = 5;
int *pnPtr = &nValue; // pnPtr points to nValue

*pnPtr = 7; // *pnPtr is the same as nValue, which is assigned 7
cout << nValue; // prints 7
```

Pointers can also be assigned and reassigned:

```
int nValue1 = 5;
int nValue2 = 7;

int *pnPtr;
```

```
pnPtr = &nValue1; // pnPtr points to nValue1
cout << *pnPtr; // prints 5

pnPtr = &nValue2; // pnPtr now points to nValue2
cout << *pnPtr; // prints 7
```

The null pointer

Sometimes it is useful to make our pointers point to nothing. This is called a **null pointer**. We assign a pointer a null value by setting it to address 0:

```
int *pnPtr;
pnPtr = 0; // assign address 0 to pnPtr
```

or shorthand:

```
int *pnPtr = 0;  // assign address 0 to pnPtr
```

Note that in the last example, the * is not a dereference operator. It is a pointer declaration. Thus we are assigning address 0 to pnPtr, not the value 0 to the variable that pnPtr points to.

C (but not C++) also defines a special preprocessor define called NULL that evaluates to 0. Even though this is not technically part of C++, it's usage is common enough that it will work in every C++ compiler:

```
int *pnPtr = NULL; // assign address 0 to pnPtr
```

Because null pointers point to 0, they can be used inside conditionals:

```
if (pnPtr)
    cout << "pnPtr is pointing to an integer.";
else
    cout << "pnPtr is a null pointer.";
```

Null pointers are mostly used with dynamic memory allocation, which we will talk about in a few lessons.

The size of pointers

The size of a pointer is dependent upon the architecture of the computer — a 32-bit computer uses 32-bit memory addresses — consequently, a pointer on a 32-bit machine is 32 bits (4 bytes). On a 64-bit machine, a pointer would be 64 bits (8 bytes). Note that this is true regardless of what is being pointed to:

```
char *pchValue; // chars are 1 byte
int *pnValue; // ints are usually 4 bytes
struct Something
{
    int nX, nY, nZ;
};
Something *psValue; // Something is probably 12 bytes

cout << sizeof(pchValue) << endl; // prints 4
cout << sizeof(pnValue) << endl; // prints 4
cout << sizeof(psValue) << endl; // prints 4
```

As you can see, the size of the pointer is always the same. This is because a pointer is just a memory address, and the number of bits needed to access a memory address on a given machine is always constant.

The main purpose of C++ programming is to add object orientation to the C programming language and classes are the central feature of C++ that supports object-oriented programming and are often called user-defined types.

A class is used to specify the form of an object and it combines data representation and methods for manipulating that data into one neat package. The data and functions within a class are called members of the class.

C++ Class Definitions:

When you define a class, you define a blueprint for a data type. This doesn't actually define any data, but it does define what the class name means, that is, what an object of the class will consist of and what operations can be performed on such an object.

A class definition starts with the keyword **class** followed by the class name; and the class body, enclosed by a pair of curly braces. A class definition must be followed either by a semicolon or a list of declarations. For example we defined the Box data type using the keyword **class** as follows:

```
class Box
{
  public:
    double length;   // Length of a box
    double breadth;  // Breadth of a box
    double height;   // Height of a box
};
```

The keyword **public** determines the access attributes of the members of the class that follow it. A public member can be accessed from outside the class anywhere within the scope of the class object. You can also specify the members of a class as **private** or **protected** which we will discuss in a sub-section.

Define C++ Objects:

A class provides the blueprints for objects, so basically an object is created from a class. We declare objects of a class with exactly the same sort of declaration that we declare variables of basic types. Following statements declare two objects of class Box:

Both of the objects Box1 and Box2 will have their own copy of data members.

Accessing the Data Members:

The public data members of objects of a class can be accessed using the direct member access operator (.). Let us try following example to make the things clear:

```
#include <iostream>

using namespace std;

class Box
{
  public:
     double length;   // Length of a box
     double breadth;  // Breadth of a box
     double height;   // Height of a box
};

int main( )
{
```

```
   Box Box1;        // Declare Box1 of type Box
   Box Box2;        // Declare Box2 of type Box
   double volume = 0.0;     // Store the volume of a box here

   // box 1 specification
   Box1.height = 5.0;
   Box1.length = 6.0;
   Box1.breadth = 7.0;

   // box 2 specification
   Box2.height = 10.0;
   Box2.length = 12.0;
   Box2.breadth = 13.0;
   // volume of box 1
   volume = Box1.height * Box1.length * Box1.breadth;
   cout << "Volume of Box1 : " << volume <<endl;

   // volume of box 2
   volume = Box2.height * Box2.length * Box2.breadth;
   cout << "Volume of Box2 : " << volume <<endl;
   return 0;
}
```

When the above code is compiled and executed, it produces following result:

```
Volume of Box1 : 210
Volume of Box2 : 1560
```

It is important to note that private and protected members can not be accessed directly using direct member access operator (.). We will learn how private and protected members can be accessed.

Virtual and Pure virtual functions

So far you have got very basic idea about C++ Classes and Objects. There are further interesting concepts related to C++ Classes and Objects which we will discuss in various sub-sections listed below:

Concept	Description
Class member functions	A member function of a class is a function that has its definition or its prototype within the class definition like any other variable.
Class access modifiers	A class member can be defined as public, private or protected. By default members would be assumed as private.
Constructor & destructor	A class constructor is a special function in a class that is called when a new object of the class is created. A destructor is also a special function which is called when created object is deleted.
C++ copy constructor	The copy constructor is a constructor which creates an object by initializing it with an object of the same class, which has been created previously.
C++ friend functions	A **friend** function is permitted full access to private and protected members of a class.

C++ inline functions	With an inline function, the compiler tries to expand the code in the body of the function in place of a call to the function.
The this pointer in C++	Every object has a special pointer **this** which points to the object itself.
Pointer to C++ classes	A pointer to a class is done exactly the same way a pointer to a structure is. In fact a class is really just a structure with functions in it.
Static members of a class	Both data members and function members of a class can be declared as static.

C++ Files and Streams

So far we have been using the **iostream** standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively.

This tutorial will teach you how to read and write from a file. This requires another standard C++ library called **fstream** which defines three new data types:

Data Type	Description
ofstream	This data type represents the output file stream and is used to create files and to write information to files.
ifstream	This data type represents the input file stream and is used to read information from files.

fstream	This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

Opening a File:

A file must be opened before you can read from it or write to it. Either the **ofstream** or **fstream**object may be used to open a file for writing and ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function which is a member of fstream, ifstream, and ofstream objects.

```
void open(const char *filename, ios::openmode mode);
```

Here the first argument specifies the name and location of the file to be opened and the second argument of the **open()** member function defines the mode in which the file should be opened.

Mode Flag	Description
ios::app	Append mode. All output to that file to be appended to the end.
ios::ate	Open a file for output and move the read/write control to the end of the file.
ios::in	Open a file for reading.
ios::out	Open a file for writing.

ios::trunc	If the file already exists, its contents will be truncated before opening the file.

You can combine two or more of these values by **OR**ing them together. For example if you want to open a file in write mode and want to truncate it in case it already exists, following will be the syntax:

Closing a File

When a C++ program terminates it automatically closes flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function which is a member of fstream, ifstream, and ofstream objects.

Writing to a File:

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an **ofstream** or **fstream** object instead of the **cout** object.

Reading from a File:

You read information from a file into your program using the stream extraction operator (<<) just as you use that operator to input information from the keyboard. The only difference is that you use an **ifstream** or **fstream** object instead of the **cin** object.

Read & Write Example:

Following is the C++ program which opens a file in reading and writing mode. After writing information inputted by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen:

```
#include <fstream>
```

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

int main ()
{

  char data[100];

  // open a file in write mode.
  ofstream outfile;
  outfile.open("afile.dat");

  cout << "Writing to the file" << endl;
  cout << "Enter your name: ";
  cin.getline(data, 100);

  // write inputted data into the file.
  outfile << data << endl;

  cout << "Enter your age: ";
  cin >> data;
  cin.ignore();

  // again write inputted data into the file.
  outfile << data << endl;

  // close the opened file.
  outfile.close();

  // open a file in read mode.
```

```
  ifstream infile;
  infile.open("afile.dat");

  cout << "Reading from the file" << endl;
  infile >> data;

  // write the data at the screen.
  cout << data << endl;

  // again read the data from the file and display it.
  infile >> data;
  cout << data << endl;

  // close the opened file.
  infile.close();

  return 0;
}
```

Above examples makes use of additional functions from cin object, like getline() function to read the line from outside and ignore() function to ignore the extra characters left by previous read statement.

File Position Pointers:

Both **istream** and **ostream** provide member functions for repositioning the file-position pointer. These member functions are **seekg** ("seek get") for istream and **seekp** ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be **ios::beg** (the default) for positioning relative to the beginning of a stream, **ios::cur** for positioning relative to the current position in a stream or **ios::end** for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are:

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

1. **throw:** A program throws an exception when a problem shows up. This is done using a**throw** keyword.
2. **catch:** A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
3. **try:** A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise and exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following:

You can list down multiple **catch** statements to catch different type of exceptions in case your**try** block raises more than one exceptions in different situations.

Throwing Exceptions:

Exceptions can be thrown anywhere within a code block using **throw** statements. The operand of the throw statements determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs:

Catching Exceptions:

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try
{
  // protected code
}catch( ExceptionName e )
{
 // code to handle ExceptionName exception
}
```

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows:

The following is an example which throws a division by zero exception and we catch it in catch block.

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

double division(int a, int b)
{
  if( b == 0 )
  {
    throw "Division by zero condition!";
  }
  return (a/b);
}
```

```
int main ()
{
  int x = 50;
  int y = 0;
  double z = 0;

  try {
    z = division(x, y);
    cout << z << endl;
  }catch (const char* msg) {
    cerr << msg << endl;
  }

  return 0;
}
```

Reference and Bibliography

1. Object Oriented Programming with C++ E.Balaguruswamy TMH 6th Edition, 2013
2. ObjectOriented Programming with C++ Robert Lafore Galgotia publication 2010
3. ObjectOriented Programming with C++ Sourav Sahay Oxford University 2006
4. Preece, J. Rogers, Y. and Sharp,H., 2007. Interaction Design: Beyond Human Computer Interaction. 2nd ed. New York: John Wiley & Sons, Inc.
5. Preece, J. Rogers, Y. and Sharp,H., 2001. Interaction Design: Beyond Human Computer Interaction. New York: John Wiley & Sons, Inc.
6. Andrew, G and Drew, P, 2009, Use Case Diagrams in Support of Use Case Modeling: Deriving Understanding from the Picture, Journal of Database Management, 20(1), 1- 24, January-March 2009.