UNIT 4

C++ Inheritance

What is Inheritance?

Inheritance is the process by which new classes called *derived* classes are created from existing classes called *base* classes. The derived classes have all the features of the base class and the programmer can choose to add new features specific to the newly created derived class.

For example, a programmer can create a *base* class named fruit and define *derived* classes as mango, orange, banana, etc. Each of these derived classes, (mango, orange, banana, etc.) has all the features of the *base* class (fruit) with additional attributes or features specific to these newly created derived classes. Mango would have its own defined features, orange would have its own defined features, banana would have its own defined features, etc.

This concept of *Inheritance* leads to the concept of *polymorphism*.

Features or Advantages of Inheritance

- ***Reusability:***

  Inheritance helps the code to be reused in many situations. The base class is defined and once it is compiled, it need not be reworked. Using the concept of inheritance, the programmer can create as many derived classes from the base class as needed while adding specific features to each derived class as needed.

- ***Saves Time and Effort:***

  The above concept of reusability achieved by inheritance saves the programmer time and effort, because the main code written can be reused in various situations as needed.

- ***Increases Program Structure which results in greater reliability.***
- ***Polymorphism***

General Format for implementing the concept of Inheritance:

class derived_classname: access specifier baseclassname

For example, if the *base* class is *exforsys* and the derived class is sample it is specified as:

class sample: public exforsys

The above makes sample have access to both *public* and *protected* variables of base class *exforsys*. Reminder about public, private and protected access specifiers:

- If a member or variables defined in a class is private, then they are accessible by members of the same class only and cannot be accessed from outside the class.
  .
- Public members and variables are accessible from outside the class.
  .
- Protected access specifier is a stage between private and public. If a member functions or variables defined in a class are protected, then they cannot be accessed from outside the class but can be accessed from the derived class.

Inheritance Example:

```
class exforsys
{
private:
int x;

public:
exforsys(void) { x=0;
} void f(int n1)
{
x= n1*5;
}

void output(void) { cout<<x;
}
};

class sample: public exforsys
{
public:
sample(void) { s1=0; }

void f1(int n1)
{
s1=n1*10;
}

void output(void)
{
```

```
exforsys::output();
cout << s1;
}

private:
int s1;
};

int main(void)
{
sample s;
s.f(10);
s.output();
s.f1(20);
s.output();
}
```

The output of the above program is

```
50
200
```

In the above example, the derived class is sample and the base class is *exforsys*. The *derived* class defined above has access to all *public* and *private* variables. *Derived* classes cannot have access to base class *constructors* and *destructors*. The derived class would be able to add new member functions, or variables, or new constructors or new destructors. In the above example, the derived class sample has new member function f1( ) added in it. The line:

```
sample s;
```

creates a derived class object named as s. When this is created, space is allocated for the data members inherited from the base class *exforsys* and space is additionally allocated for the data members defined in the derived class *sample*.

The *base* class constructor *exforsys* is used to initialize the base class data members and the *derived* class *constructor* sample is used to initialize the data members defined in *derived* class.

The access specifier specified in the line:
class sample: public exforsys

Public indicates that the *public* data members which are inherited from the *base* class by the derived class sample remains *public* in the *derived* class.

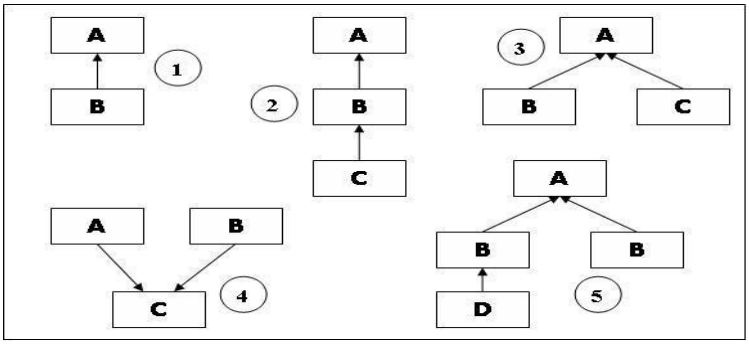
A derived class inherits every member of a base class except:

- ¬ its constructor and its destructor
- ¬ its friends
- ¬ its operator=() members

Types of Inheritance

There are five different inheritances supported in C++:

- Simple / Single
- Multilevel
- Hierarchical
- Multiple
- Hybrid



Accessibility modes and Inheritance

We can use the following chart for seeing the accessibility of the members in the Base class (first class) and derived class (second class).

		Inheritance Mode		
		public	protected	private
Members in Base Class	public	public	protected	private
	protected	protected	protected	private
	private	X	X	X
		Members in derived class		

Here X indicates that the members are not inherited, i.e. they are not accessible in the derived class.

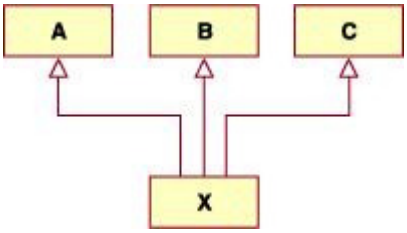
Multiple Inheritance

We can derive a class from any number of base classes. Deriving a class from more than one direct base class is called multiple inheritance.

In the following example, classes A, B, and C are direct base classes for the derived class X:

```
class A { /* ... */ };
class B { /* ... */ };
class C { /* ... */ };
class X : public A, private B, public C { /* ... */ };
```

The following inheritance graph describes the inheritance relationships of the above example. An arrow points to the direct base class of the class at the tail of the arrow:

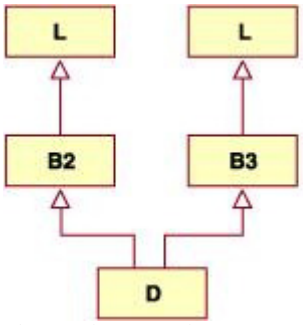


The order of derivation is relevant only to determine the order of default initialization by constructors and cleanup by destructors.

A direct base class cannot appear in the base list of a derived class more than once:

```
class B1 { /* ... */ }; // direct base class class D :
public B1, private B1 { /* ... */ }; // error
```

However, a derived class can inherit an indirect base class more than once, as shown in the following example:




```
class L { /* ... */ };              // indirect base class
class B2 : public L { /* ... */ };
class B3 : public L { /* ... */ };
class D : public B2, public B3 { /* ... */ }; // valid
```

In the above example, class D inherits the indirect base class L once through class B2 and once through class B3. However, this may lead to ambiguities because two subobjects of class L exist, and both are accessible through class D. You can avoid this ambiguity by referring to class L using a qualified class name. For example:

B2::L

or

B3::L.

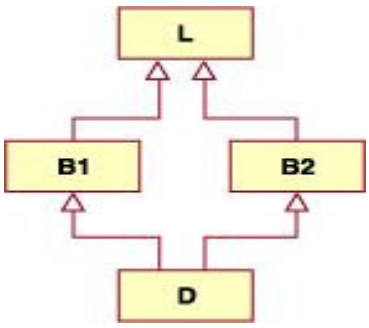
we can also avoid this ambiguity by using the base specifier virtual to declare a base class.

Virtual Base Classes

Suppose you have two derived classes B and C that have a common base class A, and you also have another class D that inherits from B and C. You can declare the base class A as virtual to ensure that B and C share the same subobject of A.

In the following example, an object of class D has two distinct subobjects of class L, one through class B1 and another through class B2. You can use the keyword virtual in front of the base class specifiers in the base lists of classes B1 and B2 to indicate that only one subobject of type L, shared by class B1 and class B2, exists.

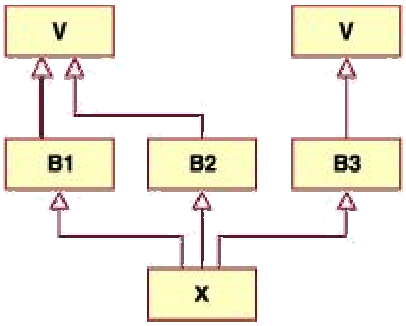
For example:




```
class L { /* ... */ }; // indirect base class
class B1 : virtual public L { /* ... */ };
class B2 : virtual public L { /* ... */ };
class D : public B1, public B2 { /* ... */ }; // valid
```

Using the keyword virtual in this example ensures that an object of class D inherits only one subobject of class L.

A derived class can have both virtual and nonvirtual base classes. For example:




```
class V { /* ... */ };
class B1 : virtual public V { /* ... */ };
class B2 : virtual public V { /* ... */ };
class B3 : public V { /* ... */ };
class X : public B1, public B2, public B3 { /* ...
*/ };
```

In the above example, class X has two subobjects of class V, one that is shared by classes B1 and B2 and one through class B3.

Abstract Classes

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 can 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:

```
class AB {
public:
virtual void f() = 0;
};
```

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:

```
class 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 {
virtual void f() = 0;
```

```
};

class A {
virtual void f() {
} };

 Error:
 Class A is an abstract class
 A g();

 Error:
 Class A is an abstract class
 void h(A);
A& i(A&);

int main() {
 Error:
 Class A is an abstract class
    A a;

   A* pa;
   B b;

// Error:
// Class A is an abstract class
//   static_cast<A>(b);
}
```

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.

For example:

```
class AB {
public:
  virtual void f() = 0;
};

class D2 : public AB {
  void g();
};

int main() {
```

```
  D2 d;
}
```

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:

```
clss 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.

Polymorphism

Polymorphism is the phenomenon where the same message sent to two different objects produces two different set of actions. Polymorphism is broadly divided into two parts:

- *Static polymorphism* – exhibited by overloaded functions.
- *Dynamic polymorphism* – exhibited by using late binding.

Static Polymorphism

Static polymorphism refers to an entity existing in different physical forms simultaneously. Static polymorphism involves binding of functions based on the number, type, and sequence of arguments. The various types of parameters are specified in the function declaration, and therefore the function can be bound to calls at compile time. This form of association is called *early binding*. The term *early binding* stems from the fact that when the program is executed, the calls are already bound to the appropriate functions.

The resolution of a function call is based on number, type, and sequence of arguments declared for each form of the function. Consider the following function declaration:

```
void add(int , int);
void add(float, float);
```

When the *add()* function is invoked, the parameters passed to it will determine which version of the function will be executed. This resolution is done at compile time.

Dynamic Polymorphism

Dynamic polymorphism refers to an entity changing its form depending on the circumstances. A function is said to exhibit dynamic polymorphism when it exists in more than one form, and calls to its various forms are resolved dynamically when the program is executed. The term *late binding* refers to the resolution of the functions at run-time instead of compile time. This feature increases the flexibility of the program by allowing the appropriate method to be invoked, depending on the context.

Static Vs Dynamic Polymorphism

- Static polymorphism is considered more efficient, and dynamic polymorphism more flexible.
- Statically bound methods are those methods that are bound to their calls at compile time. Dynamic function calls are bound to the functions during run-time. This involves the additional step of searching the functions during run-time. On the other hand, no run-time search is required for statically bound functions.
- As applications are becoming larger and more complicated, the need for flexibility is increasing rapidly. Most users have to periodically upgrade their software, and this could become a very tedious task if static polymorphism is applied. This is because any change in requirements requires a major modification in the code. In the case of dynamic binding, the function calls are resolved at run-time, thereby giving the user the flexibility to alter the call without having to modify the code.
- To the programmer, efficiency and performance would probably be a primary concern, but to the user, flexibility or maintainability may be much more important. The decision is thus a trade-off between efficiency and flexibility.

Introduction To Virtual Functions

Polymorphism, one of the three main attributes of an OOP language, denotes a process by which different implementations of a function can be accessed by the use of a single name. Polymorphism also means one interface, multiple methods.

C++ supports polymorphism both at run-time and at compile-time. The use of overloaded functions is an example of compile-time polymorphism. Run-time polymorphism can be achieved by the use of both derived classes and *virtual functions.*

Pointers to Derived Types

We know that pointer of one type may not point to an object of another type. You'll now learn about the one exception to this general rule: a pointer to an object of a base class can also point to any object derived from that base class.

Similarly, a reference to a base class can also reference any object derived from the original base class. In other words, a base class reference parameter can receive an object of types derived from the base class, as well as objects within the base class itself.

Virtual Functions

How does C++ handle these multiple versions of a function? Based on the parameters being passed, the program determines at run-time which version of the virtual function should be the recipient of the reference. It is the type of object being pointed to, not the type of pointer, that determines which version of the virtual function will be executed!

To make a function *virtual,* the virtual keyword must precede the function declaration in the base class. The redefinition of the function in any derived class does not require a second use of the virtual keyword. Have a look at the following sample program to see how this works:

```
#include <iostream.h>
using namespace std;
class bclass {
public:
virtual void whichone()
{ cout << bclass\n;
   }
};
class dclass1 : public bclass
{ public:
void whichone() {
cout << dclass1\n;
   }
};
class dclass2 : public bclass
{ public:
void whichone() {
cout << dclass2\n;
   }
};
int main()
{
bclass Obclass;
```

```
bclass *p; dclass1
Odclass1; dclass2
Odclass2;
 point to bclass
p = &Obclass;
 access bclass‘s
whichone() p->whichone();
 point to dclass1
p = &Odclass1;
// access dclass1‘s
whichone() p->whichone();
// point to dclass2
p = &Odclass2;
// access dclass2‘s whichone()
p->whichone();
return 0;
}
```

The output from this program looks like this:

```
bclass
dclass1
dclass2
```

Notice how the type of the object being pointed to, not the type of the pointer itself, determines which version of the virtual whichone() function is executed.

Virtual Functions and Inheritance

. Virtual functions are inherited intact by all subsequently derived classes, even if the function is not redefined within the derived class. So, if a pointer to an object of a derived class type calls a specific function, the version found in its base class will be invoked. Look at the modification of the above program. Notice that the program does not define the whichone() function in d class2.

```
#include <iostream.h>
using namespace std;
class bclass {
public:
virtual void whichone()
{ cout << bclass\n;
   }
};
class dclass1 : public bclass
{ public:
```

```
void whichone() {
cout << dclass1\n;
    }
};
class dclass2 : public bclass
{ };
int main()
{
bclass Obclass;
bclass *p; dclass1
Odclass1; dclass2
Odclass2; p =
&Obclass; p-
>whichone();
p = &Odclass1;
p->whichone();
p = &Odclass2;
// accesses dclass1‘s
function p->whichone();
return 0;
}
```

The output from this program looks like this:

```
bclass
dclass1
bclass
```

Rtti Constituents

The operators typeid and dynamic_cast<> offer two complementary forms of accessing the runtime type information of their operands. The operand‘s runtime type information itself is stored in a type_info object.

It‘s important to realize that RTTI is applicable solely to polymorphic objects; a class must have at least one virtual-member function in order to have RTTI support for its objects.

std::type_info

For every distinct type, C++ instantiates a corresponding std::type_info (defined in <typeinfo>) object. The interface is as follows:

```
namespace std {
class type_info
 {
```

```
public:
virtual ~type_info(); //type_info can serve as a base
class // enable comparison
bool operator==(const type_info& rhs ) const;
// return !( *this == rhs)
bool operator!=(const type_info& rhs ) const;
bool before(const type_info& rhs ) const; // ordering
//return a C-string containing the type's name
const char* name() const;
private:
//objects of this type cannot be copied
type_info(const type_info& rhs );
type_info& operator=(const type_info&
rhs); }; //type_info
}
```

All objects of the same class share a single type_info object. The most widely used member functions of type_info are name() and operator==. But before you can invoke these member functions, you have to access the type_info object itself. How? Operator typeid takes either an object or a type name as its argument and returns a matching type_info object. The dynamic type of an object can be examined as follows:

```
OnRightClick (File & file)
{
if ( typeid( file) == typeid( TextFile ) )
 {
//received a TextFile object; printing should be enabled
 }
else
 {
//not a TextFile object; printing disabled
 }
}
```

To understand how it works, look at the highlighted source line:

if (typeid(file) == typeid(TextFile)).

The if statement tests whether the dynamic type of file is TextFile (the static type of file is File, of course). The leftmost expression, typeid(file), returns a type_info object that holds the necessary runtime type information associated with the object file. The rightmost expression, typeid(TextFile), returns the type information associated with class TextFile. (When typeid is applied to a class name rather than an object, it always returns a type_info object that corresponds to that class name.)

As shown earlier, type_info overloads the operator ==. Therefore, the type_info object returned by the leftmost typeid expression is compared to the type_info object returned by the rightmost typeid expression. If file is an instance of TextFile, the if statement evaluates to true. In that case, OnRightClick should display an additional option in the menu: print(). On the other hand, if file is not a TextFile, the if statement evaluates to false, and the print() option is disabled.

dynamic_cast<>

OnRightClick() doesn‘t really need to know whether file is an instance of class TextFile (or of any other class, for that matter). Rather, all it needs to know is whether file *is-a* TextFile. An object *is-a* TextFile if it‘s an instance of class TextFile or any class derived from it. For this purpose, you use the operator dynamic_cast<>. dynamic_cast<> takes two arguments: a type name, and an object that dynamic_cast<> attempts to cast at runtime. For example:

```
//attempt to cast file to a reference
to //an object of type TextFile
dynamic_cast <TextFile &> (file);
```

If the attempted cast succeeds, the second argument *is-a* TextFile. But how do you know whether dynamic_cast<> was successful?

There are two flavors of dynamic_cast<>; one uses pointers and the other uses references. Accordingly, dynamic_cast<> returns a pointer or a reference of the desired type when it succeeds. When dynamic_cast<> cannot perform the cast, it returns NULL; or, in the case of a reference, it throws an exception of type std::bad_cast:

```
TextFile * pTest = dynamic_cast <TextFile
*> (&file); //attempt to cast
//file address to a pointer to TextFile
if (pTest) //dynamic_cast succeeded, file is-a TextFile
{
//use pTest
}
else // file is not a TextFile; pTest has a NULL value
{
}
```

Remember to place a reference dynamic_cast<> expression inside a try block and include a suitable catch statement to handle std::bad_cast exceptions.

Now you can revise OnRightClick() to handle HTMLFile objects properly:

```
OnRightClick (File & file)
{
```

```
try
 {
TextFile temp = dynamic_cast<TextFile&>
(file); //display options, including print
switch (message)
  {
case m_open:
temp.open(); //either TextFile::open or HTMLFile::open
break;
case m_print:
temp.print();//either TextFile::print or HTMLFile::print
break;
}//switch
}//try
catch (std::bad_cast& noTextFile)
 {
// treat file as a BinaryFile; excludeprint
 }
}// OnRightClick
```

The revised version of OnRightClick() handles an object of type HTMLFile properly. When the user clicks the open option in the file manager application, OnRightClick() invokes the member function open() of its argument. Likewise, when it detects that its argument is a TextFile, it displays a print option.

This hypothetical file manager example is a bit contrived; with the use of templates and more sophisticated design, it could have been implemented without dynamic_cast. Yet dynamic type casts have other valid uses in C++, as I'll show next.

Cross Casts

A *cross cast* converts a multiply-inherited object to one of its secondary base classes. To see what a cross cast does, consider the following class hierarchy:

```
struct A
{
int i;
virtual ~A () { } //enforce polymorphism; needed for
dynamic_cast };
struct B
{
bool b;
};

struct D: public A, public B
{
```

```
int k;
D() { b = true; i = k = 0;
} };

A *pa = new D;
B *pb = dynamic_cast<B*> pa; //cross cast; access the second
base //of a multiply-derived object
```

The static type of pa is A *, whereas its dynamic type is D *. A simple static_cast<> cannot convert a pointer to A into a pointer to B, because A and B are unrelated. Don't even think of a brute force cast. It will cause disastrous results at runtime because the compiler will simply assign pa to pb; whereas the B sub-object is located at a different address within D than the A sub-object. To perform the cross cast properly, the value of pb has to be calculated at runtime. After all, the cross cast can be done in a source file that doesn't even know that class D exists! The following listing demonstrates why a dynamic cast, rather than a compile-time cast, is required in this case:

```
A *pa = new D;
// disastrous; pb points to the sub-object A within d
B pb = (B) pa; bool bb = pb->b; // bb has an undefined value
 pb was not properly
 adjusted; pa and pb are identical
cout<< pa:  << pa <<  pb: <<pb <<endl;
pb = dynamic_cast<B*> (pa); //cross cast; adjust pb
correctly bb= pb->b; //OK, bb is true
 OK, pb was properly adjusted;
 pa and pb have distinct values
cout<< pa: << pa <<  pb:  << pb <<endl;
```

The code displays two lines of output; the first shows that the memory addresses of pa and pb are identical. The second line shows that the memory addresses of pa and pb are indeed different after performing a dynamic cast as needed.

Downcasting from a Virtual Base

A *downcast* is a cast from a base to a derived object. Before the advent of RTTI, downcasts were regarded as bad programming practice and notoriously unsafe. dynamic_cast<> enables you to use safe and simple downcasts from a virtual base to its derived object. Look at the following example:

```
struct V
{
virtual ~V (){} //ensure polymorphism
};
struct A: virtual V {};
```

```
struct B: virtual V {}; struct
D: A, B {};

#include <iostream> using
namespace std; int main()
{
V *pv = new D;
A* pa = dynamic_cast<A*> (pv); // downcast
cout<< pv: << pv << pa: << pa <<endl; // OK, pv and pa have //different
addresses
}
```

V is a virtual base for classes A and B. D is multiply-inherited from A and B. Inside main(), pv is declared as a V * and its dynamic type is D *. Here again, as in the cross-cast example, the dynamic type of pv is needed in order to properly downcast it to a pointer to A. A static_cast<> would be rejected by the compiler in this case, as the memory layout of a virtual sub-object might be different from that of a non-virtual sub-object. Consequently, it‘s impossible to calculate at compile time the address of the sub-object A within the object pointed to by pv. As the output of the program shows, pv and pa indeed point to different memory addresses.