**INHERITANCE**

Inheritance is the concept by which the properties of one entity are available to another. It allows new classes to be built from older and less specialized classes instead of being rewritten from scratch. Let us take an e.g. of a car. The class car inherits from another class automobiles which itself inherits from another class automobiles which itself inherits from another class called vehicles. In order to simulate this environment, C ++ allows to inherit properties and functions from another. The class that inherits properties and functions is called the subclass or the derived class and the class from which they are inherited is called the superclass or the baseclass. The derived class inherits all the properties of the base class and can add properties and refinements of its own. The bare class remains unchanged. The most important advantage of inheritance is code reusability.

Reusing existing code saves time, money and efforts and increases a program's reliability.

**NEED FOR INHERITANCE**

1 one major reason behind this is the capability to express the inheritance relationship which ensures the closeness with the real world models.

2. Another is the idea of reusability. Inheritance allows the addition of additional features to an existing class without modifying it.

3. One reason is extensive nature of inheritance. If a class A inherits properties of another class B, then all subclasses of A will automatically inherit the properties of B.

**DIFFERENT FORMS OF INHERITANCE** -

**1. Single Inheritance** - when a subclass inherits only from one base class, it is known as single inheritance.

Printed Document

Book

Figure 3.1 Single Inheritance

**2. Multiple Inheritance** - when a subclass inherits from multiple base classes, it is known as multiple inheritance.

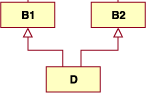


Figure 3.2 Multiple Inheritance

**3. Hierarchical inheritance** - when many subclasses inherits from a simple bare class, it is known as hierarchical inheritance.

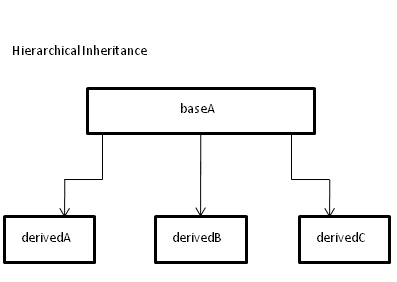


Figure 3.3 Hierarchical Inheritance

**4. Multilevel inheritance** - the transitive nature of inheritance is reflected by this form in inheritance. When a subclass inherits from a class that itself inherits form another class, it is known as multilevel inheritance.

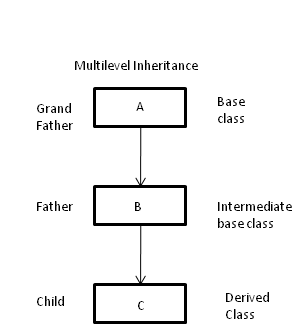


Figure 3.4 Multilevel Inheritance

**5. Hybrid inheritance**: - when a subclass inherits from multiple base classes and its entire base classes inherit from a single base class, this form of inheritance is known as hybrid inheritance.

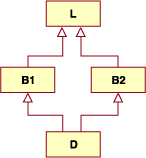


Figure 3.5 Multilevel Multipath (Hybrid) Inheritance

**DEFINING DERIVED CLASS**

A derived class can be defined by specifying its relationship with the base class in addition to its own details. The general form of defining a derived class is:-

class derived - class-name : visibility mode base-class

{

-------

------- members of derived class

-------

};

The colon indicates that the derived - class - name is derived from the base - class - name. The visibility mode is optional and, if present, may be either private or public. The default visibility mode is private.

Visibility mode specifies whether the features of the base class are privately derived or publicly derived.

eg.

Class ABC : private XYZ

// private derivation

Class ABC : public XYZ

// public derivation

when a base class is privately inherited by a derived class, 'public members' of the base class become 'private members' of the derived class and therefore the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class. A public member of a class can be accessed by its own object using dot operator. The result is that no members of the base class are accessible to the objects of the derived class.

When the base class is publicly inherited, 'public members' of the base class become 'public member' of the derived class and therefore they are accessible to the objects of the derived class.

In both the cases, the private members are not inherited and therefore, the private members of a base class will never become the members of its derived class.

**Making a private member inheritable**

C++ provides a third visibility modifies, protect, which serve a limited purpose in inheritance. A member declared as protected is accessible by the member functions within its class and any class immediately derived from it. it cannot be accessed by the functions outside these two classes.

A class can now use the three visiblity modes as illustrated below

class alpha

{

private // optional, visible

// to member functions

// within its class

protected // visible to member

// function of its own

// and derived class

public : // visible to all function

// in the program

when a protected member is inherited in public mode, it becomes protected in the derived class too and therefore is acessible by the member functions of the derived class. It is also ready for further inheritance. A protected member, inherited in the private mode derivation, becomes private in the derived class. Although it is available to the member functions of the derived class, it is not available for further inheritance. it is also possible to inherit a base class in protected mode.

In protected derivation, both the public and protected members of the base class become protected member of the derived class.

**VIRTUAL BASE CLASS**

Consider a situation where all the three kinds of inheritance, namely, multilevel, multiple and hierarchical inheritance, are involved. Let us take an e.g.

The ‘child’ has two direct base classes ‘parent1’ and ‘parent2’ which themselves have a common base class ‘grandparent’. The ‘child’ inherits the traits of ‘grandparent’, via two separate paths. it can also inherits directly as shown by broken line. the ‘grandparent’ is sometimes referred to as indirect base class.

In this case all the public and protected members of ‘grandparent’ are inherited into ‘child’ twice, first via ‘parent1’ and again via parent2’. This means, ‘child’ would have duplicate sets of the members inherited from ‘grandparent’. This introduces ambiguity and should be avoided.

The duplication of inherited members due to these multiple paths can be avoided by making the common base class as virtual base class while declaring the direct on intermediate base classes which are shown as follows:

class A

{

\_\_\_\_\_\_\_ // grand parent

\_\_\_\_\_\_\_

};

class B1 : virtual public A // parent 1

\_\_\_\_\_\_\_

\_\_\_\_\_\_\_

};

class B2 : public virtual A // parent 2

\_\_\_\_\_\_\_

\_\_\_\_\_\_\_

};

class C : public B1, public B2 // child

\_\_\_\_\_\_\_ // only one copy of A

\_\_\_\_\_\_\_ // will be inherited

};

When a class is made a virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exists between the virtual base class and a derived class.

**Generic Programming**

Generic programming is about generalizing software components so that they can be easily reused in a wide variety of situations. In C++, class and function templates are particularly effective mechanisms for generic programming because they make the generalization possible without sacrificing efficiency.

C++ provides unique abilities to express the ideas of Generic Programming through *templates*. Templates provide a form of parametric polymorphism that allows the expression of generic algorithms and data structures. The *instantiation* mechanism of C++ templates insures that when a generic algorithm or data structure is used, a fully-optimized and specialized version will be created and tailored for that particular use, allowing generic algorithms to be as efficient as their non-generic counterparts. Additionally, the C++ notion of *specialization* allows compile-time selection among alternative algorithms. The flexibility of C++ templates has made C++ an attractive language for Generic Programming.

### Function templates

Function templates are special functions that can operate with *generic types*. This allows us to create a function template whose functionality can be adapted to more than one type or class without repeating the entire code for each type.

In C++ this can be achieved using *template parameters*. A template parameter is a special kind of parameter that can be used to pass a type as argument: just like regular function parameters can be used to pass values to a function, template parameters allow to pass also types to a function. These function templates can use these parameters as if they were any other regular type.

The format for declaring function templates with type parameters is:

template <class identifier> function\_declaration;

template <typename identifier> function\_declaration;

The only difference between both prototypes is the use of either the keyword class or the keyword typename. Its use is indistinct, since both expressions have exactly the same meaning and behave exactly the same way.

For example, to create a template function that returns the greater one of two objects we could use:

|  |  |
| --- | --- |
| 1 2 3 4 | template <class myType>  myType GetMax (myType a, myType b) {  return (a>b?a:b);  } |

Here we have created a template function with myType as its template parameter. This template parameter represents a type that has not yet been specified, but that can be used in the template function as if it were a regular type. As you can see, the function template GetMax returns the greater of two parameters of this still-undefined type.

To use this function template we use the following format for the function call:

function\_name <type> (parameters);

For example, to call GetMax to compare two integer values of type int we can write:

|  |  |
| --- | --- |
| 1 2 | int x,y;  GetMax <int> (x,y); |

When the compiler encounters this call to a template function, it uses the template to automatically generate a function replacing each appearance of myType by the type passed as the actual template parameter (int in this case) and then calls it. This process is automatically performed by the compiler and is invisible to the programmer.

Here is the entire example:

|  |  |  |
| --- | --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 | // function template  #include <iostream>  template <class T>  T GetMax (T a, T b) {  T result;  result = (a>b)? a : b;  return (result);  }  int main () {  int i=5, j=6, k;  long l=10, m=5, n;  k=GetMax<int>(i,j);  n=GetMax<long>(l,m);  cout << k << endl;  cout << n << endl;  return 0;  } | 6  10 |