**Inheritance**

# Inheritance

Inheritance in most class-based object-oriented languages is a mechanism in which one object acquires all the properties and behaviours of the parent object. Inheritance allows programmers to: create classes that are built upon existing classes, to specify a new implementation to maintain the same behaviour (realizing an interface), to reuse code and to independently extend original software via public classes and interfaces.

The relationships of objects or classes through inheritance give rise to a directed graph.

Inheritance should not be confused with subtyping.

Inheritance is contrasted with object composition, where one object contains another object (or objects of one class contain objects of another class); see composition over inheritance. Composition implements a has-a relationship, in contrast to the is-a relationship of subtyping.

# Subclasses and Superclasses

An inherited class is called a subclass of its parent class or super class.

Commonly the subclass automatically inherits the instance variables and member functions of its superclasses. The general form of defining a derived class is:

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

....

....

....

};

The colon indicates that the derived-class-name is derived from the base-class-name.

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

The default visibility-mode is private. The visibility-mode is optional and, if present, may be either public, private or protected.

Note: Inheritance doesn’t work in reverse. The base class and its objects don’t know anything about any classes derived from the base class.

# What all is Inherited?

A derived class inherits from its parent.

* Every data member defined in the parent class (although such members may not always be accessible in the derived class)
* Every ordinary member function of the parent class (although such members may not always be accessible in the derived class)
* The same initial data layout as the base class

Things which a derived class doesn’t inherits from its parent:

* The base class’s constructors, destructor and operator=
* The base class’s friends

# Uninheritable classes

A class may be declared as uninheritable by adding certain class modifiers to the class declaration.

Examples

* "final" keyword in Java and C++11 onwards
* "sealed" keyword in C#

Such modifiers are added to the class declaration before the "class" keyword and the class identifier declaration.

Such sealed classes restrict reusability, particularly when developers only have access to precompiled binaries and not source code.

# Visibility of inherited members

|  |  |  |  |
| --- | --- | --- | --- |
| Base class visibility | Derived class visibility | | |
|  | **Public derivation** | **Private derivation** | **Protected derivation** |
| Private | Not inherited | Not inherited | Not inherited |
| Protected | Protected | Private | Protected |
| Public | Public | Private | Protected |

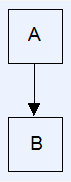
Private members of a base class are never inherited and thus will never become members of its derived class.

# Types of Inheritance in C++

1. Single Inheritance
2. Multiple Inheritance
3. Multilevel Inheritance
4. Hierarchical Inheritance
5. Hybrid (Virtual) Inheritance

## Single Inheritance

One derived class is inherited from one base class only



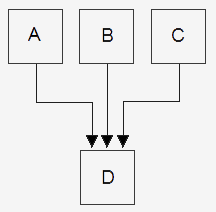
class subclass\_name : access\_mode base\_class {

//body of subclass

};

## Multiple Inheritance

A derived class is inherited from more than one base classes



class subclass\_name : access\_mode base\_class1, access\_mode base\_class2, .... {

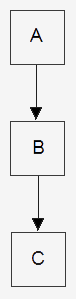
//body of subclass

};

Note: Study multiple inheritance in detail

## Multilevel Inheritance

A derived class is created from another derived class



class subclass\_name\_01 : access\_mode base\_class {

//body of subclass 01

};

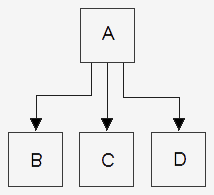
class subclass\_name\_02 : access\_mode subclass\_name\_01 {

//body of subclass 02

};

## Hierarchical Inheritance

More than one derived class is created from a single base class



class subclass\_name\_01 : access\_mode base\_class {

//body of subclass 01

};

class subclass\_name\_02 : access\_mode base\_class {

//body of subclass 02

};

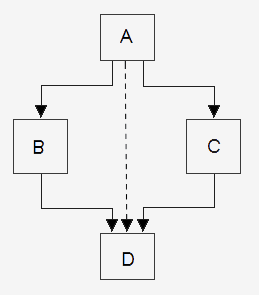
class subclass\_name\_03 : access\_mode base\_class {

//body of subclass 03

};

## Hybrid (Virtual) Inheritance

Hybrid Inheritance is implemented by combining more than one type of inheritance



class subclass\_name\_01 : access\_mode base\_class {

//body of subclass 01

};

class subclass\_name\_02 : access\_mode base\_class {

//body of subclass 02

};

class derived\_name: access\_mode subclass\_name\_01, access\_mode subclass\_name\_02 {

//body of subclass

};

# Virtual Base Classes

Virtual base class is used to avoid duplication of inherited members in case of multiple inheritance example: Diamond ring problem

Study Multiple inheritance as separate

# Virtual methods

If the superclass method is a virtual method, then invocations of the superclass method will be dynamically dispatched (late binding).

Some languages require methods to be specifically declared as virtual (e.g. C++) and in others all methods are virtual (e.g. Javarice).

An invocation of a non-virtual method will always be statically dispatched (early binding i.e. the address of the function call is determined at compile-time).

Static dispatch (early binding) is faster than dynamic dispatch (late binding) and allows optimisations such as inline expansion.

# Abstract Classes

Sometimes implementation of all function cannot be provided in a base class because we don’t know the implementation. Such a class is called abstract class.

An abstract class is one that is **not used to create objects**. An abstract class is designed only to act as a base class (to be inherited by other classes).

By definition a class can only be considered as an abstract class if it has at least one pure function.

Interesting Facts:

1. A class is abstract if it has at least **one pure virtual function**.
2. We can have pointers and references of abstract class type.
3. If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.
4. An abstract class can have constructors.

# Interface vs Abstract Classes:

An interface does not have implementation of any of its methods, it can be considered as a collection of method declarations. In C++, an interface can be simulated by making all methods as pure virtual. In Java, there is a separate keyword for interface.

# Constructors in Derived Classes

As long as no base class constructor takes any arguments, the derived class need not have a constructor.

If any base class contains a constructor with one or more arguments, then it is mandatory for the derived class to have a constructor and pass the arguments to the base class constructors.

## Why the base class’s constructor is called on creating an object of derived class?

When we create an object of derived class, all of the members of derived class must be initialized but the inherited members in derived class can only be initialized by the base class’s constructor as the definition of these members exists in base class only. This is why the constructor of base class is called first to initialize all the inherited members.

Important Points:

* Whenever the derived class’s default constructor is called, the base class’s default constructor is called automatically.
* To call the parameterised constructor of base class inside the parameterised constructor of sub class, we have to mention it explicitly.
* The parameterised constructor of base class cannot be called in default constructor of sub class, it should be called in the parameterised constructor of sub class.

## Order of Constructor/ Destructor Call in C++

The base constructor is executed first and then the constructor in the derived class is executed.

In case of multiple inheritance: base classes are constructed in the order in which they appear in the declaration of the derived class.

In case of multilevel inheritance: the constructors will be executed in the order pf inheritance

Constructors for virtual base classes are invoked before any non-virtual base classes.

If there are multiple virtual base classes, they are invoked in the order in which they are declared.

|  |  |
| --- | --- |
| Method of Inheritance | Order of execution |
| class D: public B {  }; | B(): Base constructor  D(): Derived constructor |
| class D: public B1, public B2 {  }; | B1(): base first  B2(): base second  D(): derived |
| class D: public B1, virtual public B2 {  }; | B2(): virtual base  B1(): ordinary base  D(): derived |

**Destructors in C++ are called in the opposite order of that of Constructors.**

# Applications

## Overriding

## Code reuse

# Issues and alternatives