# Ambiguity in Inheritance

- In simple as well as Multilevel

- In Hybrid Inheritance

## Ambiguity in Multilevel or Simple Inheritance

```
#include<iostream.h>
#include<conio.h>
class ClassA
{ public:
Out()
{ cout<<“hi”;}
};
class ClassB : public ClassA
{ public:
Out()
{ cout<<“hello”;}
};
void main()
{ B b1;
B1.out();   // will output hello onl
}
```

In this example, both **ClassA** & **ClassB** ha ve same function name.
.

## Ambiguity in Hybrid Inheritance

```
#include<iostream.h>
#include<conio.h>
class ClassA
{ public: int a; };
class ClassB : public ClassA
{ public: int b; };
class ClassC : public ClassA
{ public: int c; };
class ClassD : public ClassB, public ClassC
{ public: int d; };
void main()
{ ClassD obj;
obj.a = 10; //Statement Error occur //
obj.a = 100; //Statement Error occur
    }
```

In this example, both **ClassB** & **ClassC** in herit **ClassA**, they both have single copy of **ClassA**.
However **ClassD** inherit both **ClassB** & **ClassC**, therefore **ClassD** have two copies of **ClassA**, one from **ClassB**and another from **ClassC**.

# Polymorphism

- The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

poly + morphism

# Can be achieved by

- Early Binding
  - Function Overloading
  - Operator Overloading

- Late Binding
  - Virtual Function

# Virtual Function

- A virtual function a member function which is declared within base class and is re-defined (Overriden) by derived class.When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function.

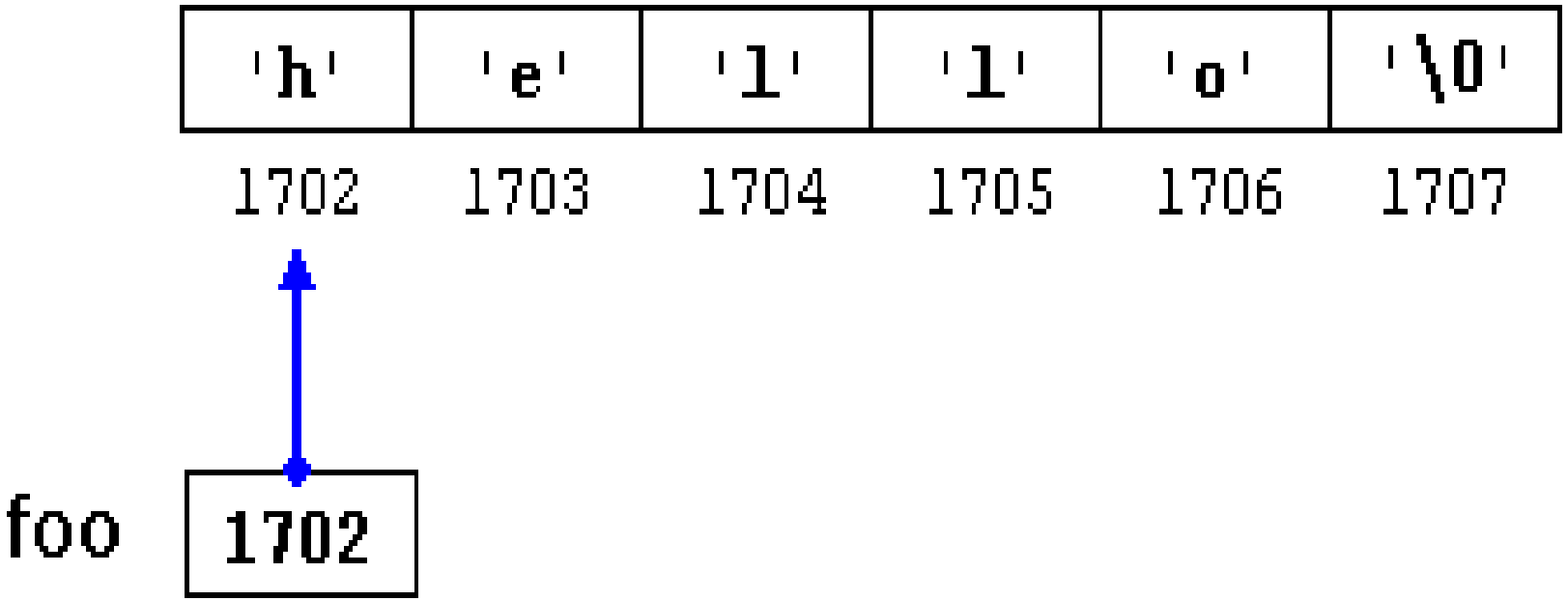
# Rule of Virtual Function

- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for function call.
- They are mainly used to achieve Runtime polymorphism
- The resolving of function call is done at Run-time.
- They Must be declared in public section of class.

# Pointer

As you know every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator which denotes an address in memory. The following statement can declare pointer variable :

- int *p;
- san *p;


'h'
'e'
'l'
'l'
'o'
'\0'
1702
1703
1704
1705
1706
1707
foo
1702

# Pure Virtual Function

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration. See the following example.

```
// An abstract class
class Test
{

public:
   // Pure Virtual Function
   virtual void show() = 0;

};
```

```
class Base
{
   int x;
public:
    virtual void fun() = 0;
    int getX() { return x; }
};

class Derived: public Base
{
    int y;
public:
    void fun() { cout << “hi”; }
};
```

# Virtual Class

**Virtual** base **classes**, used in **virtual** inheritance, is a way of preventing multiple "instances" of a given **class** appearing in an inheritance hierarchy when using multiple inheritance.

## Solution of Ambiguity in Hybrid Inheritance

```
#include<iostream.h>
#include<conio.h>
class ClassA
{ public: int a; };
class ClassB : virtual public ClassA
{ public: int b; };
class ClassC : virtual public ClassA
{ public: int c; };
class ClassD : public ClassB, public ClassC
{ public: int d; };
void main()
{ ClassD obj;
obj.a = 10; //Statement Error occur //
obj.a = 100; //Statement Error occur
    }
```

# Operators Overloading

- You can redefine or overload most of the built-in operators available in C++. Thus, a programmer can use operators with user-defined types as well.
- Overloaded operators are functions with special names the keyword operator followed by the symbol for the operator being defined. Like any other function, an overloaded operator has a return type and a parameter list.

# Syntax

class name  operator operator-symbol (parameters);

# Following is the list of operators which can be overloaded

| | | | | | |
|---|---|---|---|---|---|
| + | - | * | / | % | ^ |
| & | \| | ~ | ! | , | = |
| < | > | <= | >= | ++ | -- |
| << | >> | == | != | && | \|\| |
| += | -= | /= | %= | ^= | &= |
| \|= | *= | <<= | >>= | [] | () |
| -> | ->* | new | new [] | delete | delete [] |

# Operators cannot be Overloaded

- ::
- .*
- .?
- :
- .

# Two ways to write function body

Inside Class

- Create only one object and pass as argument subject to operator which have to overload

Outside Class

- Create two objects and pass as argument subject to operator which have to overload

# Inside Class

```
class san
{
     int a;
    public:
      in()
    { cin>>a; }
 san  operator + (san s1);
   {
            a=a-s1.a;
}
out()
{ cout<<a;}
}
};
main()
{
      san s1,s2,s3;
      s1.in();
      s2.in();
      s3=s1+s2;
      s3.out();
}
```

# Outside Class

```
class san
{
     int a;
    public:
      in()
    { cin>>a; }
 san  operator + (san s1);
out()
{ cout<<a;}
}

san  san  :: operator + (san  s1);
  {
          san s2;
           s2. a=a-s1.a;
}

};
main()
{
      san s1,s2,s3;
      s1.in();
      s2.in();
      s3=s1+s2;
      s3.out();
}
```