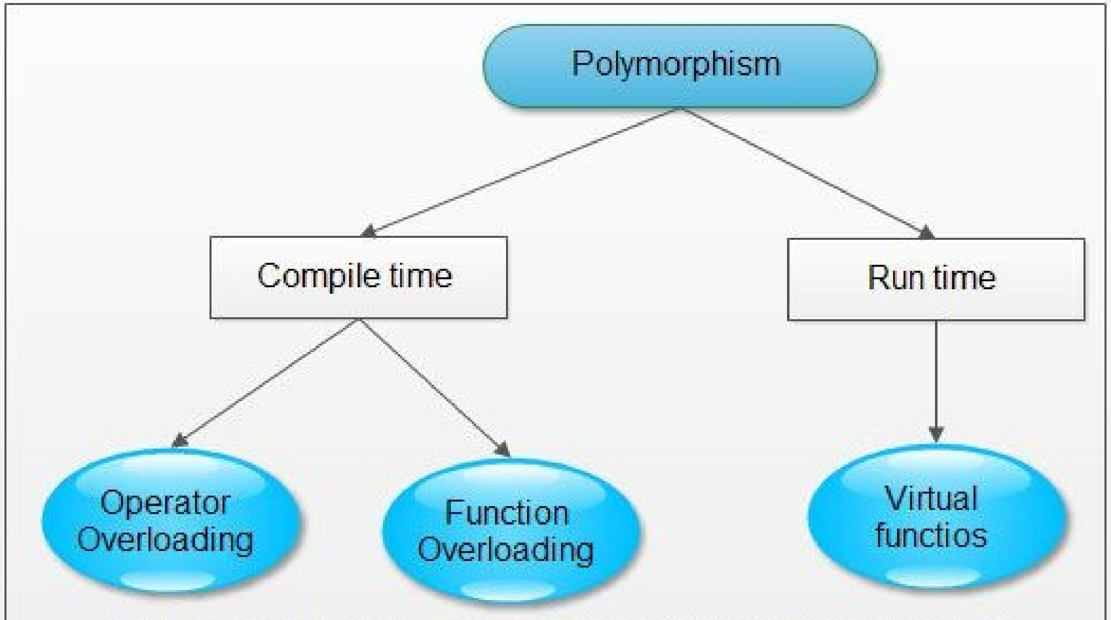
Run-time Polymorphism

Agenda

- Introduction
- Pointers to Derived Classes
- Introduction to Virtual Functions
- More about Virtual Functions

Introduction

- Polymorphism
  - Poly-many, Morphism- forms,
  - Single interface - multiple forms
- Polymorphism in C++
- Compile-time polymorphism:
  - The polymorphism can be acomplished (Implementation of an action) at compile time.
  - Function Overloading
  - Operator Overloading
- Run-time polymorphism:
  - The polymorphism cannot be acomplished at compile time, it must be doen at run time (Implementation of an action).
  - Virtual Functions (Kind of an abstraction)



Different Ways of implementing Polymorphism

Pointers to Derived Classes

- A pointer to a base class can point to any class derived from that base class.
- base *b; // base class pointer
- derived * d; // derived class pointer
- base b_ob ; // object of base class
- derived d_ob ; // object of derived class
- b = & b_ob // base class pointer can always point to base class object
- b = & d_ob ; // b can also point to derived object
- d = & d_ob // ok
- d = & o_ob // Not ok ..can not access base class object

Pointers to Derived Classes Cont..

- A base class pointer can access only those members of the derived class object that were inherited from the base.

- A base class pointer has the knowledge about base class and not about the derived class members that belong to derived class object.

- A pointer of the derived class object is not permitted to access an object of the base class.

Example

```
s base{
;
c:
seta (int i) { a = i; }
eta () { return a; }

s derived: public base

;
c :
setb (int i) { b = i; }
etb () { return y; }
```

```
int main () {
 // pointer to base class
base *bp;
// object of base
base b_ob ;
// object of derived class
derived d_ob ; bp = & b_ob ;
// access base object
bp-> seta (10) ;
cout << " Base object a: "
<< bp-> getb() << '\n';
```

```
 // point to derived class object
bp = & d_ob ;
// access derived object
bp-> seta (20) ;
// can we do bp->setb(30) ??
d_ob.setb (30) ;
cout << " Derived object a: "
<< p-> geta () << '\n';
cout << " Derived object b: "
<< d_ob.getb () << '\n';
return 0;
}
```

Output:
Base object a: 10
Derived object a: 20
Derived object b: 30

Introduction to a Virtual Function

- A virtual function is a member function that is declared within a base class and redefined by a derived class.

- When we inherit a class containing a virtual function, the derived class redefines the virtual function to fit its own purpose.

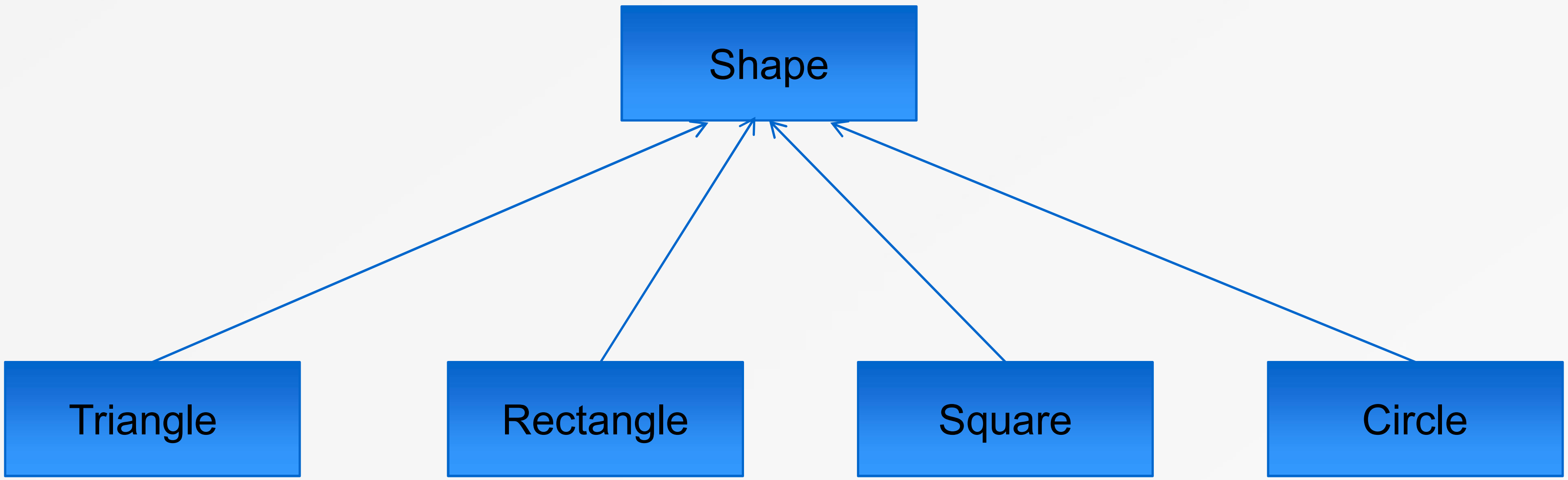
- Virtual functions implement the "one interface, multiple methods" that underlies polymorphism.

- The virtual function that is defined within the base class forms the interface to that function.

- Each redefinition of the virtual function by a derived class implements its operation as it relates specifically to the derived class.

Introduction to a Virtual Function

- Assume that there is a class named Shape.
- We have multiple types of shapes



- Draw and area functions to be common in these classes.
- Can we have a common function in the Shape class and redefine it in all the derived classes, and let an appropriate function to be called at run time?

Virtual Functions

- A member function of a class.
- Declared with “virtual” keyword.
- Usually has a different functionality in the derived class.
- Each redefinition by a derived class implements its operation as it relate specifically to the derived class.
- When redefined by a derived class, the keyword virtual is not needed.
- A function call is resolved at run-time.

Example

```
s base {
c : int i;
 (int x) { i = x; }
al void func () {
<< " Using base version of func ():

<< i << '\n'; }

s derived1 : public base{
c :
ed1 (int x) : base (x) { }
func (){
<< " Using derived1 's version of
c (): ";
<< i*i << '\n';
```

```
class derived2 : public base
{
public :
derived2 (int x) : base (x) {}
void func ()
{
cout << " Using derived2 's
 version
of func (): ";
cout << i+i << '\n';}
};
```

```
int main ()
{
base *p;
base ob (10) ;
derived1 d_ob1 (10) ;
derived2 d_ob2 (10) ;
p = &ob;
p-> func (); // use base 's func
p = & d_ob1 ;
p-> func (); // use derived1 's f
p = & d_ob2 ;
p-> func (); // use derived2 's f
return 0;
}
```

```
Using base version of func (): 10
Using derived1 's version of func (): 100
Using derived2 's version of func (): 20
```

Function Overloading vs Function Overriding

- Not same as that of function overloading- a number and type of arguments vary.
- The term **overriding** is used to describe virtual function redefinition by a derived class
- When a virtual function is redefined, all aspects of its prototype must be the same.
- If we change the prototype when we attempt to redefine a virtual function, the function will be considered overloaded by the C++ compiler, and its virtual nature will be lost.

Virtual Functions Under the Hood

- Internally, C++ compiler implements virtual functions using Virtual Table (vtable).

- Vtable holds the addresses of virtual functions defined within that class.

- Vtable is maintained per class, i.e., all objects of the class share the vtable.

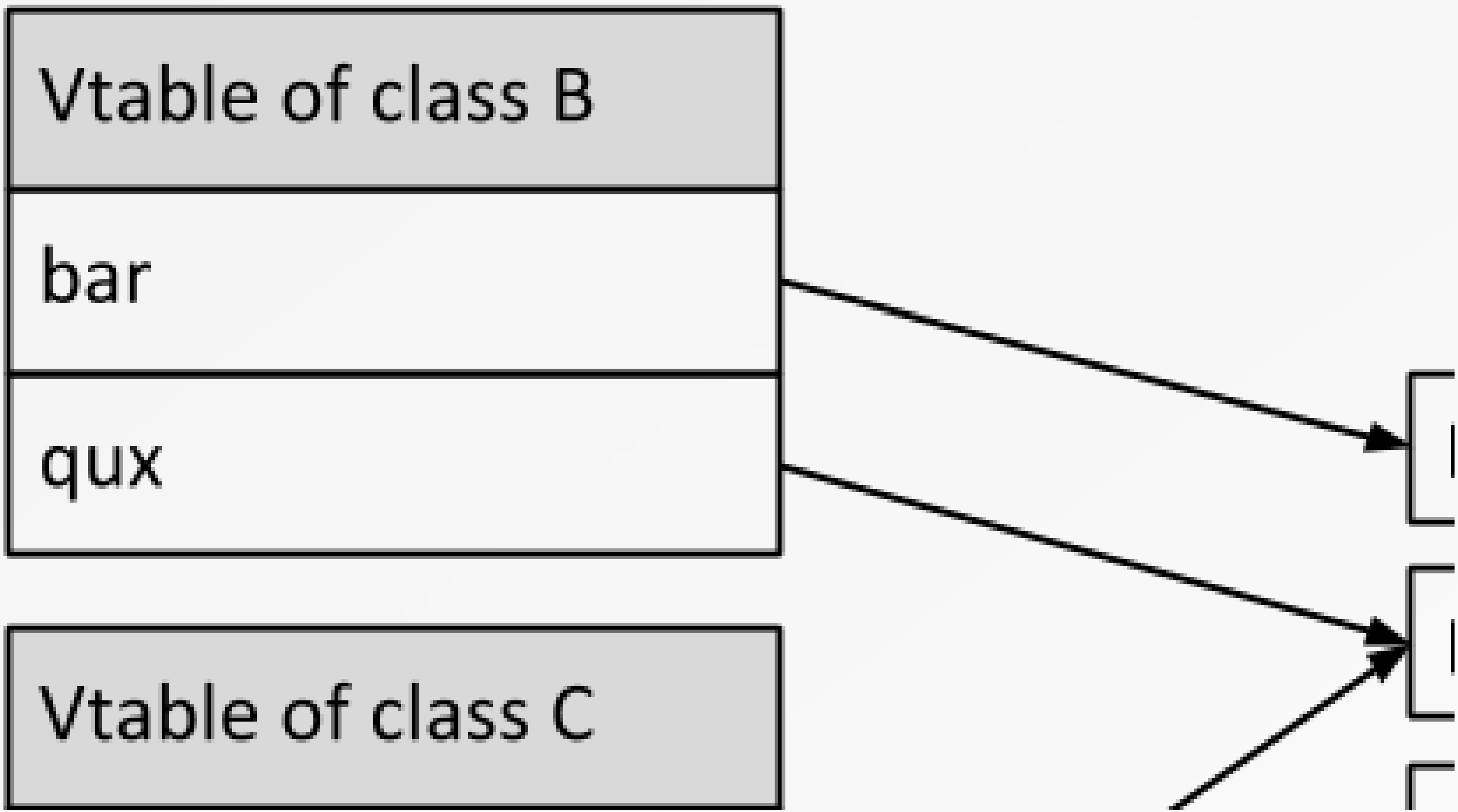
- The address of vtable is stored in virtual pointer (vptr) and is kept silently in an object.

Example

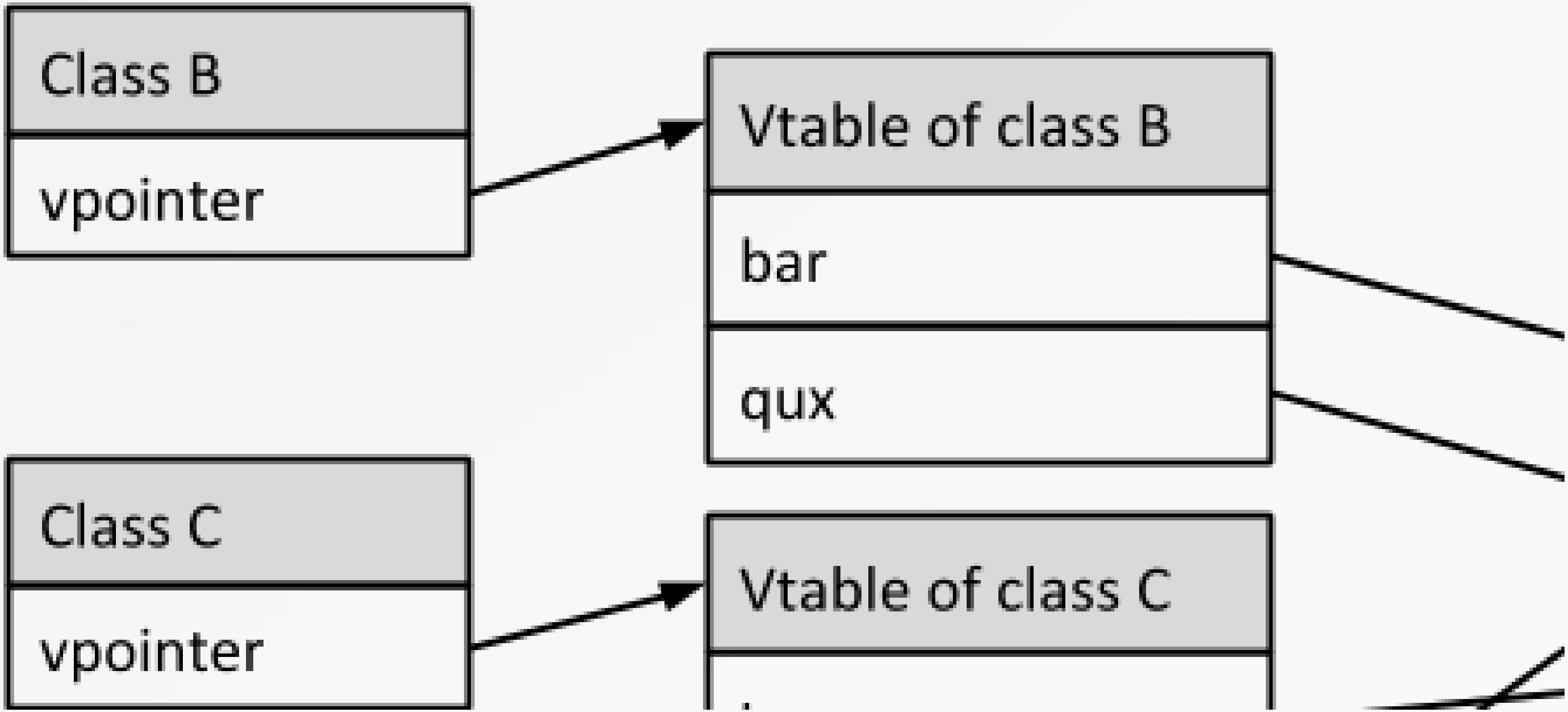
```
#include <iostream>
class B {
public:
  virtual void bar();
  virtual void qux();
};

void B::bar()
{
  std::cout << "This is B's implementation of
bar" << std::endl;
}

void B::qux() {
  std::cout << "This is B's implementation of
qux" << std::endl;
}
```

```
class C : public B
{
public:
  void bar() {
// Some def.
}
};

void C::bar()
{
  std::cout << "This is C's implementation of
bar" << std::endl;
}
main(){
B* b = new C();
b->bar();
}
```

Vtable



vpointer


Class B
vpointer
Vtable of class B
bar
qux
Class C
vpointer
Vtable of class C

Restrictions to Virtual Functions

- Virtual functions must be nonstatic members of the classes of which they are part.

- They cannot be friends.

- Constructor functions cannot be virtual, but we can have virtual destructor functions.

Virtual Destructors

```
class Base
{
   // virtual methods
};

class Derived : public Base
{
   ~Derived()
   {
      // Cleanup out here
   }
};
```

```
Base *b = new Derived();
// Make use of b
delete b; // Let us destroy derived class object
```

// Problem: the derived class object is not destroyed, but the base class pointer is deleted resulting in a memory leakage

// Can not have virtual constructors,

if we have them, then we would create derived class object first followed by base class object which is not logical

Practical Example on Virtual Functions

- Can we have a base class that holds the virtual function that one doesn't have to be defined?
- Have derived classes re-define the base class function in them.
- Let an appropriate function be called at run time.
- Example: Various geometric shapes

Example

```
class area
{
// dimensions of figure
double dim1 , dim2 ;
public :
void setarea ( double d1 , double d2)
void getdim ( double &d1 , double &d2)
virtual double getarea (){
// Override this function
return 0.0;
}
};
```

```
class rectangle : public area{
public :
double getarea (){
// define its area here } };

class triangle: public area{
public :
double getarea (){
// define its area here }};
 int main() {
 area *p;
 rectangle r;
 triangle t;
 r. setarea (3.3 , 4.5) ; t. setarea (3.3 , 4.5) ;
 p=&t;  p->getarea();
 p=&r;  p->getarea();
 }
```

Pure Virtual Functions

- If there is no meaning for a base class virtual function to perform, then any derived class must override this function.
- C++ supports pure virtual functions to ensure that this will occur.
- A pure virtual function has no definition relative to the base class.
- We need to include the function's prototype only.
- Syntax
  - virtual type func_name (parameter_list)=0;

Pure Virtual Functions Cont..

- It is an indication to the compiler that there would not be body for this function relative to the base class.
- Pure virtual function forces any derived class to override it
- A compile-time error results in case a derived class does not override this function.
- Redifinition of this function is ensured.

Abstract Class

- A class containing at least one pure virtual function is called as an abstract class.

- Abstract class is said to be technically incomplete as it does include at least one function which does not have any body.

- Therefore, objects of an abstract class can not be created.

- Base class pointers will still be created, in fact, they must be created in order to accomplish run-time polymorphism.

Example

```
class area
{
// dimensions of figure
double dim1, dim2 ;
public :
void setarea ( double d1 , double d2);
void getdim ( double &d1 , double &d2);
virtual double getarea () =0;
// Must override this function
}
};
```

```
class rectangle : public area{
public :
double getarea (){
// define its area here } };

class triangle: public area{
public :
double getarea (){
// define its area here }};
 int main() {
 area *p;
 rectangle r;
 triangle t;
 r. setarea (3.3 , 4.5) ; t. setarea (3.3 , 4.5) ;
 p=&t;  p->getarea();
 p=&r;  p->getarea();
 }
```

Early Binding

- Those events that can be known at compile time.
- Function calls that can be resolved during compilation.
- Examples:
  - Normal functions, overloaded functions, and non-virtual member and friend functions.
- The address information required to call them is known at compile time.
- Very efficient, very fast in terms of time.
- No flexibility as everything happens at compile time.

Late Binding

- Events that occur at run time.
- The address of the function to be called is not known until the program runs.
- In C++, virtual function is an object that is bound late, i.e. at run time.
- The program will have to determine at run time what type of object is being pointed to by base class pointer and then choose an appropriate overridden function to execute.
- Flexibility- at run time any random event can be responded
- More overhead associated with a function call.
- Calls will be slower.

Virtual Function through a Base Class Reference

- A virtual function is also available when called through a base-class reference.

- A base-class reference can be used to refer to an object of the base class or any object derived from that base.

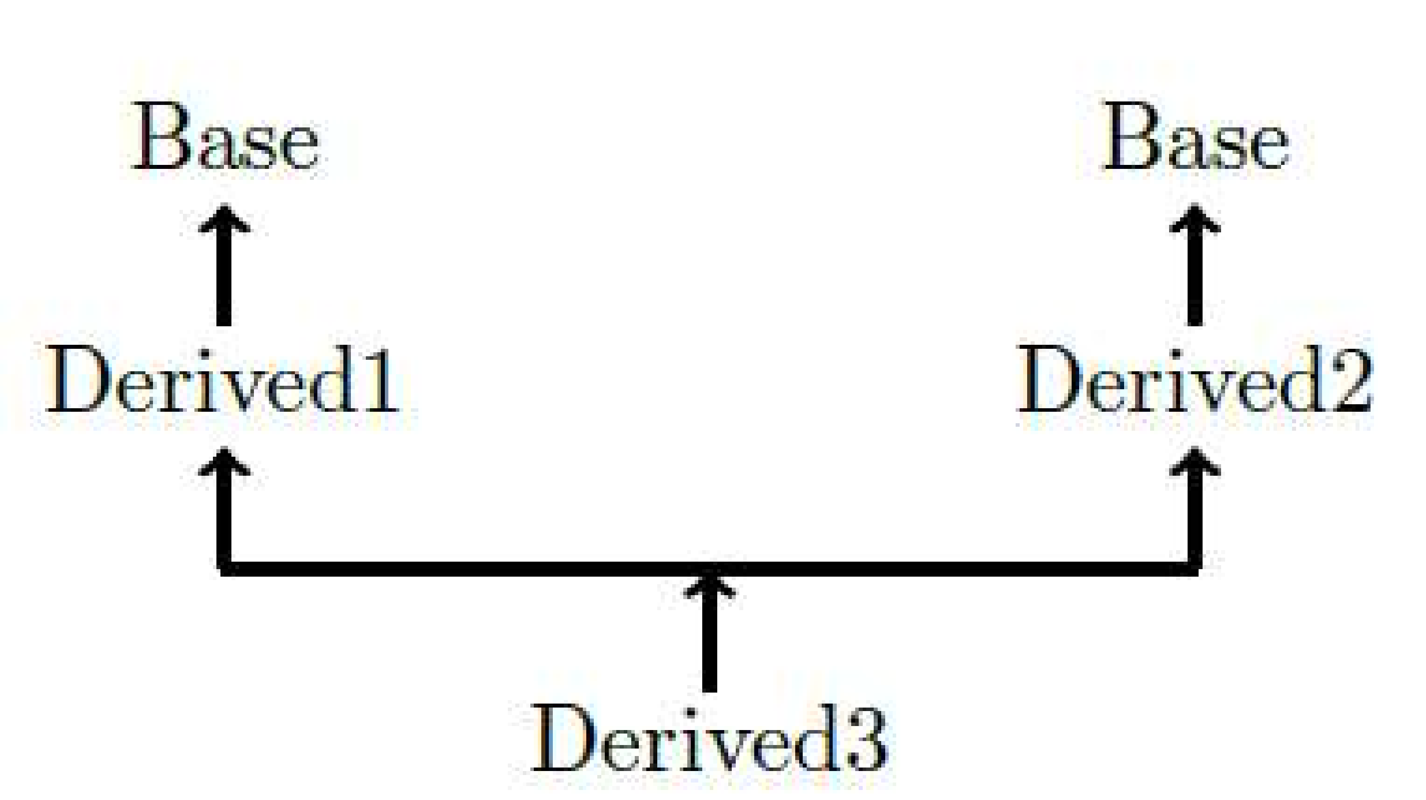
- When called through a base-class reference, the version of the function executed is determined by the object being referred to at the time of the call.

Example References

```
 Here, a base class reference is used to
ccess a virtual function. */
include <iostream>
sing namespace std;
ass base {
ublic:
rtual void vfunc() {
out << "This is base's vfunc().\n"; }

ass derived1 : public base {
ublic:
oid vfunc() {
out << "This is derived1's vfunc().\n"; }
```

```
class derived2 : public base {
public:
void vfunc() {
cout << "This is derived2's vfunc().\n"; }
};
// Use a base class reference parameter.
void f(base &r) { r.vfunc(); }
int main() {
base b;
derived1 d1;
derived2 d2;
f(b); // pass a base object to f()
f(d1); // pass a derived1 object to f()
f(d2); // pass a derived2 object to f()
return 0;
}
```

Virtual Base Class



References

- C++: The Complete Reference, 4th Edition by Herbert Schildt , McGraw-Hill
- Teach Yourself C++ 3rd Edition by Herbert Schildt,
- The C+ + Programming Language, Third Edition by Bjarne Stroustrup, Addison Wesley