Issues to be discussed in the context of compilation of virtual functions

- Illustrative examples to bring out the concept of virtual functions and their use in design of classes
- How does user of code know which version of a virtual function f() on an object o, to use given the form of a call such as : o.f() ?
- Same problem as above, except that in this case the call is of the form o->f(), where o-> would be known only at run time.

Illustration of use of Virtual Functions in C++

Consider the following C++ program.

```
class Base
{ public:
   int b;
   Base (int i) { b = i;};
   int f1(int a) {return b+a;};
   int f2(int i, int j) {return i*b+j;};
};

class D1: public Base { public: int d1;
   D1( int i, int j) : Base(i) , d1(j) { };
   virtual int f1() { return d1+b;};
};

class D2: public D1 { public: int d2;
   D2( int i, int j, int k) : D1(i, j), d2(k) { };
   virtual int f1(int k) {return d2+k;};
};
```

If the following main() is used to employ the class design given above, then the result of execution of the code is as given alongwith.

```
int main()
{ int s1, s2, s3, k, res;
  Base b(1);
  D1 dd1(5,1), dd12(10, 1);
  D2 dd2(10,6,1), dd22(5,2,1);
  Base * aa[3];
  aa[0] = &dd1; aa[1] = &dd2; aa[2] = &b;
  for (k = 0; k < 3; k++)
    {res = aa[k]->f1(k);
      cout << "res = " << res << endl;}
}
```

Output from the Porgram :

```
res = 5
res = 11
res = 3
```

However, if the class design is changed to as shown below, the output of the program also changes as given along with.

```
class Base
{ public:
   int b;
   Base (int i) { b = i;};
   virtual int f1(int a) {return b+a;};
   int f2(int i, int j) {return i*b+j;};
};
```

```
class D1: public Base { public: int d1;
   D1( int i, int j) : Base(i) , d1(j) { };
   int f1() { return d1+b;};
};

class D2: public D1 { public: int d2;
   D2( int i, int j, int k) : D1(i, j), d2(k) { };
   int f1(int k) {return d2+k;};
};
```

```
int main()
{ int s1, s2, s3, k, res;
  Base b(1); D1 dd1(5,1), dd12(10, 1);
  D2 dd2(10,6,1), dd22(5,2,1);
  Base * aa[3];
  aa[0] = &dd1; aa[1] = &dd2; aa[2] = &b;
  for (k = 0; k < 3; k++) {res = aa[k]->f1(k); cout
<< "res = " << res << endl;}
}
```

Output from the program :

```
res = 5
res = 2
res = 3
```

Again if we keep the class design the same as above, but change the main program as shown below, the compiler throws up semantic error as reproduced below.

```
int main()
{ int s1, s2, s3, k, res;
  Base b(1); D1 dd1(5,1), dd12(10, 1);
  D2 dd2(10,6,1), dd22(5,2,1);
  D1 * aa[3];
  aa[0] = &dd1; aa[1] = &dd2; aa[2] = &b;
  for (k = 0; k < 3; k++) {res = aa[k]->f1(k);
  cout << "res = " << res << endl;}
}
```

Compiler message :

In function ‘int main()’:

tut9_3.C:29:40: error: invalid conversion from ‘Base*’ to ‘D1*’

tut9_3.C:30:45: error: no matching function for call to ‘D1::f1(int&)’

tut9_3.C:14:8: note: candidate is: int D1::f1()

It is important as a revision exercise to attempt all the variations given above and get the answers right. The central question of this topic is the unravelling of the mystery by which the compiler is able to resolve the calls to virtual functions as the language defines it to be.

To implement virtual functions, C++ uses a special form of late binding known as the virtual table. The virtual table is a lookup table of functions used to resolve function calls in a dynamic/late binding manner. The virtual table sometimes goes by other names, such as “vtable”, “virtual function table”, “virtual method table”, or “dispatch table”.

Because knowing how the virtual table works is not necessary to use virtual functions, this description is useful for implementors rather than programmers.

The virtual table is an elegant idea and the details are decsribed in the following. i

- First, every class that uses virtual functions (or is derived from a class that uses virtual functions) is given it’s own virtual table. This table is simply a static array that the compiler sets up at compile time. A virtual table contains one entry for each virtual function that can be called by objects of the class. Each entry in this table is simply a function pointer that points to the most-derived function accessible by that class.
- Second, the compiler also adds a hidden pointer to the base class, which we will call *__vptr. *__vptr is set (automatically) when a class instance is created so that it points to the virtual table for that class. Unlike the *this pointer, which is actually a function parameter used by the compiler to resolve self-references, *__vptr is a real pointer. Consequently, it makes each class object allocated bigger by the size of one pointer. It also means that *__vptr is inherited by derived classes, which is important.

```
class Base
{
public:
virtual void function1() {};
virtual void function2() {};
};

class D1: public Base
{
public:
virtual void function1() {};
};

class D2: public Base
{
public:
virtual void function2() {};
};
```

Because there are 3 classes here, the compiler will set up 3 virtual tables: one for Base, one for D1, and one for D2.
The compiler also adds a hidden pointer to the most base class that uses virtual functions. Although the compiler does this automatically, we’ll put it in the next example just to show where it’s added:

```
class Base
{ public:
 FunctionPointer *__vptr;
 virtual void function1() {};
 virtual void function2() {};
};

class D1: public Base
{ public:
 virtual void function1() {};
};

class D2: public Base
{ public:
 virtual void function2() {};
};
```

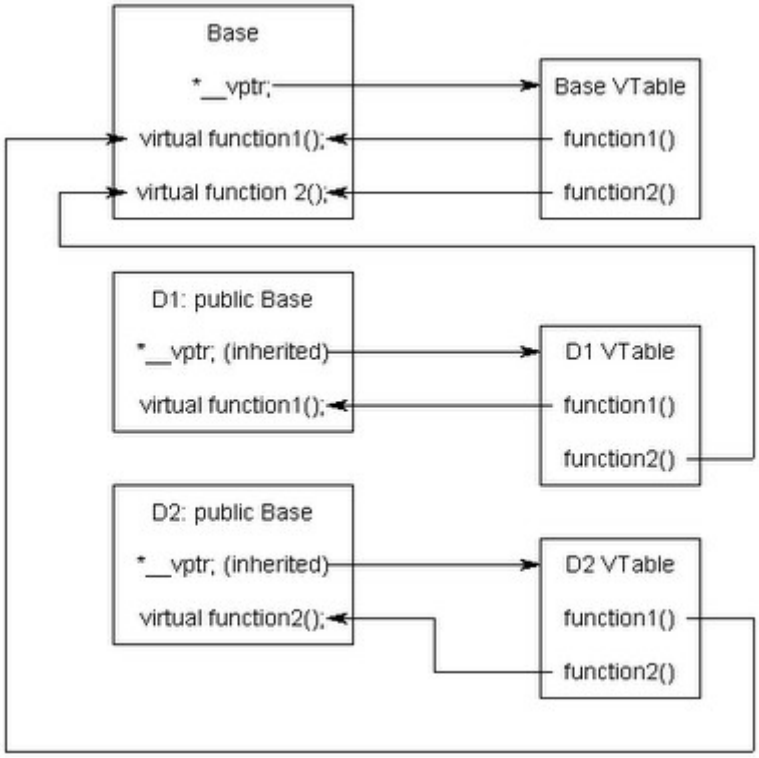
When a class object is created, *__vptr is set to point to the virtual table for that class. For example, when a object of type Base is created, *__vptr is set to point to the virtual table for Base. When objects of type D1 or D2 are constructed, *__vptr is set to point to the virtual table for D1 or D2 respectively.

The critical issue is to determine how these virtual tables are to be populated. Because there are only two virtual functions here, each virtual table will have two entries (one for function1(), and one for function2()). Remember that when these virtual tables are filled out, each entry is filled out with the most-derived function an object of that class type can call.

Base's virtual table is simple. An object of type Base can only access the members of Base. Base has no access to D1 or D2 functions. Consequently, the entry for function1 points to Base::function1(), and the entry for function2 points to Base::function2().

D1's virtual table is slightly more complex. An object of type D1 can access members of both D1 and Base. However, D1 has overridden function1(), making D1::function1() more derived than Base::function1(). Consequently, the entry for function1 points to D1::function1(). D1 hasn't overridden function2(), so the entry for function2 will continue to point to Base::function2().

D2's virtual table is similar to D1, except the entry for function1 points to Base::function1(), and the entry for function2 points to D2::function2().
Here's a picture of this graphically:



Although this diagram is kind of sphagetti like, it's really quite simple: the *__vptr in each class points to the virtual table for that class. The entries in the virtual table point to the most-derived version of the function objects of that class are allowed to call.

So consider what happens when we create an object of type D1:

```
int main()
{  D1 d1;   }
```

Because d1 is a D1 object, d1 has it's *__vptr set to the D1 virtual table.

Now, let's set a base pointer to D1:

```
int main()
{  D1 d1;
   Base *p = &d1;
}
```

Note that because p is a base pointer, it only points to the Base portion of d1. However, also note that *__vptr is in the Base portion of the class, so p has access to this pointer. It therefore implies that that p->__vptr points to the virtual table of d1! Consequently, even though p is of type Base, it still has access to D1's virtual table.

Let us now try to resolve the call p->function1()?

```
int main()
{
D1 d1;
Base *p = &d1;
p->function1();
}
```

First, the program recognizes that function1() is a virtual function (How ?). Second, uses p->__vptr to get to D1's virtual table. Third, it looks up which version of function1() to call in D1's virtual table. This has been set to D1::function1() (Why ?) . Therefore, p->function1() resolves to D1::function1()!

What if p really pointed to a Base object instead of a D1 object. Would it still call Base::function1() or D1::function1() and how will it resolve the call using this implementation echeme?

```
int main()
{
Base b;
Base *p = &b;
p->function1();
}
```

In this case, when b is created, __vptr points to Base's virtual table, not D1's virtual table. Consequently, p->__vptr will also be pointing to Base's virtual table. Base's virtual table entry for function1() points to Base::function1(). Thus, pClass->function1() resolves to Base::function1(), which is the most-derived version of function1() that a Base object should be able to call.

By using these tables, the compiler and program are able to ensure function calls resolve to the appropriate virtual function, even if an object (instance of base or derived class) is accessed using a pointer or reference to its base class.

Calling a virtual function is slower than calling a non-virtual function in view of the implementation outlined above. It involves the following actions by a compiler.

1. Dereferences the vptr attached to the object to get to the appropriate virtual table.
2. Adds the offest into this virtual table to find the location of the correct function to call.
3. Dereferences the entry to extract the pointer of the the virtual function to be called and then calls it.

As a result, the compiler generated code performs 3 operations to resolve a call to a virtual function, as compared to 2 operations for a normal indirect function call, or one operation for a direct function call.