C++ Object Model

Inside the C++

Mikael Glaad 021-418713@telia.com and Concha Batanero cbatanero@yahoo.es

22nd May 2005

Abstract

This report will focuses on some of the underlying mechanisms that support object-oriented programming within C++: constructor semantics, support for encapsulation, inheritance, and "the virtuals"–virtual functions and virtual inheritance. This report shows one general idea about how the information is shared in memory and what will happen inside C++ when our aplication is running. That is particularly interesting to understand some of the underlying implementation models which can help the programmer to code in a more efficient way.

Contents

1 Introduction

What Is the C++ Object Model?

There are two aspects to the C++ Object Model:

1. The direct support for object-oriented programming provided within the language.
2. The underlying mechanisms by which this support is implemented.

The second aspect is barely touched on in any current text. Our project will mostly treat this second aspect of the C++ Object Model.

The first aspect of the C++ Object Model is invariant. For example, under C++ the complete set of virtual functions available to a class is fixed at compile time; the programmer cannot add to or replace a member of that set dynamically at runtime. This allows for extremely fast dispatch of a virtual invocation, although at the cost of runtime flexibility.

The underlying mechanisms by which to implement the Object Model are not prescribed by the language, although the semantics of the Object Model itself make some implementations more natural than others. Virtual function calls, for example, are generally resolved through an indexing into a table holding the address of the virtual functions. This table has fixed size and is constructed prior to program execution.

Determining when to provide a copy constructor, and when not, is not something one should guess at or have adjudicated by some language. It should come from an understanding of the Object Model.

We are going to talk about the next points:

1.- Objects, A brief review about object-based and object-oriented programming paradigms supported by C++. It includes a brief tour of the Object Model.

2.- The Semantics of Constructors, we discusse how constructors works and how to increase the performance of a program. We will also talk about what happens to a constructor under inheritance.

3.- The Semantics of Data, talking about the handling of data members.

4.- The Semantics of Function, focuses on the varieties of member functions.

2 Objects

2.1 The C++ Object Model

In C++, there are two types of class data members, static and nonstatic, and three types of class member functions, static, nonstatic and virtual. Given the following declaration of a class Point:

```
Class Point {
public:
        Point( float xval);
        virtual ~Point();
        float x() const;
```

```
        static int PointCount();
Protected:
        virtual ostream&
           print( ostream &os) const;
        float _x;
        static int _point_count;
};
```

We will see how the class Point is represented within the machine:

Nonstatic data members are allocated directly within each class object. Static data members are stored outside the individual class object. Static and nonstatic function members are also hoisted outside the class object. Virtual functions are supported in two steps:

1 - A table of pointers to virtual functions is generated for each class(this is called the virtual table).

2 - A single pointer to the associated virtual table is inserted within each class object The type_info object associated with each class in support of runtime type identification is also addresses within the virtual table, usually within the table's first slot. The Standar allows the compiler the freedom to insert the vptr and the member data in whatever order.

The figure1 shows the general C++ Object Model for our Point class. The primary strength of the C++ Object Model is its space and runtime efficiency. Its primary drawback is the need to recompile unmodified code that makes use of an object of a class for which there has been an addition, removal or modification of the nonstatic class data members.

2.2 A Keyword Distinction

C++ is more complicated because have to maintain language compatibility with C. For example, overloaded function resolution would be a lot simpler if there were not eight types of integer to support. Or for example the concept of a class could be supported by a single class keyword, if C++ were not required to support existing C code and, with that, the keyword struct. If a programmer absolutely needs to use C into a C++ program, the only portable method of combining C and C++ is by composition rather inheritance:

```
struct C_point .....
class Point {
public:
        operator C_point() {return _c_point; }
        // ......
private:
        C_point _c_point;
        // ..........
};
```

One reasonable use of the C struct in C++ is when you want to pass all or part of a complex class object to a C function.

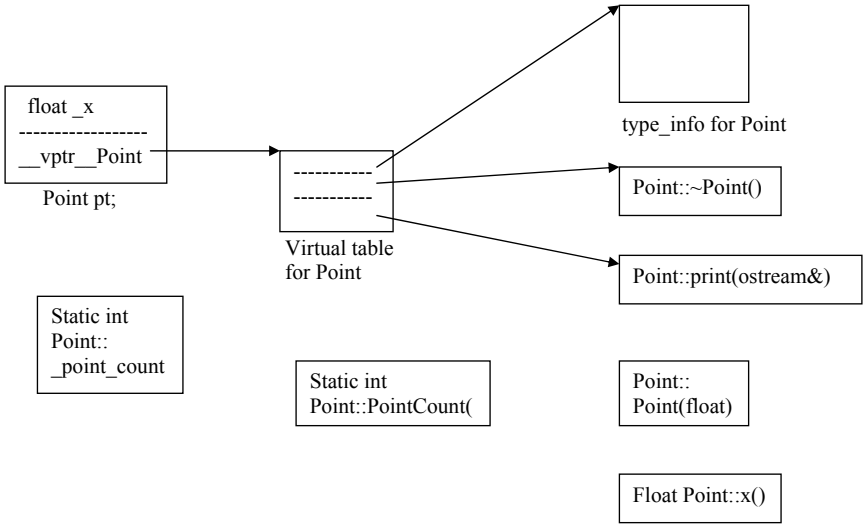


Figure 1: General C++ Object Model for Point class

2.3 An Object Distinction

The C++ programming model directly supports three programming paradigms:

1.- The procedural model as programmed in C, that it's supported within C++.

2.- The abstract data type (ADT) model, in which users of the abstraction are provided with a set of operations (the public interface), while the implementation remains hidden.

3.- The object oriented (OO) model, in which, a collection of related types are encapsulated through an abstract base class providing a common interface.

The memory requirements to represent a class object in general are the following:

* The accumulated size of its nonstatic data members.

* Plus any padding (between members or on the aggregate boundary itself) due to alignment constraints (or simple efficiency)

* Plus any internally generated overhead to support the virtuals.

The memory requirement to represent a pointer, is a fixed size regardless of the type it addresses. For example, given the following declaration of a ZooAnimal class:

```
Class ZooAnimal {
Public:
        ZooAnimal ();
        virtual ~ZooAnimal ();
        // .............
        virtual void rotate();
protected:
        int loc;
        String name;
};
ZooAnimal za( "Zoey");
ZooAnimal *pza = &za;
```

A likely layout of the class object za and the pointer is pictured in the figure 2:

2.4 Type of one Pointer

There are not differences between pointers which point to different types of data. The difference lies in the type of object being addressed. Given the pointers:

```
zooAnimal *px;
int *pi;
```

An integer pointer addressing memory location 1000 on a 32-bit machine spans the address space 1000-1003. The ZooAnimal pointer, if we presume a conventional 8-byte String (a 4-byte character pointer and an integer to hold the string length), spans the address space 1000-1015. Overhead + 8 + 4 = 16

2.5 Polymorphism

If we define now a Bear as a kind of ZooAnimal. This is done through public inheritance:

```
class Bear : public ZooAnimal {
Public:
        Bear();
        ~Bear();
        // ...
        void rotate();
        virtual void dance();
        // ...
protected:
        enum Dances  ... ;
        Dances dances_know;
        int cell_block;
```

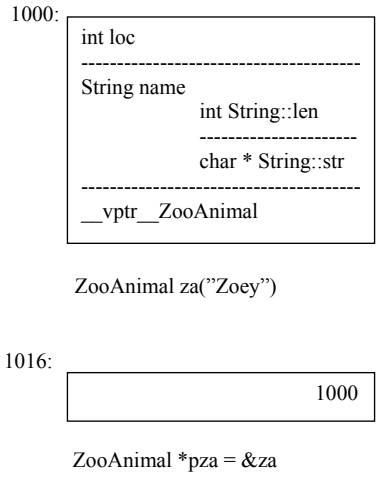


Figure 2: Layout of Object and Pointer of Independent Class

};
Bear b("Yogi");
Bear *pb = &b;
Bear &rb = *pb;

The Bear and ZooAnimal pointer addresses the same first byte of the Bear object, the difference is that the address span of pb encompasses the entire Bear object, while the span of pz encompasses only the ZooAnimal subobject of Bear. If we have the next sentences:

Bear b;
ZooAnimal za = b;
// ZooAnimal::rotate() invoked
za.rotate();

The encapsulation of the type information is maintained not in za but in the link be-

tween the object's vptr and the virtual table the vptr addresses. The instance of rotate() invoked is the ZooAnimal instance. The compiler intercedes in the initialization and assignment of the vptr. To summarize, polymorphism is a powerful design mechanism that allows for the encapsulation of related types behind an abstract public interface, such as our Library_materials hierarchy. The cost is an additional level of indirection. C++ supports polymorphism through class pointers and references. This style of programming is called object-oriented.

3 The semantics of constructors

3.1

There are four characteristics of a class under which the compiler needs to synthesize a default constructor for classes that declares no constructor at all.

The Standard refers to these as implicit, nontrival default constructors. Synthesized constructor fulfils only an implementation need. It does this by invoking member object or base class default constructor or initializing the virtual function or virtual base class mechanism for each object. Classes that do not exhibit these characteristics and that declare no constructors at all are said to have implicit, trival default constructor. In practice, these trival default constructors are not synthesized.

Within the synthesized default constructor, only the base class subobjects and member class objects are initialized. All other nonstatic data members, such as integers, pointers, and so on, are not initialized. These initializations are needs of the program, not of the implementation. If there is a program need for default constructor, such as initializing a pointer to 0, it is the programmer's responsibility to provide it in the course of the class implementation.

Programmers new to c++ often have two common misunderstandings:

1. That a default constructor is synthesized for every class does not define one.

2. That the compiler-synthesized default constructor provides explicit default initializes for each data member declared within the class.

How is a constructor implemented in the most efficient way?

When writing a constructor, you have the option of initializing class member either through the member initialization list or within the body of the constructor.

In following four cases must member initialization list be used in order to compile the program.

1. When initializing a reference member.

2. When initializing a const member

3. When invoking a base or member class constructor with a set of arguments.

In the fourth case, the program complies and executes correctly. But it does so inefficiently. For example, given

```
Class Word {
        String _name;
```

```
        Int _cnt;
public:
        //not wrong just naive
        Word(){
                _name = 0;
                _cnt = 0;
        }
};
```

This implementation of the Word constructor initializes _name once, then overrides the initialization with the assignment, resulting in the creation and the destruction of a temporary String object. Was this intentional? Unlikely. (does the compiler generate a warning)

Here is the likely internal argumentation of this constructor:

```
//Pseudo C++ Code
Word::Word(/*this pointer goes here*/)
{
        //Invoke default String constructor
        _name.String::String();
        //generate temporary
        String temp = String (0);
        // memberwise copy_name
        _name.String::operator=(temp);
        //destroy temporary
        temp.String::~String();
        _cnt = 0;
}
```

A significantly more efficient implementation would have been:

```
//Preferred implementation
Word::Word : _name(0)
{
        _cnt = 0;
}
```

This expands to something like this:

```
Word::Word(/* this pointer goes here */)
{
        //invoke String(int) constructor
        _name.String::String(0);
        _cnt = 0;
}
```

3.2 Member initializations list

What happens in the constructor when an object is created which inherit? When we define an object, such as

T object;

Exactly what happens?

If there is a constructor associated with T(either user supplied or synthesized by the compiler), it is invoked. That's obvious. What is sometimes less obvious is what the invocation of a constructor actually entails.

Constructors can contain a great deal of hidden program code because the compiler augments every constructor to a greater or lesser extent depending on the complexity of T's class hierarchy. The general sequence of compiler augmentations is as follows.

1. The data members initialized in the member initialization list have to be entered within the body of the constructor in the order of member declaration.

2. If a member class object is not present in the member initialization list but has an associated default constructor, that default constructor must be invoked.

3. Prior to that, if there is a virtual table pointer(or pointers) contained within the class object, it(they) must be initialized with the address of the appropriate virtual tables(s).

4. Prior to that, all immediate base class declaration(the order within the member initialization list is not relevant).

* if the base class is listed within the member initialization list, the explicit arguments, if any, must be passed.

* if the base class is not listed within the member initialization list, the default constructor(or default memberwise copy constructor) must be invoked, if present.

* if the base class is a second or subsequent base class, the this pointer must be adjusted.

5. Prior to that, all virtual base class constructors must be invoked in a left-to- right, depth-first search of the inheritance hierarchy defined by the derived class.

* if the class is listed within the member initialization list, the explicit arguments, if any, must be passed. Otherwise, if there is a default constructor associated with the class, it must be invoked.

* in addition, the offset of each virtual base class subobject within the class must somehow be made accessible at runtime.

* The constructors, however, may be invoked if, and only if, the class object represents the "most-derived class." Some mechanism supporting this must be put in place.

4 The semantics of data

If we have an empty class

class X ;

Her size is 1 byte, because the compiler insert a char. If we now have the next sentences:

class Y : public virtual X ;

class Z : publicvirtual X ;

class A : public Y, public Z ;

The size for both classes Y and Z is 8. This size is partially machine dependent. It is also dependent in part of the compiler implementation being used. The size, in both classes (Y and Z) is the interplay of three factors:

1.- Language support overhead. There is an associated overhead incurred in the language support of virtual base classes. Within the derived class, this overhead is reflected as some form of pointer (4 bytes).

2.- Compiler Optimization of recognized special cases. There is the 1 byte size of the virtual base class X subobject also present within Y and Z.

3.- Alignment constraints. The size of the classes at in this point is 5 bytes. On most of the machines aggregate structures have an alignment constraint so that they can be efficiently loaded from and stored to memory. Classes Y and Z requires 3 bytes of padding. The result is a final size of 8.

The size of the class A is determined by the following:

* The size of the single shared instance of class X: 1 byte.

* The size of its base classes Y and Z minus the storage allocated for class X : 4 bytes each (8 bytes total).

* The size of class A itself : 0 bytes.

* Class A must align on 4 bytes boundary, thus it requires 3 bytes of padding. This results in a total size of 12 bytes.

4.1 Static Data Members

Static data member are maintained within the global data segment of the program and do not affect the size of individual class objects. They are treated as if each were declared as a global variable, but with visibility limited to the scope of the class. Each class object, then, is exactly the size necessary to contain the nonstatic data members of its class.

If two classes each declare a static member freeList, then placing both of them in the program data segment is going to result in a name conflict. The compile resolves this by internally encoding the name of each static data member, it's called name-mangling, to yield a unique program identifier. There are as many name-mangling schemes as there are implementations, each one described in rigorous detail with tables, grammars and so on. Those uniques names can be easily recast back to the original name in case the compilation system needs to communicate with the user.

4.2 Nonstatic Data Members

Nonstatic data members are stored directly within each class object and cannot be accessed except through an explicit or implicit class object. Implict access means through the pointer this.

Access of a nonstatic data member requires the addition of the beginning address of the class object with the offset location of the data member. The offset of each nonstatic data member is know at compile time.

4.3 Inheritance

In case of inheritance without polimorphysm, it takes a lot memory space, because every derived class contains the basic classes. In case of inheritance with polimorphysm you are able to operate with either derived class member or base class member, as we can see in the next example:

```
class Point2d {
public:
        Point2d( float x = 0.0, float y = 0.0) :
        _x( x), _y( y)
        virtual float z() { return 0.0; }
        virtual void z( float ) {}
        virtual void
        operator+=( const Point2d& rhs ) {
                _x += rhs.x();
                _y += rhs.y(); }
        // ... more members
protected:
        float _x, _y;
};
class Point3d : public Point2d {
public:
        Point3d( float x = 0.0, float y = 0.0, float z = 0.0) :
        Point2d(x, y), _z( z) {}
        float z() { return _z; }
        void operator+=( const Point2d& rhs ) {
                Point2d::operator+=( rhs );
                _z += rhs.z(); }
        // ...more members
protected:
        float _z;
};
```

if we want to manipulate two and three dimensional points polymorphically, we could write:

```
void foo( Point2d &p1, Point2d &p2) {
        //...
        p1 += p2;
        // ...
}
```

where p1 and p2 may be either two or three dimensional points.

Support for this flexibility, however, does introduce a number of space and access-time overheads for the Point2d class. Each Point3d class object contains and additional vptr member object (the instance inherited from Point2d).

Also we can do that:
Point3d p3d;
Point2d *p = &p3d;
That's possible because the two objects (Point3d and Point2d) have identical data members order at the beginning.

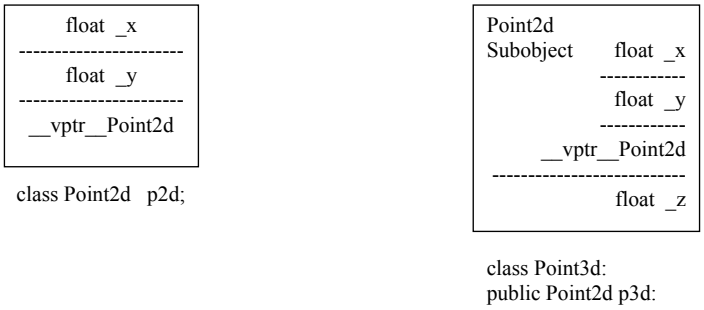


Figure 3: Data Layout: Single Inheritance with Virtual Inheritance

4.3.1 Multiple inheritance

Last assignment is impossible to do with multiple inheritance. If for example we have the next classes definition:
class Point2d {
public:

```
        // .....
protected:
        float _x, _y;
};
class Vertex {
public:
        // ...
protected:
        Vertex *next;
};
class Vertex2d :
        public Point2d, public Vertex {
public:
        // ...
protected:
        float mumble;
};
Vertex2d v2d;
Vertex *pv;
Point2d *p2d;
```

And the data layout with multiple inheritance will be:

Would be possible the next assignment:

p2d = &v2d;

because the first type of data that Vertex2d have shared in memory is Point2d, ie the type of data of p2d. But would be impossible the next assignment:

pv = &v2d;

because Vertex2d doesn't start in the same type of data that pv (Vertex).

With multiple heritance we can have the problem shown in this example:

```
class ios {.......}
class istream : public ios { ... };
class ostream : public ios { ... };
class iostream :
        public istream, public ostream { .... };
```

Both the istream and ostream classes contain an ios subobject. So, the iostream class will have repeated the ios members class.

4.3.2 Virtual inheritance

This problem is solved with virtual inheritance:

```
class ios {.......}
class istream : public virtual ios { ... };
class ostream : public virtual ios { ... };
class iostream : public istream, public ostream { .... };
```

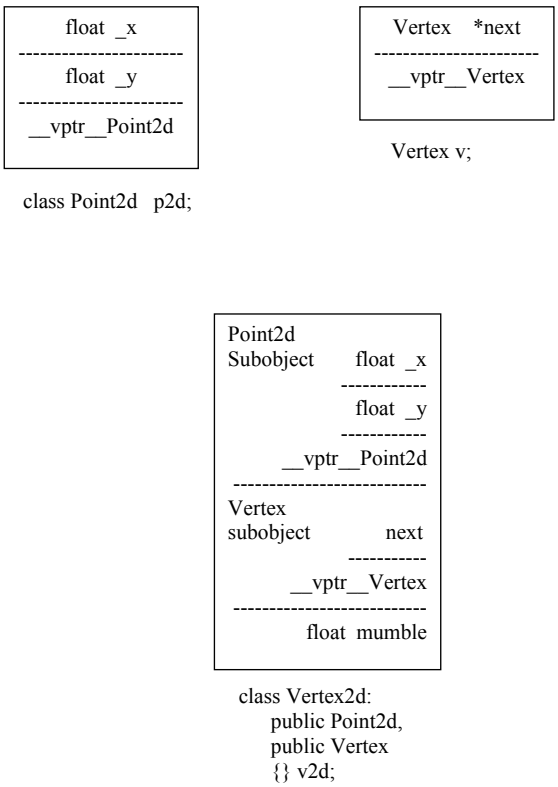


Figure 4: Data Layout: Multiple Inheritance

The general implementation solution is as follows. A class containing one or more virtual base class subobjects, such istream is divided into two regions: an invariant region and a shared region. Data within the invariant region remains at a fixed offset from the start of the object. So, members within the invariant region can be accessed directly. The shared region represent the virtual base class subobjects. The location of data within the shared region fluctuates with each derivation. So members within the shared region need to be accessed indirectly. However, one problem remains: How is the implementation to gain acces to the shared region of the class? We have two solutions:

1.- One pointer to each virtual base class is inserted within each derived class object. There are two weaknesses for this solution:

* An object of the class carries an additional pointer for each virtual base class.
* The level of indirection increases.

2.- The second solution, and the one prefered by Bjarne, is to place not the address, but the offset of the virtual base class within the virtual function table, how we show in the table at the end of the document:

In case to access to one nonstatic data member, virtual inheritance introduces an additional level of indirection in the access of its members through a base class subobject. In this case the resolution of the access must be delayed until runtime through an additional indirection.

5 The semantics of function

C++ supports three flavors of member functions: static, nonstatic, and virtual.

5.1 Nonstatic Member Functions

One C++ design criterion is that a nonstatic member function at minimum must be as efficient as its analogous nonmember function. There should be no additional overhead for choosing the member function instance. This is achieved by internally transforming the member instance into the equivalent nonmember instance. After these transformations each of its invocations must also be transformed: For example:

obj.magnitude();
becomes
magnitude_7Point3dFv(&obj);
and
ptr->magnitude();
becomes
magnitude_7Point3dFv(ptr);

In general, member names are made unique by concatenating the name of the member with that of the class. For example, given the declaration

class Bar { public: int ival; .....};
ival becomes something like
ival_3Bar

That is because we can have derivation, like this:

class Foo : public Bar { public: int ival; .....}; and the internal representation of a Foo object is the concatenation of its base and derived class members:

```
class Foo {
public :
        int ival_3Bar;
        int ival_3Foo;
......
};
```

Member functions, because they can be overloaded, require a more extensive mangling to provide each with a unique name. Transforming

```
class Point {
public:
        void x (float v );
        float x();
...
};
```

into

```
class Point {
public:
        void x_5Point( float v );
        float x_5Point();
.......
};
```

This transformation is not enought in this case because they are in the same class. The correct transformation would be:

```
class Point {
public:
        void x_5PointFf( float v );
        float x_5PointFv();
.......
};
```

5.2 Virtual Member Functions

If normalize() were a virtual member function, the call

ptr-> normalize();

would be internally transformed into

(*ptr->vptr[1])(ptr);

where the following holds:

* vptr represent the internally generated virtual table pointer inserted within each object whose class declares or inherits one or more virtual functions.
* 1, is the index into the virtual table slot associated with normalize().
* ptr in its second occurrence represents the this pointer.

5.3 Static Member Functions

If Point3d::normalize() were a static member function, both its invocations

obj.normalize();

ptr->normalize();

Would be internally transformed into "ordinary" nonmember function calls such as

// obj.normalize();

normalize_7Point3dSFv();

// ptr->normalize();

normalize_7Point3dSFv();

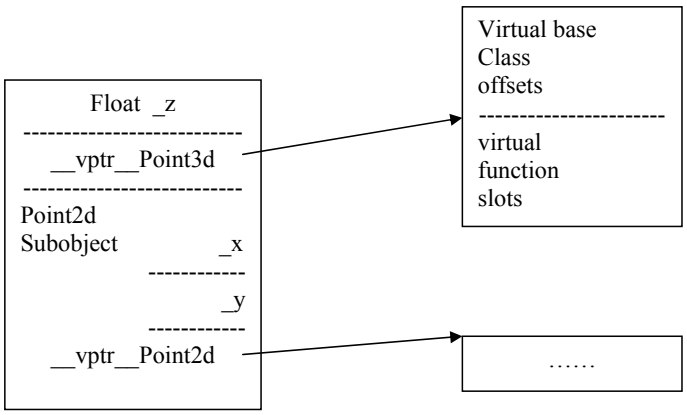
Taking the address of a static member function always yields the value of its location in memory, that is, its address. Because the static member function is without a this pointer.

6 Conclusions

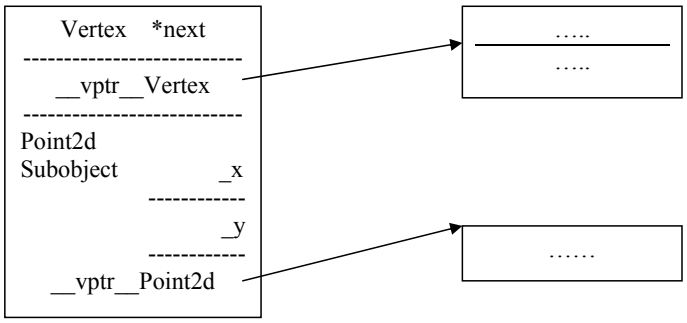
It is possible to learn how to program in an efficient way if you have knowledge about how the underlying mechanism is implemented. If you know where your data are shared and which is the way to access them, you'll be able to know wich is the best code that you can implement.

7 References

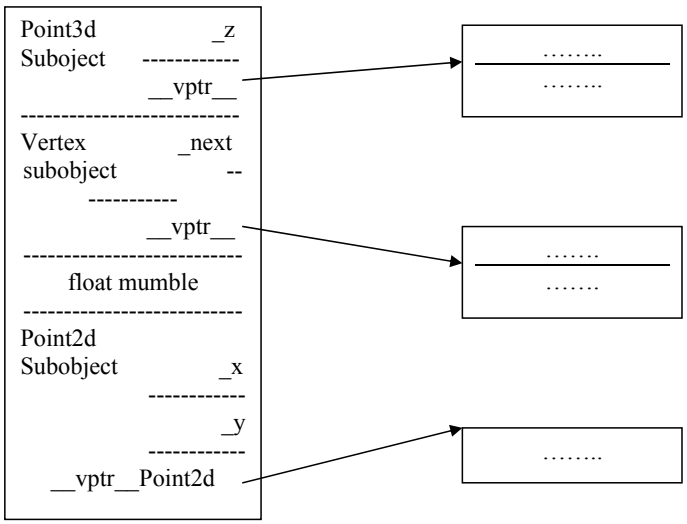
[1] S. Lippman, Inside the C++ object model,
(1996)Addison Wesley Logman, inc
[2] D. Kalev, Constructors,
http://www.informit.com/guides/content.asp?g=cplusplus&seqNum=15&rl=1
[3] C++ Course
http://www.zator.com/Cpp/E4_11_2c1.htm#Herencia%20virtual
[4] C++ Programming Language. Bjarne Stroustrup. Third edition(1997).
[5] C++
http://es.wikipedia.org/wiki/C_M%C3%A1s_M%C3%A1s
[6] Lenguajes de Programación Orientada a Objetos
http://www.inf.uach.cl/apuntes/apun_oo.pdf
[7] Object-Oriented Programming. Timothy Budd
[8] Programación Orientada a Objetos con C++



Class Point3d: virtual Point2d{…}pt3d;



class Vertex: virtual Point2d{...} v2d;



class Vertex3d:
public Point3d, public Vertex
{…}v3d;

Figure 5: Data Layout: Virtual Inheritance with Virtual Table Offset Strategy