The C++ Object Model

Magnus Lindström[1], Ulf Larsson[2], Jimmy Björklund[3]

[1] Mälardalen University, 721 23 Västerås, Sweden
dal97mlm@mds.mdh.se

[2] Mälardalen University, 721 23 Västerås, Sweden
uln98008@student.mdh.se

[3] Mälardalen University, 721 23 Västerås, Sweden
jbd99006@student.mdh.se

Abstract

In this article we will give the reader a better understanding of C++ and what is going on “under the hood”. We will present a superficial overview over the most important issues in the book “Inside the C++ Object Model” by Stanley B. Lippman. Inside the C++ Object Model focuses on the underlying mechanisms that support object-oriented programming within C++, constructor semantics, temporary generation, support for encapsulation , virtual functions, ordinary and virtual inheritance. The book shows how understanding of the underlying implementation models can help you code more efficiently. In his book Lippman points out areas in which costs and trade-offs, sometimes hidden, do exist. He then explains how the various implementation models arose, point out areas where they are likely to evolve, and why they are what they are. He covers the semantic implications of the C++ object model and how that model affects your programs.

1 Introduction

The article takes a closer look at the internal structure of the OOP supported mechanisms within C++. It tries to describe the compiler´s process when applied to different features of C++ programming such as constructors, semantics of data and virtual functions etc.

The organisation of the article is as follows. First we provide the reader with background on the object-based and object-oriented programming paradigms supported by C++. We show how the general C++ Object Model is organised and explains when constructors are synthesized by the compiler. Then the details of the C++ Object Model are discussed. The Semantics of Data, looks at the handling of data members. The semantics of methods, focuses on the varieties of member methods, with a detailed look at virtual function support. Semantics of Construction, Destruction, and Copy, deals with support of the class model and object lifetime. Runtime Semantics, looks at some of the Object Models behaviors, including the life and death of temporary objects and the support of operators new and delete.

2 The C++ Object Model

In C++, there are two different types of class data members, static and nonstatic, and three types of class member functions, static, nonstatic, and virtual. We will now show the declaration of a class Point which will be used to illustrate the general C++ Object Model.

```
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;
};
```

As shown in figure 1 nonstatic data members are allocated directly within each class object whilst static data members are kept outside. Static and nonstatic function members are also stored outside the individual class object. A table of pointers to virtual functions, the *virtual table*, is generated for each class. A pointer to the associated virtual table, the *vptr*, is inserted within each class object. For the type identification mechanism supported at runtime (RTTI) in C++ there is a *type_info* object associated with each class. This is usually addressed within the virtual table´s first slot. The strenght of the model is its space and runtime efficiency. It´s 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.

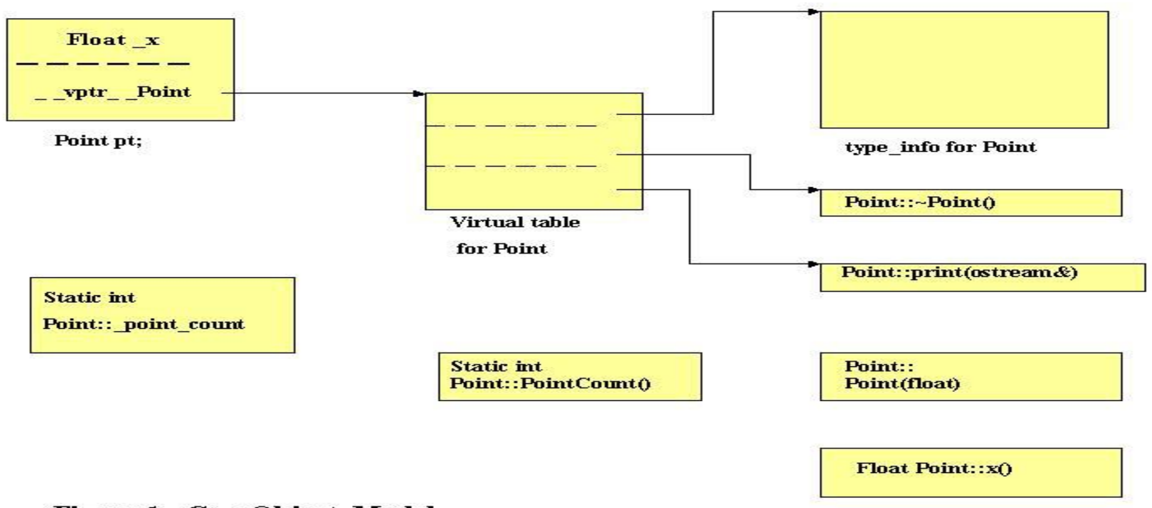


Figure 1 C++ Object Model

2.1 Polymorphism

In C++, polymorphism exists only within individual public class hierarchies. The C++ language supports polymorphism in the following ways:

1.Through a set of implicit conversions, such as the conversion of a derived class pointer to a pointer of its public base type:

```
Shape *ps = new circle ( );
```

2.Through the virtual function mechanism:

```
ps -> rotate ( );
```

3.Through the **dynamic_cast** and **typeid** operators:

```
if ( circle *ps = dynamic_cast< circle* > ( ps )) …
```

The primary use of polymorphism is to effect type encapsulation through a shared interface usually defined within an abstract base class from which specific subtypes are derived. C++ supports polymorphism through class pointers and references. This style of programming is called *object-oriented* (OO). An example of this is a Library_materials class from which actual subtypes such as Book, Video, Compact_Disc, Puppet and Laptop are derived:

```
Void
check_in (Library_materials *pmat)
{
// . . . .here should be some code
}
```

C++ also supports a concrete ADT style of programming now called *object-based* (OB) – nonpolymorphic data types. The users of the abstraction are provided with a set of operations (the public interface), while the implementation remains hidden. An example of this is a String class:

```
String girl = “Anna”;
String daughter;
….
// String::operator = ( );
daughter = girl;
…
// String::operator = = ( );
if ( girl = = daughter )
    take_to_disneyland ( girl );
```

Another programming paradigm that C++ also supports directly is the *procedural model* as programmed in C.

```
  // ….here should be some code examples in C
```

Programs written purely in the idiom of any one of these paradigms tend to be well behaved. Mixed paradigm programs, however, hold a greater potential for surprise, particularly when the mixing is inadvertent. The most common inadvertent mixing of idioms occurs when a concrete instance of a base class, such as

```
          Library_materials thing1;
```

is used to program some aspect of polymorphism:

```
// class Book : public Library_materials {. . . };

Book book;
```

//Oops: thing1 is not a Book!

//Rather, book is ´sliced´ -

//thing1 remains a Library_materials

```
thing1=book;
```

//Oops: invokes

//Library_materials::check_in( )

```
thing1.check_in();
```

rather than a pointer or reference of the base class:

//OK: thing2 now references book

```
Library_materials &thing2=book;
```

//OK: invokes Book::check_in( )

```
thing2.check_in( );
```

Although you can manipulate a base class object of an inheritance hierarchy either directly or indirectly, only the indirect manipulation of the object through a pointer or reference supports the polymorphism necessary for OO programming. The definition and use of thing2 in the previous example is a well-behaved instance of the OO paradigm. The definition and use of thing1 falls outside the OO idiom; it reflects a well-behaved instance of the ADT paradigm. Whether the behaviour of thing1 is good or bad depends on what the programmer intended. In this example, its behaviour is very likely a surprise.

2.2 Memory requirements to represent a class object

The memory requirements to represent a class object in general are the accumulated size of its nonstatic data members, any padding due to alignment constraints plus any internally generated overhead to support the virtuals.

2.3 Memory requirements to represent a pointer

The memory requirements to represent a pointer or a reference is a fixed size regardless of the type it addresses. To show this we declare the following ZooAnimal class:

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

There is no difference between a pointer to a ZooAnimal and a pointer to an integer in terms of memory requirements. The difference lies in the type of object being addressed. That is,the type of a pointer instructs the compiler as to how to interpret the memory found at a particular address and also just how much memory that interpretation should span. An integer pointer addressing memory location 1000 on a 32-bit machine spans the adress 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.

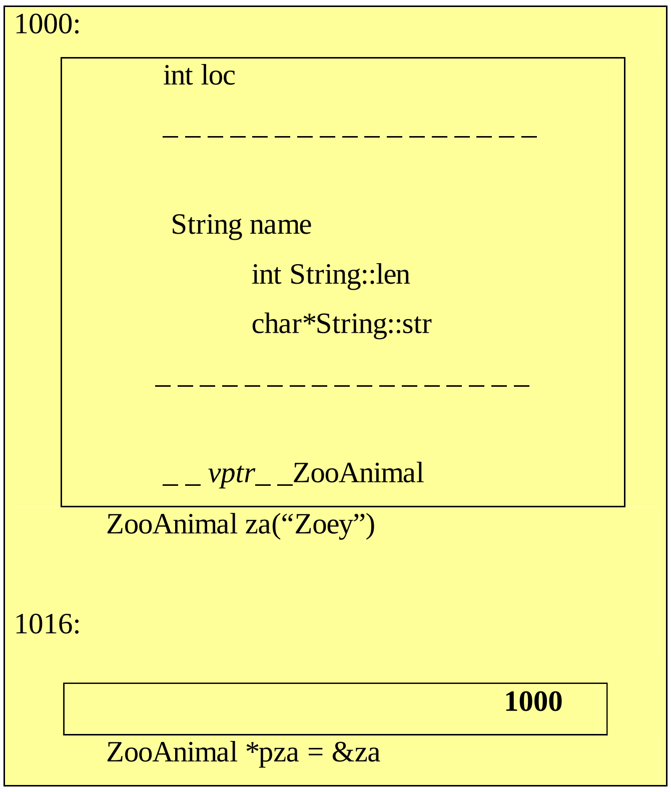


Figure 2 The layout of the class object za and the pointer pza

3 Default constructors

There are 4 cases when the compiler needs to synthesize a default constructor for classes that haven´t declared anyone.

1. If the class contains a data member from another class who has a default constructor.

2. If the class is inherited from a base class that has got a default constructor.

3. If the class declares/derives a virtual function.

4. If the class is within an inheritance hierarchy in which one or more base classes are virtual.

The synthesized constructor created by the compiler only fullfills an implementation need. It does this by invoking member object or base class default constructors or initializing the virtual function or virtual base class mechanism for each object. Within the synthesized default constructor, only the base class subobjects and member class objects are initialized. All other nonstatic data members are not initialized. These initializations are needs of the program, not of the implementation. If there is a program need for a default constructor, such as initializing a pointer to 0, it is the programmer´s responsibility to implement a default constructor for this purpose.

3.1 Copy constructors

There are 4 cases when bitwise copy semantics is not carried out.

1. If the class contains a data member from another class that has a copy constructor.

2. If the class is derived from a base class that has a copy constructor.

3. If the class declares one or more virtual functions.

4. If the class is within an inheritance hierarchy in which one or more base classes are virtual.

3.2 Program transformation semantics

When a copy constructor is applied the compiler is required to transform portions of your program. The compiler optimizes away the copy constructor invocation where possible, replacing the NRV, which stands for Named Return Value and is compiler optimization, with an additional first argument within which the value is stored directly.

3.2.1 Compiler optimization

When all return statements in a function return the same named value it is possible for the compiler to optimize the function by substituting the return variable argument for the named return value.

3.3 Member initialization list

When you write a constructor you usually have the option to choose if you want to initialize the class members through the member initialization list or through the body of the constructor but in some cases you don´t have that possibility. In the following three cases you must use the member initialization list:

1.When you are initializing a reference member.

2.When you are initializing a const member.

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

4 The semantics of data

There are a lot of myths regarding the performance of a computer program developed with the language C++. We will address a few of them in this chapter and we will also show you why they are not true. This will be done by examining how a C++ compiler builds a data object in the physical memory. When you have read this chapter you should have a good understanding how modern compilers implement the C++ object data model. If you are going to implement a C++ compiler some day this information will surely be crucial to your success.

4.1 The empty object

The intuitive size of an empty class is the size of zero. However, we still want to be able to create an instance of the class and an instance must have an address in the physical memory. This is because without a physical address there is no object. If the address of two instances of an empty class is compared they must not be the same since they are two individual objects. The C++ object model solves this by forcing an instance of an empty class to contain a single byte non-static data. Thus

```
class X {};
```

But what happens if we let another empty class Y inherit from X with virtual inheritance?

```
class X {};

class Y : public virtual X {};
```

We already knows the size of X being 1 byte. Virtual inheritance is done by letting the inheriting object have a table of pointers to the objects they have inheritied. Therefor an instance of Y has a virtual table of the size 4. In the C++ object model inheritance is done by simple concatenation giving as the size of Y being 5. This is however not always the case. Some compilers optimize the size of Y. Remember the size of X being 1 byte only to let us create an instance of the class? With the virtual pointer in an instance of Y this single byte is no longer necessary and can therefor be discarded. As a result some compilers will produce an instance of Y occupying only 4 bytes. We said earlier that the unoptimized size of Y was 5. With padding to the machines 32 bit alignment the resulting size is 8.

```
//Optimized size of Y
sizeof( Y ) == 4
//Unoptimized size of Y
sizeof( Y ) == 8
```

We will talk more about virtual pointers and tables later on.

The size of an instance of a class is partially compiler dependant, partially machine dependant. The dependancies are here described in three steps.

- The size of a pointer to a virtual base class. Size is machine dependant but in our case the size is 4 byte.

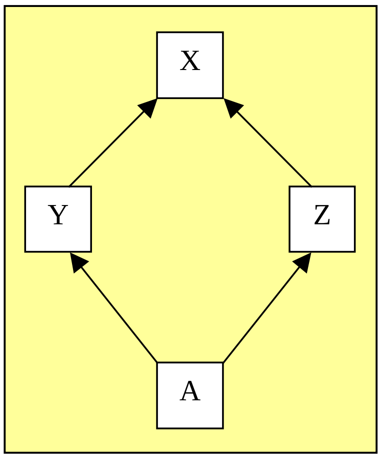
- Compiler optimizing. Some compilers optimizes the size of the empty virtual base class to 0 bytes while some compilers keep the “dummy” single byte.

- Word alignment. The compiler may need to increase the size of an object to achive a correct word alignment. In our case (32 bit) this is a multiple of 4 bytes.

4.2 Multiple inheritance

Consider the following inheritance scheme.

```
class X {};

class Y : public virtual X {};

class Z : public virtual X {};
```



We already knows the size of instances of the classes X, Y and Z. By using the dependency scheme (to the right) we find that the size of A is 12 if we are using a non-optimizing compiler.

C++ optimizes for both space and access time for non-static data members. They are all stored within the class object. A result of this is compability with the C struct.

Static data members are stored in the global data segment of the program. There will always be only one instance of the static data member no matter how many instances of the class there is. There is an exception for template classes though. The size of an instance of a class is therefor

The size of the non-static
The size of the pointers to
Alignment requirements

4.3 The binding of data members

In the early years of C++ the binding of data members introduced a few defensive styles to programming. The code below show a problem these defensive styles addressed. The problem was how data members were bound at compile time. During the early years the variable increases would be the global variable x and not the member x. To avoid this programmers always declared their data members before any inline methods.

```
int x = 5;
class A {
public:
    void increaseX( void ) { x++; }
private:
    int x;
};
```

Today this is not a problem. The binding of inline methods does not occure until the entire class definitin has been analysed. This is however not true for the parameterlist of a method. The method add in the code below expects an int as parameter while the programmer intended a float to be used.

```
typedef int TYPE;
class A {
public:
    void add( TYPE f ) { f_val += f; }
    typedef float TYPE;
private:
    float f_val;

};
```

4.4 The layout of data members

There are different ways to layout datamembers of an object in memory and the C++ object model doesn’t specify how this should be done. Todays compilers follow a common practice though. All non-static data members of the object and it’s sub-objects are stored in a single access block. They are stored in the same order as they are declared in. Any static data members are stored in the data segment of the program.

4.5 Access of data members

4.5.1 Static data members

All static data members are stored in the global data segment. The access rights to a static data member are resolved during compile time. A static data member is access by either

```
Class_name::static_member_name
```

or

```
Instance_name.static_member_name
```

or

```
Instance_pointer->static_member_name
```

It doesn’t matter which one of these alternatives is used. The static data member is access in the same way which ever you use. It doesn’t matter if the static member is declared within a complicated class hierarchy either. It is still access in the same way. This is all because the static data member is resident in the global data segment and the compiler can during compile time compute the address of the data member.

If several classes have static data members with the same name the compiler gives them unique names and they are therefor different variables even though they share name in the source code and will be resident in the global data segment.

4.5.2 Non-static data members

```
x = 4;
```

A non-static data member is access with the this pointer. Within a method this actually means

```
this->x = 4;
```

The physical address of a non-static data member is the address of the object instance plus the offset to the data member. Thus the address of a non-static data member is

```
&instance + (&ClassName::MemberName – 1);
```

The offset is known during compile time and therefor accessing a non-static data member doesn’t cost more in performance then what access to a struct member in C costs. It doesn’t even matter how complicated the class hierarchy is since all non-static data members in the object is stored in one access block which relieves us from having to traverse the class hierarchy. However, if the data member is a member of a virtual base class access time is slightly slower.

4.6 Inheritance

An object is the concatenation of it’s members with those of it’s base class(es).

4.6.1 Inheritance without polymorphism

In general, concrete inheritance adds no space nor access time overhead. However, there are a few pitfalls to avoid if performance or memory requirements are an issue.

- A Naive design might double the number of function calls to perform the same operation.

- Factoring a class into a two-level or deeper hierarchy. Splitting a class into several classes may add extra alignment due to data alignment constraints. As a result, more space is needed for the same amount of data.

4.6.2 Inheritance with polymorphism

Polymorphism adds space and access time overheads.

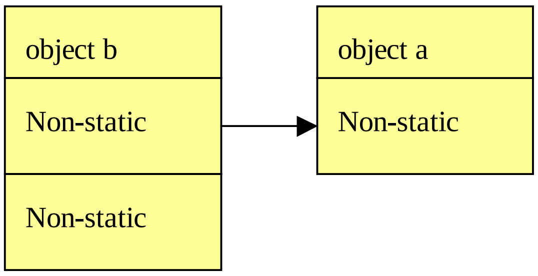
- To support polymorphism a virtual table is introduced. The size of the table is the number of virtual methods times four bytes plus four or eight bytes used for run-time type identification. There is an introduction of a vptr. This gives another 4 bytes overhead and the constructor must be extended to initialize the vptr.

- The destructor must be extended to reset the vptr.

4.6.3 Multiple inheritance

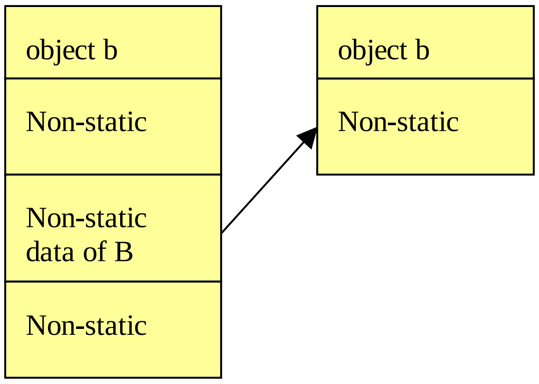
With single inheritance the code below is straight forward for the compiler. The offset to the non-static data of both the object b and object a is the same. What differs is perhaps the size of the non-static data segment. However, since the size of the non-static data segment for an object of the type A always will be less or equal to the object of type B, the only thing needed to perform a = b is a regular memory copy.

```
class A {};
class B : public A {};
B b;
A a = b;
```



Multiple inheritance is complex since you don’t know where the non-static data segment for one object begins.

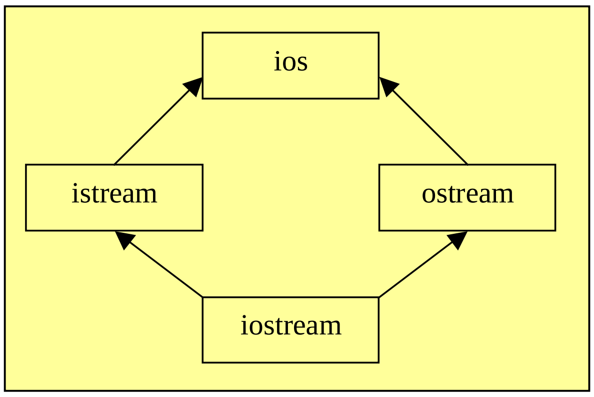
```
class A {};
class B {};
class C : public A, public B {};


C c;
B b = c;
```



The compiler calculates the offset to the data segment during compile time. Therefor, accessing members in the second or following base classes does not cause any performance overhead.

4.6.4 Virtual inheritance

The object iostream to the right only need only one instance of the object ios. This is solved with virtual inheritance.



```
class ios {…};

class istream : public virtual ios {…};

class ostream : public virtual ios {…};
```

The problem here is to find an efficient method of only having one instance of ios without breaking polymorphism between pointers to base and inherited classes. How this is solved varies between compilers and since there is no definition to how this problem should be addressed, we will not discuss this further.

5 The semantics of methods

What acctually happens when a method is called? Does it make a difference if the method is called using an object or a pointer to an object? These are a few of the questions we will answer in this chapter. There are actually three different types of methods in C++. They are static, non-static and virtual methods.

5.1 Non-static member methods

One C++ design criterion is that a non-static member function at least must be just as efficient as it's non-member counterpart. Calling a member method over a non-member function shouldn't give any performance overhead. This is achieved by transforming every member method into it's non-member counterpart at compile time. This new function accepts a pointer to the object and the other parameters the member method does.

```
//Member method

float Point3D::magnitude() const { … }
```

5.2 Virtual member methods

If magnitude() were a virtual member method, the method call would be transformed as shown below.

```
//Virtual member method call

ptr->magnitude();
```

where the following holds:

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

5.3 Static member methods

Calls to a static member method is transformed into a normal non-member function call by the compiler.

5.4 Pointer-to-member methods

Before, we saw that the value returned from taking the address of a non-static data member is the byte value of the member's position in the class layout (plus 1). One can think of it as an incomplete value. IT needs to be bound to the address of a class object before an actual instance of the member can be accessed.

The value returned from taking the address of a non-static member method, if it's not virtual, is the actual address in memory where the method is located. However, this value must also be bound to the address of an object before an actual invocation is possible.

You declare a pointer-toMember method by

```
double  //return type
(Point::*          //class the method is member
 pmf)   //name of pointer to member
();     //argument list
```

Thus one writes

```
double (Point::*coord) () = &Point::x;
```

To define, initialize and assign a value to the pointer. Invocation uses the pointer-to-member selection operators, ether

```
( origin.*coord ) ();
```

or

```
( ptr->*coord ) ();
```

The pointer-to-member method declaration syntax and selection operators serves as placeholders for the **this** pointer.

Use of a pointer-to-member method is no more expensive then a regular function pointer it it weren't for virtual methods and multiple inheritance, which complicate both the type and invocation of a pointer-to-member.

5.5 Inline methods

The first thing a compiler does with an inline method is to decide wether or not the inline method is possible to compile as inline. A to complicated or complex method may not be compiled as inline if the compiler decides not to. When compiled, the call to the method is replaced with the body of the method. For very small methods this is a very cheap way to buy performance since a method call is erased.

Inline methods provides the necessary support for encapsualing while still providing efficient access to non-public data. They are also a safe alternative to the **#define** preprocessor so common in C. However, an inline method, if invoked enough times within a program, can generate a great deal more code than may at first seem obvious from its definition.

6 Semantics of constuction, destruction and copy

6.1 Constructors

The question is what happens when a new variable is introduced: it is obvious that a constructor is called but what does the constructor actually do. To answer that question we have to discuss the structure of the class, because there are several different kinds of actions that can occur.

The first and most trivial is that the class only contains members that aren’t pointers.

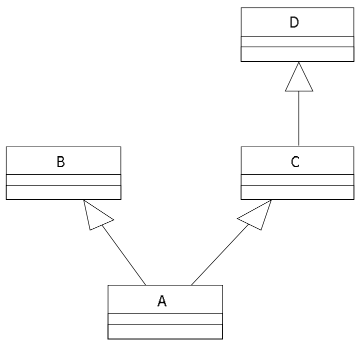
```
class ex
{
    private:
        int x;
        int y;
    public:
        some functions
};
```

Here the constructor doses not have to do anything but to set an initial value to the variables.

But if the variables on the other hand have a pointer the constructor has to make some new area for the variable on the heap. That does not realy make our life more complex but if there was an inheritance from another class, then the whole concept starts to get more complex.

The parent or parent’s constructors must also be called when there is inheritance. There is also a second thing that makes life even more complex and that is the use of virtual functions and variables.

We will first discuss the inheritance problem then return to discuss the problem with virtual. Okay then, what is the problem with inheritance. The main problem is that when you inherit a class it’s constructor must also be called and it should be called before you start to initialize the sub class.



There is also the aspect of multiple inheritance, which class should be called first and if one of those have parent classes, which should be called first?

The way to solve this problems is to first call the parents in the order they are defined in the class declaration. Take the example below for instance where the class B would be called first and then the class C and last A. But that isn't the whole truth as C has a parent class D (shown in the picture above). The actual order they are called is B, D, C, A and within each class the data has to be initialized as in the earlier discussed cases.

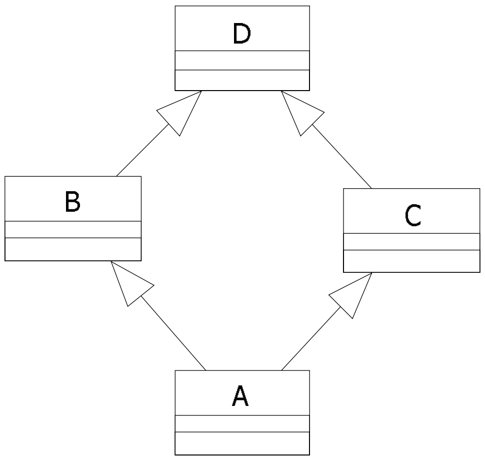
```
class a : public b , public c
```

Internally this would be represented as

```
A::A()
{
   this->B::B();
   this->C::C();
  Initials local variables.
};
```

Now we still have the problem with virtual to discuss. To illustrate the problem we can consider a case where class C and D has a virtual function f1. The question is how we shall set up the virtual tabel and which of the functions is to be called when. The standard is to let the class that is closest run the function. For exampel, if we where to call it from A we would get the function from C, but if we called it from the constructor of D we where to get the funtion that is declared in D.

Last but not least we have the problem with mutiple inharitence with a common base class. If we consider the case shown in the figur below, we will clearly see the problem with which of the two classes, B or C, should call the constructor of the class D.



6.2 Destructors

All programmers that have made some work in a large scale system knows that the use of destructors aren’t as trivial as it seems in smaller projects. There will almost always be some conflicts with how data should be deleted. The common approach is to let the creator of the data see to its destruction.

If we have a base class that have a destructor or we have variables that has one, the compiler will generate on for us. In the case of variables having one they will be called in the reversed order that they were declared and the same goes for inheritance.

If we again study the example in the internal representation of the destructor whold be:

```
A::~A()
 {
   this->C::~C();
   this->B::~B();
  deletion of local variables.
 };
```

Here we also have the problem with virtual and how the destructors are called if the virtual base class is the most-derived class.

6.3 Copy

There are two ways of copying an object. The default behavior and defining the assignment operator manually. There is another that I won’t discuss as I don’t find it important and that is to prevent coping.

The defult behavior is to copy data physicaly; this works fine if there are no references, but if one of the following occurs you have to specify a copy operation.

- The class contains an object that has an assignement operator defined.
- If the base class has an assignement operator defined
- When the class contains virtual data (can’t copy virtual pointer).

7 Runtime semantics

The run time semantics are a lot of transformations to our code done by the compiler to make the code work in a run time environment. We will start with a simpel case and work our way fom there.

Consider the ‘if’ statement below, if the variable yy is of a type we can call Y and xx is of a type we call X.

```
if ( yy == xx.getValue() )
```

Then this comparison can't occure as they are of different types. This means that we have to make a type convertion from the type X to the type Y. This is done by making an operator cast.

```
If( yy == xx.getValue().operator Y() )
```

In cases where we can't get acces to this operator the program will fail to cast and there by fail to compile. To be able to make the comparison we have to save the data from the getValue and then transform this data into another temporary variable of the type X. Finally we kan make the comparison between yy and our temporary variable. Now that we have got through the comparison we have forgotten one thing. The temporary varibals still has to get created and deleted. This is preferibly done in the beginning and end of the current block as shown in the figur bellow.

```
{
        t1.X::X();
        t2.Y::Y();
        t3.int::int();
        t1 = xx.getValue();
        t2 = t1.operator Y();
        t3 = yy.operator==(t2);
        if(t3)
        {
                ....
        }
         .....
        t3.int::~int();
        t2.Y::~Y();
        t1.X::~X();
}
```

In general it is a good idea to place the creation and destruction as close to the actual use of the data. This will spare you from unnesasary construction and destruction. This may seem to be natural to do, but as a good C or Pascal programmer this will contradict to all that you have learnd. As in both these languages the declaration of variables is always done at the start of a block.

7.1 New and delete

New is a simpel two step operation that first sets upp memory on the heep. This is done by calling a funtion called __new that in practice doesn't do anything except use malloc to allocate new memory. If this succeeds, the memory is initalized with some data, either with a basic value or by a constructor.

```
A *a = __new(sizeof(A));

If(a!=NULL)

        a=A::A();
```

Delete is a simple operation that frees the memory that new alocated. There is also protection built in to it that prevents the deletion of address 0.

```
if(a!=NULL)

    free((char*)a);
```

7.2 Arrys of objects

If we have an array as shown in the figur below, what has to be done for this array to work:

```
A a[2];
```

If it has no constuctor there is nothing to do except to allocate as much memory as needed. But if there is a constructor this must be done for each of the objects that are created. This may led to some really heavy workload as you do not know how much work is done within the consturctor of an object. The transformation that occur is shown in the figur below.

```
Vec_new( &a, sizeof(A), 2, &A::A, 0);
```

The function Vec_new is the funtion that sets up the memory needed and calls the constructor if there is one.

7.3 Global objects

As we all know, a global object is an object that is created before the main program starts and deleted when the main program ends. But these objects must be static and will be placed in the data section of the program which means that they must have a predefined size.

8 Conclusion

The book promises to explain the C++ object model. It does not aim to cover all of what is under the hood of C++, just the object-oriented features—and here it does a perfect job.We think that the book will be useful for those developers who routinely use C++ but feel the discomfort of not really knowing the underlying weight of many common operations they perform. One disadvantage with the model in our point-of-view is due to the fact that C++ has evolved a lot since 1996 when the book by Lippman was written. Consequently, many of the newer features, are not even touched. For instance namespaces are entirely forgotten. Now that the ANSI/ISO C++ committee has agreed on a final standard, we really hope Lippman will decide to take the time to update and expand his already admirable work.

9 Reference

[1] Stanley B. Lippman, Inside the C++ object model. ISBN: 0-201-83454-5

Frågor

- Vid vilka fall medför arv prestandaförsämringar ?
- Hur möjliggörs att överklassens konstruktör körs vid arv? T.ex. class A : public class B.
- Blir det någon prestandaskillnad mellan att skriva this->m() och bara m() om m är en medlems metod?