C++: Inheritance, Interfaces and Thunks
Class #: ETP-405

Niall Cooling BSc CEng MBCS MIEEE
Feabhas Ltd
5 Lowesden Works
Lambourn Woodlands
Hungerford
Berks RG17 7RY
UK
Tel. +44 1488 73050
Fax. +44 1488 73051
Web: www.feabhas.com

Abstract

Many embedded C programmers still have the misconception that C++ leads to slow, bloated programs. This has even lead to a specific subset of the C++ standard being defined for embedded systems (EC++). This FUD (Fear, Uncertainty and Doubt), which may have had limited foundation over a decade ago, is misplaced for the core aspects of C++ (Classes, Inheritance and Dynamic Polymorphism) and given a modern C++ cross-compiler it is also misplaced for the more advanced features (templates and exception handling). In this paper we will focus on the performance and memory of the core aspects, but specifically look at the use of multiple inheritance in an embedded environment.

Introduction

Today C is still the dominant programming language in embedded systems. Even with an updated version of the C standard being published in 1999 [C99], in our experience, this has not yet proliferated in embedded programming circles. Through this paper, any reference to C will denote the pre-1999 version, generally know as C90 [C90][1].

The most important item to remember when looking at the performance and memory issues of C++[C++98] is that it has its foundations in C. C was chosen for its flexibility, efficiency, availability and portability [Str94]. The C++ complier front-end (Cfront), originally developed by Stroustrup, output C that was then compiled into executable code. For many of the memory and performance demands of C++ we can draw parallels with C. If you are so inclined, you can even mimic object-oriented code in C [Coo03]. However C++ is not a pure superset of C (it was never intended to be) and there are various well documented incompatibilities [Str97].

The Class

The C++ class is a basic extension of the C struct. The structure and size of a class is resolved at compile-time (e.g. sizeof is a compile time result). Any non-static data access is resolved to <object_start_address+member_offset>. For simple C++ classes, the packing and padding rules tend to follow those well known to C programmers. Given the class declaration:

[1] For those unaware of C99, I would recommend reading Stroustrup's article [Str02].


```
class X
{
public:
  int a;
  int b;
  int c;
};
X x1;
x1.c = 10;
```

Defining an object and accessing the third data member (c and assuming `sizeof(int) == 4`) will evaluate to *<x1 address>+8*.

e.g. *ARM7 assembler using GHS MULTI (no optimization)*

```
x1.c = 10;
0x20080ac  main+0xc:  e3a0200a   MOV R2, 10                        // 10 => R2
0x20080b0  main+0x10: e59f800c   LDR R8,[PC,12](&x1 (0x20080c4))   // &x1 => R8
0x20080b4  main+0x14: e5882008   STR R2,[R8,8]                     // R2 => R8[8]
```

The significant difference between a C++ class and a C struct is the ability to declare member functions and have access specifiers (public, private, protected). Member functions differ from normal "free" functions in that they are passed an implicit pointer to the data area of the current object. The implicit object pointer, known as *this*, enables functions to access the appropriate objects data members. When a member function is called in the context of an object, the address of the object is automatically passed as an argument for the *this* parameter. Member data access within a member function involves *this* pointer dereference plus data member offset.

Therefore, the performance of a C++ member function is identical to a C function where a pointer to a struct is passed as the first parameter, e.g.

```
/* C code example */
typedef struct
{
  int a;
  int b;
  int c;
}X;


void set_c(X *const me, int value)
{
  me->c = value;
}

X x1;
set_status(&x1,10);
```

```
// C++ code example
class X
{
  int a;
  int b;
  int c;
public:
  void set_c(int value);
};

void X::set_c(int value)
{
  this->c = value;
}

X x1;
x1.set_status(10);
```

Of course the significant difference between the C and C++ code is that the C code can directly access the member (e.g. `x1.c = 10`), whereas the C++ code will generate a compile time error. This enforced encapsulation is central to ensuring low coupling within a program. High coupling is the root cause of many C programs and leads to "jenga®"[2] code.

[2] http://www.hasbro.com/jenga/

Associations

Building an object-oriented solution to a design problem involves programming up a number of interdependent objects. These objects collaborate, by sending messages to each other, to solve the overall problem. Messages (in sequential code) call on object's operations which are implemented by member functions.

If one object wants to send a message to another object then the classes must be associated (i.e. the target object must be in scope). Associations may be unidirectional (i.e. client-server) or bidirectional (i.e. peer-to-peer) [Figure 1].

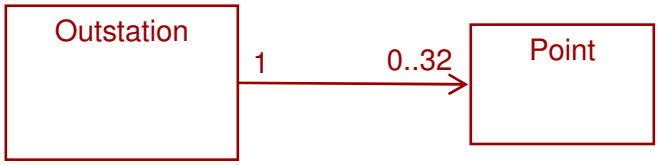

Figure 1 Class Association (unidirectional)

Associations are typically implemented as member data pointers (or a reference for single fixed associations) to the server class. The "client" object then dereferences the pointer to be able to call on published member functions of the "server" object. If the client manages a number of servers, then this is stored as a vector or array of pointers.

Inheritance

Inheritance is a cornerstone of object-oriented programming [Figure 2]. One of the major benifits of inheritance is that a base class pointer (or reference) can point a either a base class object or a derived class object[3].

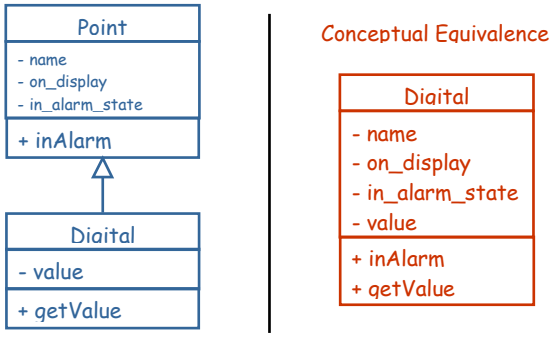

Figure 2 Inheritance

Derived classes are central to reusable software development. They allow a designer to define a new class by extending the facilities of existing, proven, classes while:

(a) Not modifying the base class
(b) Replace a base class object with a derived class object

[3] It is interesting to note that Stroustrup chose the terms *Base class* and *Derived class* in preference to the alternative *superclass* and *subclass* [Ell95]

(c) Linking clients to new server object with extended functionality without any modification to the client [Figure 3].

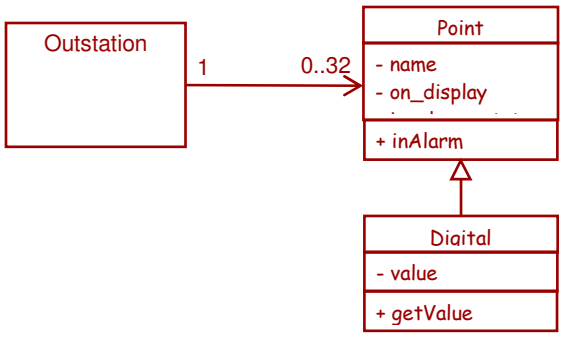

Figure 3 Association and Inheritance

Typically the data members of the derived class are appended to the memory area of the base class. Note, however, the language standard does not define this and it is left to each compiler to define what it considers the most optimal solution. For example, a compiler could chose to follow the "pimpl" idiom [Sut99] for each part of the base and derived class (however I have never seen any compiler do this).

Assuming the derived class data members are appended to the memory area of the base class members, accessing a derived member is no different from accessing a base member (i.e. *this+data_offset*). It is possible, however, that dependant on certain architectures and compilers padding may occur between the base member data set and the derived member data set (i.e. padding that would not have occurred if the class had been defined in its conceptual form).
e.g.

```
class A
{
    char a1;
    char a2;
};

class B : public A
{
    char b1;
    char b2;
};


int main()
{
    std::cout << "A " << sizeof(A) << " B " << sizeof(B) << std::endl;
}
```

Executing the example code usually prints "`A 2 B 4`", but may display "`A 2 B 6`" where two bytes have been inserted for 32-bit alignment (optimized for B pointers).

Once a derived class is define, assignment from a derived object to a base object is implicitly supported and a base class pointer may also point at a derived object. However, the opposite is illegal; base objects cannot be assigned to derived ones, and derived class pointers cannot point at base

objects. Finally, a derived pointer can also be assigned to a base pointer (the compiler will do implicit casting).

Polymorphism, Overloading and Overriding

Overloading is the ability to declare multiple functions with the same name but different signatures, e.g.

```
void f();
void f(int);
```

Where functions are overloaded the compiler must be able to determine at compile time which functions should be called based on the type of the operands (otherwise a compiler error), e.g.

```
f();    // calls f()
f(10);  // calls f(int)
```

This is referred to as *early binding* or *static polymorphism*. There is no overhead compared to normal function calls as the function address is resolved at compile/link time. However, when a derived class creates its own version of a base class member function, this is referred to as ***overriding***.

```
class Base
{
public:
    void display(void){
        cout << "In Base" << endl;
    }
};

class Derived :public Base
{
public:
    void display(void){
        cout << "In Derived" << endl;
    }
};
```

Due to the implicit *this* pointer, the overridden member function can still be statically resolved through the calling object's type. However, when an overridden function is called through a base class pointer the compiler cannot necessarily determine at compile time the actual type of an object, e.g.

```
void show(Base *bptr)
{
  bptr->display();  // ??? which function to call
}

Base *bptr = new Base;
Derived *dptr = new Derived;

bptr->display();       // Base::display(Base*)
dptr->display();       // Derived::display(Derived*)

show(bptr);
show(dptr);            // ??? which function to call
```

With *static polymorphism* the call will always resolve to `Base::display(bptr)`, irrelevant whether the object being pointed to is a base or derived object.

To overcome this limitation, C++ supports the concept of *late binding* (*dynamic polymorphism*). By adding the keyword ***virtual*** to the declaration of the function in the base class, we then allow a call to this function through a base class pointer (or reference) to be resolved at runtime (i.e. dynamically).

```
class Base
{
public:
    virtual void display(void){
        cout << "In Base" << endl;
    }
};

void show(Base *bptr)
{
  bptr->display();  // will now call either Base or Derived
}
```

As late binding has to determine which function to call a runtime it, must, therefore have an overhead compared to a normal member function call. It should be noted that that this overhead may not always play a part in calling an overridden function. For example, given the following code:

```
Base *bptr = new Base;
Derived *dptr = new Derived;

bptr->display();
dptr->display();       // early binding

show(bptr);
show(dptr);            // late binding
```

The call to `dptr->display()` would not be late bound (as the type of `dptr` is in scope). For good compilers, given the code:

```
Base *bptr = new Derived;
bptr->display();              // bind to Derived::display
```

will still early bind if it can deduce that `bptr` could not be modified between is initialization and its call.

Once a class has any virtual functions, then a virtual function table (v-table or *vtbl*) is created for that class. The table is essentially an array of function pointers, one for each declared virtual function. The v-table is constructed at compile time (i.e. it is not formed at runtime). Each object, in addition, has an attribute, call a v-table pointer (*vtptr*), added to its data member set. When the complier is required to resolve a dynamic polymorphic call, it now involves:

(1) Dereferencing the *this* pointer to get at the object
(2) Dereferencing the v-table pointer to get to the v-table
(3) Index into the table for the appropriate function address
(4) Calling the function passing the *this* pointer as a parameter (remember all member functions require a *this* pointer)

This resolved (C style) to "`this->vtptr[n](this,…)`" [Figure 4].

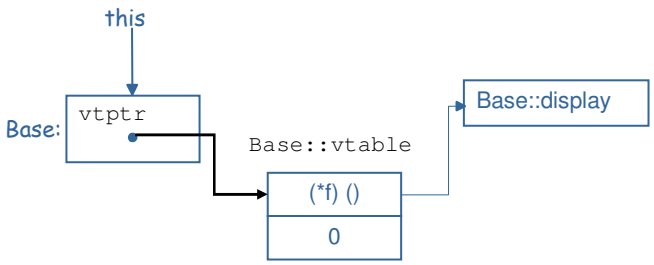

Figure 4 Dynamic Polymorphism

The placement of the v-table pointer with the member data structure is implementation defined. Originally compilers append it to the end of the base classes data member set. However, most modern compilers place it as the first data member, thus offsetting all other members by the pointer size (e.g. four bytes on a 32-bit processor) [Lip96]. Placing the *vtptr* as the first member has the advantage that the *this* pointer is already pointing at the *vtptr*, so no adjustment is necessary during dynamic polymorphic calls. In addition the base data and derived data are still a contiguous group which is less likely to lead to padding between within the derived class. For example, modifying our previous example:

```
class A
{
    char a1;
    char a2;
public:
    virtual void f();
    virtual ~A(){}
};

class B : public A
{
    char b1;
    char b2;
};


int main()
{
    std::cout << "A " << sizeof(A) << " B " << sizeof(B) << std::endl;
}
```

Without virtual functions and for a pair of compilers that both output "`A 2 B 4`", when virtual functions are added and they differ in their *vtptr* placement, then the outputs are likely to be:
"`A 8 B 8`" for the compiler placing the *vtptr* at the start [Figure 5a], and
"`A 8 B 12`" for the complier with the *vtptr* at the end of A's data segment [Figure 5b].

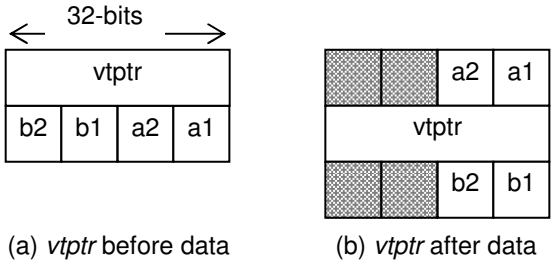

Figure 5 vtptr offset

A derived class automatically gets its own copy of the base class's v-table (note this is a copy). Initially this copied v-table's entries will be pointing at all the base class virtual functions. When a virtual function is override the entry at the appropriate index is overwritten to point at the derived class's version of that function. However the polymorphic call actual resolves to the same code, i.e. "`this->vtptr[n](this,…)`", because, in reality, C++ cannot resolve calls at runtime as it has no inbuilt operating system or virtual machine. Thus *dynamic polymorphic* function calls are still compile time resolved except pointer manipulation is used to support indirect addressing.

As had been started, using *dynamic polymorphic* functions must add some overhead to using statically resolved function calls. In addition to the actual call, the general space and time overheads include:

- Introduction of a virtual table with an entry for each virtual function (static data)
- A *vtptr* in each object (runtime data)
- Augmentation of constructor to initialize object *vtptr* (code + time)
- Augmentation of destructor to reset *vtptr* (code + time)

These are all notable, but relatively minor in most cases. One of the potential greatest overheads, for an embedded system, is the placement of the v-tables in memory. As they are compile time resolved, they are, in effect, constant. Thus they will, by default for most linkers, be bundled in the const memory area. In many embedded systems this is placed in Flash memory (or EPROM). This means that for any dynamic call, part of the function pointer dereferencing involves reads of address information from Flash rather than RAM. This may end up being significant is any fast-tight code. For most linkers this can resolved by specifying the table be located in RAM (but ensure, like initialized externs, that the values are copied from Flash to RAM). There is one potentially issue with placing the v-table in RAM; it can then be manipulated and thus lead to program exploits [Pin04].

One other function impacted by virtual functions is the *operator* = (assignment). For simple objects many compilers do the equivalent of a *memcpy*[4] (note that the standard specifies it should be a member-by-member copy). However, as a derived object can be assigned to a base object, if a simple *memcpy* was used then the base object's *vtptr* would be overwritten with the address of the derived v-table (and all manner of interesting behavior would ensue!).

Multiple Inheritance

A class can inherit from any number of base classes. The use of more than one direct base class is typically referred to as multiple inheritance [Figure 6]. In a simple case the general behavior is straightforward, e.g.

[4] C standard library call

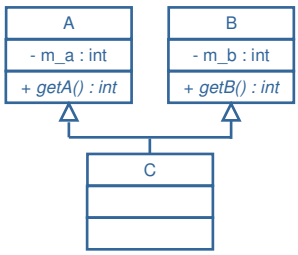

Figure 6 Multiple Inheritance

```
class A
{
    int m_a;
public:
    virtual int getA(){return m_a;}
};

class B
{
    int m_b;
public:
    virtual int getB(){return m_b;}
};

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

In the example above, a *C* object can be called in the context of *A*'s member functions or *B*'s member functions. The order in which a derived class inherits from multiple base classes is functionally not important (except the order of construction is defined by the order of inheritance – e.g. *A* will be constructed before *B*). However, we shall see from a performance perspective there are differences. The actual runtime memory layout is, again, implementation dependent, but typically the memory is laid out in the order that the classes are inherited. Given the example above, investigation of the runtime memory would typically have `m_a` as the first data member followed by `m_b` [Figure 7]. Taking this general case, we shall refer to the first base class in the inheritance list (e.g. *A*) as the Primary Base Class (PBC) and all subsequent base classes as Secondary Base Classes (SBC).

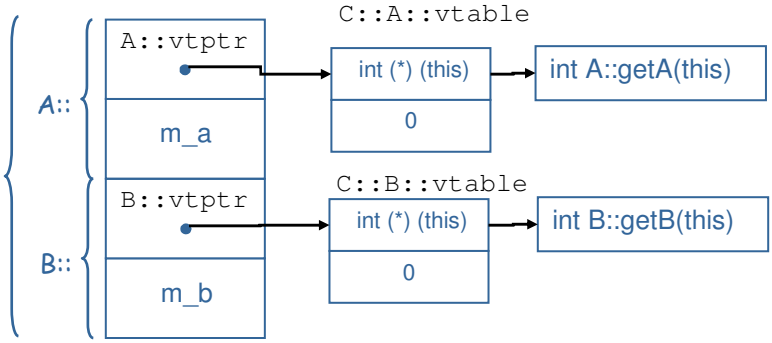

Figure 7 MI Layout

Once a derived class multiply inherits, the compile must start making adjustments throughout the code. We have already described the expected memory layout for each object type. Also, we know, the

access to member data via a member function is resolved at compile time, e.g. a member function accessing the first member data would use *<this+4>* on a 32-bit machine.

As we have stated, a base class pointer can point at a derived object. This means that base class pointers for any of the base classes can point at the same derived object. From hereon in the paper will describe the more common implementations, but remember, all memory layout is implementation specific. Given an object of the derived type (*C*), with an address (*&c*), we would expect this address to be the start PBC data part, and thus the PBC's v-table pointer. Assigning the address of the object to an *A* pointer (*A**) and calling any member function of *A* involves no pointer adjustment.

However, if we then assign the address of the object to a SBC pointer (e.g. *B**) and then call a *B* member function we have a potential problem. A *B* member function is expecting a *this* pointer that holds the address of the start of a "*B*" object. Accessing the first data member is resolved to *<this+4>*. However, we also know that the address of the *C* object points at the PBC part not the SBC part. Thus before calling the *B* member function we must adjust the *this* pointer to correctly align with the SBC data part (e.g. *B*). For example:

```
C c;
A* aptr = &c;
x = aptr->getA();      // A::getA(this){return [this+4]}

B* bptr = &c;
x = bptr->getB();      // B::getB(this){return [this+4]} ????
```

The adjustment can be made in one of two places. First the compiler must calculate the difference between the natural *this* pointer address (e.g. *&c*) and the start of the SBC data region. This we can call *Delta(B)*, and in our example is eight bytes. In the first approach the compile can adjust the this pointer just prior to the member function call:

```
B::getB( (B*)(((char*)this)+8) ){return [this+4]}
```

More typically, though the compiler makes the adjustment when the SBC pointer is assigned the address of the derived object, e.g.

```
B* bptr = &c;          // bptr = (B*)(((char*)&c)+8)
x = bptr->getB();      // B::getB(this){return [this+4]}
```

This has the advantage that the call to the SBC member function is much simpler and does not require knowledge of the calling object.

However, a compiler will typically have to make run time adjustments when a SBC member function is being called directly in the context of a derived object, e.g.

```
int x = c.getA();      // A::getA(this){return [this+4]}
int y = c.getB();      // B::getB(this+8){return [this+4]}
```

so it may choose to always do this pointer adjustment on function call.

MI Name Clashes

In the previous example each base class's member function had a different name. But situations arise where both base classes have member functions with a common name, e.g.

```
class A
{
    int m_a;
public:
    virtual int f();
};

class B
{
    int m_b;
public:
    virtual int f();
};

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

Given this case, a direct invocation of the member function from the derived object will cause a compile time error, e.g.

```
C c;
int x = c.f();          // ERROR ambiguous A::f or B::f
```

This could be resolved by explicit name resolution (`c.A::f()`) but this is pretty ugly and indicates poor design. Nevertheless, both functions can be invoked through appropriate base class pointers, e.g.

```
A* aptr = &c;
x = aptr->f();          // Okay A::f

B* bptr = &c;
x = bptr->f();          // Okay B::f
```

If the clashing member functions in the base classes are both virtual functions, then it is perfectly legal for the derived class to override both base class functions with a single function of the same name, e.g.

```
class C : public A, public B
(
    int m_c;
public:
    virtual int f()
    {
       return m_c;
    }
};
```

This removes the ambiguity and eliminates the previous error. It has the added benefit that whether the function is invoked directly or through a base class pointer, the same function will always execute e.g.

```
C c;
int x = c.f();          // Okay C::f

A* aptr = &c;
x = aptr->f(); // C::f
```



```
B* bptr = &c;
x = bptr->f(); // C::f
```

Member data access for the derived class doesn't change. The derived member data will be appended either after the PBC's data segment, or appended to the end of the overall data segment (i.e. PBC + SBC data area). Either way, member data access in member functions is still via a *this* pointer offset access. In addition the derived object will have two separate v-tables; one from the PBC (*A*) and one from the SBC (*B*). In each inherited v-table the entry will be replaced with the address of the derived overridden function's address (`&C::f`). Assuming the object memory layout (for a C object) is [Figure 8]:

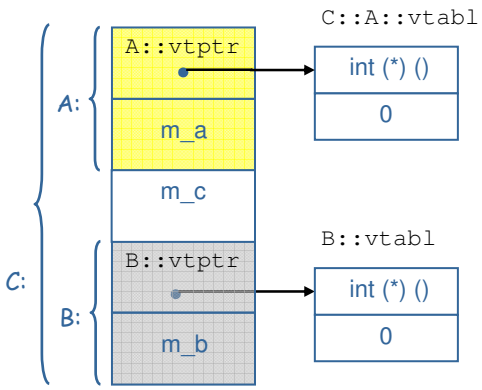

Figure 8 Object Memory Layout

The access to "`m_c`" will be <*this*+*8*>. If the function is called dynamically through a PBC pointer (*A**), then no pointer adjustment is required.

However, we have a problem brewing. We have looked at the situation where when a member function is called via SBC pointer (*B**); the compiler must adjust the *this* pointer to compensate for the derived classes this offset. So, for example:

```
C c;
int x = c.f();

B* bptr = &c;  // B* bptr = (B*)( ((char*)&c)+12)
```

However, now when we make the polymorphic call, which is going to resolve to the derived classes function (`C::f`), the *this* pointer now is not pointing at the start of the object as expected, e.g.

```
x = bptr->f(); // Opps!!! C::f(this+12)
```

This means the memory beyond the actual object is going to be accesses, which of course will lead to incorrect behavior and possibly memory corruption.

So we appear to have a stalemate. One the one hand we know we have to adjust the *this* pointer for SBC calls (which is likely to be done at point assignment time thus having no knowledge of the actual call) and yet we need to readjust it back by *delta(SBC)* before calling the derived member function.

One approach is to extend the v-table so each virtual function has two entries:

- the address of the function to be called
- this pointer adjustment value (*-delta(SBC)*)

Modern compiler tend to use an alternative that doesn't change the size of the v-table. Instead, where this pointer adjustment is required (i.e. *–delta(SBC)*) then the compiler creates a small piece of code that does the adjustment, manipulates the stacks return address (so the execution thread can directly return from the derived function) and then calls the derived member function (`C::f`) [Figure 9]. The address of this code is then inserted into the appropriate SBC v-table entry (`C::B::vtable`). The code that does this adjustment is usually referred to as a *thunk*[5]. Thunks naturally add both a code size and performance overhead; therefore with MI it is advisable to make the PBC to more frequently accessed interface.

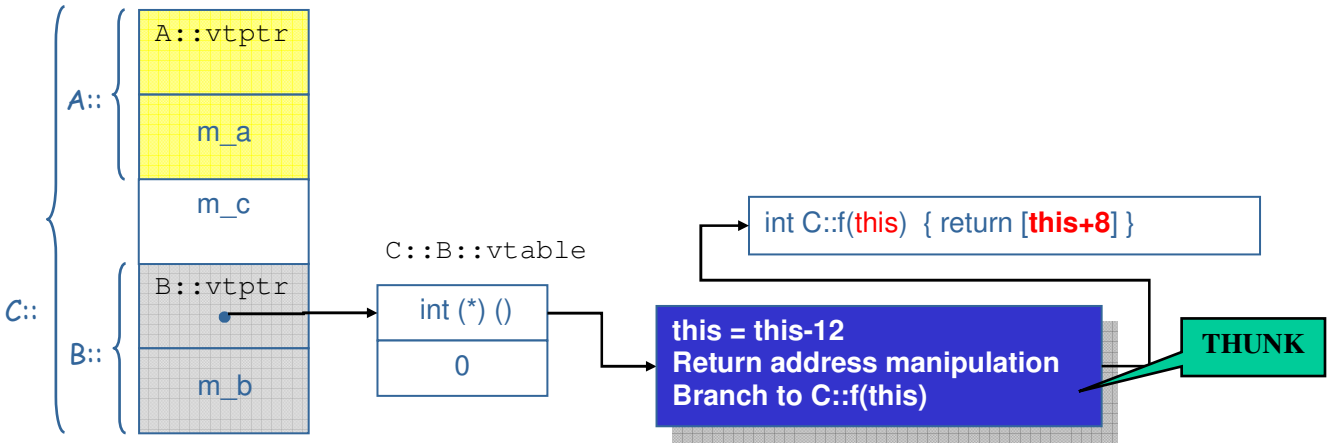

Figure 9 Thunk

Deadly Diamond of Death

Clearly, a base class can itself be derived from another class. So we can introduce the terms *direct base class* (those that appear in our inheritance list) and *indirect base class* (those which our base classes inherit from). With simple inheritance structures this is just an extension of the rules we have followed so far. Things start to become interesting when a base class (either direct or indirect) appears multiple times in ancestry e.g.

```
class A
{
// stuff
};

class B : public A {};

class C : public A {};

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

[5] Apparently the term dates back to Algol60 [Ell95]

Conceptually we hope for the situation of just one base class, i.e. [Figure 10].

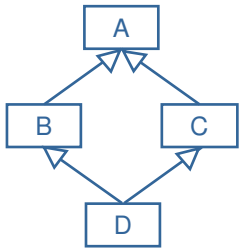

Figure 10 Common Base Class

but what we get, by default, is [Figure 11]:

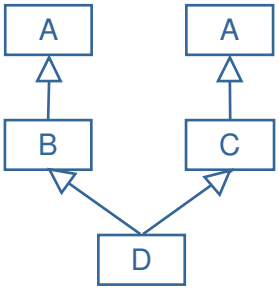

Figure 11 Repeated MI

This has been referred to as the "deadly diamond" [Mey98] or the "diamond of death" [Sut99].

If the root base class (*A*) declares a function, which is then used in the context of the most derived object (*D*), the code will encounter compiler errors messages about ambiguous functions, e.g.

```
class A
{
private:
    int m_value;
public:
    int get_value(void){  return m_value;  }
};

class B : public A {};

class C : public A {};

class D : public B, public C
{
public:
    void show_value(void)
   {
      std::cout << get_value();  // ERROR!!!
    }
};

error #266: "D::get_value" is ambiguous
```

Thinking, as there is only one root class, we can resolve the error by calling the function by fully qualify the name with the scope resolution operator, as in:

```
std::cout << A::get_value();
```


This, however, leads to a similar, but with a different (and slightly more confusing) error message, e.g.

```
error #286: base class "A" is ambiguous
```

The problem can be resolved using scope resolution operator, but only by using one of the direct base class names (even though they don't define the function).

```
B::get_value() or C::get_value()
```

This is far from an ideal solution. Once we start constructing base classes, the problem becomes even more apparent. For example, given:

```
class A
{
protected:
   int m_value;
public:
   A(int a): m_value(a){ cout << "A's ctor with value " << m_value << endl; }
};

class B: public A
{
   int m_b;
public:
   B(int b) : A(b), m_b(b) { cout << "B's ctor with value "<< m_b << endl;}
};

class C : public A
{
   int m_c;
public:
   C(int c) : A(c), m_c(c) {  cout << "C's ctor with value "<< m_c << endl;}
};

class D : public B, public C
{
public:
    D(int b, int c) : B(b), C(c){};
};

D d1(20,30);   // Create an object d1 of type D
```

When a *D* object (*d1*) is created, its constructor is called. This, in turn, calls its direct base class constructors. In this example *B* is the PBC of *D*, so on *D* being created, *B*'s constructor will be called first (ahead of *C*'s), with an argument of 20. However, *B* has its own direct base class (*A*), so before *B*'s constructor runs, *A*'s constructor is called being passed the argument 20. The first printed output is:

```
A's ctor with a value of 20
```

Once the *A* constructor is complete, *B*'s constructor then completes, printing out:

```
B's ctor with a value of 20
```

Once *B* is fully constructed, the thread of execution returns back to *D*. However, *D* has a SBC (*C*) and its constructor is called (argument 30), which in turn calls *A*'s constructor again (but this time with an argument of 30), printing out:


```
A's ctor with a value of 30
C's ctor with a value of 30
```

Here we have the problem that *A* is repeated, consequently its constructor is called twice. However, each *A* constructor initializes its own memory [Figure 12].

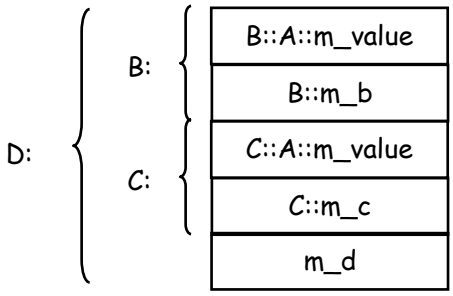

Figure 12 Repeated MI Memory

The keyword **virtual** when added to a base class specifier ensures that only one copy of the base class is used in the derived class. e.g.

```
class A
{
private:
    int m_value;
public:
    int get_value(){  return m_value;  }
};

class B : virtual public A {};

class C : virtual public A {};

class D : public B, public C
{
public:
    void show_value(){
      std::cout << get_value();  // Okay now only one version of A
    }
};
```

So how is this resolved? There are again, a number of different approaches. The earlier compilers solved the issues of repeated multiple inheritance by adding a pointer to the virtual base class part in the derived object [Figure 13]. When the virtual class is repeated in the hierarchy the pointers both point at the common base class's data area.

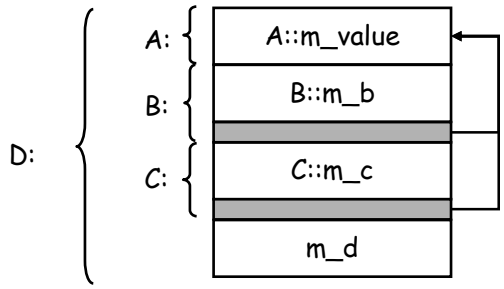

Figure 13 Virtual Inheritance

An alternative approach, used by many modern compilers is to utilize the v-table. Compilers use negative relative indexes (e.g. vtable[-1]) to the store the address of the base classes part[6]. This has the advantage that the object size is unaffected by virtual inheritance.

However, both *D*'s direct base classes (*B* and *C*) have their own direct base class. As we have seen in the previous example, they both call the constructor for *A*. What value will *A* be initialized with, the one passed by *B*'s or *C*'s constructor?

The answer is neither; you will get a compile time error, e.g.

```
error #291: no default constructor exists for class "A"
      D(int b, int c):B(c),C(c){}
                              ^
```

Huh?

To eliminate this error message, the derived class (*D*) has to explicitly call the repeated (and indirect) base class's (*A*) constructor, e.g.

```
class A
{
protected:
   int m_value;
public:
   A(int a): m_value(a){ cout << "A's ctor with value " << m_value << endl; }
};

class B: public A
{
   int m_b;
public:
   B(int b) : A(b), m_b(b) { cout << "B's ctor with value "<< m_b << endl;}
};

class C : public A
{
   int m_c;
public:
   C(int c) : A(c), m_c(c) {  cout << "C's ctor with value "<< m_c << endl;}
};

class D : public B, public C
{
public:
    D(int a, int b, int c) : A(a), B(b), C(c){};
```

[6] Many compilers apply the same technique to manage RTTI


```
};

D d1(10, 20,30);       // Create an object d1 of type D
```

This ensures (because we have used the virtual keyword) that the constructor for *A* is called once and once only (ugly but true).

This leads to one subtlety, if a *B* object is constructed then it must call *A*'s constructor. However, if *B* is constructed as part of *D*'s construction it must not call *A*'s constructor. This is, typically, achieved through the compiler adding an additional Boolean parameter to *B*'s constructor (and *C*'s in this case) which will flag whether to call *A*'s constructor.

Unique Call Path

Given the following code:

```
class A
{
public:
    virtual void display()
    {
      cout << "In A\n";
    }
};
class B : public virtual A {};

class C : public virtual A {};

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

And the following calls:

```
A* ap = new D; ap->display();
B* bp = new D; bp->display();
C* cp = new D; cp->display();
D* dp = new D; dp->display();
```

The output will be:

```
In A
In A
In A
In A
```

If the derived class were to override the virtual function with its own member function:

```
class D : public B, public C {
public:
    virtual void display()
    {
      cout << "In D\n";
    }
};
```

Then for the same code we will get:

```
In D
In D
In D
In D
```


This is our expected behavior based on everything we have discussed so far.

However, there is one case worth considering. What should the behavior be if the common base class's function was overridden by one the intermediate classes and not by the most derived class? e.g.

```
class B : public virtual A {
public:
   void display() { cout << "In B\n"; }
};
```

Certainly it would be logical if

```
A* ap = new D; ap->display();
B* bp = new D; bp->display();
D* dp = new D; dp->display();
```

all printed "`In B`" (which they do). But what about:

```
C* cp = new D; cp->display();
```

Here we have a sibling class (of *B*), not a direct descendent. If we have an object of that type (*C*) and called the member function we will get "`In A`" printed. However because *A* is not repeated, there can be one and only one version of a virtual function in a call tree (otherwise ambiguity arises and subsequent compiler errors). In our example as `B::display` is considered "closer" to *D* than *A*, then it is said to *dominate* [Ell95]. So in answer to the question, when the call is made:

```
cp->display();
```

the output is also "`In B`".

Object overheads

So what are some of the overheads associated with virtual inheritance (assume 32-bit)? Given:

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

The memory requirements vary quite significantly depending on whether the compiler uses embedded pointers to the common base class, or uses the v-table offset.

Virtual implementation	Embedded Pointer	v-table Offset
A	1	1
B	8 *(4 bytes for the virtual base class pointer + base class part + alignment padding)*	4
C	8	4
D	12 *(A+B+C, but A is not repeated)*	8 *(B::vtable & C::vtable)*

Adding virtual functions to all the base classes, e.g.


```
class A{ public: virtual void f() {}};
class B : public virtual A{ public: virtual void g() {}};
class C : public virtual A{ public: virtual void h() {}};
class D : public B, public C{};
```

Virtual implementation	Embedded Pointer	v-table Offset
A	4	4 *(A::vtable)*
B	12 *(4 for the virtual base class pointer + 4 for its own v-table pointer + base class (A) part))*	4
C	12	4
D	20 *(A+B+C, but A is not repeated)*	8 *(B::vtable & C::vtable)*

The above tables are only looking at runtime memory and do not include the actual v-table sizes. However, as it is unlikely to be using virtual inheritance without virtual functions, the second table is a better comparison of the difference.

Summary

This paper has demonstrated that using the full object-oriented features of C++ (classes, inheritance and dynamic polymorphism) introduces both memory and performance overheads. However, these overheads, in most cases, are minimal and can be predicted. What should be apparent is the selection of compiler will make a greater impact on memory and performance demands than the C++ language itself. Used badly (e.g. poor use of multiple inheritance) C++ can lead to code that is harder to maintain and debug than C. However, used well, C++ is a far safer, robust and more extensible language than C, which should incur minimal overheads when being compared to a well written, well structured C program.

References

[C99] ISO/IEC 9899:1999, Programming Languages - C

[C90] ISO/IEC 9899:1990, Programming Languages - C

[C++98] ISO/IEC 14882, Standard for the C++ Language

[Coo03] Cooling, N.S. *UML for Embedded C.* ACCU Spring Conference 2004, www.accu.org

[Ell95] Ellis, M. and Stroustrup, B. *The Annotated C++ Reference Manual*, Addison-Wesley, 1995, ISBN 0-201-51459-1

[Lip96] Lippman, S. *Inside the C++ Object Model*, Addison-Wesley. 1996. ISBN 0-201-83454-5

[Mey98] Scott Meyers *Effective C++: 50 Specific Ways to Improve Your Programs and Design, 2nd Edition* Addison Wesley Professional 1998 ISBN: 0201924889

[Pin04] Pincus, J., and Baker, B. *Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns*, IEEE Security & Privacy, July/August 2004, Vol2 #4, pp20-27

[Str94] Stroustrup, B. *The Design and Evolution of C++.* Addison-Wesley. 1994. ISBN 0-201-54330-3

[Str97] Stroustrup, B. *The C++ Programming Language (3rd Edition).* Addison-Wesley. 1997. ISBN 0-201-88954-4

[Str02] Stroustrup, B. *Sibling Rivalry: C and C++.* Research Technical Report, AT&T Labs, January 2002. http://www.research.att.com/~bs/sibling_rivalry.pdf

[Sut99] Sutter, H. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions,* Reading,MA:Addison-Wesley Longman ISBN 0201615622

