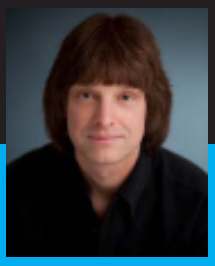


Scott Meyers

Presentation Materials

Effective C++ in an Embedded Environment

artima

Effective C++ in an Embedded Environment *Sample*

Thank you for downloading this sample from the presentation materials for Scott Meyers' *Effective C++ in an Embedded Environment* training course. If you'd like to purchase the complete copy of these notes, please visit:

http://www.artima.com/shop/effective_cpp_in_an_embedded_environment

Artima Press is an imprint of Artima, Inc.
P.O. Box 390122, Mountain View, California 94039

Cover photo by Stephan Jockel. Used with permission.

Effective C++ in an Embedded Environment

Scott Meyers, Ph.D.
Software Development Consultant

smeyers@aristeia.com
http://www.aristeia.com/

Voice: 503-638-6028
Fax: 503-974-1887

Last Revised: 10/4/12

These are the official notes for Scott Meyers' training course, "Effective C++ in an Embedded Environment". The course description is at http://www.aristeia.com/c++-in-embedded.html. Licensing information is at http://aristeia.com/Licensing/licensing.html.

Please send bug reports and improvement suggestions to smeyers@aristeia.com.

In these notes, references to numbered documents preceded by N (e.g., N3092) are references to C++ standardization document. All such documents are available via http://www.open-std.org/jtc1/sc22/wg21/docs/papers/.

[Comments in braces, such as this, are aimed at instructors presenting the course. All other comments should be helpful for both instructors and people reading the notes on their own.]

Important!

In this talk, I assume you know *all* of C++.

You may not.

When you see or hear something you don't recognize,
***please ask*!**

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Overview

Day 1 (Approximate):

- "C++" and "Embedded Systems"
- A Deeper Look at C++
  - ➡ Implementing language features
  - ➡ Understanding inlining
  - ➡ Avoiding code bloat
- 3 Approaches to Interface-Based Programming
- Dynamic Memory Management
- C++ and ROMability

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Overview

Day 2 (Approximate):

- Modeling Memory-Mapped IO
- Implementing Callbacks from C APIs
- Interesting Template Applications:
  - ➡ Type-safe void*-based containers
  - ➡ Compile-time dimensional unit analysis
  - ➡ Specifying FSMs
- Considerations for Safety-Critical and Real-Time Systems
- Further Information

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2012 Scott Meyers, all rights reserved.

Always on the Agenda

- **Your questions, comments, topics, problems, etc.**
  - ➧ Always top priority.

The primary course goal is to cover what *you want to know*.

- It doesn't matter whether it's in the prepared materials.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Overview

Day 1 (Approximate):

- "C++" and "Embedded Systems"
- A Deeper Look at C++
  - ➡ Implementing language features
  - ➡ Understanding inlining
  - ➡ Avoiding code bloat
- 3 Approaches to Interface-Based Programming
- Dynamic Memory Management
- C++ and ROMability

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

"C++"

Timeline and terminology:

- 1998: **C++98**: "Old" standard C++.
- 2003: C++03: Bugfix revision for C++98.
- 2005: **TR1**: Proposed extensions to standard C++ library.
  - ➡ Common for most parts to ship with current compilers.
  - ➡ Overview comes later in course.
- 2011: **C++11**: "New" standard C++.
  - ➡ Common for many parts to ship with latest compiler releases.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2012 Scott Meyers, all rights reserved.

"Embedded Systems"

Embedded systems using C++ are diverse:

▪ Real-time?	Maybe.
▪ Safety-critical?	Maybe.
▪ Challenging memory limitations?	Maybe.
▪ Challenging CPU limitations?	Maybe.
▪ No heap?	Maybe.
▪ No OS?	Maybe.
▪ Multiple threads or tasks?	Maybe.
▪ "Old" or "weak" compilers, etc?	Maybe.
▪ No hard drive?	Often.
▪ Difficult to field-upgrade?	Typically.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

[The goal of this slide is to get people to recognize that their view about what it means to develop for embedded systems may not be the same as others' views. The first time I taught this class, I had one person writing code for a 4-bit microprocessor used in a digital camera (i.e., a mass-market consumer device), and I also had a team writing real-time radar analysis software to be used in military fighter planes. The latter would have a very limited production run, and if the developers needed more CPU or memory, they simply added a new board to the system. Both applications were "embedded," but they had almost nothing in common.]

Developing for Embedded Systems

In general, little is "special" about developing for embedded systems:

- Software must respect the constraints of the problem and platform.
- C++ language features must be applied judiciously.

These are true for non-embedded applications, too.

- **Good embedded software development is just good software development.**

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Overview

Day 1 (Approximate):

- "C++" and "Embedded Systems"
- A Deeper Look at C++
  - ➡ Implementing language features
  - ➡ Understanding inlining
  - ➡ Avoiding code bloat
- 3 Approaches to Interface-Based Programming
- Dynamic Memory Management
- C++ and ROMability

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Implementing C++

Why Do You Care?

- You're just curious: how do they do that?
- You're trying to figure out what's going on while debugging.
- You're concerned: do they do that efficiently enough?
  - ➡ That's the focus of this presentation
  - ➡ Baseline: C size/speed

Have faith:

- C++ was designed to be competitive in performance with C.
- Generally speaking, you don't pay for what you don't use.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Implementing Virtual Functions

Abandon All Hope, Ye Who Enter!

- Compilers are allowed to implement virtual functions in any way they like:
  - There is no mandatory "standard" implementation
- The description that follows is *mostly* true for most implementations:
  - I've skimmed over a few details
  - None of these details affects the fact that virtual functions are typically implemented *very* efficiently

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2012 Scott Meyers, all rights reserved.

Implementing Virtual Functions

Consider this class:

```
class B {
public:
   B();

   virtual ~B();
   virtual void f1();
   virtual int f2(char c) const;
   virtual void f3(int x) = 0;

   void f4() const;
   ...
};
```

Compilers typically number the virtual functions in the order in which they're declared. In this example,

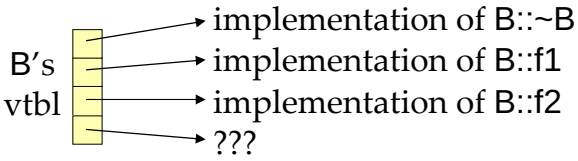
- The destructor is number 0
- f1 is number 1, f2 is number 2, f3 is number 3

Nonvirtual functions get no number.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Implementing Virtual Functions

A *vtbl* ("virtual table") will be generated for the class. It will look something like this:



Notes:

- The vtbl is an array of pointers to functions
- It points to virtual function implementations:
  - The *i*th element points to the virtual function numbered *i*
  - For pure virtual functions, what the entry is is undefined.
    - It's often a function that issues an error and quits.
- Nonvirtual functions (including constructors) are omitted:
  - Nonvirtual functions are implemented like functions in C

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

According to the "Pure Virtual Function Called" article by Paul Chisholm (see the "Further Information" slides at the end of the notes), the Digital Mars compiler does not always issue a message when a pure virtual function is called, it just halts execution of the program.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Aside: Calling Pure Virtual Functions

Most common way to call pure virtuals is in a constructor or destructor:

```
class B {
public:
   B() { f3(10); }                  // call to pure virtual
   virtual void f3(int x) = 0;
   ...
};
```

This is easy to detect; many compilers issue a warning.

The following case is trickier:

```
class B {
public:
   B() { f1(); }                    // call from ctor to "impure" virtual; looks safe
   virtual void f1() { f3(10); }    // call to pure virtual from non-ctor;  looks safe
   virtual void f3(int x) = 0;
   ...
};
```

Compilers rarely diagnose this problem.

For the first example, gcc 4.4-4.7 issue warnings. VC9-11 do not.

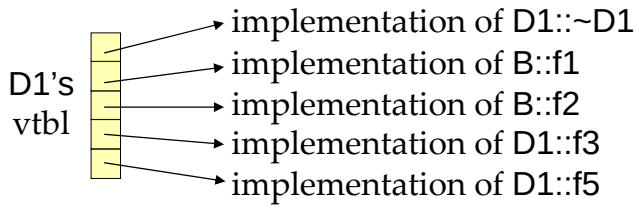
For the second example, none of the compilers issues a warning.

Implementing Virtual Functions

Now consider a derived class:

```
class D1: public B {
public:
   D1();                                  // nonvirtual
   virtual void f3(int x);                // overrides base virtual
   virtual void f5(const std::string& s); // new virtual
   virtual ~D1();                         // overrides base virtual
   ...
};
```

It yields a vtbl like this:



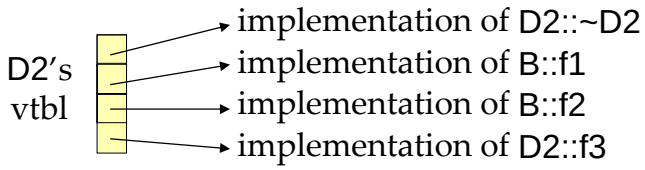
Note how corresponding function implementations have corresponding indices in the vtbl.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Implementing Virtual Functions

A second derived class would be treated similarly:

```
class D2: public B {
public:
   D2();
   virtual void f3(int x);
   ...
};
```

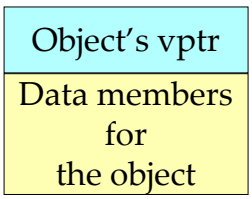


- D2's destructor is automatically generated by the compiler.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2012 Scott Meyers, all rights reserved.

Implementing Virtual Functions

Objects of classes with virtual functions contain a pointer to the class's vtbl:



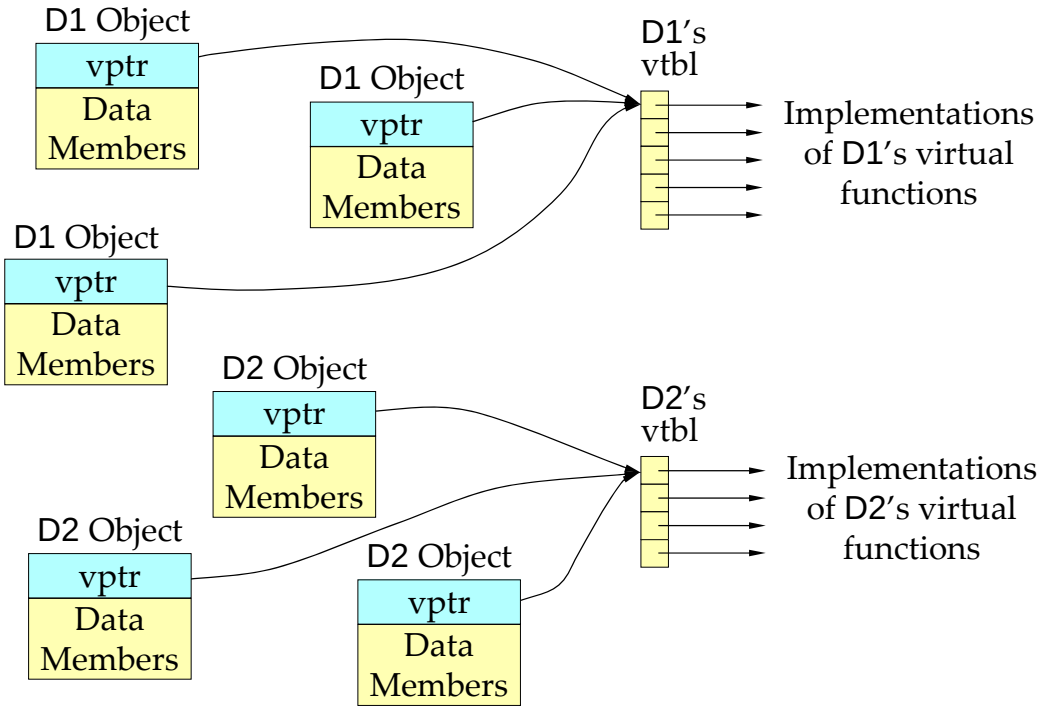
This pointer is called the *vptr* ("virtual table pointer").

- Its location within an object varies from compiler to compiler

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Implementing Virtual Functions

Vptrs point to vtbls:

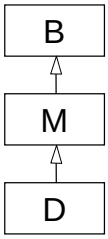


Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Implementing Virtual Functions

Vptrs are set by code compilers insert into constructors and destructors.

- In a hierarchy, each class's constructor sets the vptr to point to that class's vtbl
- Ditto for the destructors in a hierarchy.



Compilers are permitted to optimize away unnecessary vptr assignments.

- E.g., vptr setup for a D object could look like this:

```
D obj;

   Set vptr to B's vtbl;          // may be optimized away
   Set vptr to M's vtbl;          // may be optimized away
   Set vptr to D's vtbl;
   ...
   Set vptr to M's vtbl;          // may be optimized away
   Set vptr to B's vtbl;          // may be optimized away
```

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

B = "Base", M = "Middle", D = "Derived".

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2012 Scott Meyers, all rights reserved.

Implementing Virtual Functions

Consider this C++ source code:

```
void makeACall(B *pB)
{
   pB->f1();
}
```

The call to f1 yields code equivalent to this:

```
(*pB->vptr[1])(pB);          // call the function pointed to by
                             // vtbl entry 1 in the vtbl pointed
                             // to by pB->vptr; pB is passed as
                             // the "this" pointer
```

One implication:

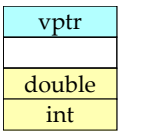
- When a virtual function changes, every caller must recompile!
  - e.g., if the function's order in the class changes
    - i.e., its compiler-assigned number.
  - e.g., if the function's signature changes.

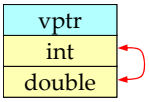
Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Implementing Virtual Functions

Size penalties:

- Vptr makes each object larger
  - ➡ Alignment restrictions could force padding
    - ◆ Reordering data members often eliminates problem





- Per-class vtbl increases each application's data space

Speed penalties:

- Call through vtbl slower than direct call:
  - ➡ But usually only by a few instructions
- Inlining usually impossible:
  - ➡ This is often inherent in a virtual call

But compared to C alternatives:

- *Faster and smaller* than if/then/else or switch-based techniques
- Guaranteed to be right

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

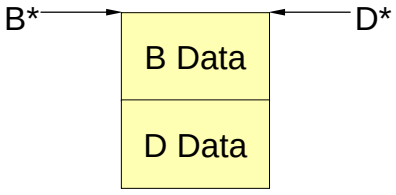
The diagram shows that if the first data member declared in a class has a type that requires double-word alignment (e.g., double or long double), a word of padding may need to be inserted after the vptr is added to the class. If the second declared data member is a word in size and requires only single-word alignment (e.g., int), reordering the data members in the class can allow the compiler to eliminate the padding after the vptr.

Object Addresses under Multiple Inheritance

Under SI, we can generally think of object layouts and addresses like this:

```
class B { ... };

class D: public B { ... };
```



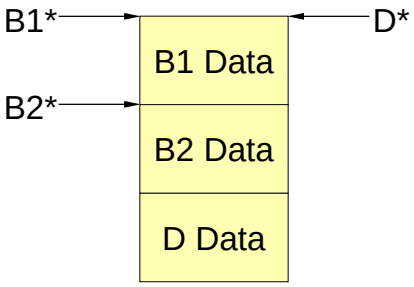
- An exception (with some compilers) is when D has virtual functions, but B doesn't.

Under MI, it looks more like this:

```
class B1 { ... };

class B2 { ... };

class D: public B1,
         public B2 { ... };
```



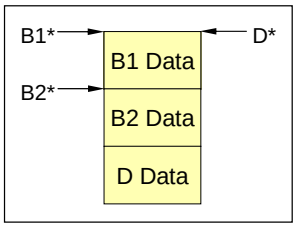
- D objects have multiple addresses:
  - ➡ One for B1* and D* pointers.
  - ➡ Another for B2* pointers.

SI = "Single Inheritance." MI = "Multiple Inheritance."

Object Addresses under Multiple Inheritance

There is a good reason for this:

```
void f(B1 *pb1);        // expects pb1 to point
                        // to the top of a B1
void g(B2 *pb2);        // expects pb2 to point
                        // to the top of a B2
```



Some calls thus require *offset adjustments*:

```
D *pd = new D;          // no adjustment needed
f(pd);                  // no adjustment needed
g(pd);                  // requires D* ⇒ B2* adjustment
B2 *pb2 = pd;           // requires D* ⇒ B2* adjustment
```

Proper adjustments require proper type information:

```
if (pb2 == pd) …                    // test succeeds (pd converted to B2*)
if ((void*)pb2 == (void*)pd) …      // test fails
```

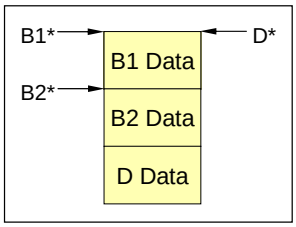
Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Null pointers never get an offset. At runtime, a pointer nullness test must be performed before applying an offset.

Virtual Functions under Multiple Inheritance

Consider the plight of your compilers:

```
class B1 {
public:
   virtual void mf();          // may be overridden in
   ...                         // derived classes
};
class B2 {
public:
   virtual void mf();          // may be overridden in
   ...                         // derived classes
};
void g(B2 *pb2)                // as before
{
   pb2->mf();                  // requires offset adjustment
}                              // before calling mf?
```



An adjustment is needed only if D overrides mf *and* pb2 really points to a D.

What should a compiler do? When generating code for the call,

- It may not know that D exists.
- It can't know whether pb2 points to a D.

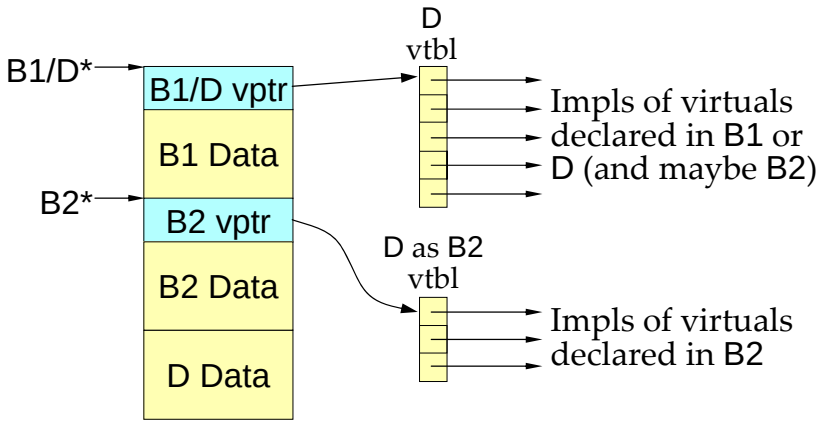
I don't remember the details, but both B1 and B2 need to declare mf for the information on this slide to be true for VC++. For g++, I believe it suffices for only B2 to declare mf.

Virtual Functions under Multiple Inheritance

The problem is typically solved by

- Creating special vtbls that handle offset adjustments.
- For derived class objects, adding new vptrs to these vtbls, one additional vptr for each base class after the first one:

```
class B1 { … };
class B2 { … };
class D:
   public B1,
   public B2 { … };
```



These special vptrs and vtbls apply only to derived class objects.

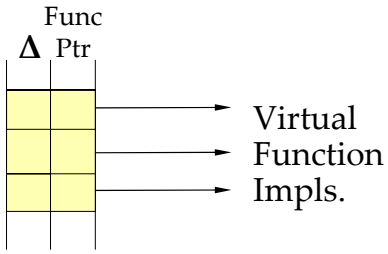
- Virtual functions for B1 and B2 objects are implemented as described before.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2012 Scott Meyers, all rights reserved.

Virtual Functions under Multiple Inheritance

Offset adjustments may be implemented in different ways:

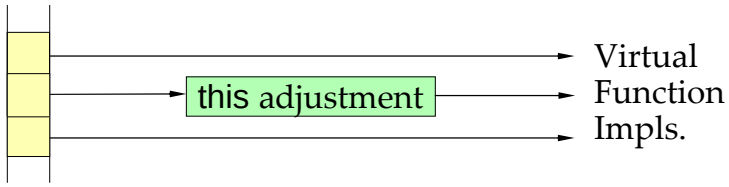
- Storing deltas in the vtbl:



(*pObject->vptr[index])(pObject+Δ);

  - ➧ Typically, most deltas will be 0, especially under SI.

- Passing virtual calls through thunks:

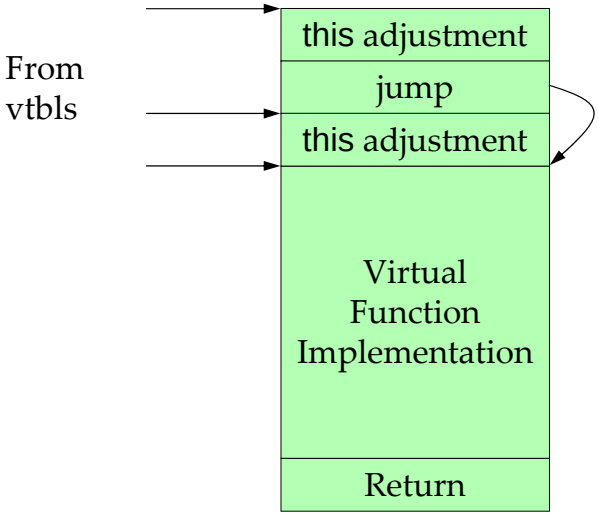


  - ➧ Thunks are generated only if an adjustment is necessary.
  - ➧ This approach is more common.

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Thunk Implementation

Often a function with multiple entry points:



Scott Meyers, Software Development Consultant
http://www.aristeia.com/

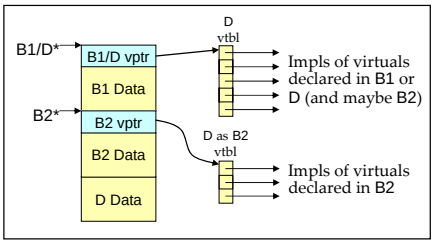
Slide 28

I'm guessing about the jump in the diagram. An alternative would be for one thunk to fall through to the next, with the sum of the offset adjustments calculated to ensure that the proper this value is in place when the function body is entered.

Virtual Functions under Multiple Inheritance

The details of vtbl layout and usage under MI vary from compiler to compiler.

- When a virtual is inherited from only a non-leftmost base, it may or may not be entered into both vtbls:




```
class B1 { ... };            // declares no mf

class B2 {
public:
   virtual void mf();
   ...
};

class D: public B1, public B2 { ... };

D *pd = new D;

pd->mf();                    // may use either B2's or D's vptr,
                             // depending on the compiler
```


As I recall, g++ enters the function into both vtbls, but VC++ enters it into only the vtbl for B2. This means that the call in red shown above would use the B2 vtbl under VC++, and that means that there'd be a D*→B2* offset adjustment made prior to calling through the B2 vtbl.

Virtual Functions, MI, and Virtual Base Classes

The general case involves:

- Virtual base classes with nonstatic data members.
- Virtual base classes inheriting from other virtual base classes.
- A mixture of virtual and nonvirtual inheritance in the same hierarchy.

Lippman punts:

> *Virtual base class support wanders off into the Byzantine...*
> *The material is simply too esoteric to warrant discussion...*

I punt, too :-)

The quote is from Lippman's *Inside the C++ Object Model*, for which there is a full reference in the "Further Information" slides at the end of the notes.