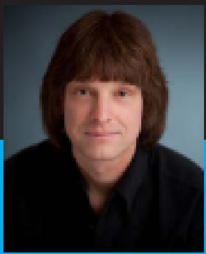


Scott Meyers

Presentation Materials

Effective C++ in an Embedded Environment

artima

Effective C++ in an Embedded Environment Version 2

Artima Press is an imprint of Artima, Inc.
P.O. Box 305, Walnut Creek, California 94597

First version published April 26, 2010
This version published October 4, 2012
Produced in the United States of America

Cover photo by Stephan Jockel. Used with permission.

Effective C++ in an Embedded Environment

Scott Meyers, Ph.D.
Software Development Consultant

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*!**

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

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
© 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

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

"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
© 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.

[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

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
© 2012 Scott Meyers, all rights reserved.

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

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

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,

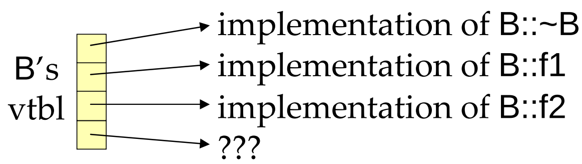
- The destructor is number 0
- f1 is number 1, f2 is number 2, f3 is number 3

Nonvirtual functions get no number.

Scott Meyers, Software Development Consultant

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

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.

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.

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

Slide 15

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

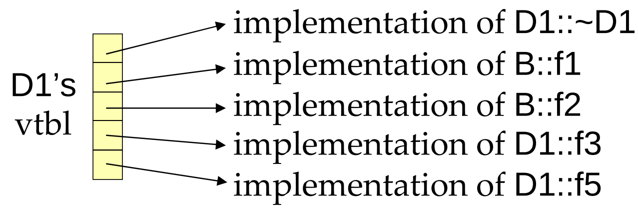
For the second example, none of the compilers issues a warning.

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:



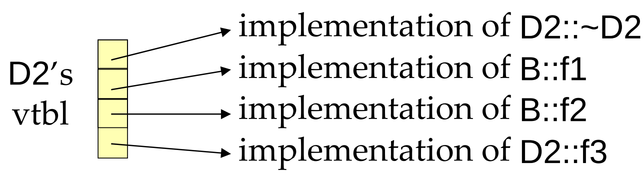
Note how corresponding function implementations have corresponding indices in the vtbl.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Implementing Virtual Functions

A second derived class would be treated similarly:

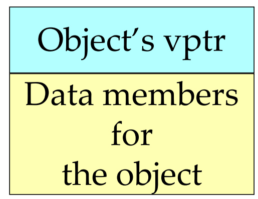
```
class D2: public B {
public:
  D2();
  virtual void f3(int x);
  ...
};
```



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

Implementing Virtual Functions

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



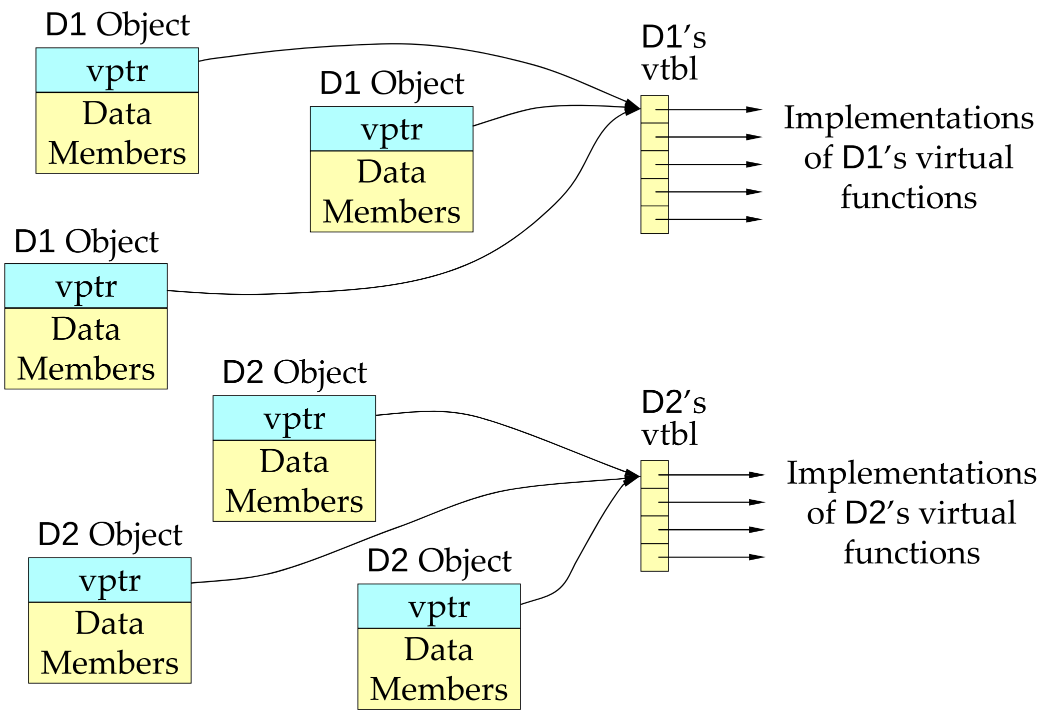
This pointer is called the *vptr* ("virtual table pointer").

- Its location within an object varies from compiler to compiler

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Implementing Virtual Functions

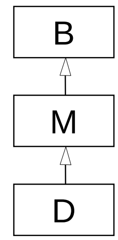
Vptrs point to vtbls:



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".

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:

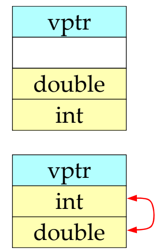
- When a virtual function changes, every caller must recompile!
  - e.g., if the function’s order in the class changes
    - i.e., its compiler-assigned number.
  - e.g., if the function’s signature changes.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

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/

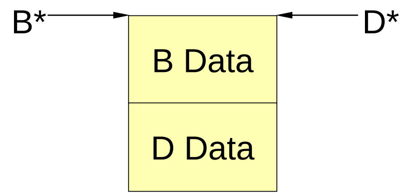
Slide 22

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:

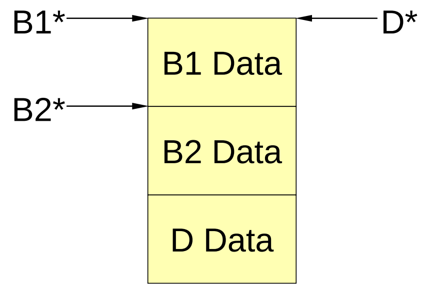
```
class B { ... };

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 { ... };
```

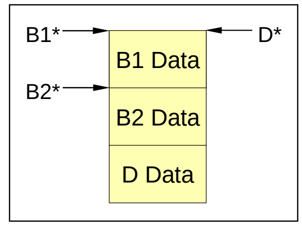


- D objects have multiple addresses:
  - One for B1* and D* pointers.
  - Another for B2* pointers.

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
```

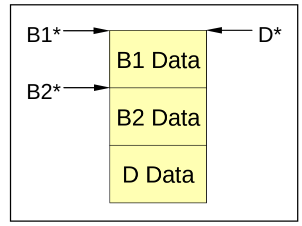
Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Slide 24

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.

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

Slide 25

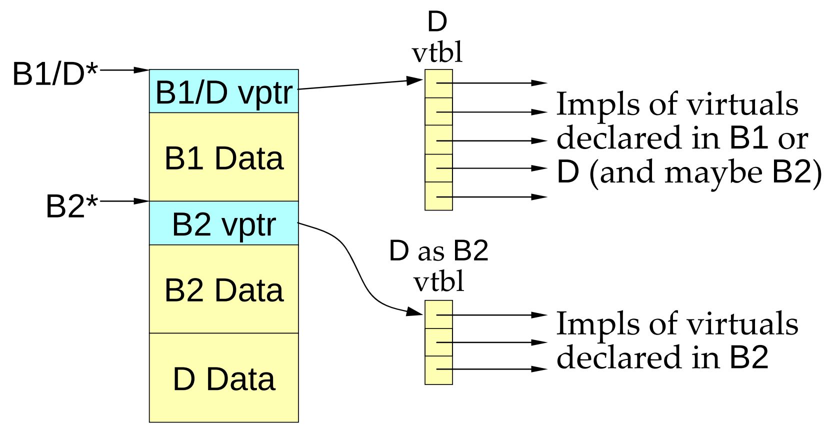
I don't remember the details, but both B1 and B2 need to declare mf for the information on this slide to be true for VC++. For g++, I believe it suffices for only B2 to declare mf.

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.

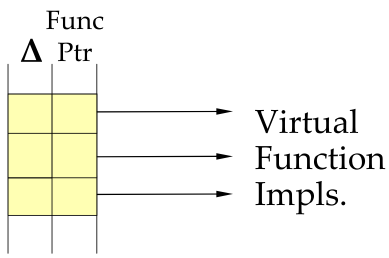
- Virtual functions for B1 and B2 objects are implemented as described before.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Virtual Functions under Multiple Inheritance

Offset adjustments may be implemented in different ways:

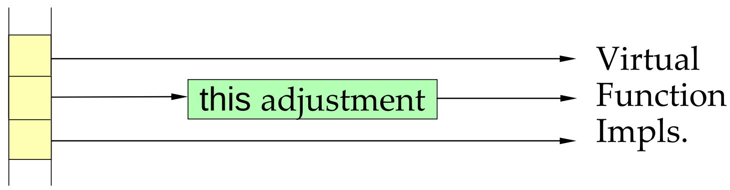
- Storing deltas in the vtbl:




```
(*pObject->vptr[index])(pObject+Δ);
```

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

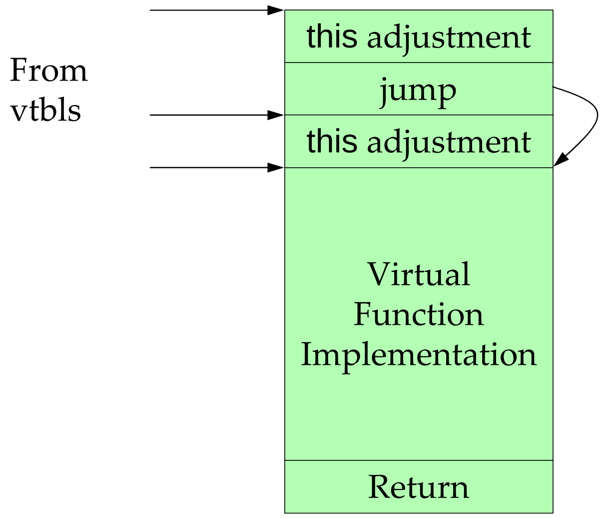
- Passing virtual calls through thunks:



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

Thunk Implementation

Often a function with multiple entry points:

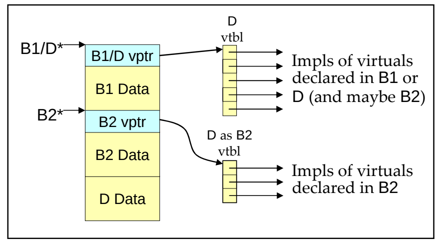


I'm guessing about the jump in the diagram. An alternative would be for one thunk to fall through to the next, with the sum of the offset adjustments calculated to ensure that the proper this value is in place when the function body is entered.

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
```

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

Slide 29

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 :-)

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

Slide 30

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.

"No-Cost" C++ Features

These exact a price only during compilation. In object code, they look like C:

- All the C stuff: structs, pointers, free functions, etc.
- Classes
- Namespaces
- Static functions and data
- Nonvirtual member functions
- Function and operator overloading
- Default parameters:
  - Note that they are *always* passed. Poor design can thus be costly:

```
void doThat(const std::string& name = "Unnamed");            // Bad

const std::string defaultName = "Unnamed";
void doThat(const std::string& name = defaultName);          // Better
```

  - Overloading is typically a cheaper alternative.

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

Slide 31

This slide begins a summary of the costs of various C++ language features (compared to C).

More "No-Cost" C++ Features

These look like they cost you something, but in truth they rarely do (compared to equivalent C behavior):

- Constructors and destructors:
  - They contain code for *mandatory* initialization and finalization.
  - However, they may yield chains of calls up the hierarchy.
- Single inheritance
- Virtual functions
  - Abstract classes with no virtual function implementations (i.e., "Interfaces") may still generate vtbls.
    - Some compilers offer ways to prevent this.
- Virtual inheritance

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

Both MS and Comeau offer the __declspec(novtable) mechanism to suppress vtbl generation and vptr assignment for Interface classes. Apparently the Sun compiler will optimize away unnecessary vtbls in some cases without any manual user intervention. From what I can tell, as of gcc 4.x, there is no comparable feature in g++.

Still More "No-Cost" C++ Features

- new and delete:
  - By default, new = malloc + constructor(s) and delete = destructor(s) + free
  - Note that error-handling behavior via exceptions is built in.

Important: new is useful even in systems where all memory is statically allocated.

- *Placement* new allows objects to be constructed at particular locations:
  - E.g., in statically allocated memory.
  - E.g., at memory-mapped addresses.
- We'll see examples later.

Note: for all of the preceding features, if you don't use them, you don't pay.

Scott Meyers, Software Development Consultant

"Low-Cost" C++ Features

You may pay for these features, even if you don't use them:

- Exceptions: a small speed and/or size penalty (code)
  - When evaluating the cost of exceptions, be sure to do a fair comparison.
  - Error handling costs you something, no matter how it is implemented.
    - E.g., Saks reports object code increases of 15-40% for error handling based on return values.

Details on Dan Saks' analysis is in the *Embedded Systems Design* article referenced in the "Further Information" slides at the end of the notes.

Approaches to Implementing Exceptions

Consider the problem of local object destruction:

```
{
  V1
  ...
  {
    ...
    V2
    V3
    ...
  }
  ...
  V4
  V5
  ...
}
```

Which objects should be destroyed if an exception is thrown?

- There are two basic approaches to keeping track.

Approaches to Implementing Exceptions

One is to keep a shadow stack of objects requiring destruction if an exception is thrown.

- Code size increases to include instructions for manipulating the shadow stack.
- Runtime data space increases to hold the shadow stack.
- Program runtime increases to allow for shadow stack manipulations.
- Performance impact?
  - Unknown. Apples-to-apples comparisons are hard to come by.
  - Ballpark guesstimate: 5-10% hit in both time and space.
    - “Guesstimate” = “Speculation”

This is sometimes known as the “Code Approach.”

- Microsoft uses it for 32 bit (but not 64 bit) Windows code.
- g++ distributions for Windows use it, too.

Scott Meyers, Software Development Consultant

Approaches to Implementing Exceptions

The alternative maps program regions to objects requiring destruction:

```
{
  V1                     R0
---------------------------
  ...
  {                      R1
    ...
    V2
---------------------------
    V3                   R2
---------------------------
    ...                  R3
  }
---------------------------
  ...                    R4
  V4
---------------------------
  V5                     R5
---------------------------
  ...                    R6
}
---------------------------
```

Region	Objects
R0	None
R1	V1
R2	V1, V2
R3	V1, V2, V3
R4	Same as R1
R5	V1, V4
R6	V1, V4, V5

- This analysis is simplified, e.g., it ignores the possibility that destructors may throw.
- Most compilers for Unix use this approach. The 64 bit Itanium ABI also uses it.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Approaches to Implementing Exceptions

Implications of this "Table Approach:"

- Program speed is unaffected when no exceptions are thrown.
- Program size increases due to need to store the code to use the tables.
- Static program size increases due to need to store the tables.
  - When no exception is thrown, these tables need not be in memory, in working set, or in cache.
- Throwing exceptions is *slow*:
  - Tables must be read, possibly after being swapped in, possibly after being uncompressed.
  - However, throwing exceptions should be ... exceptional.

Scott Meyers, Software Development Consultant

Exceptions and Dynamically Allocated Memory

Some compilers try to use heap memory for exception objects.

- This can be unacceptable in some embedded systems.

Don't let this scare you:

- Implementations reserve some non-heap memory for exception objects.
  - ➨ They have to be able to propagate std::bad_alloc exceptions!
  - ➨ Platforms with no heap should still be able to use exceptions.

One platform that uses heap memory for exceptions (when it can) is g++.

"Low-Cost" C++ Features

More features you may pay for, even if you don't use them:

- Multiple inheritance: a small size penalty (vtbls that store Δs)
- dynamic_cast and other RTTI features: a small size penalty (vtbls)
  - Each *use* of dynamic_cast may be linear in the number of base classes (direct and indirect) of the object being cast.
    - Each use may involve a call to strcmp for each class in the hierarchy.
  - QOIs vary. The *Technical Report on C++ Performance* provides details.

"QOI" = "Quality of Implementation"

C++ Features that can Surprise Inexperienced C++ Programmers

These can cost you if you're not careful:

- Temporary objects, e.g., returned from `a+b`:
  - ➡ Many techniques exist to reduce the number and/or cost of such temporaries.
  - ➡ I'll provide some references at the end of this talk.
- Templates:
  - ➡ We'll discuss techniques based on inheritance and `void*`-pointers that can eliminate code bloat.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Common Questions

Why are simple "hello world" programs in C++ so big compared to C?

- iostream vs. stdio
- "hello world" is an atypical program:
  - ➡ For small programs, C++ programmers can still use stdio

Why do C developers moving to C++ often find their code is big and slow?

- C++ isn't C, and C programmers aren't C++ programmers
- C++ from good C++ developers as good as C from good C developers

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Efficiency Beyond C

C++ can be *more efficient* than C:

- C++ feature implementation often better than C approximations:
  - E.g., virtual functions
- Abstraction + encapsulation ⇒ flexibility to improve implementations:
  - std::strings often outperform char*- based strings:
    - May use reference counting
    - May employ "the small string optimization"
- STL-proven techniques have revolutionized library design:
  - Shift work from runtime to compile-time:
    - Template metaprogramming (TMP), e.g., "traits"
    - Inlined operator()s
  - Sample success story: C++'s sort is faster than C's qsort.

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

Slide 43

Reference-counting implementations for std::string are valid in C++03, but not in C++11.

C++ Implementation Summary

- C++ designed to be competitive with C in size and speed
- Compiler-generated data structures generally better than hand-coded C equivalents
- You generally don't pay for what you don't use
- C++ is successfully used in many embedded systems, e.g.:
  - Mobile devices (e.g., cell phones, PDAs)
  - Air- and Spacecraft
  - Medical devices
  - Video game consoles
  - Networking/telecom hardware (e.g., routers, switches, etc.)
  - Shipping navigations systems

Scott Meyers, Software Development Consultant

Overview

Day 1 (Approximate):

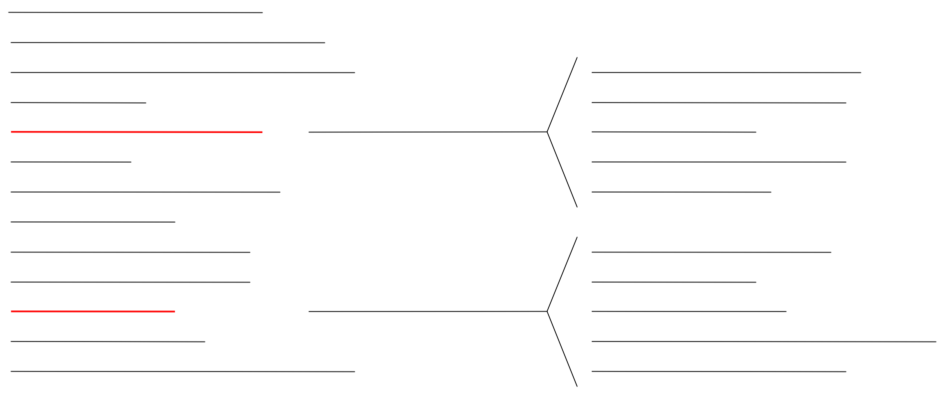
- "C++" and "Embedded Systems"
- A Deeper Look at C++
  - Implementing language features
  - Understanding inlining
  - Avoiding code bloat
- 3 Approaches to Interface-Based Programming
- Dynamic Memory Management
- C++ and ROMability

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

The Pros and Cons of Inlining

Advantages of inlining:

- Function call overhead is eliminated:
  - ➡ For very small functions, overall code size may shrink!
  - ➡ Essential for decent performance in layered systems
- Allows modular source code with branch-free object code.
  - ➡ Function calls in source code yield straight-line object code.
- Often allows for better object code optimization by compilers:



The lines in the diagram represent intermediate code generated by the compiler. Black lines are not function calls, red lines are. The two red lines on the left expand into the black lines on the right if the calls represented by the red lines are inlined.

The Pros and Cons of Inlining

Disadvantages:

- Debuggers can't cope:
  - ➡ How do you set a breakpoint in a function that doesn't exist?
- Overall system code size typically increases.
  - ➡ This can decrease cache hit rate or increase paging.
- Constrains binary compatibility for upgrade releases.

The Pros and Cons of Inlining

- Some "small" functions may result in a lot of code being generated:
  - Overhead to support EH may be significant
  - Constructors may set vptrs, call base class constructors, etc.

```
class Base {
   T1 x, y;
   ...
};

class Derived: public Base {
   T2 z;

public:
   Derived(){}
   ...
};
```

`Derived(){}` generates:

1. If on heap, call operator new
2. Call Base::Base
3. Make vptr point to Derived vtbl
4. Call constructor for z

Scott Meyers, Software Development Consultant

The Pros and Cons of Inlining

inline is only a request — compilers are free to ignore it:

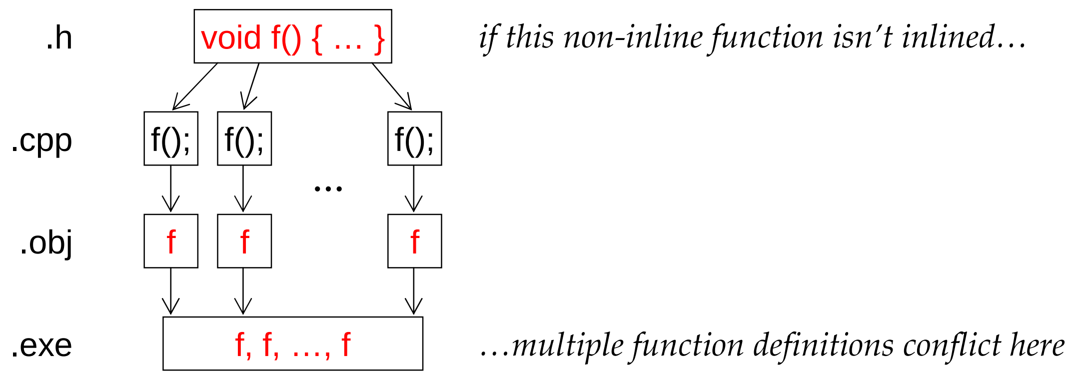
- Compilers rarely inline virtual function calls:
  - Inlining occurs at build-time, but virtuals are resolved at runtime.
  - Optimizations are sometimes possible:
    - Virtuals invoked on *objects* (not pointers or references).
    - Explicitly qualified calls (e.g., *ClassName::*virtualFunctionName()).
- Compilers often ignore inline for "complex" functions, e.g., those containing loops
- Compilers must ignore inline when they need a pointer to the function, e.g., constructors and destructors for arrays of objects

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Automatic Inlining

Compilers may inline functions not declared inline, but this is uncommon.

- To inline a function, compilers need its definition, but non-inline functions are not defined in header files.
  - ➨ They'd cause duplicate symbol errors during linking:



- Non-inline functions are thus *declared* in headers, not defined there.
- The rules for function templates are a bit different….

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

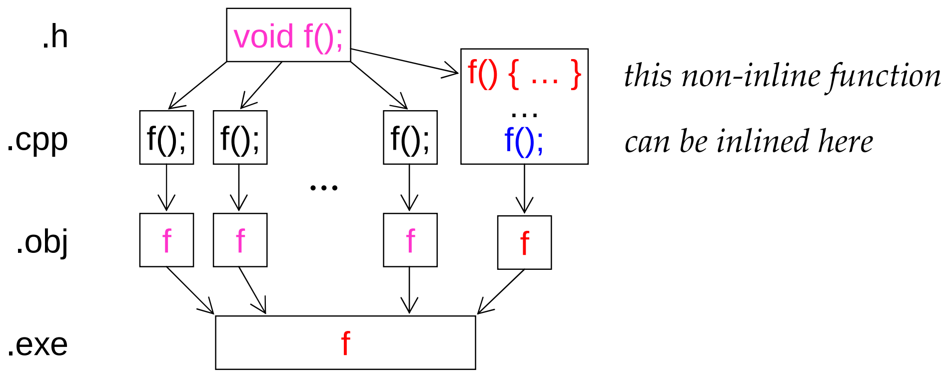
Slide 50

Code in red is function definitions, code in black is function calls.

Automatic Inlining

Compilers rarely inline functions only declared in headers.

- They need to know the function body to inline it.
  - ➡ When they do know it, inlining is easy (and common).
    - ◆ E.g., in the .cpp file defining the function:



Code in red is function definitions, code in magenta is function declarations (or in an object file, references to external symbols), code in black is un-inlined function calls, code in blue is inlined function calls.

Link-Time Inlining

Linkers may also perform inlining:

- Many already do (with appropriate options enabled).
  - E.g., Microsoft, Gnu, Intel, Sun.

Still, manual inline declarations remain a necessary evil.

Options that enable link-time inlining are typically named *whole program optimization* (WPO) or *link-time optimization* (LTO).

Link-time optimization became available in gcc as of version 4.5.

The Pros and Cons of Inlining

Bottom line:

- Inlining is almost always a good bet for small, frequently called functions.
  - ➨ Overall runtime speed is likely to increase.
- Imprudent inlining can lead to code bloat.
- Minimize inlining if binary upgradeability is important.

Scott Meyers, Software Development Consultant

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

Code Bloat in C++

C++ has a few features you pay for (in code size and/or runtime speed), even if you don't use them:

- Support for exceptions.
- Support for generalized customizable iostreams.
  - I.e., streams of other than `char` or `wchar_t`.

These things may reasonably be considered bloat.

Possible workarounds:

- Disable exceptions during compilation.
  - Practical only if you know that no code (including libraries, plug-ins, etc.) throws.
- Use stdio instead of iostreams.

Scott Meyers, Software Development Consultant

Code Bloat in C++

However, most bloat accusations are unfair, traceable to either:

- Comparing functionality in C++ with lesser functionality in C:
  - E.g., C++ virtual functions do more than C functions.
- Improper use of the language:
  - E.g., Putting inessential code in constructors/destructors.

The feature most associated with bloat is templates.

- That's what I'll focus on here.
- Most problems with "template code bloat" arise from:
  - Misunderstandings of template rules.
  - Improper use of templates.

Scott Meyers, Software Development Consultant

Templates, Header Files, and Inlining

Consider:

```
template<typename T>              // header file for a class
class SomeClass {                 // template
public:
   SomeClass() { ... }            // implicitly declared inline
   void mf1() { ... }             // implicitly declared inline
   void mf2();                    // not implicitly declared inline
   ...
};

template<typename T>              // template funcs are typically
void SomeClass<T>::mf2() { ... }  // defined in header files, but
                                  // this does not automatically
                                  // declare them inline
```

Critical:

- Don't declare template functions inline simply because they are defined in headers.
  - ➡ Unnecessary inlining will lead to bloat.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Templates, Header Files, and Inlining

Templates need not be defined in headers:

```
template<typename T>
class SomeClass {
public:
   SomeClass() { ... }                   // still implicitly inline
   void mf1() { ... }                    // still implicitly inline
   void mf2();                           // declaration only; no definition
   ...                                   //   provided in this file
};
```

Code using this header will compile fine.

- But if SomeClass::mf2 is called, it won't link.
  - We'll cover how to fix that in a moment.
- Templates are typically defined in header files to avoid such problems.

Templates, Header Files, and Inlining

The convention of putting all template code in headers has an advantage:

- Single point of change for client-visible code, e.g., function declarations.
  - ➡ No need to change both header and implementation files.

And some disadvantages:

- Increased compilation times.
- Increased compilation dependencies for clients.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Instantiating Templates

Templates that aren't used aren't instantiated.

- They thus generate no code and no data.
  - ➡ But the need to read template headers usually slows compilation.
- Templates can thus generate *less* code than non-templates!

```
class C {                  // Even if C is never used, object files
public:                    // typically contain f1..fn. Few linkers
  void f1();               // will remove all code and data related
  ...                      // to uncalled functions.
  void fn();
};

template<typename T>       // Object files should contain only those
class C {                  // functions that are called.
public:
  void f1();
  ...
  void fn();
};
```

➡ Templates can thus help avoid linking dead code into executables.

Safety-critical systems often require the elimination of dead code, so the fact that templates can avoid generating it in the first place is attractive to people developing such systems.

Instantiating Templates

Instantiated templates may generate both code and data:

```
SomeClass<int> sc;                 // SomeClass<int> instantiated;
                                   // some code generated, memory
                                   // for static class data set aside
```

Instantiating a class shouldn't instantiate all its member functions:

- Only member functions that are used should be instantiated.
  - ➡ You shouldn't pay for what you don't use.
- A few compilers (typically older ones) get this wrong.
  - ➡ They instantiate all member functions of a class if any is used.
  - ➡ We'll discuss how to avoid this in a moment.

Instantiating Templates

Most templates are *implicitly instantiated*:

- Compiler notes used functions, instantiates them automatically.
- To create the functions, it needs access to their definitions.
  - This is why template code is typically in header files.
- Without a definition, compiler generates reference to external symbol.
  - Hence SomeClass::mf2 callable w/o a definition, but a link-time error will result.

Templates can also be *explicitly instantiated*:

- You can force a class or function template to be instantiated.
  - For class templates, *all* member functions are instantiated.
  - Individual member functions can also be instantiated.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Explicit Instantiation

In a .h file:

```
template<typename T>                  // as before
class SomeClass {
public:
   SomeClass() { ... }
   void mf1() { ... }
   void mf2();
   ...
};
```

In a .cpp file:

```
...                                   // Definitions of SomeClass's
                                      // non-inline functions go here

template                              // explicitly instantiate all SomeClass
class SomeClass<double>;              // mem funcs for T=double; compiled
                                      // code will go in this .cpp's .obj file

template                              // explicitly instantiate SomeClass::mf2
void SomeClass<int>::mf2();           // for T=int; compiled code will go in
                                      // this .cpp's .obj file
```

Explicit Instantiation

Explicit instantiation can be a lot of work:

- You must manually list each template and set of instantiation parameters to be instantiated.

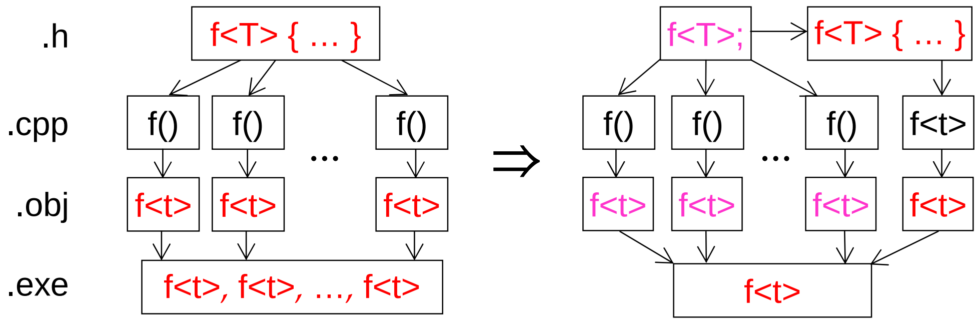
But it can be useful:

- To create libraries of instantiations.
- To put instantiations into particular code sections.
- To avoid code bloat arising from bad compilers/linkers.
  - ➡ Details on next page.

Explicit Instantiation

Your executable might end up with multiple copies of an instantiation:

- If your compiler (incorrectly) instantiates all class template member functions when only some are used.
- If your linker is bad:



- If you use dynamic linking.

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

Slide 65

In the diagrams, code in red is function definitions, code in magenta is function declarations (or in an object file, references to external symbols), and code in black is function calls.

Avoiding Code Duplication

Consider:

```
class IntClass {
public:
  void usageInfo(std::ostream& s);           // summarize usage info to s
  ...
};

class DoubleClass {
public:
  void usageInfo(std::ostream& s);           // summarize usage info to s
  ...
};
```

Both usageInfo functions will do essentially the same thing.

- This is code duplication.
- It leads to code bloat.

Note that no templates are involved here.

Avoiding Code Duplication

A common way to eliminate such duplication is to move the duplicated code to a base class:

```
class Base {
public:
  void usageInfo(std::ostream& s);          // summarize usage info to s
  ...
protected:
  ...                                       // data storing usage info
};

class IntClass: public Base {
  ...                                       // no declaration of usageInfo
};

class DoubleClass: public Base {
  ...                                       // no declaration of usageInfo
};
```

Now there's only one copy of usageInfo in the program, regardless of how many classes inherit from Base.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Avoiding Code Duplication

Exactly the same reasoning applies when writing templates:

```
template<typename T>                  // a template leading to bloat
class SomeClass {
  ...
  void usageInfo(std::ostream& s);    // leads to code duplication if
  ...                                 // usageInfo makes no use of T
};
```

The solution is the same:

```
class Base {                          // same as on previous page
public:
  void usageInfo(std::ostream& s);
  ...
};

template<typename T>                  // a template avoiding bloat
class SomeClass: public Base {
  ...                                 // no declaration of usageInfo
};
```

Avoiding Code Duplication

Moving type-invariant code into a base class sometimes called *code hoisting*.

It can help avoid code bloat due to multiple pointer types:

```
template<typename T>                  // general template
class Stack { ... };

class GenericPtrStack { ... };        // non-template using void*s

template<typename T>                  // partial specialization for
class Stack<T*>:                      // pointers; uses void*-based
  private GenericPtrStack {           // base class for all real work

  ...                                 // all inline casting functions;
};                                    // they generate no code
```

All Stack instantiations for pointer types thus share their code.

- We'll see this example in detail later.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Avoiding Code Duplication

Code hoisting works well with inlining to avoid duplication arising from non-type template parameters:

```
template<typename T, std::size_t BUFSZ>   // Suspect design: each
class Buffer {                            // BUFSZ value will yield a
  T buffer[BUFSZ];                        // new set of member functions

public:
  ...
};


template<typename T>                      // Better design: BufferBase
class BufferBase {                        // is independent of BUFSZ
  ...
};

template<typename T, std::size_t BUFSZ>   // Buffer does only BUFSZ-
class Buffer: public BufferBase<T> {      // dependent operations.
  ...                                     // Ideally, all are inline, so
};                                        // Buffer classes cost nothing
```


Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Avoiding Code Duplication

Avoiding code bloat with templates fundamentally calls for disciplined *commonality and variability analysis*:

- The parts of a template that don't depend on the template parameters (the *common* parts) should be moved out of the template.
- The remaining parts (the *variable* parts) should stay in the template.

This kind of analysis is critical to avoiding code duplication in any guise:

- Features common to multiple classes should be moved out of the classes.
  - Maybe to a base class.
  - Maybe to a class template.
- Features common to multiple functions should be moved out of the functions:
  - Maybe to a new function.
  - Maybe to a function template.

Need to distinguish here between source code duplication and object code duplication. Templates and inlines can reduce source code duplication, but can lead to object code duplication.

Code Bloat Summary

Most bloat can be eliminated by careful design. Arrows in your quiver:

- Consider disabling support for exceptions.
- Consider stdio instead of iostreams.
- Avoid excessive inlining, especially with templates.
- Judiciously use explicit instantiation to avoid code duplication.
- Hoist parameter-independent code out of templates.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Dealing with Function Templates

We've discussed only class templates, but bloat elimination techniques for function templates are similar:

```
template<typename T>            // template leading to bloat
void doSomething(const T& obj)
{
  ...                           // code making use of T or obj
  ...                           // code independent of T or obj
  ...                           // code making use of T or obj
}
```

A "hoisting" alternative:

```
void doSomethingHelper();       // "hoisted" code in non-template
                                // function; not inline

template<typename T>            // revised template avoiding bloat
void doSomething(const T& obj)
{
  ...                           // code making use of T or obj
  doSomethingHelper();
  ...                           // code making use of T or obj
}
```


Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Data Bloat

Not all bloat is due to code. Unnecessary classes can yield data bloat, too:

- Some classes have a vtbl, so unnecessary classes ⇒ unnecessary vtbls.
  - Such unnecessary classes could come from templates.
- Functions must behave properly when exceptions are thrown, so unnecessary non-inline functions ⇒ unnecessary EH tables.
  - Such unnecessary functions could come from templates.
  - This applies only to the Table Approach to EH.

An important exception to these issues are class templates that:

- Contain only inline functions.
  - Hence no extra EH tables.
- Contain no virtual functions.
  - Hence no extra vtbls.

We'll see examples of such "bloat-free" templates later.

Scott Meyers, Software Development Consultant

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

Interface-Based Programming

Interface-based programming:

- Coding against an interface that allows multiple implementations.
  - ➡ Function interface.
  - ➡ Class interface.
- Client code unaware which implementation it uses.
  - ➡ It depends only on the interface.

Scott Meyers, Software Development Consultant

Polymorphism

Polymorphism:

- The use of multiple implementations through a single interface.

Key question: **when is it known which implementation should be used?**

- **Runtime:** each *call* may use a different implementation.
  - ➡ Use inheritance + virtual functions.
- **Link-time:** each *link* may yield a different set of implementations.
  - ➡ Use separately compiled function bodies.
  - ➡ Applies to both static and dynamic linking.
- **Compile-time:** each *compilation* may yield a different set of implementations.
  - ➡ Use computed typedefs.

Scott Meyers, Software Development Consultant

Runtime Polymorphism

- The "normal" meaning of interface-based programming.
  - In much OO literature, the only meaning.
    - Unnecessarily restrictive for C++.
- The most flexible.
  - Can take advantage of information known only at runtime.
- The most expensive.
  - Based on vptrs, vtbls, non-inline function calls.

Scott Meyers, Software Development Consultant

Runtime Polymorphism Example

```
class Packet {                          // base class ("interface")
public:
  …
  virtual bool isWellFormed() const = 0;
  virtual std::string payload() const = 0;
  …
};

class TCPPacket: public Packet {        // derived class ("implementation")
  …
  virtual bool isWellFormed() const;
  virtual std::string payload() const;
  …
};

class CANPacket: public Packet {        // derived class ("implementation")
  …
  virtual bool isWellFormed() const;
  virtual std::string payload() const;
  …
};
```

Runtime Polymorphism Example

```
std::auto_ptr<Packet>                          // factory function;
nextPacket( /* params */ );                    // generate next packet

…

std::auto_ptr<Packet> p;
while (p = nextPacket( /* params */), p.get() != 0) {
  if (p->isWellFormed()) {                     // use Packet interface

    …

  }

  …
}
```

Runtime polymorphism is reasonable here:

- Types of packets vary at runtime.

Note: As of C++11, std::unique_ptr is preferable to std::auto_ptr, and nullptr is preferble to 0.

Link-Time Polymorphism

- Useful when information known during linking, but not during compilation.
- No need for virtual functions.
- Typically disallows inlining.
  - ➡ Most inlining is done during compilation.

Link-Time Polymorphism Example

Software can be deployed on two kinds of boxes:

- Expensive, high-performance box.
  - Uses expensive, fast components.
- Cheaper, lower-performance box.
  - Uses cheaper, lower-performance components.
- Essentially the same software runs on both boxes.
  - Component driver implementations differ.
    - A common interface can be defined.

Approach:

- One class definition for both drivers.
- Different component-dependent implementations.
- Implementations selected during linking.
  - This is "C" polymorphism.

Scott Meyers, Software Development Consultant

Link-Time Polymorphism Example

device.h:

```
namespace Drivers {
  class Impl;
  class DeviceDriver {              // all nonvirtual non-inline functions
  public:
    DeviceDriver();
    ~DeviceDriver();
    void reset();
    …
  private:
    Impl *pImpl;                    // ptr to data for driver
  };
}
```

All client code #includes this header and codes against this class.

- Note lack of virtual functions.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Link-Time Polymorphism Example

EFDevice.cpp (generates EFDevice.o, EFDevice.obj, or EFDevice.dll, etc.):

- EFDevice = "Expensive Fast Device"

```
namespace Drivers {
  struct Impl { … };                  // data needed by EFDevice driver
  DeviceDriver::DeviceDriver()        // ctor code for EFDevice
  { … }
  DeviceDriver::~DeviceDriver()       // dtor code for EFDevice
  { … }
  void DeviceDriver::reset()          // reset code for EFDevice
  { … }
  …
}
```

All functions in this file have access to the Impl struct defined here.

Link-Time Polymorphism Example

CSDevice.cpp (generates CSDevice.o, CSDevice.obj, or CSDevice.dll, etc.):

- CSDevice = "Cheap Slow Device"

```
namespace Drivers {
  struct Impl { … };                     // data needed by CSDevice driver
  DeviceDriver::DeviceDriver()           // ctor code for CSDevice
  { … }
  DeviceDriver::~DeviceDriver()          // dtor code for CSDevice
  { … }
  void DeviceDriver::reset()             // reset code for CSDevice
  { … }
  …
}
```

All functions in this file have access to the Impl struct defined here.

- Impl in this file typically different from that in EFDevice.cpp.
- Function bodies in this file also typically different.

Link-Time Polymorphism Example

Link with:

- EFDevice.o if building for expensive, high-performance box.
  - Or link dynamically with e.g. EFDevice.dll.
- CSDevice.o if building for cheaper, lower-performance box.
  - Or link dynamically with e.g. CSDevice.dll.

Link-time polymorphism is reasonable here:

- Deployment platform unknown at compilation, known during linking.
  - No need for flexibility or expense of runtime polymorphism.
    - No vtbls.
    - No indirection through vtbls.

Scott Meyers, Software Development Consultant

Compile-Time Polymorphism

- Useful when:
  - Implementation determinable during compilation.
  - Want to write mostly implementation-independent code.
- No need for virtual functions.
- Allows inlining.
- Based on *implicit interfaces*.
  - Other forms of polymorphism based on *explicit interfaces.*

Scott Meyers, Software Development Consultant

Device Example Reconsidered

Goal:

- **Device class to use determined by platform's #bits/pointer.**
  - This is known during compilation.

Approach:

- Create 2 or more classes with "compatible" interfaces.
  - I.e., support the same implicit interface.
    - E.g., must offer a reset function callable with 0 arguments.
- Use compile-time information to determine which class to use.
- Define a typedef for this class.
- Program in terms of the typedef.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Compile-Time Polymorphism Example

Revised device.h:

```
#include "NASDevice.h"      // NAS = "Normal Address Space" (32 bits);
                            // defines class NASDevice
#include "BASDevice.h"      // BAS = "Big Address Space" (>32 bits);
                            // defines class BASDevice
#include "SASDevice.h"      // SAS = "Small Address Space" (<32 bits);
                            // defines class SASDevice
...                         // remainder of device.h (coming soon)
```

By design, each class has a compatible interface.

- Members with identical names, compatible types, etc.

Scott Meyers, Software Development Consultant

Compile-Time Polymorphism Example

Driver classes may use any language features:

- **Especially inlining.**

```
class NASDevice {
public:
   …
   void reset() { … }              // inline function
   …
};
class BASDevice {
public:
   …
   void reset() { … }              // inline function
   …
};
class SASDevice {
   …
   void reset();                   // non-inline function
   …
};
```


Scott Meyers, Software Development Consultant

Compile-Time Polymorphism Example

Clients refer to the correct driver type this way:

```
Driver::type d;                    // d's type is either NASDevice,
d.reset();                         // BASDevice, or SASDevice,
                                   // depending on # of bits/pointer
```

- Driver "computes" the proper class for type to refer to.
  - Implementation on next page.

Compile-time polymorphism is reasonable here:

- Device type can be determined during compilation.
  - No need for flexibility or expense of runtime polymorphism.
  - No need to configure linker behavior or give up inlining.

Scott Meyers, Software Development Consultant

Compile-Time Polymorphism Example

Revised device.h (continued):

```
template<int PtrBitsVs32> struct DriverChoice;
template<> struct DriverChoice<-1> {              // When bits/ptr < 32
  typedef SASDevice type;
};
template<> struct DriverChoice<0> {               // When bits/ptr == 32
  typedef NASDevice type;
};
template<> struct DriverChoice<1> {               // When bits/ptr > 32
  typedef BASDevice type;
};

struct Driver {
  enum { bitsPerVoidPtr = CHAR_BIT * sizeof(void*) };
  enum { ptrBitsVs32 = bitsPerVoidPtr >   32 ? 1 :
                       bitsPerVoidPtr ==  32 ? 0 :
                                              -1
       };
  typedef DriverChoice<ptrBitsVs32>::type type;
};
```


As far as I know, this can't be done with the preprocessor, because you can't use sizeof in a preprocessor expression.

Summary: Interface-Based Programming

- One interface, multiple implementations.
- Polymorphism used to select the implementation.
  - Runtime polymorphism uses virtual functions.
  - Link-time polymorphism uses linker configuration.
  - Compile-time polymorphism uses typedefs.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

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

Dynamic Memory Management

Embedded developers often claim heap management isn't an issue:

- Client: "We don't have a heap."
- Me: "You're right. You have five heaps."

Dynamic memory management is present in many embedded systems.

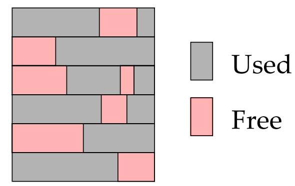
- Even if malloc/free/new/delete never called.
- Key indicator:
  - Variable-sized objects going in fixed-size pieces of memory.
    - E.g., event/error logs, rolling histories, email messages, etc.

The quote at the top of the slide is based on my first interaction with an embedded team. They warned me that they had no heap, but when I examined their design, I saw that they had five pools of dynamically allocated memory. What they meant was that they didn't call new or delete, but they still performed dynamic memory management. Effectively, they had five heaps.

Dynamic Memory Management

Four common worries:

- **Speed:**
  - Are new/delete/malloc/free fast enough?
  - How much variance, i.e., how deterministic?
- **Fragmentation:**
  - Will heap devolve into unusably small chunks?
    - This is *external* fragmentation.
- **Memory leaks:**
  - Will some allocations go undeallocated?
- **Memory exhaustion:**
  - What if an allocation request can't be satisfied?



Each concern can be addressed.

This is not an exhaustive list of concerns, just a list of common ones.

A Survey of Allocation Strategies

Each less general than malloc/free/new/delete.

- Typically more suited to embedded use.

We'll examine:

- Fully static allocation
- LIFO allocation
- Pool allocation
- Block allocation
- Region allocation
  - An optimization that may be combined with other strategies.

Scott Meyers, Software Development Consultant

Fully Static Allocation

No heap. Objects are either:

- On the stack: Local to a function.
- Of *static storage duration*:
  - At **global scope**.
  - At **namespace scope**.
  - **static at file, function, or class scope.**

Useful when:

- Exact or maximum number of objects in system statically determinable.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Fully Static Allocation

"Allocation" occurs at build time. Hence:

- **Speed**: essentially infinite; deterministic.
- **External Fragmentation**: impossible.
- **Memory leaks**: impossible.
- **Memory exhaustion**: impossible.

But:

- Initialization order of static objects in different TUs indeterminate.

TU = "Translation Unit."

"Heap Allocation"

Two common meanings:

- **Dynamic allocation outside the runtime stack.**
- *Irregular* dynamic allocation outside the runtime stack.
  - Unpredictable numbers of objects.
  - Unpredictable object sizes.
  - Unpredictable object lifetimes.

We'll use the first meaning.

- The second one is just the most general (i.e., hardest) case of the first.

User-controlled non-heap memory for multiple variable-sized objects entails heap management:

```
unsigned char buffer[SomeSize];      // this is basically a heap
…                                    // create/destroy multiple different-
                                     // sized objects in buffer
```

The C++ Memory Management Framework

User-defined memory management typically built upon:

- User-defined versions of malloc/free
- User-defined versions of operator new/new[], operator delete/delete[]
- New handlers:
  - Functions called when operator new/new[] can't satisfy a request.

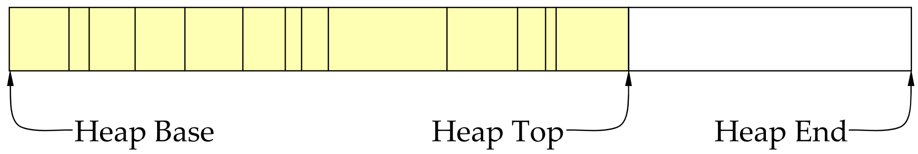
Interface details are in Further Information.

- Here we focus on allocation strategies suitable for embedded systems.

Scott Meyers, Software Development Consultant

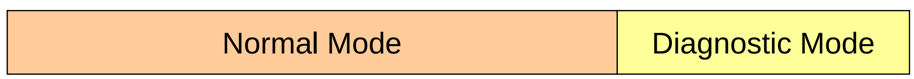
LIFO Heap Allocation

Dynamic allocation is strictly LIFO (like a stack).

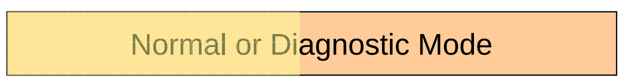


Easy way to implement a "union" for multiple-mode operations:

- E.g., a system in "normal" or "diagnostic" mode.
  - ➡ Static allocation requires the *sum* of the two modes' memory needs.



  - ➡ LIFO allocation only the *maximum* of the modes' needs.



LIFO allocation/deallocation is fast in its own right, but another speed benefit is that an allocation following a deallocation is likely to refer to memory that is already in the data cache.

LIFO allocation (a natural candidate for region allocation) is good in video games, where each level can reuse the same memory for its LIFO heap.

LIFO Heap Allocation

A first cut at an implementation:

```
class LIFOAllocator {                                   // provides behavior
public:                                                 // of new/delete via
  LIFOAllocator(unsigned char* heapAddr,                // allocate/deallocate
                std::size_t heapSize)
  : heapBase(heapAddr), heapEnd(heapAddr+heapSize),
    heapTop(heapAddr)
  {}

  void* allocate(std::size_t sz) throw (std::bad_alloc);      // shown shortly
  void deallocate(void* ptr, std::size_t sz) throw ();        // ditto

private:
  unsigned char * const heapBase;
  unsigned char * const heapEnd;
  unsigned char *heapTop;
};
```

- allocate/deallocate behave like class-specific new/delete.
- Pointer data member ⇒ copying functions should be declared.
- If LIFOAllocator templatized, ctor params could be template params.
  - The MMIO section has an example.

This "first cut" implementation is suitable only for use as a class-specific allocator, because the deallocate function requires that a size be passed. The next implementation shown allows for the size of the allocated block to be hidden in the block itself, hence could be used by non-class operator new.

LIFO Heap Allocation

Classes can easily build custom new/delete using LIFOAllocator:

```
unsigned char heapSpace[HeapSpaceSize];        // memory for heap

LIFOAllocator customAllocator( heapSpace,       // typically at global scope
                               HeapSpaceSize);

void* Widget::operator new(std::size_t bytes) throw (std::bad_alloc)
{
  return customAllocator.allocate(bytes);
}
void Widget::operator delete(void *ptr, std::size_t size) throw ()
{
  customAllocator.deallocate(ptr, size);
}
```

Here there's one global heap, but per-class or per-thread heaps are easy.

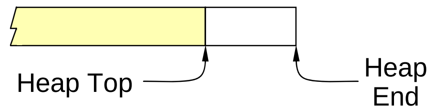
- Create a LIFOAllocator for each memory block to be used as a LIFO heap.
  - For per-class allocators, make the LIFOAllocators static and private.
  - For per-thread allocators, use thread-local storage (TLS) for the memory.

Scott Meyers, Software Development Consultant

LIFOAllocator::allocate

Implemented just like global operator new:

```
void* LIFOAllocator::allocate(std::size_t bytes) throw (std::bad_alloc)
{
  if (bytes == 0) bytes = 1;
  while (true) {
    if (heapTop + bytes <= heapEnd) {          // overflow?
      unsigned char *pMem = heapTop;           // alignment?
      heapTop += bytes;
      return pMem;
    }
    std::new_handler currentHandler = std::get_new_handler();
    if (currentHandler) currentHandler();
    else throw std::bad_alloc();
  }
}
```



- Comments indicate issues we're ignoring.
- With this design, hard for new handler to increase available memory.

The only way that 0 bytes could be requested is that somebody explicitly calls *className*::operator new(0); it's not possible to get it from a new expression. Proper deallocation in that case would be tricky, because the caller would have to explicitly call *className*::operator delete(ptr, 1), i.e., know *a priori* that a 0-byte request yields a 1-byte allocation. I don't know of a simple way to address this problem.

The comments "overflow?" and "alignment?" show places where these issues have to be considered. In the skeletal code in thise slides, they are simply flagged and ignored.

The only standard-conforming way to address the alignment issue is to make sure that this function always returns a pointer to memory that is aligned for any data type.

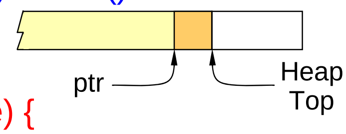
std::get_new_handler is new to C++11. Earlier compilers must do the following instead:

```
std::new_handler currentHandler = std::set_new_handler(0);
std::set_new_handler(currentHandler);
```

The diagram is supposed to make it easy to refer to the memory layout of the LIFO heap.

LIFOAllocator::deallocate

```
void LIFOAllocator::deallocate(void *ptr, std::size_t size) throw ()
{
  if (ptr == nullptr) return;
  if (heapTop != static_cast<unsigned char*>(ptr) + size) {
    // either client usage error or heap-related data structures are invalid
    Log the problem, then call exit or abort or restart/reboot the system.
  }
  heapTop -= size;
}
```



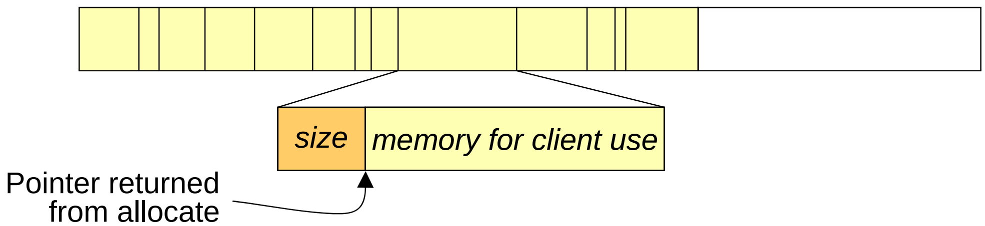
- Exception specification ⇒ throwing an exception isn't an option.

The diagram is supposed to make it easy to refer to the memory layout of the LIFO heap

Supporting Global new/delete

Global operator delete lacks size info, but heapTop -= size still needed, so:

- Have LIFOAllocator::allocate optionally hide the size in the memory.



- Overload LIFO::deallocate to take only a ptr and use hidden size info.

```
class LIFOAllocator {
public:
   …
   void* allocate(std::size_t sz, bool hideSize) throw (std::bad_alloc);
   void deallocate(void* ptr, std::size_t sz) throw ();
   void deallocate(void* ptr) throw ();
};
```

It'd be better software engineering to use an enum instead of a bool…

I have not seen a convincing explanation for why ::operator delete is not specified to take a size_t parameter. A 2006 discussion thread addressing the issue can be found at http://tinyurl.com/yhslcnz.

Supporting Global new/delete

Global new/delete are then easy to implement:

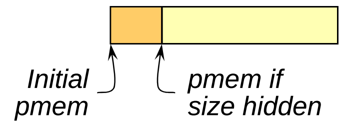
```
void* operator new(std::size_t bytes) throw (std::bad_alloc)
{
  return customAllocator.allocate(bytes, true);
}

void operator delete(void *ptr) throw ()
{
  customAllocator.deallocate(ptr);                // note lack of size param
}
```

As are class-specific versions:

```
void* Widget::operator new(std::size_t bytes) throw (std::bad_alloc)
{
  return customAllocator.allocate(bytes, false);
}

void Widget::operator delete(void *ptr, std::size_t size) throw ()
{
  customAllocator.deallocate(ptr, size);
}
```


Scott Meyers, Software Development Consultant

Supporting Global new/delete




```
void* LIFOAllocator::allocate(std::size_t bytes, bool hideSize) throw (std::bad_alloc)
{
  if (bytes == 0) bytes = 1;

  if (hideSize)
    bytes += sizeof(std::size_t);        // add space for size;
                                         // overflow?
  while (true) {
    if (heapTop + bytes <= heapEnd) {                   // overflow?
      unsigned char *pMem = heapTop;                    // alignment?
      if (hideSize) {
        *reinterpret_cast<std::size_t*>(pMem) = bytes;  // alignment?
        pMem += sizeof(std::size_t);                    // alignment?
      }

      heapTop += bytes;
      return pMem;
    }

    check/use the new handler as usual;

  }
}
```

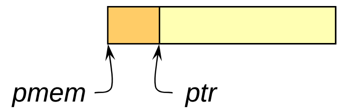

In a situation analogous to the one mentioned before, this code will have a problem if bytes is 0 and hideSize is false. As before, that can happen only if somebody explicitly calls *className*::operator new(0).

The comments "overflow?" and "alignment?" show places where these issues have to be considered. In the skeletal code in thise slides, they are simply flagged and ignored.

The only standard-conforming way to address the alignment issue is to make sure that this function always returns a pointer to memory that is aligned for any data type.

The diagram is supposed to make it easy to refer to the memory layout of an allocated block where the size has been stored at the beginning.

Supporting Global new/delete




```
void LIFOAllocator::deallocate(void *ptr) throw ()
{
  if (ptr == nullptr) return;

  unsigned char *pMem =
    static_cast<unsigned char*>(ptr) – sizeof(std::size_t);

  std::size_t size = *reinterpret_cast<std::size_t*>(pMem);

  if (heapTop != pMem + size) {

    // either client usage error or heap-related data structures are invalid
    Log the problem, then call exit or abort or restart/reboot the system.
  }

  heapTop -= size;
}
```

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

Slide 110

The diagram is supposed to make it easy to refer to the memory layout of an allocated block where the size has been stored at the beginning.

LIFO Heap Allocation

- **Speed**: extremely fast; deterministic.
  - ➡ Assuming you don't run out of memory.
- **External Fragmentation**: possible, but easy to detect (as shown).
- **Memory leaks**: possible, easy to detect.
- **Memory exhaustion**: possible.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Pool Allocation

Heap allocations are all the same size.

- Typically because all heap objects are one size.
  - Well-suited for class-specific allocators.
- Can also work when all heap objects are *nearly* the same size.
  - Then all allocations are the size of the largest objects.

Basic approach:

- Treat heap memory as an array.
  - Each element is the size of an allocation unit.
    - No need to store the size of each allocation.
- Unallocated elements are kept on a *free list*.
- Allocation/deallocation is a simple list operation:
  - Removing/adding to the front of the free list.

Pool Allocation

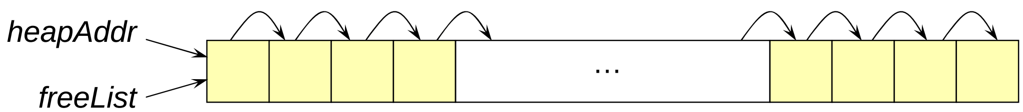
```
template<std::size_t ElementSize>
class PoolAllocator {
public:
  PoolAllocator(unsigned char* heapAddr,
                std::size_t heapSize);                    // on next page

  void* allocate(std::size_t sz) throw (std::bad_alloc);   // coming soon
  void deallocate(void* ptr, std::size_t sz) throw ();     // ditto

private:
  union Node {                                             // pool element
    unsigned char data[ElementSize];                       // when in use
    Node *next;                                            // on free list
  };

  Node *freeList;
};
```

- Pointer data member ⇒ copying functions should be declared.
- If PoolAllocator untemplatized, template param could be ctor param.
- Ideally, we'd ensure that ElementSize > 0.

There's no deallocate taking only a void*, because Pool allocators are virtually always used inside classes, i.e., when operator delete gets a size argument.

PoolAllocator Constructor

```
template<std::size_t ElementSize>
PoolAllocator<ElementSize>::PoolAllocator(unsigned char* heapAddr,
                                          std::size_t heapSize)
: freeList(reinterpret_cast<Node*>(heapAddr))
{
  const std::size_t nElems = heapSize / ElementSize;

  for (std::size_t i = 0; i < nElems-1; ++i)          // link array elements together
    freeList[i].next = &freeList[i+1];

  freeList[nElems-1].next = nullptr;
}
```



To avoid alignment problems, this code should check heapAddr to see if it is suitably aligned. If not, an exception could be thrown or sufficient bytes could be skipped at the beginning of the memory to get to a suitably aligned address.

PoolAllocator::allocate

```
template<std::size_t ElementSize>
void* PoolAllocator<ElementSize>::allocate(std::size_t bytes)
   throw (std::bad_alloc)
{
   if (bytes != ElementSize) return ::operator new(bytes);
   while (true) {
      if (freeList != nullptr) {
         void *pMem = freeList;                                  // alignment?
         freeList = freeList->next;
         return pMem;
      }
      std::new_handler currentHandler = std::get_new_handler();
      if (currentHandler) currentHandler();
      else throw std::bad_alloc();
   }
}
```

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

Slide 115

The proper place to deal with the alignment issue is the constructor. See the comment on the slide for that code.

std::get_new_handler is new to C++11. Earlier compilers must do the following instead:

```
std::new_handler currentHandler = std::set_new_handler(0);
std::set_new_handler(currentHandler);
```

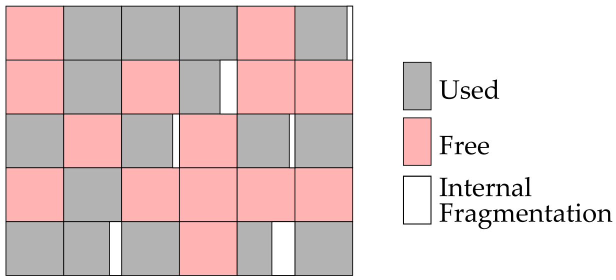
PoolAllocator::deallocate

```
template<std::size_t ElementSize>
void PoolAllocator<ElementSize>::deallocate(void *ptr, std::size_t size)
   throw ()
{
   if (ptr == nullptr) return;

   if (size != ElementSize) {
      ::operator delete(ptr);
      return;
   }

   Node *p = static_cast<Node*>(ptr);
   p->next = freeList;
   freeList = p;
}
```

PoolAllocator::allocate

Variation: allow bytes <= ElementSize, i.e., that the request fits.

- More flexible, but can lead to *internal* fragmentation.



Pool Allocation

Client code:

- "Clients" are implementers of operators new/delete.
- Left as an exercise for the attendee :-)
  - operator new calls allocate
  - operator delete calls deallocate
  - Similar to LIFOAllocator.

Scott Meyers, Software Development Consultant

Pool Allocation

- **Speed**: extremely fast; deterministic.
  - Assuming:
    - No wrong-sized requests.
    - You don't run out of memory.
- **External Fragmentation**: impossible.
- **Memory leaks**: possible.
- **Memory exhaustion**: possible.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Block Allocation

Essentially a set of pools with different element (block) sizes:

Pool for allocations of size $\mathbf{s_1}$	Pool for allocations of size $\mathbf{s_2}$	Pool for allocations of size $\mathbf{s_3}$	Pool for allocations of size $\mathbf{s_4}$	Pool for allocations of size $\mathbf{s_5}$

n-byte requests handled by first pool with size ≥ *n* and non-null free list.

Useful when:

- Allocations needed for a relatively small number of object sizes.
  - ➡ Otherwise internal fragmentation ⇒ wasted memory.

Many RTOSes offer native support for block allocation.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Block Allocation

- **Speed**: fast; nearly deterministic (and boundable).
  - Assuming:
    - No requests larger than handled by the largest-chunk pool.
    - You don't run out of memory.
- **External Fragmentation**: impossible.
- **Memory leaks**: possible.
- **Memory exhaustion**: possible.

Speed isn't totally deterministic, because you may need to examine multiple pools to find one with sufficient free memory.

General Variable-Sized Allocation

What new/delete/malloc/free already do.

- Desirable only if vendor-supplied routines unacceptable.

Possible motivations:

- Detect overruns/underruns.
- Gather heap usage data.
  - Size and lifetime distributions, temporal usage patterns, etc.
- Support data structure clustering.
- Avoid thread-safety penalty.
  - ST applications.
  - Thread-local allocators in MT applications.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Region Allocation

An optimization for when memory for all of a heap's objects can be released at once.

- Clients call a region member function at the appropriate time.
  - Faster than deallocating each object's memory individually.
- Common with LIFO allocators, but compatible with pools, blocks, etc.
- operator delete for individual objects a no-op, hence very fast.
  - Can still use delete operator to invoke destructors:

```
delete p;               // invoke *p's dtor, then operator delete on p;
                        // if *p in a region, operator delete is a no-op
```

Summary: Dynamic Memory Management

- Many embedded systems include dynamic memory management.
- Key issues are speed, fragmentation, leaks, and memory exhaustion.
- LIFO is fast and w/o fragmentation, but object lifetimes must be LIFO.
- Pools are fast and w/o fragmentation, but object sizes are limited.
- Block allocation is essentially multiple pool allocators.
- Regions excel when all heap objects can be released simultaneously.

Scott Meyers, Software Development Consultant

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
© 2012 Scott Meyers, all rights reserved.

C++ and ROM

Anything can be burned into ROM and loaded into RAM prior to program execution.

- Provided the architecture allows it.
  - Harvard does not.

The more interesting question is:

- What may remain in ROM as the program runs?

The C++ Standard is silent on ROMing:

- It allows essentially anything, guarantees nothing.
- What's ROMable is thus up to your compiler and linker.

In what follows, I discuss what is *technically possible*.

- Your compiler/linker probably imposes some restrictions.
- We'll discuss those first.

Scott Meyers, Software Development Consultant

C++ and ROM

To understand the restrictions, we need to know what a "POD type" is.

- "POD" = "Plain Old Data"
- All C data types are POD types.
- C++11 classes, structs, and unions are generally POD types if they lack:
  - Base classes
  - Virtual functions
  - Non-static data members of reference type
  - User-defined constructors, destructor, or assignment operators
  - Non-static data members of non-POD types

Essentially, a C++11 class or struct is a POD type if it's "laid out like C and its semantics are preserved if it's `memcpy`ed."

- But note that non-virtual member functions are allowed.
- Static data and static member functions are allowed, too.

The definition of POD types in C++98/03 is stricter, because protected and private non-static data members are precluded.

C++ and ROM

Common restrictions on ROMing data:

- Many compilers/linkers will ROM only statically initialized POD types.
  - ➡ As we'll see, it is technically possible for some dynamically initialized non-PODs to be ROMed.
- Some compilers/linkers will ROM structs, but not classes.
  - ➡ There is no technical reason for this distinction.

Scott Meyers, Software Development Consultant

C++ and ROM

Program instructions can always be ROMed.

Data in a C++ program can be ROMed if it meets two criteria:

- Its value is known before runtime.
  - I.e., either the compiler or the linker knows it or can compute it.
- It can't be modified at runtime.

The following examples are largely based on the ROMability section of the *Technical Report on C++ Performance*.

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

Slide 129

A full reference for the *Technical Report on Performance* is given in the "Further Information" slides at the end of the notes.

C++ and ROM

If it's ROMable in C, it's ROMable in C++:

```
static const int table[] = { 1, 2, 3 };     // table is ROMable

const char *pc1 = "Hello World";            // "Hello World" is ROMable
                                            // (but pc1 is not)

const char * const pc2 = "World";           // "World" is ROMable (and
                                            // may be shared with
                                            // "Hello World");
                                            // pc2 is also ROMable
```


Scott Meyers, Software Development Consultant

C++ and ROM

Integral scalar constants of static storage duration are ROMable, but they are often optimized away entirely:

```
const unsigned bufsize = 128;          // typically optimized away unless
                                       // bufsize's address is taken
```

Enums take no storage, and they're safer than #defines:

```
enum { bufsize = 128 };                // almost always optimized away
```

- No portable way to specify the size of an enumerant in C++03.
  - ➡ Supported directly in C++11:

```
enum: unsigned short { bufsize = 128 };     // C++11
```

Such constants typically become immediate operands in instructions.

- If they're not, they can definitely be ROMed.
- There is no advantage to using #defines in these cases.
  - ➡ #defines don't respect scope.
  - ➡ #defines can't be private or protected.

const objects have internal linkage, so if no const propagation is performed, the first example on this page could yield multiple copies of bufsize in an executable.

Per the 2003 ISO C++ Standard (section 4.5, paragraph 2), "An rvalue of…an enumeration type can be converted to an rvalue of the first of the following types that can represent all the values of its underlying type: int, unsigned int, long, or unsigned long." This means that even anonymous enums can benefit from the C++11 ability to specify the underlying type, because that can affect overload resolution when enumerants are passed as parameters.

C++ and ROM

For non-integral scalar constants, consts are safer than #defines and may be more efficient:

```
#define pi 3.14159                  // ROMable, but subject to
                                    // macro drawbacks

const double pi = 3.14159;          // ROMable, but not subject
                                    // to macro drawbacks
```

Floating point values can rarely be turned into immediate operands:

- They're ROMable in both forms above.
- With a bad compiler, the macro form might result in multiple copies of pi in an object file.
  - ➡ This shouldn't happen with the const.
    - ◆ It should never yield more than one copy in an object file.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

C++ and ROM

Objects may be ROMed if the following are true:

- They are declared const at their point of definition.
- They contain no mutable data members.
- They are initialized with values known during compilation.
  - Such "knowledge" might come from dataflow analysis, etc.

```
struct Point {
   int x, y;
};

const Point origin = { 0, 0 };          // origin is ROMable

struct Widget {                         // all Widgets can be bitwise
   int a;                               // initialized from a ROMed
   const char *p;                       // Widget initialized with
   Widget(): a(7), p("xyzzy") { }       // { 7, "xyzzy" }
};

const Widget w;                         // w is ROMable (even though
                                        // it's a non-POD requiring
                                        // dynamic initialization)
```

C++ and ROM

C++98 POD types have limited encapsulation. A wrapper can often help:

```
class Widget {                        // non-POD in C++98
  int x, y;                           // (has private data)
};
const Widget w = { 0, 0 };            // illegal – w isn't an aggregate

struct Widget {                       // POD (and an aggregate)
  int x, y;
};
const Widget w = { 0, 0 };            // typically will be ROMed

class WidgetWrapper {
  struct Widget { int x, y; }         // POD
  static const Widget w;              // ROMable
};
const WidgetWrapper::Widget
  WidgetWrapper::w = { 0, 0 };        // typically will be ROMed
```

For details, consult Herity's 1998 *Embedded Systems Programming* article.

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

Slide 134

In C++11, class Widget is a POD, but it's still not an aggregate, nor can it be brace-initialized without adding a constructor taking a std::initializer_list parameter.

C++ and ROM

Some compiler generated data structures can usually be ROMed:

- Virtual function tables
- RTTI tables and type_info objects
- Tables supporting exception handling

ROMing these objects may be impossible if they are dynamically linked from shared libraries.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

C++ and ROM

What's not ROMable? Objects that may be modified at runtime.

- Objects with nontrivial constructors or destructors.

```
class Widget {
public:
  Widget();                     // Widget objects won't be ROMable
  ...
};
```

- Objects with mutable members.

```
class Widget {
  mutable int lastValue;        // Widget objects won't be ROMable
  ...
};
```

- Objects not defined to be const.

```
  int x = 14;                   // x isn't const, hence not ROMable
  std::string s = "xyzzy";      // s isn't const, hence not ROMable
```

  - Of course, 14 and "xyzzy" can still be ROMed.

Summary: C++ and ROM

- Most compilers/linkers are willing to ROM statically initialized POD types.
  - Aggressive build chains may go beyond this.
- ROMable PODs can be encapsulated by making them protected or private in a non-POD type.
- Compiler-generated data structures are typically ROMable.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

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

Modeling Memory-Mapped IO

Many systems map IO devices to fixed parts of a program's address space.

- Input registers are often separate from output registers.
- Control/status registers are often separate from data registers.
  - ➡ Different status register bits convey information such as readiness or whether device interrupts are enabled.

C++ makes it easy to make memory-mapped IO devices look like objects with natural interfaces.

- At zero cost.
  - ➡ Provided you have a decent compiler :-)

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Modeling Memory-Mapped IO

Memory-mapped devices may require special handling, e.g.,

- Atomic reads/writes may require explicit synchronization.
- Individual bits may sometimes be read-only, other times write-only.
- Clearing a bit may require assigning a *1* to it.
- One status register may control more than one data register.
  - E.g., bits 0-3 are for one data register, bits 4-7 for another.

What follows is a *framework* for modeling memory-mapped IO, not a prescription.

- The framework tells you where to put whatever special handling your devices require.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Modeling a Control Register

Suppose we have a four-byte control register such that:

- Bit 0 indicates readiness: 1 ⇒ ready, 0 ⇒ not ready.
- Bit 2 indicates whether interrupts are enabled: 1 ⇒ enabled, 0 ⇒ not.

Assuming an int is four bytes in size, we can model the register like this:

```
enum { bit0 = 0x1, bit1 = 0x2, ... , bit31 = 0x80000000 };

class ControlReg {
public:
   bool ready() const                { return regValue & bit0; }
   bool interruptsEnabled() const    { return regValue & bit2; }
   void enableInterrupts()           { regValue |= bit2; }
   void disableInterrupts()          { regValue &= ~bit2; }

private:
   volatile unsigned regValue;             // data in the register may change
};                                         // outside program control
```

All functions are inline, so their existence should incur no cost.

- Assuming they are actually inlined.

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

Slide 141

Some lint-like tools will complain about the implementations of ready and interruptsEnabled, because they return the result of bit operations as bools. To quiet such tools, it can be preferable to write them more like this:

```
bool ready() const { return (regValue & bit0) == true; }
```

enableInterrupts and disableInterrupts use read/modify/write instructions, so they may be subject to race conditions in multithreaded systems.

Modeling a Control Register

Notes:

- In this example, we assume that an unsigned int has the proper size and alignment for the register:
  - If it doesn't, you'll need to choose a data type that does.
  - The header <stdint.h> (offering e.g., uint32_t) can be helpful here.
    - Standard in C99 and C++11.
- We also assume that the register is both readable and writable.
  - If it's write-only, you'll need to cache the most recently written value.

In theory, you could use std::aligned_storage (or std::tr1::aligned_storage) to solve the alignment problem, but then you'd have to worry that the total amount of underlying storage might be larger than the register you are modeling. For this kind of application, it seems to me that you want more precise control over the amount of storage allocated than std::aligned_storage gives you.

Caching the most recently written value is tricky, because adding a data member to the class is unacceptable. One possible approach is to have an external data structure indexed by MMIO address (e.g., a map) that holds auxillary device information, e.g., the most recently written value. The cached value would then be accessed as something like *auxillaryData[this]*.

Masks vs. Bitfields

- The design deliberately uses manual masking instead of bitfields.
  - Compilers need not map bitfields in the "obvious" fashion:

```
class ControlReg {                                          // suspect design
public:
   bool ready() const { return readyBit; }
   ...
private:
   volatile unsigned  readyBit            :1,     // Unreliable!
                      /* unused */        :1,     // Compilers need not
                      interruptEnabledBit :1,     // map readyBit to
                      /* unused */        :29;    // bit 0, etc.
};
```

  - However, on some platforms, bitfields may be faster.

Scott Meyers, Software Development Consultant

Aside: new and Placement new

A *new expression* like T *p = new T does two things:

1. Call an operator new function to find out where to put the T object.
2. Call the appropriate T constructor.

Important: operator new's fundamental job is not to allocate memory, it's to identify *where an object should go*.

- Usually, this results in dynamic memory allocation.
- Sometimes you know where you want an object to be placed:
  - You have an MMIO address where you want to put an object.
  - You have a memory buffer you'd like to construct an object in.
- You can pass operator new where you want to put something, and it will return that location:

```
void* operator new(std::size_t, void *ptrToMemory)
{ return ptrToMemory; }
```

- This form of operator new is called *placement new*.
  - It's a standard form available everywhere.

Aside: new and Placement new

Because an expression like T *p = new T calls two functions, we need a way to pass two lists of parameters.

- This passes constructor arguments: T *p = new T(*ctor args*);
- This passes arguments to operator new: T *p = new (*op new args*) T;
- This does both: T *p = new (*op new args*) T(*ctor args*);
  - ➧ You can thus use any constructor on an object you are creating via placement new.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Modeling a Control Register

Memory-mapped IO registers occur at specific addresses. To create a ControlReg object at the correct address, we use placement new.

```
// create a ControlReg object at address 0xFFFF0000 and
// make pcr point to it
ControlReg * const pcr =
  new (reinterpret_cast<void*>(0xFFFF0000)) ControlReg;
```

- Remember, with placement new, no memory is being allocated.

Once you have pcr, you can use it to communicate with the device:

```
while (!pcr->ready()) ;              // wait until the ready bit is on
pcr->enableInterrupts();             // enable device interrupts
if (pcr->interruptsEnabled()) ...    // if interrupts are enabled...
```


Scott Meyers, Software Development Consultant

Modeling a Control Register

You can avoid the pointer syntax by binding the dereferenced pointer to a reference:

```
ControlReg& cr =
  *new (reinterpret_cast<void*>(0xFFFF0000)) ControlReg;

while (!cr.ready()) ;                   // wait until the ready bit is on

cr.enableInterrupts();                  // enable device interrupts

if (cr.interruptsEnabled()) ...         // if interrupts are enabled...
```


Scott Meyers, Software Development Consultant

Placement new vs. Raw Casts

Our use of placement new calls the default ControlReg constructor.

- It's implicit, inline, and empty.
  - It should optimize away (i.e., to zero instructions).
  - If it doesn't and you care, consider a reinterpret_cast instead of placement new:

```
ControlReg * const pcr =                          // pointer version
  reinterpret_cast<ControlReg*>(0xFFFF0000);

ControlReg& cr =                                  // reference version
  *reinterpret_cast<ControlReg*>(0xFFFF0000);
```


Scott Meyers, Software Development Consultant

Placement new vs. Raw Casts

Placement new is typically preferable to a raw reinterpret_cast:

- It works if the device object's constructor has work to do.
- A raw cast will behave improperly in that case.

But there are times when reinterpret_cast can be superior:

- If placement new isn't optimized to zero instructions (and you care).
- If you want to ROM the address of an IO register *and*
  - Your compiler will ROM the result of a reinterpret_cast *and*
  - It won't ROM the result of a use of placement new.

Placing Objects via Compiler Extensions

With some compilers/linkers, there is another alternative:

- Use a compiler extension to place an object at a specific address.

Examples:

- Altium Tasking compilers offer this kind of syntax:

```
ControlReg cr __at(0xFFFF0000);
```

- The Wind River Diab compiler offers this:

```
#pragma section MMIO address=0xFFFF0000
#pragma use_section MMIO cr
ControlReg cr;
```

Such extensions may impose restrictions:

- E.g., such manually-placed objects may have to be POD types.

Payoffs:

- No need to access MMIO objects indirectly through a pointer.
- No need to allocate space for a pointer to each MMIO object.

Placing Objects via Linker Commands

Linkers may support specification of where objects are to be placed:

- C++ source code is "normal."
  - No object placement information is present.

```
ControlReg cr;                    // in C++ source file
```

- Linker scripts map C++ objects to memory locations, often by:
  - Mapping objects to sections.
    - The linker sees only *mangled* names.
  - Mapping sections to address ranges.

Result is more portable C++ code.

- Platform-specific addresses mentioned only in linker scripts.
- "Hardware engineers exercise their reign of terror on someone else."

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

Slide 151

For this approach to work, objects to be placed (e.g., cr) must presumably have external linkage.

I have no example excerpt from a linker script, because I was unable to find or develop a simple example. gcc supports linker scripts, and basic information about them (e.g., reference manuals) is easy to find via Google.

Modeling an Output Device, Take 1

An int-sized output device register can be modeled as a ControlReg object bundled with the int-sized data it controls:

```
class OutputDevice1 {
public:
  OutputDevice1(unsigned controlAddr,
                unsigned dataAddr);                // see next page

  ControlReg& control() { return *pcr; }          // get ControlReg

  void write(unsigned value)                      // write data to
  { *pd = value; }                                // device

private:
  ControlReg * const pcr;               // ptr to ControlReg; note const
  volatile unsigned * const pd;         // ptr to data; note const and volatile
};
```

Modeling an Output Device, Take 1

The constructor makes pcr and pd point to the correct addresses:

```
inline
OutputDevice1::OutputDevice1(unsigned controlAddr, unsigned dataAddr)
:  pcr(new (reinterpret_cast<void*>(controlAddr)) ControlReg),
   pd(new (reinterpret_cast<void*>(dataAddr)) unsigned)
{}
```

Clients use the class like this:

```
OutputDevice1 od( 0xFFFF0000,                // ctrl reg addr
                  0xFFFF0004);               // data reg addr

unsigned x;

...

while (!od.control().ready()) ;              // wait until the ready bit is on

od.write(x);                                 // write x to od

...
```


Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Modeling an Output Device, Take 2

MMIO addresses are compile-time constants, so it shouldn't be necessary to store them as data members like pcr and pd.

A template with non-type parameters makes it easy not to:

```
template<unsigned controlAddr, unsigned dataAddr>
class OutputDevice2 {
public:
                                                   // ctor now gone
  static ControlReg& control()
  { return *reinterpret_cast<ControlReg*>(controlAddr); }

  static void write(unsigned value)
  { *reinterpret_cast<volatile unsigned*>(dataAddr) = value; }

                                                   // data members now gone
};
```

OutputDevice2 uses static member functions to avoid passing this pointers.

Static member functions such as control and write can be invoked before objects of type OutputDevice2 have been constructed, and that could be problematic, both in general and in this example if, as on the next page, ControlReg requires construction before use. Such problems can be avoided by using non-static member functions. Calling such functions would lead to an unnecessary this pointer being passed to the member functions (modulo optimization).

Modeling an Output Device, Take 2

This example assumes the ControlReg constructor/destructor do nothing.

- Otherwise OutputDevice2 will need a constructor/destructor that call them (e.g., via placement new).

```
template<unsigned controlAddr, unsigned dataAddr>
class OutputDevice2 {
public:
  OutputDevice2()
  { new (reinterpret_cast<ControlReg*>(controlAddr)) ControlReg; }
  …
};
```

- Such initialization/cleanup must occur only once!
  - Problematic if multiple OutputDevice2 objects exist for a single hardware device.
    - Use Singleton to prevent multiple instantiations?
    - Use static alreadyInitialized/alreadyCleanedup flags?

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Modeling an Output Device, Take 2

Client code looks almost the same as before:

```
OutputDevice2<0xFFFF0000, 0xFFFF0004> od;
unsigned x;
...
while (!od.control().ready()) ;          // wait until the ready bit is on
od.write(x);                             // write x to od
...
```

Advantages of this approach:

- OutputDevice2 objects are smaller than OutputDevice1 objects.
- OutputDevice2 code may also be smaller/faster than OutputDevice1 code.
  - ➡ No need to go indirect via a this pointer.

Thanks to Siegward Jäkel for the essence of this approach.

Modeling an Output Device, Take 3

If, as in this case, the control and data registers are in contiguous memory, you can use a third design:

- Create objects *directly on* the MMIO locations:

```
class OutputDevice3 {
public:
  ControlReg& control()      { return cr; }
  void write(unsigned value) { data = value; }

private:
  OutputDevice3(const OutputDevice3&);          // prevent copying
  ControlReg cr;
  volatile unsigned data;
};
```

- Have clients use placement new (or bare reinterpret_cast) themselves:

```
// create OutputDevice3 object at address 0xFFFF0000

OutputDevice3& od =
  * new (reinterpret_cast<void*>(0xFFFF0000)) OutputDevice3;
```


Scott Meyers, Software Development Consultant

Modeling Hardware Directly

There are thus two kinds of classes for modeling memory-mapped IO.

One kind models hardware *directly*.

- Objects of such classes are created by clients at specific addresses.
  - Via placement new or reinterpret_cast.
- They contain only non-static data that maps to MMIO registers:

```
 class OutputDevice3 {                    // class designed to be
 public:                                  // instantiated at MMIO
   ...                                    // addresses
private:
   OutputDevice3(const OutputDevice3&);
   ControlReg cr;                         // data members map directly
   volatile unsigned data;                // to MMIO device registers
};
```

  - Static data is okay.
- They never contain virtual functions.
  - Virtual functions leads to a vptr *somewhere* within each object.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Modeling Hardware Indirectly

The other kind models hardware *indirectly*.

- Objects of such classes are *not* created at specific addresses.
  - ➡ Clients pass MMIO addresses as template or constructor arguments.
- They may contain "extra" data members.
  - ➡ I.e., that don't correspond to MMIO device registers.

```
class OutputDevice1 {
  ...
private:
  ControlReg * const pcr;           // data members that don't map
  volatile unsigned * const pd;     // to MMIO device registers

  unsigned lastValueWritten;        // useful for write-only registers
};
```


Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Modeling Hardware Indirectly

- They may contain virtual functions:

```
class DeviceBase {
public:
  virtual void reset() = 0;
  …
};

class OutputDevice1: public DeviceBase {
public:
  virtual void reset();
  …
};

template<unsigned controlAddr, unsigned dataAddr>
class OutputDevice2: public DeviceBase {
public:
  virtual void reset();
  …
};

…                                        // continued on next slide…
```

Modeling Hardware Indirectly

```
OutputDevice1 od1a(0xFFFF0000, 0xFFFF0004);
OutputDevice1 od1b(0xFFFF0010, 0xFFFF0014);
…
OutputDevice2<0xEEEE0000, 0xEEEE0010> od2a;
OutputDevice2<0xEEEE0020, 0xEEEE0040> od2b;
…
DeviceBase* registers[] = { &od1a, &od1b, …, &od2a, &od2b, … };

const std::size_t numRegisters = sizeof(registers)/sizeof(registers[0]);

…

for (std::size_t i =0; i < numRegisters; ++i)          // reset all registers
  registers[i]->reset();                               // in system
```


Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Modeling Hardware Indirectly

Indirect modeling more expensive than direct modeling:

- Memory for "extra" data members (if present).
- Indirection to get from the object to the register(s) (if needed).

It's also more flexible:

- May add other data members.
- May declare virtual functions.

Scott Meyers, Software Development Consultant

Preventing Likely Client Errors

Classes that model hardware directly can easily be misused, e.g.:

- Clients might instantiate them at non-MMIO addresses.
- Clients who use placement new might think they need to call delete.
  - ➡ They don't, though they may need to manually call the destructor.

Scott Meyers, Software Development Consultant

Preventing Likely Client Errors

To prevent such errors, consider declaring the destructor private:

```
class OutputDevice3 {                     // class modeling hardware
  ...                                     // directly; prevents some
private:                                  // kinds of client errors
  ~OutputDevice3() {}

  ControlReg cr;
  volatile unsigned data;
};

OutputDevice3 d;                          // error! implicit destructor
                                          // invocation. (We want to
                                          // prevent MMIO objects from
                                          // being placed on the stack.)
OutputDevice3* pd =
  new (reinterpret_cast<void*>(0xFFFF0000)) OutputDevice3;   // fine
...
delete pd;                                // error! another implicit
                                          // destructor invocation
```


Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Preventing Likely Client Errors

But if the destructor has work to do, clients must manually invoke it.

- This suggests that such classes need a public destructor:

```
class OutputDevice3 {
public:
  ~OutputDevice3();
  ...
};

OutputDevice3* pd =
  new (reinterpret_cast<void*>(0xFFFF0000)) OutputDevice3;
...
pd->~OutputDevice3();
```

But we just decided that public destructors might lead to client errors...

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Preventing Likely Client Errors

The proverbial additional level of indirection lets you have your cake and eat it, too:

```
class OutputDevice3 {
public:
  void destroy() { this->~OutputDevice3(); }        // "this->" is required for
  ...                                               // a correct parse
private:
  ~OutputDevice3() { ... }
  ...
};

OutputDevice3 d;                                    // still an error

OutputDevice3* pd =                                                    // still
  new (reinterpret_cast<void*>(0xFFFF0000)) OutputDevice3;             // fine
...
delete pd;                                          // still an error

pd->~OutputDevice3();                               // error!

pd->destroy();                                      // fine
```


Scott Meyers, Software Development Consultant

Preventing Likely Client Errors

Clients can still put MMIO devices at invalid addresses, but we can prevent that, too. One way:

```
class InDevCtrlRegAddr { ... };                  // classes representing
class OutDevCtrlRegAddr { ... };                 // valid MMIO addresses
...
const OutDevCtrlRegAddr ODCRA1(0xFFFF0000);      // an object for each
...                                              // MMIO address
class OutputDevice {
public:
  static void* operator new( std::size_t,        // op. new taking
                             OutDevCtrlRegAddr);  // only MMIO objs
  ...
};
OutputDevice& od = *new (ODCRA1) OutputDevice(params);
```

This can also eliminate the need for clients to do reinterpret_casts when creating objects.

- But there must be a way to get a void* from an OutDevCtrlRegAddr object.

Scott Meyers, Software Development Consultant

Preventing Likely Client Errors

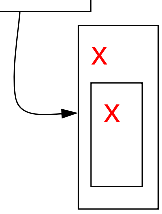
Because class-specific operator news hide all other operator news, this also prevents the creation of MMIO objects on the heap!

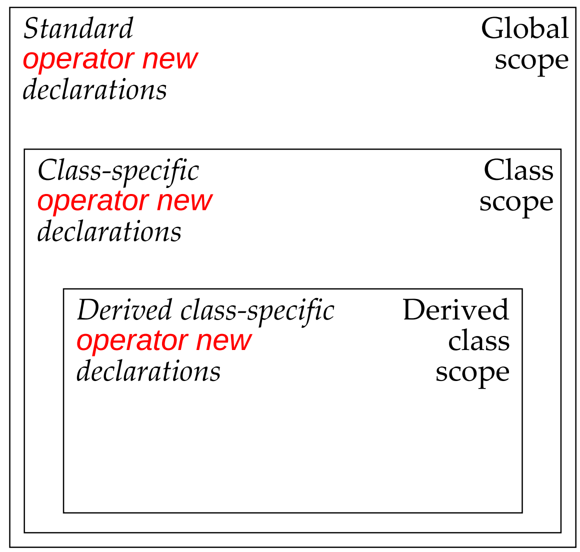
```
OutputDevice& od = *new OutputDevice(params);   // error! op. new
                                                // requires an
                                                // OutDevCtrlRegAddr
                                                // object
```

Basis:

- A name in an inner scope hides that name in an outer scope.
- Derived class scopes are nested inside base class and global scopes.

```
int x;
{
   int x;
   ...
}
```





The boxed code snippet (with two declarations for x) points to a depiction of the nested scopes that explain why the inner x hides the outer x. The nested scopes to the right correspond to how derived class operator news hide any other operator new declarations at base class or global scope.

Preventing Likely Client Errors

Details of this approach are left as an exercise, but:

- A fundamental design goal is that *design violations should not compile*.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Generalizing via Templates

Devices are likely to vary in several ways:

- Number of bits in each register.
- Which bits correspond to ready and interrupt status, etc.

Templates make it easy to handle such variability:

```
template<typename RegType,
         RegType ReadyBitMask,
         RegType InterruptBitMask>
class ControlReg {
public:
  bool ready() const              { return regValue & ReadyBitMask; }
  bool interruptsEnabled() const { return regValue & InterruptBitMask; }
  void enableInterrupts()         { regValue |= InterruptBitMask; }
  void disableInterrupts()        { regValue &= ~InterruptBitMask; }

private:
   volatile RegType regValue;
};
```


Scott Meyers, Software Development Consultant

Generalizing via Templates

```
// CRui03 is a control register the size of an unsigned int where
// bit 0 is the ready bit and bit 3 is the interrupt bit
typedef ControlReg<unsigned int, bit0, bit3> CRui03;
CRui03& cr1 = * new (reinterpret_cast<void*>(0xFFFF0000)) CRui03;
...                                           // use cr1

// CRuc15 is a control register the size of an unsigned char where
// bit 1 is the ready bit and bit 5 is the interrupt bit
typedef ControlReg<unsigned char, bit1, bit5> CRuc15;
CRuc15& cr2 = * new (reinterpret_cast<void*>(0xFFFF0010)) CRuc15;
...                                           // use cr2
```


Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Summary: Modeling Memory-Mapped IO

C++ tools you'll probably want to use:

- Classes
- Class templates (with both type and non-type parameters)
- Inline functions
- Placement new and reinterpret_cast
- const pointers
- volatile memory
- References
- Private member functions, e.g., copy constructor, destructor.

Scott Meyers, Software Development Consultant

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
© 2012 Scott Meyers, all rights reserved.

Implementing Callbacks from C

APIs often support C callback functions, e.g.:

- Hardware interrupts ⇒ calls to ISRs in C.
- OS signals ⇒ calls to signal handlers in C.
- Application events ⇒ calls to event handlers in C.

Goal:

- Write the callbacks in C++.
- Preserve full flexibility:
  - Access to all C++ language features.
  - Be able to create/configure/install/replace callbacks dynamically.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Implementing Callbacks from C

Callback details vary widely:

- Callback function parameters, e.g.:

```
void callbackFcn();                              // no params
void callbackFcn(int eventID);                   // eventID only
void callbackFcn(int eventID, void *pEventData); // eventID + arbitrary
                                                 // user-defined data
```

  - ➡ Other parameter lists (types, number of parameters) are possible.
- Return types aren't always void.
- Constraints on callback behavior:
  - ➡ E.g., ISRs and signal handlers must be fast, safe to call asynchronously, etc.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

ISRs as Example Callbacks

Our example is ISRs, but our focus is on *design* issues:

- Real ISR code is more complex:
  - May require special ISR calling conventions.
  - Behavior often constrained:
    - May require disabling other interrupts during execution.
    - May safely manipulate only atomic `volatile` data.
    - May have a hard real-time limit on execution time.
- We'll ignore such details.
  - They don't affect the basic design options.

Scott Meyers, Software Development Consultant

Example C Callback API

Assume this ISR API implemented in C:

```
typedef void (*ISR_t)(int);              // param = interrupt ID
void setISR(int interruptID, ISR_t isr);
```


Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

C-Like Function Pointers

Conceptually, two kinds of C++ function pointers can be passed to setISR:

- Non-member functions, e.g., global or namespace-scoped functions.
- Static member functions.

Reason: neither has a this pointer.

- Pointers to non-static member functions require a this pointer.
  - ➡ They're not-layout compatible with "normal" function pointers.

Static member functions are preferable:

- Reduced namespace pollution: their names are local to their class.
- Encapsulation opportunities: they can be protected or private.
- Access privileges: they can access protected or private (static) members.

Our first goal is to use static member functions as callbacks.

Scott Meyers, Software Development Consultant

C vs. C++ Linkage

Linkage can be an issue:

- C functions have C linkage.
- C++ functions have C linkage only if declared extern "C":

```
void f1(int);                 // non-member fcn:  default C++ linkage

extern "C" void f2(int);      // non-member fcn:  explicit C linkage

class Widget {
public:
  static void smf(int);       // member fcn:  always C++ linkage
};
```

For C++ compilation, ISR_t and setISR need C linkage, so:

```
extern "C" {                  // use C calling conventions
  typedef void (*ISR_t)(int);
  void setISR(int interruptID, ISR_t isr);
}
```


Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

C vs. C++ Linkage

Compilers *may* treat C and C++ linkage differently:

```
setISR(0, &f1);                    // may or may not compile/link/run
setISR(1, &Widget::smf);           // ditto

setISR(2, &f2);                    // typically compiles/links/runs
```

In general, only non-member extern "C" functions are valid C callbacks.

- Even then only for compatible C and C++ object code.
  - ➡ There is no standard ABI for C or C++ linkage.

Scott Meyers, Software Development Consultant

Static Member Functions as Callbacks

The non-member can make an inline call to a static member function:

```
class InterruptMgr {
public:
   …
   static void isr(int interruptID)            // effective ISR
   { code to handle interrupt; }
};

extern "C" {
   void isrHelper(int interruptID)             // function to pass to C API
   { InterruptMgr::isr(interruptID); }         // inline call to effective ISR
}

setISR(0, &isrHelper);                         // install callback
```

On some platforms, the non-member function can be omitted:

- On such platforms, C and C++ linkage are the same.

```
   setISR(0, &InterruptMgr::isr);              // works on some platforms
```


Scott Meyers, Software Development Consultant

Static Member Functions as Callbacks

The static member function usually calls another function to actually service the interrupt, so a better function name is advisable:

```
class InterruptMgr {
public:
  …
  static void isrDispatcher(int interruptID)
  { invoke function to handle interrupt; }
};
extern "C" {
  void isrHelper(int interruptID)
  { InterruptMgr::isrDispatcher(interruptID); }
}
setISR(0, &isrHelper);                              // as before
```

Preventing Exception Propagation into C

Exceptions thrown from C++ must not propagate into C.

- C stack frames may be laid out differently from C++ stack frames!

If callback code can throw, prevent exception propagation, e.g.:

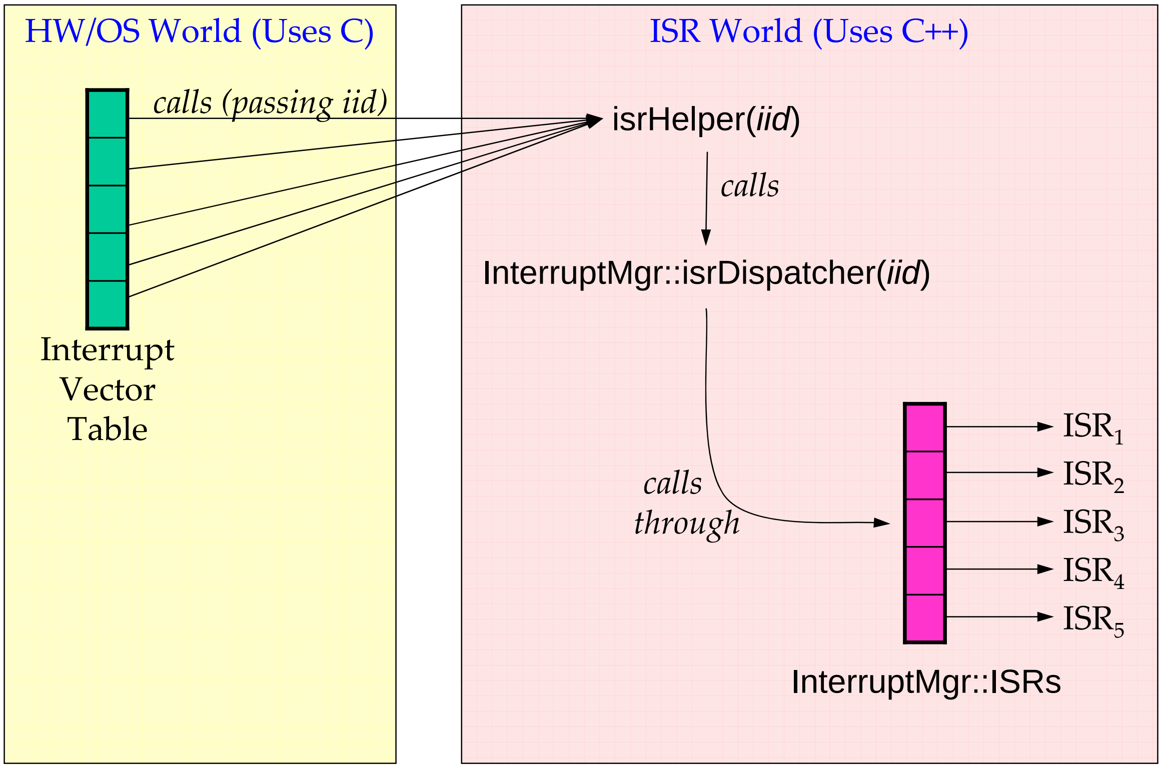
```
extern "C" {
  void isrHelper(int interruptID)          // function to pass to C API
  {
    try {
      InterruptMgr::isrDispatcher(interruptID);
    }
    catch (…) {
      set errno, log exception, whatever….
    }
  }
}
```

A try block may incur a runtime cost:

- A preferable design may be to ban exceptions in callback code.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

The Basic Callback-Handling Strategy



The Basic Callback-Handling Strategy

InterruptMgr typically works like this:

```
class InterruptMgr {
public:
   typedef ??? ISRType;                           // details in a moment
   …
   static void registerISR(int interruptID, ISRType isr)
   { ISRs[interruptID] = isr; }
   static void isrDispatcher(int interruptID)
   { call ISRs[interruptID](interruptID); }       // details in a moment
private:
   static ISRType ISRs[NUM_INTERRUPTS];           // decl. arr. of actual ISRs
};
InterruptMgr::ISRType                             // define array of actual
   InterruptMgr::ISRs[NUM_INTERRUPTS];            // ISRs
```

The ISRs array mimics the system's interrupt vector table.

- But we can make it an array of *anything* in C++:
  - ➡ It could hold objects, pointers to objects, member func. ptrs., etc.
  - ➡ We're now in the world of C++.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Callbacks via Virtual Functions

Create a base class for objects that handle interrupts:

```
class ISRBase {                                  // classes implementing ISRs
public:                                          // inherit from this
  …
  virtual void isr(int interruptID) = 0;         // or maybe operator()(int)
};
```

InterruptMgr can then look like this:

```
class InterruptMgr {
public:
  typedef ISRBase* ISRType;
  …
  static void registerISR(int interruptID, ISRType isr) { … }
  static void isrDispatcher(int interruptID)
  {
    ISRs[interruptID]->isr(interruptID);         // invoke ISR via virtual call
  }
private:
  static ISRType ISRs[NUM_INTERRUPTS];   // array of ptrs to objects w/ISRs
};
```

Callbacks via Virtual Functions

Derived classes provide actual ISRs:

```
class TimerISR: public ISRBase {
public:
   …
   virtual void isr(int interruptID);
};
```

```
class KeyboardISR: public ISRBase {
public:
   …
   virtual void isr(int interruptID);
};
```

Objects of these types are then created and registered:

```
TimerISR t;
KeyboardISR k;
InterruptMgr::registerISR(TIMER_INT_NUM, &t);
InterruptMgr::registerISR(KEYBOARD_INT_NUM, &k);
```

Result:

- Interrupt number TIMER_INT_NUM is handled by t.isr.
- Interrupt number KEYBOARD_INT_NUM is handled by k.isr.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Callbacks via Virtual Functions

Consider what happens when a timer interrupt occurs:

1. C API calls non-member function isrHelper.
2. isrHelper calls static member function InterruptMgr::isrDispatcher.
3. InterruptMgr::isrDispatcher calls member function ISRs[TIMER_INT_NUM]->isr.
   - This virtual call resolves to t.isr (i.e., TimerISR::isr on t).

InterruptMgr::isrDispatcher is inline, so at runtime we expect:

1. C API calls isrHelper.
2. isrHelper calls t.isr via vtbl.

Similarly, when a keyboard interrupt occurs:

1. C API calls isrHelper.
2. isrHelper calls k.isr via vtbl.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Assessment: Using Virtual Functions

Advantages:

- Clear, clean, "object-oriented" solution.
- Heap objects are allowed, but not required.
  - Note that t and k could be globals, namespace-local, or file static.

Disadvantages:

- Must introduce a base class and virtual functions.
  - Virtual functions ⇒ vtbl.
  - Base class may exist only to support callbacks.
    - Can lead to many small "interface" classes (and the files they're in).
- Actual ISRs must be non-static member functions.
  - Even if static member functions or non-members would do.
    - Of course, the non-static member functions could call them.
- Actual ISRs must have the same signature (including constness).
  - Modulo covariant return types….

Scott Meyers, Software Development Consultant

Callbacks via std::function and std::bind

Functionality present in C++11 and TR1.

- std::function: generalized function pointer; holds any callable entity.
- std::bind: creates function objects holding callable entities and some parameter values.
  - ➡ Can do more, but this suffices here.
  - ➡ Result often stored in a std::function object.

TR1 versions are in a nested namespace:

- std::function ⇒ std::tr1::function.
- std::bind ⇒ std::tr1::bind.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Function Types and std::function

A function's type is its declaration w/o any names:

```
void f1(int x);                     // type is void(int)
double f2(int x, std::string& s);   // type is double(int, std::string&)
```

std::function's template parameter specifies its target function type:

```
std::function<void(int)> func;      // func can hold any callable entity
                                    // compatible with the type void(int)
```


Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Callbacks via std::function

Changes to InterruptMgr are small:

```
class InterruptMgr {
public:
   typedef std::function<void(int)> ISRType;
   …
   static void registerISR(int interruptID, ISRType isr) { … }
   static void isrDispatcher(int interruptID)
   {
      ISRs[interruptID](interruptID);          // note lack of "->isr";  function
   }                                           // call syntax is used instead
private:
    static ISRType ISRs[NUM_INTERRUPTS];  // array of std::function objects
};
```

An ISRType object:

- Holds *anything callable with an* int *and returning anything*.
  - ➡ It's a generalized function pointer.
- Is invoked using function syntax.
  - ➡ E.g., inside InterruptMgr::isrDispatcher.

Scott Meyers, Software Development Consultant

Callbacks via std::function

TimerISR and KeyboardISR are unchanged, except:

- They have no base class.
  - ➡ ISRBase is no longer necessary.
- The isr functions need not be virtual.
  - ➡ Their signatures (e.g., constness) need not be identical, either.

```
class TimerISR {                        // no base class
public:
   …
   void isr(int interruptID);           // nonvirtual function (non-const)
};
class KeyboardISR {                     // no base class
public:
   …
   void isr(int interruptID) const;     // nonvirtual function (const)
};
```


Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Callbacks via std::function and std::bind

To register an ISR with InterruptMgr, we create a function object that holds:

- The ISR member function.
- The object on which that function should be invoked.

```
TimerISR t;                                  // as before
KeyboardISR k;

// on a timer interrupt, call t.isr
InterruptMgr::registerISR( TIMER_INT_NUM,
                           std::bind(&TimerISR::isr, &t, _1));

// on a keyboard interrupt, call k.isr
InterruptMgr::registerISR(KEYBOARD_INT_NUM,
                           std::bind(&KeyboardISR::isr, &k, _1));
```

- Details on bind coming soon.

The call to bind on this page won't compile as shown unless std::placeholders::_1 has been made visible (e.g., via a using declaration). This is virtually always done in code that uses bind.

Callbacks via std::function and std::bind

Consider what happens when a timer interrupt occurs:

1. As before, C API calls isrHelper.
2. As before, isrHelper calls InterruptMgr::isrDispatcher.
3. InterruptMgr::isrDispatcher calls member function held by std::function object ISRs[TIMER_INT_NUM].
   - This call resolves to t.isr via member function pointer.

InterruptMgr::isrDispatcher is still inline, so at runtime we expect:

1. C API calls isrHelper.
2. isrHelper calls t.isr via member function pointer.

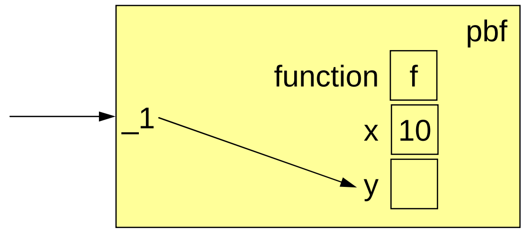
Similarly, when a keyboard interrupt occurs:

1. C API calls isrHelper.
2. isrHelper calls k.isr via member function pointer.

Scott Meyers, Software Development Consultant

Details on std::bind

```
void f(int x, int y);                   // function to ultimately call

std::function<void(int)> pbf =          // pbf = "partially bound f";
  std::bind(f, 10, _1);                 //   f's x param set to 10
```




```
…

pbf(20);                                // same as f(10, 20)
```

Note that pbf is called like a function.

The call to bind on this page won't compile as shown unless std::placeholders::_1 has been made visible (e.g., via a using declaration). This is virtually always done in code that uses bind.

The diagram is conceptual rather than rigorously accurate. In particular, it fails to show how pbf contains a copy of the object produced by bind, depicting instead that what's inside that bind-produced object is inside pbf. There is no return value shown in the diagram, because pbf's signature has a void return type.

Details on std::bind

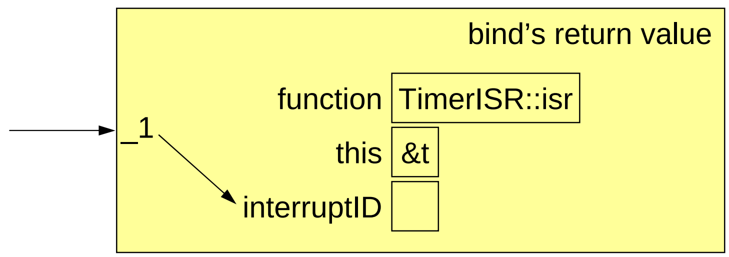
When binding non-static member functions, *this object is parameter #1:

```
class TimerISR {                          // as before
public:
  …
  void isr(int interruptID);              // std::bind sees two params:
};                                        // *this is #1, interruptID is #2
```

So

```
  std::bind(&TimerISR::isr, &t, _1)
```

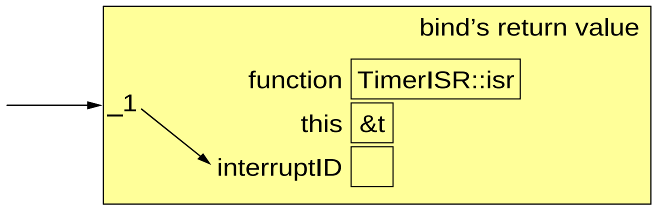
yields:



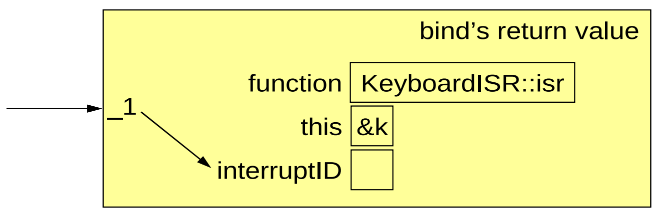
Callbacks via std::function and std::bind

Hence:

```
TimerISR t;                                  // as before
KeyboardISR k;

InterruptMgr::registerISR( TIMER_INT_NUM,
                           std::bind(&TimerISR::isr, &t, _1));
```




```
InterruptMgr::registerISR( KEYBOARD_INT_NUM,
                           std::bind(&KeyboardISR::isr, &k, _1));
```



The calls to bind on this page won't compile as shown unless std::placeholders::_1 has been made visible (e.g., via a using declaration). This is virtually always done in code that uses bind.

Flexibility in std::function

std::function works with static member functions and with non-member functions, too:

```
void handleKbdInt(int interruptID);                    // non-member function

InterruptMgr::registerISR( KEYBOARD_INT_NUM,
                           &handleKbdInt);
```

Any *compatible* signature is allowed – an exact match is not required:

```
class Timer {
public:
  …
  static void onInterrupt(long interruptID);           // static mem. func. w/
};                                                     // long param (not int)

InterruptMgr::registerISR( TIMER_INT_NUM,
                           &Timer::onInterrupt);       // okay
```

Assessment: Using std::function and std::bind

Advantages:

- No need for user-defined base classes or virtual functions.
- Callbacks may be
  - Function objects:
    - E.g., bound non-static member functions produced by std::bind
  - Static member functions
  - Non-member functions
- Callback signatures need only be compatible with a target signature.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Assessment: Using std::function and std::bind

Disadvantages:

- TR1 implementations common, but not ubiquitous.
  - Some compilers ship with none of TR1.
    - Open-source and commercial versions of bind and function exist.
  - Some developers unfamiliar with TR1 components.
- std::function objects have costs:
  - They may allocate heap memory.
  - Some implementations may use virtual functions.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

(Simple) Performance Comparison

Using VC11 beta						
	VC11 beta /Ox (milliseconds)					
	Non-Members	Virtuals	std::function			
Average of 5 trials	238.6	391.4	590.8			
Ratio to non-member call	1.0	1.6	2.5			
std::function/Virtuals			1.5			
Using g++ TR1						
				g++ 4.7 -O3 (milliseconds)		
				Non-Members	Virtuals	std::function
Average of 5 trials				291.9	426.4	755.4
Ratio to non-member call				1.0	1.5	2.6
std::function/Virtuals						1.8
Using Boost 1.49						
	VC11 beta /Ox (milliseconds)			g++ 4.7 -O3 (milliseconds)		
	Non-Members	Virtuals	boost::function	Non-Members	Virtuals	boost::function
Average of 5 trials	259.5	390.4	921.5	293.1	468.6	787.9
Ratio to non-member call	1.0	1.5	3.6	1.0	1.6	2.7
boost::function/Virtuals			2.4			1.7

- Virtuals notably slower than non-members.
- Performance using std::function varies with libraries and compilers.
  - As little as 50% slower, as much as 140% slower.

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

Slide 202

These numbers correspond to experiments I performed in June 2012.

(Simple) Performance Comparison

Benchmark setup:

- Lenovo W510 laptop (Intel quad-core Core i7, 4GB RAM, Win64)
- Do-nothing callbacks, i.e., empty bodies.
  - Only callback overhead was measured.
    - Callback execution often – typically? – swamps calling overhead.
- Maximum compiler optimizations enabled.
- All language features enabled.
  - Embedded developers often disable EH and RTTI.

If performance is important to you, do your own tests.

- And let me know what you find out….

Scott Meyers, Software Development Consultant

When User Data is Part of the Callback

Some callbacks are passed arbitrary user data, e.g.,:

```
extern "C" {
  typedef void (*ISR_t)(int, void *pData);                    // callback APIs
  void setISR(int interruptID, ISR_t isr, void *pData);       // may be like this
}
```

This change in signature propagates:

```
extern "C" {
  void isrHelper(int interruptID, void *pData)
  {
    try {
      InterruptMgr::isrDispatcher(interruptID, pData);
    }
    catch (…) {
      set errno, log exception, whatever….
    }
  }
}
```

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

When User Data is Part of the Callback

InterruptMgr's ISRs array can then be eliminated.

- The user data is the function pointer or object to be invoked:

```
class InterruptMgr {
public:
   typedef std::function<void(int)> ISRType;
   …
   static void registerISR(int interruptID, ISRType *pFunc)
   {
      setISR(interruptID, isrHelper, pFunc);
   }
   static void isrDispatcher(int interruptID, void *pFunc)
   {
      (*static_cast<ISRType*>(pFunc))(interruptID);
   }
};
```


Note that whatever pFunc points to when passed to registerISR must continue to exist when isrDispatcher invokes it. That is, the lifetime of the functor passed to registerISR must extend to the last time isrDispatcher will invoke that functor. Among other things, this means that pointers to temporaries must not be passed to registerISR.

When User Data is Part of the Callback

Sample client code:

```
InterruptMgr::ISRType f(std::bind(&TimerISR::isr, &t, _1));
InterruptMgr::registerISR(TIMER_INT_NUM, &f);
```

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

The call to bind on this page won't compile as shown unless std::placeholders::_1 has been made visible (e.g., via a using declaration). This is virtually always done in code that uses bind.

Summary: Implementing Callbacks from C

- C API callbacks in C++ can't be to non-static member functions.
  - Some platforms allow calls to static member functions.
  - Some support only callbacks to non-members declared extern "C".
- 2 basic approaches to getting into member functions:
  - Virtual functions.
  - std::function objects.
- Approaches vary in several ways:
  - Need to declare base classes and virtual functions.
  - Whether non-member functions are directly supported.
  - Whether callback signatures may vary.
  - Use of “non-standard” features (i.e., TR1 or C++11 components).
  - Use of heap memory and/or vtbls.
  - Invocation speed.

Scott Meyers, Software Development Consultant

TR1

- Standard C++ Committee Library "Technical Report 1."
- Basis for new library functionality in C++11.
- TR1 functionality is in namespace std::tr1.
- TR1-like functionality in C++11 is in std.
  - ➡ Such functionality not identical to that in TR1.
    - ◆ Uses new C++11 language features.
    - ◆ Tweaks APIs based on experience with TR1.
  - ➡ Calling interfaces largely backwards compatible
    - ◆ C++11 primarily offers "enhanced" TR1 functionality

Among other things, C++11 versions of function and shared_ptr offer allocator support not present in TR1, and tuples in C++11 offer concatenation functions (tuple_cat) not in TR1.

TR1 Summary

New Functionality	Summary
Reference Wrapper	Objects that act like references
Smart Pointers	Reference-counting smart pointers
Getting Function Object Return Types	Useful for template programming
Enhanced Member Pointer Adapter	2nd-generation mem_fun/mem_fun_ref
Enhanced Binder	2nd-generation bind1st/bind2nd
Generalized Functors	Generalization of function pointers
Type Traits	Compile-time type reflection
Random Numbers	Supports customizable distributions
Mathematical Special Functions	Laguerre polynomials, beta function, etc.
Tuples	Generalization of pair
Fixed Size Array	Like vector, but no dynamic allocation
Hash Tables	Hash table-based set/multiset/map/multimap
Regular Expressions	Generalized regex searches/replacements
C99 Compatibility	64-bit ints, <cstdint>, new format specs, etc.

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

Slide 209

Regarding random numbers, C supports only rand, which is expected to produce a uniform distributions. TR1 supports both "engines" and "distributions." An engine produces a uniform distribution, while a distribution takes the result of an engine and produces an arbitrary distribution from it. TR1 specifies default versions for the engine and distributions, but it also allows for customized-versions of both.

TR1 Itself

TR1 is a *specification*:

- Aimed at implementers, not users.
- Lacks background, motivation, rationale for functionality it specifies.
- Doesn't stand on its own.
  - E.g., assumes information in C++03.

Scott Meyers, Software Development Consultant

Understanding TR1

To *understand* the functionality in TR1:

- Consult the Further Information.
- Look at the extension proposals.
  - Links are available at Scott Meyers' TR1 Information web page, http://www.aristeia.com/EC3E/TR1_info.html.

EC++ Page	*Effective C++, Third Edition* Name	TR1 Name	Proposal Document
265	Smart Pointers	Smart Pointers	n1450
265	tr1::function	Polymorphic Function Wrappers	n1402
266	tr1::bind	Function Object Binders	n1455
266	Hash Tables	Unordered Associative Containers	n1456
266	Regular Expressions	Regular Expressions	n1429
266	Tuples	Tuple Types	n1403 (PDF)
267	tr1::array	Fixed Size Array	n1479
267	tr1::mem_fn	Function Template mem_fn	n1432
267	tr1::reference_wrapper	Reference Wrappers	n1453
267	Random Number Generation	Random Number Generation	n1452
267	Mathematical Special Functions	Mathematical Special Functions	n1422
267	C99 Compatibility Extensions	C Compatibility	n1568
267	Type Traits	Metaprogramming and Type Traits	n1424
267	tr1::result_of	Function Return Types	n1454

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

Slide 211

[It's a good idea to have a an open browser window showing the web page depicted here so that you can click on the links.]

What is Boost?

- A volunteer organization and a web site (boost.org).
- A repository for C++ libraries that are
  - Open-source
  - Portable
  - Peer-reviewed
  - Available under a "non-viral" license.
- A place to try out prospective standard C++ library enhancements.

Scott Meyers, Software Development Consultant

Boost and TR1

Boost motivated most of and implements all of TR1:

- Boost libraries are executable, TR1 isn't.

Other full or partial TR1 implementations are available:

- Microsoft:
  - 12/14 libs included in VC++ 2010-11 (VC10-11).
  - C++11 versions ship with VC++ 2011.
- Dinkumware: full TR1 impls for selected platforms.
- Gnu: 10/14 libs ship with gcc 4.

10 of 14 libraries in TR1 are modeled on Boost libraries.

Libraries missing from the VC9 TR1 update are mathematical special functions and C99 compatibility. The same is true in VC10-11.

Using Boost instead of native library implementations is a way to reduce variability (e.g., in implementation and performance) across platforms.

TR1 and Boost

Boost ≠ TR1:

- Boost offers *much* more functionality than in TR1.
  - Libraries rarely consider embedded issues.
    - But performance always a concern.
- Boost APIs don't always match corresponding TR1 APIs.
  - E.g., Bind and Tuple have some namespace "issues".
- Other TR1 implementations may differ from Boost implementations.
  - TR1 specifies *interfaces*, not implementations.

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

Slide 214

[This is a good time to show attendees the Boost web site, if time allows.]

Boost/TR1 Summary

- TR1 is a specification for standard library functionality beyond C++03.
- Boost is the premier repository of open-source, portable, peer-reviewed C++ libraries.
- Much TR1 functionality is available from Boost and others.
- Boost offers many non-TR1 libraries, too.

© 2012 Scott Meyers, all rights reserved.

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
© 2012 Scott Meyers, all rights reserved.

Using Templates to Eliminate Common Casts

Consider a Stack class template:

```
template<typename T>
class Stack {
public:
   Stack();
   ~Stack();
   void push(const T& object);
   T pop();

private:
   ...
};
```

Each different type will yield a new class:

- This could result in a lot of duplicated code.
- You may not be able to afford such code bloat.

Scott Meyers, Software Development Consultant

A Generic Stack Class for Pointers

A class using void* pointers can implement any kind of (pointer) stack:

```
class GenericPtrStack {
public:
   GenericPtrStack();
   ~GenericPtrStack();

   void push(void *object);
   void * pop();

private:
   ...
};
```


Scott Meyers, Software Development Consultant

A Generic Stack Class for Pointers

GenericPtrStack is good for sharing code:

```
GenericPtrStack stringPtrStack;
GenericPtrStack intPtrStack;

std::string *newString = new std::string;
int *newInt = new int;

stringPtrStack.push(newString);                  // these execute
intPtrStack.push(newInt);                        // the same code
```

But it's easy to misuse:

```
stringPtrStack.push(newInt);                     // uh oh...

std::string *sp =
  static_cast<std::string*>(intPtrStack.pop());  // uh oh (reprise)...
```

Code-sharing is important, but so is type-safety:

- We want both.

Scott Meyers, Software Development Consultant

Type-Safe Interfaces

We can partially specialize Stack to generate type-safe void*-based classes:

```
template<typename T>
class Stack<T*> {
public:
  void push(T *ptr) { s.push(ptr); }

  T * pop() { return static_cast<T*>(s.pop()); }

private:
  GenericPtrStack s;                          // implementation
};
```

At runtime, the cost of Stack<T*> instantiations is *zero*:

- All instantiations use the code of the single GenericPtrStack class
- All Stack<T*> member functions are implicitly inline

The cost of type-safety is *nothing*.

Type-Safe Interfaces

How force programmers to use the type-safe classes *only*?

Prevent direct use of GenericPtrStack by making everything protected:

```
class GenericPtrStack {
protected:
   GenericPtrStack();
   ~GenericPtrStack();

   void push(void *object);
   void * pop();

private:
   ...                                    // same as before
};

GenericPtrStack stringStack;              // error!
GenericPtrStack intStack;                 // error!
```


Scott Meyers, Software Development Consultant

Type-Safe Interfaces

But now Stack<T*> won't compile:

```
template<typename T>
class Stack<T*> {
public:
  void push(T *ptr)
  { s.push(ptr); }                // error! GenericPtrStack::push
                                  // is protected
  ...
private:
  GenericPtrStack s;
};
```

Type-Safe Interfaces

Private inheritance gives access to protected members:

```
template<typename T>
class Stack<T*>: private GenericPtrStack {
public:
   void push(T *objectPtr)
   { GenericPtrStack::push(objectPtr); }

   T * pop()
   { return static_cast<T*>(GenericPtrStack::pop()); }
};
```

Net result:

- Maximal type safety
- Maximal efficiency

How did we get here?

- void* Pointers
- Templates
- Private Inheritance
- Inlining
- Protected Members

Scott Meyers, Software Development Consultant

Code Bloat, Containers of Pointers, and QOI

Your C++ implementation may spare you the need to do this kind of thing:

- Some standard library vendors take care of this for you.
- Some compiler vendors (e.g., Microsoft) eliminate replicated code arising from template instantiations.
  - Approach applies to more than just containers of pointers.
    - Also optimizes Template<int> and Template<long> when int and long are the same size.

Before looking for ways to manually eliminate code bloat, make sure it's really an issue.

QOI = "Quality of Implementation"

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Summary: Eliminating Common Casts

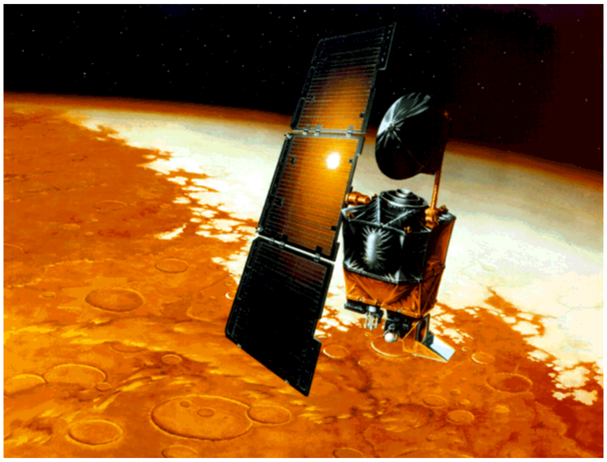
- Templates can generate type-safe wrappers around type-unsafe code.
- Inlining wrapper member functions can eliminate any runtime cost.
- Careful implementation choices can enforce design objectives.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Enforcing Dimensional Unit Correctness

Proper unit use is crucial:

- Nonsensical to assign or compare time to distance.
- Nonsensical to assign or compare pounds to newtons.
  - ➡ 1.00 pounds ≅ 4.45 newtons.
  - ➡ Loss of NASA's Mars Climate Orbiter, September 1999.



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

Slide 226

From http://en.wikipedia.org/wiki/Mars_Climate_Orbiter#The_metric_mixup: "The metric mixup which destroyed the craft was caused by a software error. The software was used to control thrusters on the spacecraft which were intended to control its rate of rotation, but by using the wrong units, the ground station underestimated the effect of the thrusters by a factor of 4.45. This is the difference between a pound force - the imperial unit - and a newton, the metric unit."

Enforcing Dimensional Unit Correctness

Alas, most software ignores units:

```
double t;                          // time - in seconds
double a;                          // acceleration - in meters/sec2
double d;                          // distance - in meters
...
std::cout << d/(t*t) - a;          // okay, subtracts meters/sec2
std::cout << d/t - a;              // nonsensical, but compiles
```

Enforcing Dimensional Unit Correctness

Typedefs just disguise the problem:

```
typedef double Acceleration;
typedef double Time;
typedef double Distance;

Time t;
Acceleration a;
Distance d;

...

std::cout << d/t - a;                       // still nonsensical, still compiles
```

Goal: use the C++ type system to:

- Detect unit compatibility errors during compilation.
- Incur minimal runtime performance impact.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Enforcing Dimensional Unit Correctness

Observations:

- Dimensional types are determined by dimension exponents:
  - Velocity = $distance^1/time^1$ = $distance^1 * time^{-1}$
  - Acceleration = $distance^1/time^2$ = $distance^1 * time^{-2}$
  - Time = $distance^0 * time^1$
- Each combination of exponents should be a different type.
  - In principle, the number of combinations is unlimited.
- Templates generate types.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Enforcing Dimensional Unit Correctness

```
template<int m,                              // exponent for mass
         int d,                              // exponent for distance
         int t>                              // exponent for time
class Units {
public:
  explicit Units(double initVal = 0): val(initVal) {}

  double value() const { return val; }
  ...

private:
  double val;
};
```

Now we can say:

```
Units<1, 0, 0> m;                           // m is of type mass
Units<0, 1, 0> d;                           // d is of type distance
Units<0, 0, 1> t;                           // t is of type time

m = t;                                      // error! type mismatch
```


The highlighting of `val` is to show that the template is just wrapping a `double`.

Adding typedefs for Cosmetic Purposes

Typedefs can hide the ugly type names:

```
typedef Units<1, 0, 0> Mass;
typedef Units<0, 1, 0> Distance;
typedef Units<0, 0, 1> Time;

Mass m;
Distance d;
Time t;

m = t;                                  // still an error
```


Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Enforcing Dimensional Unit Correctness

Arithmetic operations on these kinds of types are important, so we can augment Units as follows:

```
template<int m, int d, int t>
class Units {
public:
  ...                                              // as before
  Units<m, d, t>& operator+=(const Units<m, d, t>& rhs)
  {
    val += rhs.val;
    return *this;
  }
  Units<m, d, t>& operator*=(double rhs)
  {
    val *= rhs;
    return *this;
  }
  ...
};
```

Operators for subtraction and division are analogous.

Scott Meyers, Software Development Consultant

Enforcing Dimensional Unit Correctness

Non-assignment operators are best implemented as non-members:

```
template<int m, int d, int t>
Units<m, d, t> operator+(const Units<m, d, t>& lhs,
                         const Units<m, d, t>& rhs)
{
  Units<m, d, t> result(lhs);
  return result += rhs;
}
template<int m, int d, int t>
Units<m, d, t> operator*(double lhs,
                         const Units<m, d, t>& rhs)
{
  Units<m, d, t> result(rhs);
  return result *= lhs;
}
template<int m, int d, int t>
Units<m, d, t> operator*(const Units<m, d, t>& lhs,
                         double rhs)
{
  Units<m, d, t> result(lhs);
  return result *= rhs;
}
```

`operator-` and `operator/` are defined analogously.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Creating Typed Values

Useful (typed) constants from the SI system:

```
const Mass kilogram(1);                 // in each case, the internal
const Distance meter(1);                // double is set to 1.0
const Time second(1);
```

Other useful (typed) constants are easy to define:

```
const Mass pound(kilogram/2.2);         // Avoirdupois pound

const Time minute(60 * second);

const Distance inch(.0254 * meter);
```

As are variables:

```
int rawLength;                          // untyped length from outside
std::cin >> rawLength;                  // source, known to be in inches

Distance length(rawLength * inch);      // typed length
```

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

Slide 234

The term “pound” is used for both mass and force. As a unit of mass, it’s more formally known as “Avoirdupois pound.” As a unit of force, it’s more formally known as “pound-force.” Both are sometimes abbreviated as “lb”.

Enforcing Dimensional Unit Correctness

The real fun comes when multiplying/dividing Units:

```
template<int m1, int d1, int t1,
         int m2, int d2, int t2>
Units<m1+m2, d1+d2, t1+t2>
operator*(const Units<m1, d1, t1>& lhs,
          const Units<m2, d2, t2>& rhs)
{
  typedef Units<m1+m2, d1+d2, t1+t2> ResultType;

  return ResultType(lhs.value() * rhs.value());
}

template<int m1, int d1, int t1,
         int m2, int d2, int t2>
Units<m1-m2, d1-d2, t1-t2>
operator/( const Units<m1, d1, t1>& lhs,
           const Units<m2, d2, t2>& rhs)
{
  typedef Units<m1-m2, d1-d2, t1-t2> ResultType;

  return ResultType(lhs.value() / rhs.value());
}
```


Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Enforcing Dimensional Unit Correctness

Real implementations typically use more template arguments for Units:

- One specifies the precision of the value (typically float or double)
- The others are for the exponents of the seven SI units:
  - Mass
  - Distance
  - Time
  - Current
  - Temperature
  - Luminous intensity
  - Amount of substance

Scott Meyers, Software Development Consultant

Enforcing Dimensional Unit Correctness

```
template<class T, int m, int d, int t, int q, int k, int i, int a>
class Units {
public:
   explicit Units(T initVal = 0) : val(initVal) {}
   T& value() { return val; }
   const T& value() const { return val; }
   ...
private:
   T val;
};

template<class T, int m1, int d1, int t1, int q1, int k1, int i1, int a1,
                  int m2, int d2, int t2, int q2, int k2, int i2, int a2>
Units<T, m1+m2, d1+d2, t1+t2, q1+q2, k1+k2, i1+i2, a1+a2>
operator*(const Units<T, m1, d1, t1, q1, k1, i1, a1>& lhs,
          const Units<T, m2, d2, t2, q2, k2, i2, a2>& rhs)
{
   typedef Units<T, m1+m2, d1+d2, t1+t2, q1+q2, k1+k2, i1+i2, a1+a2>
           ResultType;

   return ResultType(lhs.value() * rhs.value());
}
```


Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Dimensionless Quantities

Dimensionless quantities (i.e., objects of type Units<T,0,0,0,0,0,0,0>) should be type-compatible with unitless types (e.g., int, double, etc.).

- Partial template specialization can help:

```
template<typename T>
class Units<T, 0, 0, 0, 0, 0, 0, 0> {
public:
  ...
  Units(T initVal = 0): val(initVal) {}        // allow implicit conversion
  operator T() const { return val; }           // to/from values of type T

  Units& operator=(T newVal)                   // allow assignments from
  { val = newVal; return *this; }              // values of type T
  ...
private:
  T val;
};
```

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

Slide 238

If partial template specialization is unavailable, you can totally specialize for e.g., T = double and/or T = float.

Efficiency

Some compilers won't put objects in registers:

- A Units<double, ...> may yield worse code than a raw double.

Idea:

- **Two sets of headers, a checking set and a no-op set.**
  - Both provide typedefs for all named unit types.
  - Checking header uses Units template, non-checking header doesn't:

```
// checking header
template<…>
class Units {
   …
};
typedef Units<...> Mass;
typedef Units<...> Distance;
typedef Units<...> Velocity;
typedef Units<...> Force;
...
```

```
// no-op header
typedef double UnitPrecision;

typedef UnitPrecision Mass;
typedef UnitPrecision Distance;
typedef UnitPrecision Velocity;
typedef UnitPrecision Force;
...
```


The "Idea" sketched on this slide and the next is just that. I have not implemented it, so there may be problems I have not anticipated.

Efficiency

- **Choose header set at build time.**
  - ➨ Using checking headers catches all errors.
  - ➨ Using no-op headers generates optimal code.
- **Clean compilation with checking headers ⇒ no unit errors:**
  - ➨ No-op headers can then safely be used.
    - ◆ E.g., velocity-acceleration errors can't exist in code.

Caveats:

- Overloading on unit types could be problematic:

```
double computeValue(Mass m);            // 3 functions with
double computeValue(Distance d);        // checking headers,
double computeValue(double d);          // only 1 with typedefs
```

  - ➨ Could lead to undefined behavior via ODR violation.

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

Slide 240

ODR = "One Definition Rule".

If the three versions of computeValue are compiled separately with the checking headers and linked with object file compiled with the unchecked headers, the system will have an inconsistent definition of computeValue.

Industrial-Strength Dimensional Analysis

State-of-the-art implementations more sophisticated than what I've shown:

- Allow fractional exponents (e.g., $\text{distance}^{1/2}$)
- Support multiple unit systems (beyond just SI)
- Use template metaprogramming for compile-time computation.
  - E.g., to compute GCDs when reducing fractional exponents.
    - $\text{distance}^{1/2} = \text{distance}^{4/8}$

Scott Meyers, Software Development Consultant

Industrial-Strength Dimensional Analysis

It can determine whether this “simple” formula,

$$\frac{1}{X_0} = 4\,\alpha\,r_e^2\,\frac{N_A}{A}\,\left\{Z^2\,[L_{rad} - f(Z)] + Z\,L'_{rad}\right\}$$

is correctly modeled by this C++:

```
Energy<> finalEnergy(Element<> const & material, Density<> const dens,
                     Length<> const thick, Energy<> const initEnergy) {
  AtomicWeight<> const A = material->atomicWeight;
  AtomicNumber<> const Z = material->atomicNumber;

  Number<> const L_rad = log( 184.15 / root<3>( Z ) );
  Number<> const Lp_rad = log( 1194. / root<3>(Z*Z) );

  Length<> const X_0 = 4.0 * alpha * r_e * r_e * N_A / A *
                      ( Z * Z * L_rad + Z * Lp_rad );

  return initEnergy / exp( thick / X_0 );
}
```

(It’s not. There are three dimensional type errors.)

Everything on this slide is from Walter E. Brown’s paper, which is referenced in the “Further Information” slides at the end of the notes.

Not Quite Foolproof

Some unit combinations correspond to more than one physical quantity.

- Energy and torque both correspond to Distance * Force.

Our approach can't tell them apart:

```
typedef Units<1, 2, -2> Energy;
typedef Units<1, 2, -2> Torque;

Energy e;
Torque t;

…

e = t;                                  // nonsensical, but will compile
```

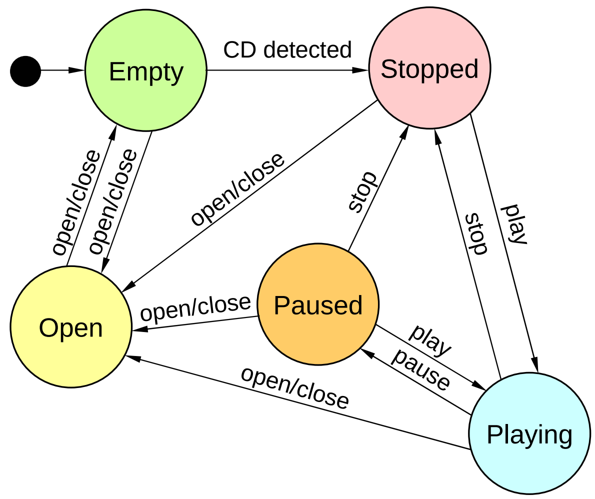
Summary: Enforcing Dimensional Unit Correctness

- Templates can be used to add new kinds of type safety.
- Non-type template parameters are both powerful and useful.
- Templates can add type safety to code with little or no runtime penalty

Scott Meyers, Software Development Consultant

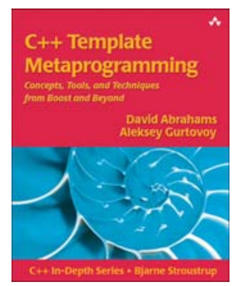
Specifying FSMs

Consider this FSM (Finite State Machine) for a CD player:



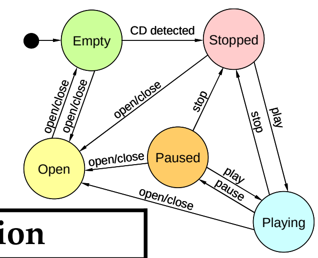
Taken (as is the entire FSM example) from Abrahams' and Gurtovoy's *C++ Template Metaprogramming.*

- Full citation in Further Information.
- I've modified their material slightly for presentation.



The legal folks at Pearson require that I note that the FSA diagram and FSA table in this section of the notes is adapted from Abrahams/Gurtovoy, C++ TEMPLATE METAPROGRAMMG: CONCEPTS TOOLS & TECHNIQUES FROM BOOST AND BEYOND, 2005 Pearson Education, Inc. and is used with permission of Pearson Education, Inc.

Specifying FSMs



Here's a table version that also shows transition actions:

Current State	Event	Next State	Transition Action
Empty	Open/Close	Open	Open drawer
Empty	CD-Detected	Stopped	Store CD info
Stopped	Play	Playing	Start playback
Stopped	Open/Close	Open	Open Drawer
Open	Open/Close	Empty	Close drawer; collect CD info
Paused	Play	Playing	Resume playback
Paused	Stop	Stopped	Stop playback
Paused	Open/Close	Open	Stop playback; open drawer
Playing	Stop	Stopped	Stop playback
Playing	Pause	Paused	Pause playback
Playing	Open/Close	Open	Stop playback; open drawer

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Domain-Specific Embedded Languages

Both diagram and table are *declarative specifications* of FSMs.

- They specify *what* should happen, not *how*.

TMP makes it possible for such specifications to be given in C++.

- **Via Domain-Specific Embedded Languages (DSELs).**
  - ➡ Domain-specific languages embedded within C++.
- With DSELs, specifications *are* programs.

Scott Meyers, Software Development Consultant

The FSM DSEL

For the FSM DSEL,

- Clients specify:
  - States
  - Events
  - Transitions
  - Transition actions

The compiler generates the FSM code automatically.

- Via template instantiation.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

The FSM DSEL

In this presentation we examine only the DSEL's **client interface**:

- How clients *use* a TMP-based FSM library.
  - Goal: demonstrate what can be accomplished.
- The library *implementation* is in *C++ Template Metaprogramming*.
  - It's Chapter 11 of an 11-chapter book.
    - No time here to cover chapters 1-10 :-)

Only fundamental functionality is shown.

- No state-entry/exit actions, no guards, no state hierarchies, etc.
- **Goal is to demonstrate unobvious template functionality**, not to show how to implement FSMs.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Specifying FSMs and States

An FSM is specified by a class, its states by a nested enum:

```
class player : public state_machine<player>       // player = FSM for
{                                                 // the CD player
private:
  enum states {                                   // states in the FSM
    Empty, Open, Stopped, Playing, Paused
    , initial_state = Empty                       // initial FSM state
  };
  …
};
```

- Base class state_machine provides generic FSM functionality.
  - TMP-generated code will go in this class.
  - Base class takes the derived class as a template parameter.
    - "The Curiously Recurring Template Pattern"
- States are private.
  - FSM clients don't need to access them.
    - FSM clients just cause events to be generated.
    - The FSM reacts accordingly.

States are identified by enumerants so that they can be passed as template parameters and also stored as the value of a data member. A more natural design (IMO) is to model them as classes, an approach that's taken in the Boost Statechart library. How that library keeps track of which state the FSM is in, I don't know.

Specifying FSM Events

Events are classes. In this example,

- They are defined outside the FSM class.
  - If desired, they could be nested inside.
- They are largely empty.
  - In a real system, they could be arbitrarily complex.

```
struct play {};

struct open_close {};

struct pause {};

struct stop {};

class cd_detected {
public:
   cd_detected(char const* cdName,
                     std::vector<std::clock_t> const& trackLengths) { … }
   …
};
```

Specifying FSM Transition Actions

Transition actions are FSM member functions:

```
class player : public state_machine<player>
{
private:
   enum states { … };

   void start_playback(play const&);
   void open_drawer(open_close const&);
   void close_drawer(open_close const&);
   void store_cd_info(cd_detected const&);
   void stop_playback(stop const&);
   void pause_playback(pause const&);
   void resume_playback(play const&);
   void stop_and_open(open_close const&);

 …
};
```

- Like states, transition actions are private.
  - FSM clients just generate events.
  - The FSM automatically reacts.

Scott Meyers, Software Development Consultant

Specifying FSM Transitions

FSM structure is specified in table form.

- Base class state_machine declares a row template:

```
template<class Derived>
class state_machine
{
...
protected:
  template<
      int CurrentState
    , class Event
    , int NextState
    , void (Derived::*action)(Event const&)
  >
  struct row { … };
…
};
```

Current State	Event	Next State	Transition Action
Empty	Open/Close	Open	Open drawer
Empty	CD-Detected	Stopped	Store CD info.
Stopped	Play	Playing	Start playback
Stopped	Open/Close	Open	Open Drawer
Open	Open/Close	Empty	Close drawer; collect CD info.
Paused	Play	Playing	Resume playback
Paused	Stop	Stopped	Stop playback
Paused	Open/Close	Open	Stop playback; open drawer
Playing	Stop	Stopped	Stop playback
Playing	Pause	Paused	Pause playback
Playing	Open/Close	Open	Stop playback; open drawer

Scott Meyers, Software Development Consultant

Specifying FSM Transitions

The FSM creates the table as a collection of rows:

```
class player : public state_machine<player> {
private:
  …                                    // states and transitions
  typedef player p;                    // makes transition table cleaner
  struct transition_table : mpl::vector11<
  //    Start      Event          Next       Action
  //  +---------+-------------+---------+---------------------+
  row < Stopped , play         , Playing , &p::start_playback  >,
  row < Stopped , open_close   , Open    , &p::open_drawer     >,
  //  +---------+-------------+---------+---------------------+
  row < Open    , open_close   , Empty   , &p::close_drawer    >,
  //  +---------+-------------+---------+---------------------+
  …
 > {};
};
```

- A fixed-width font makes the table form clearer.
- mpl::vector11 is a vector-like TMP container of 11 types.
  - In this case, 11 row instantiations.

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

Slide 254

The name of the transition table must be transition_table, because the base class state_machine<T> refers to it by that name.

Specifying FSM Transitions

Here's the complete table:

```
class player : public state_machine<player> {
…
struct transition_table : mpl::vector11<
  //    Start     Event         Next      Action
  //  +---------+-------------+---------+---------------------+
  row < Empty   , open_close  , Open    , &p::open_drawer      >,
  row < Empty   , cd_detected , Stopped , &p::store_cd_info    >,
  //  +---------+-------------+---------+---------------------+
  row < Stopped , play        , Playing , &p::start_playback   >,
  row < Stopped , open_close  , Open    , &p::open_drawer      >,
  //  +---------+-------------+---------+---------------------+
  row < Open    , open_close  , Empty   , &p::close_drawer     >,
  //  +---------+-------------+---------+---------------------+
  row < Paused  , play        , Playing , &p::resume_playback >,
  row < Paused  , stop        , Stopped , &p::stop_playback    >,
  row < Paused  , open_close  , Open    , &p::stop_and_open    >,
  //  +---------+-------------+---------+---------------------+
  row < Playing , stop        , Stopped , &p::stop_playback    >,
  row < Playing , pause       , Paused  , &p::pause_playback   >,
  row < Playing , open_close  , Open    , &p::stop_and_open    >
  //  +---------+-------------+---------+---------------------+
 > {};
};
```

FSM in C++ vs. FSM in a Table

Compare with the original table:

Current State	Event	Next State	Transition Action
Empty	Open/Close	Open	Open drawer
Empty	CD-Detected	Stopped	Store CD info
Stopped	Play	Playing	Start playback
Stopped	Open/Close	Open	Open Drawer
Open	Open/Close	Empty	Close drawer; collect CD info
Paused	Play	Playing	Resume playback
Paused	Stop	Stopped	Stop playback
Paused	Open/Close	Open	Stop playback; open drawer
Playing	Stop	Stopped	Stop playback
Playing	Pause	Paused	Pause playback
Playing	Open/Close	Open	Stop playback; open drawer

- The table *is* the source code!

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Using FSMs

Client code just generates events:

- Again, this is taken from *C++ Template Metaprogramming*.

```
player p;                                // An instance of the FSM

p.process_event(open_close());           // user opens CD player
p.process_event(open_close());           // inserts CD and closes
p.process_event(                         // CD is detected
    cd_detected(  "louie, louie"
                , std::vector<std::clock_t>( /* track lengths */ ))
);

p.process_event(play());                 // etc.
p.process_event(pause());
p.process_event(play());
p.process_event(stop());
```


The member function process_event is defined by the state_machine<T> base class.

Summary: Specifying FSMs

- Template metaprogramming makes it possible to create Domain-Specific Embedded Languages (DSELs).
- DSELs facilitate a declarative programming style.
- Declarative code tends to be easier to create, understand, and enhance.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Other Approaches to FSMs

There are many ways to specify and implement FSMs in C++. They vary in:

- **Expressiveness**, e.g., support for hierarchical and concurrent states.
  - UML supports both (think "Statecharts + Petri Nets")
- Support for **both static and dynamic FSM specification**.
- **Type safety** of source code.
- Demands on **compiler template support**.
- **Size and speed** of executable code.
- Use of **heap memory** at runtime.
- Support for **multithreading**.
- **Debuggability**.
- **Coupling** between, e.g., states and transitions.

A plethora of approaches are described in the Further Information.

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

Slide 259

Regarding coupling, the interface just shown has all states in a global list (high coupling), but states have no knowledge of transitions or actions (low coupling). In Boost.Statechart, it's the opposite: there is no global list of states, but states know about transitions and actions.

Summary: Interesting Template Applications

- **Templates are useful for a lot more than just containers**
- Templates can generate type-safe wrappers around type-unsafe code.
- Templates can be used to enforce novel kinds of type safety (e.g., dimensional units).
- Domain-Specific Embedded Languages (DSELs) can be built on templates.

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
© 2012 Scott Meyers, all rights reserved.

C++ in Safety-Critical Systems

Safety-Critical: System failure ⇒ loss of (human) life or serious injury.

Some application areas:

- **Transportation**: airplanes, cars, trains, ships, spacecraft, etc.
- **Medicine**: radiation machines, heart-lung machines, drug-delivery equipment, etc.
- **Communication**: battlefield radios, emergency response (e.g., 911 in USA, 112 in Europe), etc.

Current C++ use in such systems?

- **Extensive:** All application areas above.

Scott Meyers, Software Development Consultant

Safety-Critical Software and Risk

Safety-critical software is like “normal” software, except:

- The risk of incorrect behavior must be extraordinarily low.

The key is therefore simple:

- **Minimize risk of incorrect behavior.**

Scott Meyers, Software Development Consultant

Minimizing Risk

General approaches:

- Very detailed specifications.
  - Plus rigorous change management.
- Comprehensive testing.
  - **At multiple levels**, e.g., unit, module, system.
  - **Includes performance**.
    - Adequate performance is a correctness criterion.
- Constrained programmer discretion.
  - Via coding guidelines.
- Extensive static analysis:
  - Ensure coding guidelines are obeyed.
  - Look for problems unlikely to be exposed by testing.
  - Analyses performed by both machines and humans.
    - Lint-like tools.
    - Formal code inspections.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Minimizing Risk

- Independent redundant computation:
  - Independent teams implement the same functionality.
    - Possibly using different programming languages.
  - At runtime, all implementations execute in parallel.
  - When implementations produce different results,
    - Vote?
    - Shut down?
    - Revert to known state?

Scott Meyers, Software Development Consultant

Minimizing Risk

Recap:

- Detailed specifications.
- Comprehensive testing.
- Constrained programmer discretion.
- Extensive static analysis.
- Independent redundant computation.

Nothing above is specific to C++.

- ***Development process*** **vastly dominates programming language.**
- The only thing C++-specific is the coding guidelines employed.

Scott Meyers, Software Development Consultant

Coding Guidelines

Goals:

- Maximize code clarity and comprehensibility.
  - For both humans and static analysis tools.
- Maximize code's behavioral predictability.

Means:

- Requirements and prohibitions regarding coding practices.

Ideally, guideline violations can be automatically detected.

- Ideal rarely achieved.
  - E.g., Hatton notes that 5-10% of MISRA-C rules not so enforceable.
- Human static analysis must enforce rules not automatically checkable.

"Hatton" is "Les Hatton," author of *Safer C* and a researcher on, among other things, factors affecting software correctness, especially in safety-critical systems. His comment on this slide is, I believe, from a personal conversation I had with him. His web site is http://www.leshatton.org/.

Guideline Levels

Guidelines usually have multiple levels of stringency.

- E.g., Joint Strike Fighter (JSF) rules use three levels:
  - **Should**: advisory.
  - **Will**: mandatory, verification not required.
  - **Shall**: mandatory, verification required.
- Other guideline sets distinguish *required rules* from *advisory rules*, etc.

Lower stringency levels increase programmer discretion.

- Higher levels are therefore preferable.
- Violating even a JSF "Should" rule requires a manager's approval.

The Joint Strike Fighter is also known as the F-35. All the avionics code is apparently written in C++ following the JSF coding standard.

Guideline Types

Lexical guidelines:

- No effect on execution semantics.
- Reduce programmer-to-programmer variation.
  - Improves code/system clarity and comprehensibility.
- Examples:
  - **Naming rules**:

    JSF AV Rule 45: All words in an identifier will be separated by the '_' character.
  - **Formatting rules**:

    High Integrity C++ Rule 3.1.1: Organise 'class' definitions by access level, in the following order: 'public', 'protected', 'private'.

    MISRA C++ Rule 2-13-4: Literal suffixes shall be upper case. [E.g., "2.5f" is disallowed, but "2.5F" is okay.]

JSF = "Joint Strike Fighter", AV = "Air Vehicle"

Guideline Types

Language use guidelines:

- Specify acceptable language features and constructs.
  - Remove "unnecessary" and "dangerous" things.
    - Identify the acceptable language subset.
- Reduce code's complexity.
- Increases its behavioral predictability.
- Make it more amenable to static analysis.

For C++, coding guidelines often based on MISRA-C++ or MISRA-C.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Example Guidelines

- **Make no use of "underspecified" behavior.**
  - Many official terms: "undefined", "unspecified", etc.
  - Common in C++.
    - From *An Investigation of the Unpredicatable Features of the C++[98] Language*:

Category	Language issues	Library Issues	Total Number of Issues
Unspecified behaviour	25	25	50
Undefined behaviour	77	29	106
Implementation behaviour	58	23	81
Indeterminate behaviour	5	0	5
Behaviour that requires no diagnostic	18	0	18

    - Example:

```
f(calcThis(), calcThat());        // prohibited:  eval order undefined

int thisVal = calcThis();
int thatVal = calcThat();
f(thisVal, thatVal);              // fine, eval order fully defined
```


Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Example Guidelines

- **Avoid "surprising" behavior:**
  - *JSF AV Rule 177:* User-defined conversion functions should be avoided.
    - Prevents unexpected implicit conversions.
  - *MISRA C++ Rule 5-0-1:* The value of an expression shall be the same under any order of evaluation that the standard permits.
    - Prevents compiler- or optimization-dependent behavior.
  - *High Integrity C++ Rule 3.3.14:* Declare the copy assignment operator protected in an abstract class.
    - Prevents partial assignments to derived objects via base class pointers/references.

Scott Meyers, Software Development Consultant

Example Guidelines

- **Avoid overly complex code:**
  - *JSF AV Rule 3:* All functions shall have a cyclomatic complexity number of 20 or less.
  - *High Integrity C++ Rule 4.1:* Do not write functions with an excessive McCabe Cyclomatic Complexity.
    - Recommended maximum is 10.
  - *MISRA C++ Rule 6-6-3:* The `continue` statement shall only be used within a well-formed `for` loop.

The motivation for the MISRA rule is as follows: "Over-use of the `continue` statement can lead to unnecessary complexity within the code. This complexity may impede effective testing as extra logic must be tested. The required testing may not be achievable due to control flow dependencies."

Risk Reduction via Language Choice

C++ can reduce risk compared to C:

- Coding occurs at a higher level of abstraction
  - Code looks more like the design.
  - Direct support for multiple paradigms:
    - **OO**: Encapsulation, classes, inheritance, dynamic binding, etc.
    - **Generic**: Templates
    - **Procedural**
    - **Functional** (closures and lambdas supported only in C++11, alas)
- Language features reduce the need for preprocessor usage.
  - E.g., C macros often become C++ consts and inlines.
- Templates offer type-safe alternatives to type-unsafe C practices.
  - E.g. prevent confusing pointers and arrays:
    - std::unique_ptr<T> or std::shared_ptr<T> (or std::auto_ptr<T>) ⇒ single object.
    - std::vector<T> or std::array<T, n> ⇒ array of objects.

Riskiness of C++ compared to Java, Ada, C#, etc. hotly debated :-)

std::unique_ptr is in only C++11. std::shared_ptr and std::array are present in both C++11 and TR1. In the latter, they are in namespace std::tr1.

Tool-Related Risks

Compilers, linkers, runtime systems, OSes, etc. are software.

- They also contribute to the reliability of safety-critical systems.
- Reducing risk means addressing the risks they introduce, too.

Approaches:

- **Commercial validation suites:**
  - ➨ E.g., for compiler/library conformance to standard C++.
  - ➨ E.g., against DO-178B.
- **Manual analysis of generated code.**
  - ➨ Typically in conjunction with a restricted source code subset.
- **Testing, testing, testing.**

C++ compilers typically not certified in any standard way.

- Green Hills' compiler for Embedded C++ has been certified at DO-178B Level A.

Summary: C++ in Safety-Critical Systems

- Fundamentally a matter of reducing risk.
- Development process more important than programming language.
- Coding guidelines plus extensive static analysis are key.
- Reliability of ancillary software tools/components also important.
- C++ currently employed in many safety-critical application areas.

Scott Meyers, Software Development Consultant

C++ in Real-Time Systems

Real-Time:

- **Hard**: Timing deadlines missed ⇒ system failure.
  - E.g., engine controllers, pacemakers, elevators, etc.
- **Soft**: Timing deadlines missed ⇒ reduced behavioral quality.
  - E.g., Music and video players, IP network buffer managers, etc.

Key characteristic is not speed, but **determinism** in timing:

- RT systems fully or largely guarantee their ability to satisfy timing constraints.
  - *Fully* for hard RT.
  - *Largely* for soft RT.

In TCP/IP communication, TCP guarantees packet delivery, but IP does not. So if the IP layer misses a deadline and drops a packet, the TCP layer will detect that and make sure the packet is retransmitted. So ultimately no data is lost, but throughput decreases.

Approaching RT Development

RT development for C++ is essentially the same as for C:

- Determine timing constraints.
- Avoid language features with indeterminate timing behavior:
  - C: "Out of the box" malloc/free/memcpy
  - C++:
    - **"Out of the box" malloc/free/memcpy/new/delete**
    - **RTTI**: dynamic_cast, comparisons of type_info objects
    - **Exceptions**: try/throw/catch
  - Custom malloc/free/memcpy, etc., may have deterministic timing.
- Perform execution time analysis.
  - For functions, tasks, and the entire system.
  - Hard RT: typically Worst Case Execution Time (WCET) analysis.
  - Soft RT: often average case execution time analysis.

"Language features" includes library functionality, because malloc, free, memcpy, etc., are library features, not language features.

Timing Behavior Variations

Execution time for language features depends on:

- Compiler and linker.
  - Including optimization settings
- Call context (for inline functions).
- Hardware features:
  - E.g., caching, pipelining, speculative instruction execution, etc.

In addition:

- Library features may be implemented in different ways:
  - Container/algorithm implementations in the STL.
    - E.g., std::string may or may not use SSO or COW.
    - E.g., std::sort may use quicksort or introsort.
  - Different heap management algorithms for malloc/free/new/delete.

SSO = "Small String Optimization," COW = "Copy on Write"

COW is not a valid std::string implementation technique in C++11.

Analyzing Execution Time

Approaches to block/function WCET analysis:

- **Static analysis of source code.**
  - By humans, tools, or both.
  - Templates can be handled by explicit instantiation and per-instantiation analysis.
- **Dynamic analysis of code under test.**
  - Observe how long it takes to execute blocks/functions.
- **A combination of the above.**
  - Dynamic analysis of basic blocks' WCETs.
  - Static analysis of paths through blocks.
    - Testing for 100% path coverage is difficult.

For system WCET, combine:

- Per-task WCET analysis.
- Task schedulability analysis.

Scott Meyers, Software Development Consultant

Analyzing Execution Time

Approaches to average-case analysis:

- **Same options as for WCET.**
  - But determine average-case time, not worst-case.
- **Multiply C++ statement count by a fudge factor.**
  - A bigger fudge factor than C.
    - C++ statements typically do more than C statements.
  - Useful for ballparking execution time during development.
    - Reduces need for fine-tuning later.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Summary: C++ in Real-Time Systems

- Fundamental approach the same as for C.
- Typically avoid the use of heap operations, RTTI, and exceptions.

Scott Meyers, Software Development Consultant

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

Further Information

Using C++ in embedded systems:

- "Abstraction and the C++ Machine Model," Bjarne Stroustrup, Keynote address at ICESS04, December 2004, available at http://www.research.att.com/~bs/abstraction-and-machine.pdf.
  - An overview of the strengths of C++ for embedded systems.
- "OO Techniques Applied to a Real-time, Embedded, Spaceborne Application," Alexander Murray and Mohammad Shababuddin, *Proceedings of OOPSLA 2006*.
  - Describes how OO and C++ are being used in the development of satellite software.
- "Reducing Run-Time Overhead in C++ Programs," Embedded Systems Conference, Dan Saks, 1998 and subsequent years.
  - How to avoid common C++ performance "gotchas".
  - 2002 paper available at http://www.open-std.org/jtc1/sc22/wg21/docs/ESC_SF_02_405_&_445_paper.pdf

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Further Information

More on using C++ in embedded systems:

- "C++ in Embedded Systems: Myth and Reality," Dominic Herity, *Embedded Systems Programming*, February 1998.
  - Dated, but a good, comprehensive overview of C++ vs. C. Considers code size, code speed, exceptions, ROMability, etc.
- "Embedded Programming with C++," Stephen Williams, Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS), 1997.
  - Summarizes the design, functionality, and performance of a C++ runtime library for embedded systems; the runtime replaces the OS.
- "C++ in der Automotive-Software-Entwicklung," Matthias Kessler, Oliver Müller, and Gerd Schäfer, *Elektronik automotive*, May 2006.
  - An overview of C++ language features and how they've proven useful in the development of embedded automotive software.
    - Article is in German.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Further Information

General information on how C++ is implemented:

- *Technical Report on C++ Performance*, ISO/IEC TR 18015:2006(E), February 2006, http://www.open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf.
  - ➡ Summarizes likely overhead for various language features.
- *Inside the C++ Object Model*, Stanley B. Lippman, Addison-Wesley, 1996, ISBN 0-201-83454-5.
  - ➡ Information is sometimes outdated, incorrect, or cfront-specific.
- *Secure Coding in C and C++*, Robert Seacord, Addison-Wesley, 2006, ISBN 0-321-33572-4.
  - ➡ Good treatments of runtime data structures and how undefined behavior can lead to security vulnerabilities.
- "C++ Exceptions and the Linux Kernel," Halldór Ísak Glyfason and Gísli Hjálmtysson, *Dr. Dobbs Journal*, September 2005.
  - ➡ Focuses on exceptions, but mentions other language features, too.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Further Information

More general information on how C++ is implemented:

- "Vtbl layout under MI ," comp.lang.c++.moderated thread, initiated 27 May 2005 by Scott Meyers. Available at http://tinyurl.com/rs3rb.
  - Note gcc's "-fdump-class-hierarchy" option.
- "Exception Handling," Josée Lajoie, *C++ Report*, March-April 1994 (Part 1) and June 1994 (Part 2).
  - Overview of a table-based EH implementation.
  - Part 1 was reprinted in *C++ Gems*, Stanley Lippman (Ed.), SIGS Books & Multimedia, 1996, ISBN 1-884842-37-2.
- "Itanium C++ ABI," *CodeSourcery* web site, available at http://www.codesourcery.com/cxx-abi/abi.html.
  - Includes an ABI specification for table-based EH implementations (for 64 bit applications).
- "Fast Dynamic Casting," Michael Gibbs and Bjarne Stroustrup, *Software Practice & Experience*, December 2005.
  - Describes a constant-time implementation of dynamic_cast.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Further Information

Information on how Microsoft's C++ is implemented:

- "How a C++ Compiler Implements Exception Handling," Vishal Kochhar, *The Code Project* web site, April 2002, available at http://www.codeproject.com/cpp/exceptionhandler.asp.
  - An excruciatingly detailed description of how EH is implemented in MSVC6-7.
- "The Cost of C++ Exception Handling on Windows," Kevin Frei, Presentation to the *Northwest C++ Users Group*, October 18, 2006.
- "C++ Under the Hood," Jan Gray, in *Black Belt C++: The Masters Collection*, M&T Books, 1994, ISBN 1-55851-334-5
  - Describes how language features are (were?) implemented in MSVC.
- "Microsoft C++ Name Mangling Scheme," Kang Seonghoon, http://mearie.org/documents/mscmangle/.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Further Information

Program behavior when pure virtual functions are called:

- "'Pure Virtual Function Called': An Explanation," *The C++ Source*, February 26, 2007, http://www.artima.com/cppsource/pure_virtual.html.

Cost of error handling without exceptions:

- "Bail, return, jump, or…throw," Dan Saks, *Embedded Systems Design*, March 2007.
  - ➡ Estimates object code size increase of 15-40% for using return values to report error conditions (vs. no error detection/propagation).

Scott Meyers, Software Development Consultant

Further Information

Controlling the generation and cost of temporary objects:

- *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Scott Meyers, Addison-Wesley, 1996, ISBN 0-201-63371-X.
  - Items 19-22 cover the basics of controlling temporary creation.
  - Item 29 describes how reference counting can make object creation inexpensive.
  - A copy of the book's table of contents is attached.
- Template metaprogramming (TMP), including expression templates and TMP-based libraries:
  - "Using C++ Template Metaprograms," Todd Veldhuizen, *C++ Report*, May 1995.
  - *C++ Templates*, David Vandevoorde and Nicolai Josuttis, Addison-Wesley, 2003, ISBN 0-201-73484-2, chapters 17-18.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Further Information

Code bloat:

- *Techniques for Scientific C++*, Todd Veldhuizen, Indiana University Computer Science Technical Report # 542, August 2000. Available at http://www.osl.iu.edu/~tveldhui/papers/techniques/.
  - ➡ Section 1.5 is entitled "Managing Code Bloat."
- "Code Bloat due to Templates," comp.lang.c++.moderated thread, initiated 16 May 2002. Available at http://tinyurl.com/xnhn.
- "C++ Templates vs. .NET Generics," comp.lang.c++.moderated thread, initiated 4 October 2003. Available at http://tinyurl.com/xnhf.
  - ➡ The 13 November posting by Mogens Hansen initiates a good subthread on commonality/variability analysis.
- "Efficient Run-Time Dispatching in Generic Programming with Minimal Code Bloat," Lubomir Bourdev and Jaakko Järvi, *Library-Centric Software Design* (LCSD '06), 22 October 2006.
  - ➡ Describes an advanced code hoisting technique.
- "Minimizing Dependencies withing Generic Classes for Faster and Smaller Programs," Dan Tsafrir *et al.*, OOPSLA '09, October 2009.
  - ➡ Discusses performance costs of unnecessarily template class nesting.

Scott Meyers, Software Development Consultant

Further Information

Inlining:

- "Inline Redux," Herb Sutter, *C/C++ Users Journal*, November 2003.
- "Inline Functions," Randy Meyers, *C/C++ Users Journal*, July 2002.
- *Efficient C++*, Dov Bulka and David Mayhew, Addison-Wesley, 2000, ISBN 0-201-37950-3.
  - *Three chapters* on getting the most out of inlining!
- "Link-Time Code Generation," Matt Pietrek, *MSDN Magazine*, May 2002.
  - Describes link-time inlining in Visual C++ .NET, including how the system determines which functions will be inlined.

Scott Meyers, Software Development Consultant

Further Information

Using link-time polymorphism:

- "Effective Test-Driven Development for Embedded Software," Michael Karlesky *et al.*, IEEE 2006 Electro/Information Technology Conference, May 2006.
  - Uses link-time polymorphism to achieve TDD for C.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Further Information

ROMing objects:

- "Static vs. Dynamic Initialization," Dan Saks, *Embedded Systems Programming*, December 1998 and "Ensuring Static Initialization in C++," *Embedded Systems Programming*, March 1999.
  - ➨ Summarizes when compilers are most likely to ROM data.
- *Technical Report on C++ Performance*, ISO/IEC TR 18015:2006(E).
  - ➨ Discusses what can be ROMed, at least in theory.

Scott Meyers, Software Development Consultant

Further Information

Memory management in embedded systems:

- *Small Memory Software*, James Noble and Charles Weir, Addison-Wesley, 2001, ISBN 0-201-59607-5, chapter 5.
  - ➡ Also explores other topics related to embedded systems software.
- *Real-Time Design Patterns*, Bruce Powel Douglass, Addison-Wesley, 2003, ISBN 0-201-69956-7, chapter 6.
- "Improving Performance for Dynamic Memory Allocation," Marco Varlese, *Embedded Systems Design*, May 2009.
  - ➡ Describes implementation of a block allocator.

Guidelines for understanding, using, and writing `new` and `delete`:

- *Effective C++, Third Edition*, Scott Meyers, Addison-Wesley, 2005, ISBN 0-321-33487-6.
- *Effective C++, Second Edition*, Scott Meyers, Addison-Wesley, 1998, ISBN 0-201-92488-9.
- *More Effective C++*, Scott Meyers, Addison-Wesley, 1996, ISBN 0-201-63371-X.

Both *Small Memory Software* and *Real-Time Design Patterns* are written as collections of patterns, but *Small Memory Software* makes better use of the form, IMO.

Further Information

General information on new, delete, and memory management:

- "Efficient Memory Allocation," Sasha Gontmakher and Ilan Horn, *Dr. Dobbs Journal*, January 1999, pp. 116ff.
- *Modern C++ Design*, Andrei Alexandrescu, Addison-Wesley, 2001, ISBN 0-201-70431-5, chapter 4.
- "Memory Management & Embedded Databases," Andrei Gorine and Konstantin Knizhnik, *Dr. Dobbs Journal*, December 2005.
- "Boost Pool Library," Stephen Cleary, http://www.boost.org/libs/pool/doc/index.html.
- "A Memory Allocator," Doug Lea, http://gee.cs.oswego.edu/dl/html/malloc.html.
- "Scalable Lock-Free Dynamic Memory Allocation," Maged M. Michael, *Proceedings of the 2004 Conference on Programming Language Design and Implementation (PLDI)*, available at http://www.cs.utah.edu/~wilson/compilers/papers/pldi04-michael.pdf.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Further Information

STL allocators:

- *Effective STL*, Scott Meyers, Addison-Wesley, 2001, ISBN 0-201-74962-9.
- *The C++ Programming Language (Third Edition)*, Bjarne Stroustrup, Addison-Wesley, 1997, ISBN 0-201-88954-4, pp. 567-576.
- "Custom STL Allocators," Pete Isensee, http://www.tantalon.com/pete/customallocators.ppt.
  - PPT materials from a 2003(?) Game Developers Conference Talk.
- "Improving Performance with Custom Pool Allocators for STL," Anthony Aue, *Dr. Dobbs Journal*, September 2005.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Further Information

Modeling memory-mapped IO:

- "Register Access in C++," Pete Goodliffe, *C/C++ Users Journal*, May 2005.
- Columns by Dan Saks in *Embedded Systems Programming*, *Embedded Systems Design*, or at embedded.com:
  - "Mapping Memory," September 2004.
  - "Mapping Memory Efficiently," November 2004.
  - "More Ways to Map Memory," January 2005.
  - "Sizing and Aligning Device Registers," May 2005.
  - "Alternative models for memory-mapped devices," May 2010.
  - "Memory-mapped devices as C++ classes," June 22, 2010.
  - "Accessing memory-mapped classes directly," September 2010.
  - "Bundled vs. unbundled monostate classes," November 11, 2010.
  - "Measuring instead of speculating," December 2010.

Embedded Systems Programming was renamed *Embedded Systems Design* in October 2005.

Further Information

More on Modeling memory-mapped IO:

- "Representing and Manipulating Hardware in Standard C and C++," Embedded Systems Conference, Dan Saks, 1999 and subsequent years.
  - 2002 paper available at http://www.open-std.org/jtc1/sc22/wg21/docs/ESC_SF_02_465_paper.pdf.
- *Technical Report on C++ Performance*, ISO/IEC TR 18015:2006(E).
  - Includes information on C's <iohw.h> and C++'s <hardware>.
- "The Embedded C Extension to C," Marcel Beemster *et al.*, *C/C++ Users Journal*, August (Part 1) and September (Part 2), 2005.
  - Discusses <iohw.h>.
    - Purely a C approach – no C++.

Further Information

Compilers and volatile:

- "Volatiles are Miscompiled, and What to Do about It," Eric Eide and John Regehr, *Proc. Eighth ACM and IEEE Intl. Conf. on Embedded Software (EMSOFT)*, October 2008.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Further Information

Implementing callbacks from C:

- "Interrupts in C++," Alan Dorfmeyer and Pat Baird, *Embedded Systems Programming*, August 2001.
- "Applying Design Patterns to Simplify Signal Handling," Douglas C. Schmidt, *C++ Report*, April 1998.
- "Use Member Functions for C-Style Callbacks and Threads – a General Solution," Daniel Lohmann, *The Code Project*, 8 July 2001, http://www.codeproject.com/win32/callback_adapter.asp.
- "Generalizing C-Style Callbacks," http://www.crystalclearsoftware.com/cgi-bin/boost_wiki/wiki.pl?Generalizing_C-Style_Callbacks.
- "Serial Port Design Pattern," http://www.eventhelix.com/RealtimeMantra/PatternCatalog/serial_port_design_pattern.htm.
- "Encapsulating ISRs in C++," Daniel G. Rusch, *Embedded Systems Programming*, February 1998.
- "Interoperability & C++ Compilers," Joe Goodman, *C/C++ Users Journal*, March 2004.

Further Information

ISR issues:

- "Reduce RTOS latency in interrupt-intensive apps," Nick Lethaby, *Embedded Systems Design*, June 2009.
- "Minimize Your ISR Overhead," Nigel Jones, *Embedded Systems Design*, January 2007.
  - ➨ Primarily about avoiding unnecessary register saves/restores during ISR invocations.
- "Modeling Interrupt Vectors," Dan Saks, *Embedded Systems Design*, September 2006.
  - ➨ Focuses on getting ISR addresses into interrupt vector tables.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Further Information

TR1 and Boost:

- *The C++ Standard Library Extensions*, Pete Becker, Addison-Wesley, 2007, ISBN 0-321-41299-0.
  - A comprehensive reference for TR1.
- Scott Meyers' TR1 Information web page, http://www.aristeia.com/EC3E/TR1_info.html.
  - Contains links to proposal documents, articles, books, etc.
- *Effective C++, Third Edition*, Scott Meyers, Addison-Wesley, 2005.
  - Item 35 explains and demonstrates use of tr1::function.
  - The TOC is attached.
- "Generalized Function Pointers," Herb Sutter, *C/C++ Users Journal Experts Forum*, August 2003.
  - Describes std::tr1::function.
- Boost web site, http://www.boost.org/
- *Beyond the C++ Standard Library: An Introduction to Boost*, Björn Karlsson, Addison-Wesley, 2006, ISBN 0-321-13354-4.
  - An overview of selected Boost libraries, including bind and function.

© 2012 Scott Meyers, all rights reserved.

Further Information

Compile-time dimensional unit analysis:

- *Boost.Units*, Matthias C. Schabel and Steven Watanabe, http://www.boost.org/doc/libs/1_46_0/doc/html/boost_units.html.
- "Library for checking dimensional unit correctness?," comp.lang.c++.moderated thread, initiated 22 October 2005. Available at http://tinyurl.com/yrgwt5.
  - ➡ Includes links (from 2005!) to several libraries.
- "Applied Template Metaprogramming in SIUNITS: the Library of Unit-Based Computation," Walter E. Brown, Second Workshop on C++ Template Programming, October 2001. Available at http://www.oonumerics.org/tmpw01/brown.pdf.
- "Dimension Checking of Physical Quantities," Michael Kenniston, *C/C++ Users Journal*, November 2002.
  - ➡ A description of an implementation for less conformant compilers (e.g., Visual C++ 6).

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Further Information

Implementing FSMs:

- *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, David Abrahams and Aleksey Gurtovoy, Addison-Wesley, 2005, ISBN 0-321-22725-5, Chapter 11.
  - Code for FSM example also available at http://boost.org/libs/mpl/example/fsm/player1.cpp.
  - Source code used per the Boost Software License, Version 1.0:

Scott Meyers, Software Development Consultant

Further Information

More on implementing FSMs:

- "UML Tutorial: Finite State Machines," Robert C. Martin, *C++ Report*, June 1998.
- "Yet Another Hierarchical State Machine," Stefan Heinzmann, *Overload*, December 2004.
- "Hierarchical State Machine Design in C++," Dmitry Babitsky, *C/C++ Users Journal*, December 2005.
- "State Machine Design Pattern," Anatoly Shalyto *et al.*, *Proceedings of .NET Technologies 2006*, May 2006.
- "State Machine Design in C++," David Lafreniere, *C/C++ Users Journal*, May 2000.
- "The Anthology of the Finite State Machine Design Patterns," Paul Adamczyk, *Proceedings of PLoP 2003*, September 2003.
  - ➡ A summary of 24 FSM design patterns!
- "The Boost Statechart Library," Andreas Huber Dönni, April 2007, http://www.boost.org/doc/libs/1_36_0/libs/statechart/doc/index.html.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Further Information

The Curiously Recurring Template Pattern (CRTP):

- "Curiously Recurring Template Patterns," James O. Coplien, *C++ Report*, February 1995.
- Many template-oriented C++ books include a discussion of CRTP.

Template metaprogramming:

- *Modern C++ Design: Generic Programming and Design Patterns Applied*, Andrei Alexandrescu, Addison-Wesley, 2001, ISBN 0-201-70431-5.
- *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, David Abrahams and Aleksey Gurtovoy, Addison-Wesley, 2005, ISBN 0-321-22725-5.

Scott Meyers, Software Development Consultant

Further Information

Guidelines for using C++ in safety-critical software:

- *Guidelines for the Use of the C++ Language in Critical Systems*, MISRA, June 2008, ISBN 978-906400-03-3 (hardcopy) or 978-906400-04-0 (PDF).
- *High-Integrity C++ Coding Standard Manual (Version 2.4)*, Programming Research, December 2006.
- "The Power of 10: Rules for Developing Safety-Critical Code," Gerard J. Holzmann, *IEEE Computer*, June 2006.
- *Joint Strike Fighter Air Vehicle C++ Coding Standards for the System Development and Demonstration Program*, Lockheed Martin Corporation, December 2005.
- *Cert C++ Coding Standard*, CERT, http://tinyurl.com/za3hr.
  - ➡ Focuses on developing secure code.
- *Guidelines for the Use of the C Language in Critical Systems*, MISRA, October 2004, ISBN 0 9524156 2 3 (hardcopy) or 0 9524156 4 X (PDF).
  - ➡ Superceded for C++ development by the C++ version (above).

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Further Information

Aspects of C++ with unpredictable (e.g., undefined) behavior:

- *An Investigation of the Unpredicatable Features of the C++ Language*, M. G. Hill and E. V. Whiting, QinetiQ Ltd., May 2004.

Static analysis for safety-critical software:

- "Using Static Analysis to Evaluate Software in Medical Devices," Raoul Jetley and Paul Anderson, *Embedded Systems Design*, April 2008.

Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

Further Information

C++ in real-time systems:

- "Worst Case Execution Time," Wikipedia, http://en.wikipedia.org/wiki/Worst_case_execution_time.
- "Use of Modern Processors in Safety-Critical Applications," Iain Bate *et al.*, *The Computer Journal*, June 2001.
  - Section 4.4. is devoted to WCET analysis.
- "You Can't Control what you Can't Measure...," Nat Hillary and Ken Madsen, *Proceedings of the 2nd Intl. Workshop on Worst Case Execution Time Analysis*, June 2002.
  - Overview of pros/cons of some approaches to analyzing RT software.
    - Written by marketing managers – and it shows.

Scott Meyers, Software Development Consultant

Further Information

C++11:

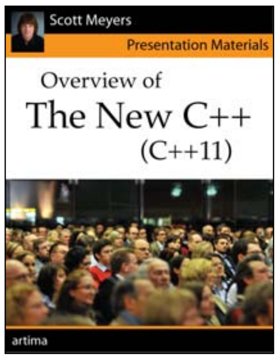
- *"C++11," Wikipedia*.
- *The C++ Standard Library, Second Edition*, Nicolai M. Josuttis, Addison-Wesley, 2012.
- *Overview of the New C++ (C++11)*, Scott Meyers, http://www.artima.com/shop/overview_of_the_new_cpp.
  - ➡ Annotated training materials (analogous to these).
- *C++11 - the recently approved new ISO C++ standard*, Bjarne Stroustrup, http://www.stroustrup.com/C++11FAQ.html.
- cppreference.com.
  - ➡ Reference source for standard C++ and standard C.

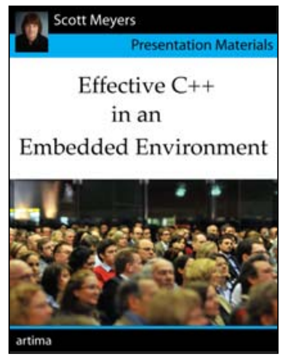
Licensing Information

Scott Meyers licenses materials for this and other training courses for commercial or personal use. Details:

- Commercial use: http://aristeia.com/Licensing/licensing.html
- Personal use: http://aristeia.com/Licensing/personalUse.html

Courses currently available for personal use include:





Scott Meyers, Software Development Consultant
© 2012 Scott Meyers, all rights reserved.

About Scott Meyers

Scott is a trainer and consultant on the design and implementation of C++ software systems. His web site,

http://www.aristeia.com/

provides information on:

- Training and consulting services
- Books, articles, other publications
- Upcoming presentations
- Professional activities blog

Contents

Reprinted from *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*, by Scott Meyers, Addison-Wesley Publishing Company, 2005.

Reprinted from *Effective C++, Third Edition: 55 Specific Ways to Improve Your Programs and Designs*, by Scott Meyers, Addison-Wesley Publishing Company, 2005.

Contents

Reprinted from *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, by Scott Meyers, Addison-Wesley Publishing Company, 1996.

Contents

Reprinted from *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*, by Scott Meyers, Addison-Wesley Publishing Company, 2001.

Reprinted from *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*, by Scott Meyers, Addison-Wesley Publishing Company, 2001.