The Case for C++ in Embedded Systems

Scott Meyers, Ph.D.
Software Development Consultant

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

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

Last Revised: 10/11/10

Beyond Language Choice

Reliably high-quality software based on:

- **Competent management + reasonable development process.**
  - Genuine concern for quality.
  - Suitable requirements analysis and change management.
  - Suitable scheduling/deployment decisions.
  - Suitable resource provision.
- **Competent developers.**
  - Architects, designers, programmers.
  - Understand problem domain + development tools.
    - Language, compiler, linker.
    - Unit testing tools, static + dynamic analysis tools.
  - Apply tools judiciously.
    - E.g., language features.

Beyond Language Choice

Applicable regardless of language and problem domain.

- Bad management/process/developers ⇒ bad software.

Language Usage Constraints

Usage constraints on C and C++ common.

- Generally stricter in embedded environments.
- Stricter still in safety-critical environments.
  - Higher cost of failure ⇒ more restrictive constraints.
- Language subsetting typically part of constraints.
  - From MISRA C and C++:

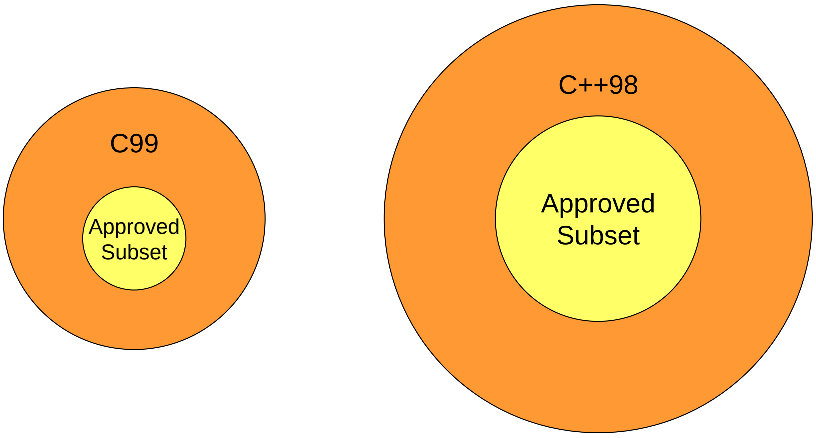
    Identifiers declared in an inner scope shall not hide an identifier declared in an outer scope.

    The comma operator shall not be used.
  - From MISRA C++:

    A class destructor shall not exit with an exception.

Language Subsetting

So which is preferable?



The Case for C++

Constructors and destructors:

- **Automates initialization/finalization of UDTs.**
  - Can't forget.
  - Can't overlook control paths.
- **Enables generalized automatic resource management:**
  - RAII ("Resource Acquisition is Initialization"):
    - Constructor acquires or holds resource.
    - Destructor releases it.

RAII

Many standard library examples:

```
std::mutex m;                                  // C++0x
{
  std::vector<int> v(1000);                    // allocate heap array
  std::ofstream f("data.txt");                 // open file
  std::auto_ptr<Widget> p(new Widget);         // note heap object
  std::lock_guard<std::mutex> lg(m);           // lock mutex (C++0x)
  ...                                          // arbitrarily complex;
                                               // may throw
}                                              // unlock mutex
                                               // delete heap object
                                               // close file
                                               // deallocate heap array
```


RAII

Straightforward to customize:

```
class HoldResourceMgr {                              // e.g.,  std::auto_ptr
private:                                             //        std::shared_ptr
  Resource r;
public:
  explicit HoldResourceMgr(const Resource& src)
  : r(src) {}
  ~HoldResourceMgr() { releaseResource(r); }
  ...                                                // handle copying
};
class AcquireResourceMgr {                           // e.g.,  std::vector,
private:                                             //        std::ofstream,
  Resource r;                                        //        std::lock_guard
public:
  explicit AcquireResourceMgr(const DataForResource& d)
  : r(getResource(d)) {}
  ~AcquireResourceMgr() { releaseResource(r); }
  ...                                                // handle copying
};
```

Beyond Simple RAII

```
class Tracer {                    // log function calls & time spent in them
public:
  explicit Tracer( const char *funcName,
                   std::ostream& stream = std::clog)
  : fn(funcName), log(stream)
  {
    log << "Entering " << fn << '\n';
    startTime = std::time(NULL);
  }
  ~Tracer()
  {
    double ms = std::difftime(std::time(NULL), startTime) * 1000;
    log << "Leaving " << fn << '[' << ms << " ms]\n";
  }
private:
  const char * const fn;          // function name
  std::time_t startTime;
  std::ostream& log;
};
```

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

Slide 9

Beyond Simple RAII

Calls to

```
void someFunction( parameters )
{
  Tracer t(__func__);             // start timer, log entry (C++0x)
   ...
}                                 // stop timer, log exit
```

produce e.g., (in std::clog):

```
Entering someFunction
Leaving someFunction[1000 ms]
```

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

Slide 10

The Case for C++

Classes:

- **Encapsulate members by default.**
  - ➡ Private data members accepted as good practice.
- **Encourages interface/implementation separation.**
- **Encourages programming to interfaces.**
- **Facilitates changing internals w/o breaking client code.**

Changing Class Implementations

```
class Tracer {
private:
  const std::string fn;                          // new
  MyCustomTimeClass startTime;                   // internals
  std::ostream& log;

public:
  explicit Tracer(const char *funcName,          // old
                  std::ostream& stream = std::clog); // interface
  ~Tracer();
};

void someFunction( parameters )
{
  Tracer t(__func__);                            // as before
  ...
}
```

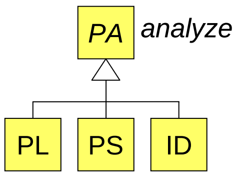
The Case for C++

Inheritance and virtual functions:

- **Manifests cross-type interface commonality.**

```
class Packet { ... };

class PacketAnalyzer {
public:
  virtual void analyze(const Packet& p) = 0;
  ...
};
class PacketLogger:
  public PacketAnalyzer { ... };
class PasswordSniffer:
  public PacketAnalyzer { ... };
class IntrusionDetector:
  public PacketAnalyzer { ... };
```



Acting Polymorphically

- **Automates type-specific implementation selection.**

```
bool getPacket(Packet);
std::vector<PacketAnalyzer*> analyzers;
...
Packet p;
while (getPacket(p)) {
  for (std::vector<PacketAnalyzer*>::iterator it = analyzers.begin();
       it != analyzers.end();
       ++it) {
    (*it)->analyze(p);              // perform appropriate analysis
  }
}
```

Gratuitous Animal Photo

Giant Anteater

The Case for C++

Operator overloading:

- **More readable UDT-based code:**

```
std::vector<Widget> v;
Widget w;
 ...
 v[5] = w;              // std::vector::operator[], Widget::operator=
```

  - **Smart pointers an especially nice application:**
    - C++0x's std::shared_ptr automates reference counting.

```
p1 = p2;           // ++RC for *p1, --RC for *p2
```

    - Can combine with RAII on temps returned from operator->:

```
p->f();            // possibly grab lock, invoke f, release lock;
                   // or start timer, invoke f, stop timer,
                   // etc.
```

- **Inlined operator() faster than call through function pointer.**
  - Makes C++'s sort faster than C's qsort.

The Case for C++

Templates:

- **Facilitate type safety.**
  - The obvious kind, e.g., wrapping `void*` implementations:

```
template<typename T>           // offers push(T), pop(T)
class Stack { ... };

template<typename T>           // for when T is pointer type
class Stack<T*> {
public:
  void push(T* p) { data.push_back(p); }
  T* pop() {
    T* p = (T*) data.back();
    data.pop_back();
    return p;
  }
private:
  std::vector<void*> data;
};
```

  - Clients see type-safe interfaces, object code only for `void*`s.

Dimensional Analysis

- The less obvious kind, i.e., user-defined type relationships.

```
template <int m, int d, int t>          // dimensionally safe
class Units {                           // wrapper for double
  ...
private:
  double value;                         // standardized value,
};                                      // e.g., kg, meters, etc.

typedef Units<0, 1, 0> Distance;
typedef Units<0, 0, 1> Time;
typedef Units<0, 1, -1> Velocity;       // distance/time
typedef Units<0, 1, -2> Acceleration;   // distance/time2

Distance d;
Time t1, t2;

Velocity v = d/t1;                      // okay
d = t1;                                 // error!
Acceleration a = v/t2;                  // okay
a = d/t1;                               // error!
```

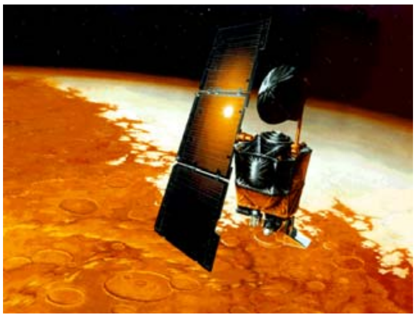

Scott Meyers, Software Development Consultant
http://www.aristeia.com/

Dimensional Analysis

Used to statically dimensionally check, e.g.:

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

Standard internal unit representation could have prevented 1999 loss of NASA's Mars Climate Orbiter.



The Case for C++

More expressive standard library:

- **Containers and algorithms.**
  - Increases maintainability/comprehensibility.
  - Likely better vetted than home-grown versions.
    - Often more efficient, e.g., std::remove, std::sort.
  - Reduces tendency to always use array or list.
    - Deques, balanced trees, hash tables (C++0x) always on call.

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

The Case for C++

- **Better type safety:**
  - ➡ Type-safe support for UDTs.

```
std::list<Widget*> lwp;          // lists are type-safe, use same
std::list<Gadget*> lgp;          // source code, probably same
                                 // object code
```

  - ➡ Different library "helpers" for single vs. array-like objects:

```
std::auto_ptr<int> api;          // object pointer
api[4] = 5;                      // error!  operator[] unavailable
*api = 5;                        // okay

std::shared_ptr<int> spi;        // object pointer (C++0x)
spi[4] = 5;                      // error! still no operator[]
*spi = 5;                        // okay

std::deque<int> d;               // array-like object
d[4] = 5;                        // okay
*d = 5;                          // error! operator* unavailable
```


The Case for C++

Wider choice of third-party libraries:

- **C++ designed to take advantage of C APIs.**
  - ➡ Hence can call anything callable from C.
- **C not designed to call C++ APIs.**

Summary

Compared to C, C++ offers:

- Automatic UDT initialization and finalization.
- RAII-based resource management and derived techniques.
- UDT data encapsulation by default.
- Ability to express cross-type interface commonality.
- Automatic type-appropriate interface implementation selection.
- Natural operator syntax for UDTs.
- Greater type safety.
- More expressive standard library.
- Wider selection of third-party libraries.

Both languages requires developer competence in the language.

Further Information

- "Abstraction and the C++ Machine Model," Bjarne Stroustrup, Keynote address at ICESS04, December 2004.
  - ➡ Overview of 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 use of OO and C++ in 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.
- "C++ in Embedded Systems: Myth and Reality," Dominic Herity, *Embedded Systems Programming*, February 1998.
  - ➡ Dated (but good) overview of C++ vs. C.

Further Information

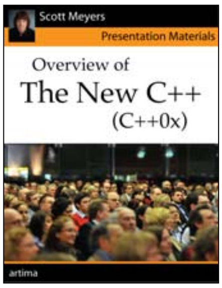
- "Embedded Programming with C++," Stephen Williams, *Third USENIX Conference on Object-Oriented Technologies and Systems (COOTS)*, 1997.
  - ➜ Summarizes design/functionality/performance of a C++ runtime library for embedded systems.
- "C++ in der Automotive-Software-Entwicklung," Matthias Kessler et al., *Elektronik automotive*, May 2006.
  - ➜ How C++ has been useful in embedded automotive software.
- "Applied Template Metaprogramming in SIUNITS: the Library of Unit-Based Computation," Walter E. Brown, *Second Workshop on C++ Template Programming*, October 2001.

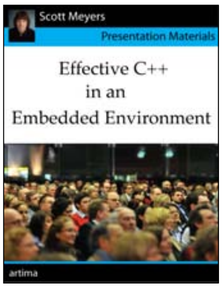
Licensing Information

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:





About Scott Meyers

Scott is a trainer and consultant on the design and implementation of software systems, typically in C++. His web site,

http://www.aristeia.com/

provides information on:

- Training and consulting services
- Books, articles, other publications
- Upcoming presentations
- Professional activities blog

Scott Meyers, Software Development Consultant
http://www.aristeia.com/
© 2010 Scott Meyers, all rights reserved.