Modern C++ in embedded systems – Part 1: Myth and Reality

Dominic Herity (/user/Dominic Herity)

FEBRUARY 17, 2015

Virtual functions

Virtual member functions allow us to derive class B from class A and override a virtual member function of A with one in B and have the new function called by code that knows only about class A. Virtual member functions provide polymorphism, which is a key feature of object-oriented design.

A class with at least one virtual function is referred to as a 'polymorphic' class. The distinction between a polymorphic and a non-polymorphic class is significant because they have different trade-offs in runtime cost and functionality.

Virtual functions have been controversial because they exact a price for the benefit of polymorphism. Let us see, then, how they work and what the price is.

Virtual functions are implemented using an array of function pointers, called a vtable, for each class that has virtual functions. Each object of such a class contains a pointer to that class's vtable. This pointer is put there by the compiler and is used by the generated code, but it is not available to the programmer and it cannot be referred to in the source code. But inspecting an object with a low level debugger will reveal the vtable pointer.

When a virtual member function is called on an object, the generated code uses the object's vtable pointer to access the vtable for that class and extract the correct function pointer. That pointer is then called.

Listing 13 shows an example using virtual member functions. Class A has a virtual member function f(), which is overridden in class B. Class A has a constructor and a member variable, which are actually redundant, but are included to show what happens to vtables during object construction.

```
// Classes with virtual functions

class A {
private:
    int value;
public:
```

MOST READ

11.01.2013
Inline Code in C and C++ (/design/programming-languages-and-tools/4423679/Inline-Code-in-C-and-C-)

11.08.2013
How to know when to switch your SCM/version control system (/design/programming-languages-and-tools/4424039/How-to-know-when-to-switch-your-SCM-version-control-system)

10.26.2013
ARM design on the mbed Integrated Development Environment - Part 1: the basics (/design/programming-languages-and-tools/4423344/ARM-design-on-the-mbed-Integrated-Development-Environment---Part-1--the-basics)

EMBEDDED (/VIDEOS) TV (/VIDEOS) (/videos) video library (/videos)

```
    A();
    virtual int f();
};

A::A() {
    value = 0;
}

int A::f() {
    return 0;
}

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

B::B() {
}

int B::f() {
    return 1;
}

int main() {
    B b;
    A* aPtr = &b;
    aPtr->f();
    return 0;
}
```

Listing 13: Virtual Functions

Listing 14 shows what a C substitute would look like. The second last line in `main()` is a dangerous combination of casting and function pointer usage.

```
/* C substitute for virtual functions */

struct A {
    void **vTable;
    int value;
};


int f_A(struct A* this);

void* vTable_A[] = {
    (void*) &f_A
};

void AConstructor(struct A* this) {
    this->vTable = vTable_A;
    this->value = 1;
}

int f_A(struct A* this) {
    return 0;
}

struct B {
    struct A a;
};

int f_B(struct B* this);

void* vTable_B[] = {
    (void*) &f_B
};

void BConstructor(struct B* this) {
    AConstructor((struct A*) this);
    this->a.vTable = vTable_B;
}

int f_B(struct B* this) {
    return 1;
}

int main() {
```

```
    struct B b;
    struct A* aPtr;

    BConstructor(&b);
    typedef void (*f_A_Type)(struct A*);

    aPtr = (struct A*) &b;
    ((f_A_Type)aPtr->vTable[0]) (aPtr);
    return 0;
}
```

Listing 14: C substitute for virtual functions

This is the first language feature we have seen that entails a runtime cost. So let us quantify the costs of virtual functions.

The first cost is that it makes objects bigger. Every object of a class with virtual member functions contains a vtable pointer. So each object is one pointer bigger than it would be otherwise. If a class inherits from a class that already has virtual functions, the objects already contain vtable pointers, so there is no additional cost. But adding a virtual function can have a disproportionate effect on a small object. An object can be as small as one byte and if a virtual function is added and the compiler enforces four-byte alignment, the size of the object becomes eight bytes. But for objects that contain a few member variables, the cost in size of a vtable pointer is marginal.

The second cost of using virtual functions is the one that generates most controversy. That is the cost of the vtable lookup for a function call, rather than a direct one. The cost is a memory read before every call to a virtual function (to get the object's vtable pointer) and a second memory read (to get the function pointer from the vtable). This cost has been the subject of heated debate and it is hard to believe that the cost is typically less than that of adding an extra parameter to a function. We hear no arguments about the performance impact of additional function arguments because it is generally unimportant, just as the cost of a virtual function call is generally unimportant.

A less discussed, but more significant, cost of virtual functions is their impact on code size. When an application is linked after compilation, the linker can identify regular, non-virtual functions that are never called and remove them from the memory footprint. But because each class with virtual functions has a vtable containing pointers to all its virtual functions, the pointers in this vtable must be resolved by the linker. This means that all virtual functions of all classes used in a system are linked. Therefore, if a virtual function is added to a class, the chances are that it will be linked, even if it is never called.

So virtual functions have little impact on speed, but their effects on code size and data size should be considered. Because they involve overheads, virtual functions are not mandatory in C++ as they are in other object-oriented languages. So if, for a given class, you find the costs outweigh the benefits, you can choose not to use virtual functions.

Templates

C++ templates are powerful, as shown by their use in the Standard C++ Library. A class template is rather like a macro that produces an entire class as its expansion. Because a class can be produced from a single statement of source code, careless use of templates can have a devastating effect on code size. Older compilers will expand a templated class every time it is encountered, producing a different expansion of the class in each source file where it is used. Newer compilers and linkers, however, find duplicates and produce at most one expansion of a given template with a given parameter class.

Used appropriately, templates can save a lot of effort at little or no cost. After all, it's a lot easier and probably more efficient to use complex<float> from the Standard C++ Library, rather than write your own class.

Listing 15 shows a simple template class A<T>. An object of class A<T> has a member variable of type T, a constructor to initialize and a member function `A::f()` to retrieve it.

```
// Sample template class

template<typename T> class A {
private:
    T value;
public:
    A(T);
    T f();
};

template<typename T> A<T>::A(T initial) {
    value = initial;
}

template<typename T> T A<T>::f() {
    return value;
}

int main() {
    A<int> a(1);
```

```
    a.f();
    return 0;
}
```

Listing 15: A C++ template

The macro `A(T)` in **Listing 16** approximates a template class in C. It expands to a `struct` declaration and function definitions for functions corresponding to the constructor and the member function. We can see that although it is possible to approximate templates in C, it is impractical for any significant functionality.

```
/* C approximation of template class */

#define A(T)                                                \
    struct A_##T {                                          \
        T value;                                            \
    };                                                      \
                                                            \
    void AConstructor_##T(struct A_##T* this, T initial) { \
        (this)->value = initial;                            \
    }                                                       \
                                                            \
    T A_f_##T(struct A_##T* this) {                         \
        return (this)->value;                               \
    }

A(int) /* Macro expands to 'class' A_int */

int main() {
    struct A_int a;
    AConstructor_int(&a, 1);
    A_f_int(&a);
    return 0;
}
```

Listing 16: A C 'template'

Exceptions

Exceptions are to `setjmp()` and `longjmp()` what structured programming is to `goto`. They impose strong typing, guarantee that destructors are called, and prevent jumping to a disused stack frame.

Exceptions are intended to handle conditions that do not normally occur, so implementations are tailored to optimize performance for the case where no exceptions are thrown. With modern compilers, exception support results in no runtime cost unless an exception is thrown. The time taken to throw an exception is unpredictable and may be long due to two factors. The first is that the emphasis on performance in the normal case is at the expense of performance in the abnormal case. The second factor is the runtime of destructor calls between an exception being thrown and being caught.

The use of exceptions also causes a set of tables to be added to the memory footprint. These tables are used to control the calling of destructors and entry to the correct catch block when the exception is thrown.

For detailed information on the costs of exceptions with different compilers, see **Effective C++ in an Embedded Environment (http://www.aristeia.com/books.html).**

Because of the cost of exception support, some compilers have a 'no exceptions' option, which eliminates exception support and its associated costs.

```
// C++ Exception example

#include <iostream>
using namespace std;

int factorial(int n) throw(const char*) {
    if (n<0)
        throw "Negative Argument to factorial";
    if (n>0)
        return n*factorial(n-1);
    return 1;
}

int main() {
   try {
         int n = factorial(10);
         cout << "factorial(10)=" << n;
    } catch (const char* s) {
         cout << "factorial threw exception: " << s << "\n";
    }
    return 0;
}
```

Listing 17: A C++ exception example

Listing 17 above shows an example of an exception and **Listing 18** below shows a C substitute that has several shortcomings. It uses global variables. It allows `longjmp(ConstCharStarException)` to be called either before it is initialized by

`setjmp(ConstCharStarException)` or after `main()` has returned. In addition, substitutes for destructor calls must be done by the programmer before a `longjmp()`. There is no mechanism to ensure that these calls are made.

```
     /* C approximation of exception handling */

     #include <stdio.h>
     #include <setjmp.h>

     jmp_buf ConstCharStarException;
     const char* ConstCharStarExceptionValue;

     int factorial(int n) {
         if (n<0) {
         ConstCharStarExceptionValue = "Negative Argument to
factorial";
             longjmp(ConstCharStarException, 0);
         }
         if (n>0)
         return n*factorial(n-1);
         return 1;
     }

     int main() {
         if (setjmp(ConstCharStarException)==0) {
             int n = factorial(10);
             printf("factorial(10)=%d", n);
         } else {
             printf("factorial threw exception: %s\n",
ConstCharStarExceptionValue);
         }
         return 0;
     }
```

Listing 18: A C 'exception' example

For language features discussed up to this point, it has been possible to entertain the possibility of a C substitute as a practical proposition. In this case of exceptions, however, the additional complexity and opportunities for error make a C substitute impractical. So if you're writing C, the merits of exception-safe programming are a moot point.

Runtime type information
The term 'runtime type information' suggests an association with purer object-oriented languages like Smalltalk. This association raises concerns that efficiency may be compromised. This is not so. The runtime cost is limited to the addition of a type_info object for each polymorphic class and type_info objects aren't large.

To measure the memory footprint of a type_info object, the code in **Listing 19** was compiled to assembly. The output was put through the name demangler at www.demangler.com and the result was annotated to highlight the size of the type_info object and the class name string. The result was 30 bytes. This is about the cost of adding a one line member function to each class.

Many compilers have an option to disable runtime type information, which avoids this cost for an application that does not use type_info objects.

```
     // type_info test classes /////////////////
     class Base {
     public:
       virtual ~Base() {}
     };
     class Derived: public Base {};
     class MoreDerived: public Derived {};

/* type_info for more_derived generated by g++ 4.5.3 for cygwin.
Total 30 bytes */
_typeinfo for MoreDerived:
    .long    _vtable for __cxxabiv1::__si_class_type_info 8        /*
4 bytes */
    .long    _typeinfo name for MoreDerived                        /*
4 bytes */
    .long    _typeinfo for Derived                                 /*
4 bytes */
/* ... */
_vtable for MoreDerived:
    .long    0
    .long    _typeinfo for MoreDerived                             /*
4 bytes */
    .long    _MoreDerived::~MoreDerived()
    .long    _MoreDerived::~MoreDerived()
/* ... */
_typeinfo name for MoreDerived:
    .ascii "11MoreDerived\0"                                     /*14
bytes */
```

Listing 19: type_info Memory Footprint Measurement

Part 2: Modern C++ in Embedded Systems: Evaluating C++

(http://www.embedded.com/design/programming-languages-and-tools/4438679/Modern-C--embedded-systems---Part-2--Evaluating-C--?isCmsPreview=true)

Dominic Herity is a Principal Software Engineer at Faz Technology Ltd (http://www.faztechnology.com/). He has 30 years' experience writing software for platforms from 8 bit to 64 bit with full life cycle experience in several commercially successful products. He served as Technology Leader with Silicon & Software Systems and Task Group Chair in the Network Processing Forum. He has contributed to research into Distributed Operating Systems and High Availability at Trinity College Dublin. He has publications on various aspects of Embedded Systems design and has presented at several conferences in Europe and North America. He can be contacted at dherity@gmail.com (mailto:dherity@gmail.com).

< Previous (/design/programming-languages-and-tools/4438660/2/Modern-C--in-embedded-systems---Part-1--Myth-and-Reality) Page 3 of 3 Next >

16 COMMENTS

WRITE A COMMENT

Freddie Chopin (/User/Freddie%20Chopin) **POSTED: DEC 3, 2016 5:12 PM EST**

Yeah... This whole mythology surrounding C++ in embedded will probably never change. But I'm also trying to fight this myth by doing something that many people will consider a complete nonsense - I'm writing an RTOS for ARM Cortex-M microcontrollers in C++11 - http://distortos.org/ (http://distortos.org/) . So far it's working perfectly fine, and now I don't have to write all those C++ wrappers for "other" RTOSes to use a member function in a thread or to have a mutex globaly initialized by a constructor. Not to mention all the simplification caused by RAII and similar patterns not even imaginable in pure C. Try using a lambda with 10 arguments as a thread function or passing a complex object via message queue. Not possible in a typical RTOS written in C. Quite easy to do in C++11.

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F3%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)

Dominic Herity (/User/Dominic%20Herity) **POSTED: DEC 18, 2016 2:00 PM EST**

Writing a small footprint RTOS in C++11 is a good way to demonstrate its suitability for the environment.

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F3%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)

Freddie Chopin (/User/Freddie%20Chopin) **POSTED: JAN 25, 2017 3:35 AM EST**

Hello Dominic and thanks for your reply!

Personally I find the importance of size to be a bit overstated. Don't get me wrong - I don't say that it's OK for the RTOS to use 1MB of flash and 256kB of RAM just to blink a LED - I'm far from that. But at the same time I also far from saying that any scheduler which uses more than 1000 bytes of flash and 50 bytes of RAM is bloated and should never be used. The numbers are obviously picked at random, but I guess you see my point. It's all about balance and trade-offs.

The first reason is that the size of firmware is usually directly proportional to the features it offers. A framework which provides 10 "features" will be smaller than a framework which provides the same 10 "features" and one feature more. A RTOS which provides detachable threads or POSIX signals will be bigger than a RTOS which only provides the bare minimum.

Reason number two - the size and speed are often in opposition. Bubble sort will no doubt be smaller than heapsort, but the later will definitely be faster. Usually (not always, but quite often) the same is true for any kind of algorithm or piece of code.

Last but not the least - for large projects using C++ is quite a good idea, as it simplifies a lot of things. Projects which need multithreading are usually large anyway. In a large project - which uses ~100kB of flash - it makes little difference whether a RTOS uses 1kB or 10kB of flash, especially when some parts of these 10kB are most likely used by the application anyway (some standard functions from C or C++ libraries).

Anyway - the whole RTOS that I've written fits into a typical ARM Cortex-M microcontrollers without problems, leaving plenty of space for the application (;

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F3%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)

OdinHolmes (/User/OdinHolmes) **POSTED: JUL 4, 2016 6:07 AM EDT**

I think a lot of C fans are missing the point. If you write C++ code like you would write C code there is no difference. If you write C++ code like your were writing code for a desktop then it will be slow and huge. C++ brings new tools and new tools bring new paradigms and patterns. Most notably zero cost abstraction brings encapsulation of expertise. Templates bring -smarter- libraries, policy based class design and static aliasing. Its a brave new world and we need more library devs ;)

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F3%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)

Dominic Herity (/User/Dominic%20Herity) **POSTED: DEC 18, 2016 2:02 PM EST**

All true.

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F3%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)

FredHansen (/User/FredHansen) **POSTED: APR 24, 2016 6:55 PM EDT**

It's not quite true that C++ is a superset of C. Many of the things are violate the superset idea are good (stricter type checking for example).

I agree that C++ does not always result in larger compiled code. However, final implementation are often bigger and slower when written in C++. I think the issue is that slick/cute new options in C++ (new/delete, templates, multiple inheritance, ...) are like new toys for FW engineers. We can help playing with them. The final product is often bigger as a result.

I was writing embedded system code back when everyone said you had to use assembly and C was the new comer. We worried that the final code would be too big and too slow, ... Eventually compilers got good enough and fast processors and memory got cheap enough that C became the right answer. Also the problems got complex enough that we just could not reasonably attack them with assembly.

The same thing will likely happen again. Either C++ or some other OO language will eventually dominate embedded systems. If there is that much C code around then we simply not there yet.

I'd also point out that C++ is a LOT bigger than C. It has a lot of toys in it. The transition might happen sooner if C++ were a bit smaller.

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F3%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)

Dominic Herity (/User/Dominic%20Herity) **POSTED: APR 25, 2016 2:11 PM EDT**

All true. My motivation for the original 1998 article was to challenge the often stated proposition that C++ is _inherently_ unsuitable for embedded systems. It was aimed at C programmers who were thinking about trying it, but deterred by inaccurate assertions.
I wanted to shed some light on how it works and suggested that it be adopted piecemeal to maintain productivity and minimize risk while getting comfortable with the language. I hope it convinced some people to try it and that they found the experience rewarding.

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F3%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)

HannaD573 (/User/HannaD573) **POSTED: APR 14, 2016 10:35 AM EDT**

Why I will never use c++ for small mcus:
1. Too often, it is assumed that a highly skilled C engineer can just start using C++ as a better "C". Not true - C++ has a learning curve far beyond "C". I dare say many C++ engineers do not fully understand C++.

2. When I malloc, I know the exact size of the memory block, and need to carfully track and control this - when I "new" an object I really have no idea how much I malloc'ed - further the object could possibly have contained objects that are allocated in the ctor.
3. Due to the constrained size of the environment, I can't really use OO design effectively.
Instead of your Quixote crusade why not accept that this is not going to happen, and instead push for better embedded design.

Clean Code is not about the language.

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F3%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)

Dominic Herity (/User/Dominic%20Herity) **POSTED: APR 15, 2016 5:04 PM EDT**

I don't just assume that C++ can be used as a better C. I assert it.
Take your example of malloc versus new. I don't agree that malloc if preferable, but there's nothing to stop you using it in C++, if that's what you really want.
Anything you can do in C, you can do in C++. No problem.
My own experience is that OOD makes code clearer when it reaches somewhere between 100 and 1000 lines, so I would contend that OOD can improve clarity even in systems with a few KB of code.
Of course, C++ offers more than just OOD - exceptions, templates, etc. Templates can benefit very small systems by doing compile time calculations that can't be done in C.
Of course, small systems have to be handled with skill and are unforgiving of poor design choices.
Think of C++ as a bigger toolbox than C. Some of the extra tools should be used with care, but don't blame the toolbox if you misuse a tool. The purpose of the article was to help C programmers understand how some of the extra tools work, so that they can make an informed decisions about when to use them.
I'm all for better embedded design and refusing to use a more powerful, more expressive, language because it can be misused is a mistake. All languages can be misused and more powerful ones can be misused more.

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F3%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)

Wouter Van Ooijen (/User/Wouter%20van%20ooijen) **POSTED: JAN 3, 2016 11:56 AM EST**

I'd like to add one disadvantage of virtual functions: they are an inbline & optimization barrier. This can make a hughe difference in performance (in addition to code size) for functions at or near the edges of the call tree. when the object relations are known only at run time, this is unavoidable. But for situations where the object tree is known at compile time compile-time composition via templates can (IMO nicely) solve this problem. (As I argue in "Objects? No Thanks", check http://www.embedded.com/design/programming-languages-and-tools/4428377/Objects--No--thanks---Using-C--effectively-on-small-systems-(http://www.embedded.com/design/programming-languages-and-tools/4428377/Objects--No--thanks---Using-C--effectively-on-small-systems-) or https://www.youtube.com/watch?v=k8sRQMx2qUw (https://www.youtube.com/watch?v=k8sRQMx2qUw))

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F3%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)

Dominic Herity (/User/Dominic%20Herity) **POSTED: JAN 3, 2016 3:07 PM EST**

This is true. Thanks for pointing it out. I was comparing a virtual function to a function in a separate translation unit that the compiler can't access for optimisation, but this can't be assumed.

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F3%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)

OdinHolmes (/User/OdinHolmes) **POSTED: JUL 4, 2016 6:13 AM EDT**

Building on the idea in "Objects? No Thanks" policy based class design in general allows "open closed" idiom with zero overhead. A well designed C++ library can have a fraction of the flash footprint of its C counterpart.

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F3%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)

Onkar Raut (/User/Onkar%20Raut) **POSTED: JUN 12, 2015 1:20 PM EDT**

Out of curiosity, what would be your opinion on working with peripheral functions? From my experience, singleton classes are a fair choice, but seem to be an inappropriate when you have multiple iterations of the same peripheral e.g. a timer function.

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F3%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)

Dominic Herity (/User/Dominic%20Herity) **POSTED: JUN 17, 2015 12:21 PM EDT**

Singletons have many problems. See http://stackoverflow.com/questions/137975/what-is-so-bad-about-singletons.(http://stackoverflow.com/questions/137975/what-is-so-bad-about-singletons.)
For cases like you describe, I tend to use a function that returns a reference to an object that is statically allocated inside the function.
Such an object is constructed the first time the function is called, which gives you lazy initialization, one of the good things about the singleton.
So you can do something like this:

```
class uart;
uart& console() {
```

```
static uart the_console(...);
return the_console;
}
```

Its a little more complicated if multiple threads can make the first call.

If you have multiple objects of the same class, the function could take a numeric parameter and return a reference to the correct one. This case is more complex because the objects will generally need different constructor parameters, so a statically allocated array won't do it.
Something like:

```
class timer;
timer& system_timer(unsigned n) {
static std::vector<timer> the_timers;
if (the_timers.empty()) {
// Set them up
}
return the_timers[n];
}
```

You could also use a template like this.

```
template <unsigned n> system_timer() {
static timer the_timer(...); // constructor oparameters depend on n.
return the_timer;
}
```

You'd need to make sure the template is only instantiated once for each n. This has the benefit that you get a compile time error if you try to access a non-existent device.

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F3%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)

OdinHolmes (/User/OdinHolmes) **POSTED: JUL 4, 2016 6:31 AM EDT**

Here my rebuttal of the SO post for embedded specific:
[1] embedded programs are smaller which mitigates the problem, model the singleton as a template class and you can always pass the class object around (its empty anyway the optimizer will remove it in most cases) as a kind of handle.

[2] its usually most efficient to initialize at start up any way in embedded so initialization is less of an issue. There are cases where initialization order is an issue which is really the last standing argument against singletons.

[3] if your singletons are template classes you can use the traits pattern to mock them out, no show stopper here.

[4] mock them out in unit tests. Endless object lifetime avoids the heap (fragmentation, non deterministic timing etc.) and is therefore an advantage in my mind. Heap must be as big as the worst case set of simultaneously used objects plus heap overhead plus fragmentation penalty. If the singleton can be part of the worst case set of objects removing it from the heap and giving it static lifetime is actually a huge RAM savings (saves overhead and fragmentation penalty) although this may seem counter intuitive at first.

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F3%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)

OdinHolmes (/User/OdinHolmes) **POSTED: JUL 4, 2016 6:15 AM EDT**

Make the peripheral instance a template parameter, its like function overloading but for singletons.

REPLY (/LOGIN?ASSETID=203&SUCCESSFULLOGINREDIRECT=HTTP%3A%2F%2FWWW.EMBEDDED.COM%2FDESIGN%2FPROGRAMMING-LANGUAGES-AND-TOOLS%2F4438660%2F3%2FMODERN-C--IN-EMBEDDED-SYSTEMS---PART-1--MYTH-AND-REALITY)

Connectivity (/development/connectivity)

Debug & Optimization (/development/debug-and-optimization)

MCUs, Processors & SoCs (/development/mcus-processors-and-socs)

Operating Systems (/development/operating-systems)

Power Optimization (/development/power-optimization)

Programming Languages & Tools (/development/programming-languages-and-tools)

Prototyping & Development (/development/prototyping-and-development)

Real-time & Performance (/development/real-time-and-performance)

Real-world Applications (/development/real-world-applications)

Safety & Security (/development/safety-and-security)

System Integration (/development/system-integration)

Webinars (/education-training/webinars)

Courses (/education-training/courses)

Tech Papers (/education-training/tech-papers)

Newsletters (/archive/Embedded-com-Tech-Focus-Newsletter-Archive)

Videos (/videos)

Collections (/collections/all)

Advertising (/advertising)

Editorial Contributions (/editorial-contributions)

Site Map (/site-map)

GLOBAL NETWORK **EE Times Asia (http://www.eetasia.com/)** **EE Times China (http://www.eet-china.com/)** **EE Times Europe (http://www.electronics-eetimes.com/)** **EE Times India (http://www.eetindia.co.in/)** **EE Times Japan (http://eetimes.jp/)** **EE Times Korea (http://www.eetkorea.com/)** **EE Times Taiwan (http://www.eettaiwan.com/)** **EDN Asia (http://www.ednasia.com/)** **EDN China (http://www.ednchina.com/)** **EDN Japan (http://ednjapan.com/)** **ESC Brazil (http://www.escbrazil.com.br/en/)**

(http://ubmcanon.com/)

Communities

EE Times (http://www.eetimes.com/) | **EDN (http://www.edn.com/)** | **EBN (http://www.ebnonline.com/)** | **DataSheets.com (http://www.datasheets.com/)** | **Embedded (http://www.embedded.com/)** | **TechOnline (http://www.techonline.com/)** | **Planet Analog (http://www.planetanalog.com/)**

Working With Us: About (http://www.aspencore.com/) | Contact Us (http://www.aspencore.com/#contact) | Media Kits (http://go.aspencore.com/mediakit)