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

Dominic Herity (/user/Dominic Herity)

FEBRUARY 17, 2015

In 1998, I wrote an article for Embedded Systems Programming called **C++ in Embedded Systems – Myth and Reality (http://www.eetindia.co.in/ARTICLES/1998FEB/PDF/EEIOL_1998FEB02_EMS_TA.pdf)**. The article was intended to inform C programmers concerned about adopting C++ in embedded systems programming.

A lot has changed since 1998. Many of the myths have been dispelled, and C++ is used a lot more in embedded systems. There are many factors that may contribute to this, including more powerful processors, more challenging applications, and more familiarity with object-oriented languages.

C99 (an informal name for ISO/IEC 9899:1999) adopted some C++ features including const qualification and inline functions. C++ has also changed. C++11 and C++14 have added some cool features (how did I manage without the auto type specifier?) and some challenges, like deciding when to use constexpr functions.

But C++ has not displaced C, as I thought it would in 1998. C is alive and well in the Linux kernel, and there is a body of opinion implacably opposed to C++ in that environment.

The suspicion lingers that C++ is somehow unsuitable for use in small embedded systems. For 8- and 16-bit processors lacking a C++ compiler, that may be a concern, but there are now 32-bit microcontrollers available for under a dollar supported by mature C++ compilers. As this article series will make clear, with the continued improvements in the language most C++ features have no impact on code size or on speed. Others have a small impact that is generally worth paying for. To use C++ effectively in embedded systems, you need to be aware of what is going on at the machine code level, just as in C. Armed with that knowledge, the embedded systems programmer can produce code that is smaller, faster and safer than is possible without C++.

My history with C++

When I started a new microcontroller project a few years ago, I had to choose a tool-chain for the project. The MCU used (NXP LPC2458) was a 72MHz ARM7 with 512KB FLASH and 64KB RAM. Some toolchain vendors were surprised to be asked about the memory footprint of C++ libraries. When one vendor was pressed on the issue of a bloated library component, they said not many people are using C++ in such resource-constrained devices and it's hard to justify the cost of improving the library. Bear in mind that this "resource-constrained device" was somewhat more powerful than the DOS platform that ran commercial software written in C++ in the 90s.

So in 2015, it seems that there's still a need to de-mystify C++ for software engineers who are expert in embedded systems and in C, but wary of C++. If you're not familiar

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)

with C++, if you find that not many people are using it for applications like yours and if it's considered unsuitable for the Linux kernel, this wariness is understandable.

This is a revised version of the 1998 article addressing this issue. Less attention is given to features present in C99, since C programmers are likely to be familiar with them. The reader is assumed to be familiar with C99, which is used in the C code examples. The reader is also assumed to understand the C++ language features discussed, but doesn't need to be a C++ expert. A reader that is unfamiliar with some language features can still get value from this article by skipping over those features. The intended use of C++ language features and why they might be preferable to alternatives is also beyond the scope of this article.

This article aims to provide a detailed understanding of what C++ code does at the machine code level, so that readers can evaluate for themselves the speed and size of C++ code as naturally as they do for C code.

To examine the nuts and bolts of C++ code generation, we will discuss the major features of the language and how they are implemented in practice. Implementations will be illustrated by showing pieces of C++ code followed by the equivalent (or near equivalent) C code. We will then discuss some pitfalls specific to embedded systems and how to avoid them.

We will not discuss the uses and subtleties of the C++ language or object-oriented design, as these topics have been well covered elsewhere. See http://en.cppreference.com/w/ for explanations of specific C++ language features.

C++11 and C++14 features are discussed separately in sections towards the end. The bulk of the article applies to the C++03 version of the language. C++11 is backward compatible with C++03 and C++14 is backward compatible with C++11. This helps the reader to ignore advanced features on a first reading and come back to them later.

Myths about C++. Some of the perceptions that discourage the use of C++ in embedded systems are:

- C++ is slow.
- C++ produces bloated machine code.
- Objects are large.
- Virtual functions are slow.
- C++ isn't ROMable.
- Class libraries make large binaries.
- Abstraction leads to inefficiency.

Most of these ideas are wrong. When the details of C++ code generation are examined in detail, hopefully it will be clear what the reality behind these myths is.

Anything C does, C++ can do. One property of C++ is so obvious that it is often overlooked. This property is that C++ is almost exactly a superset of C. If you write a code fragment (or an entire source file) in the C subset, the compiler will usually act like a C compiler and the machine code generated will be what you would get from a C compiler. (See Compatibility of C and C++ (http://en.wikipedia.org/wiki/Compatibility_of_C_and_C%2B%2B) for information about C constructs that won't compile as C++)

Because of this simple fact, anything that can be done in C can also be done in C++. Existing C code can typically be re-compiled as C++ with about the same amount of difficulty that adopting a new C compiler entails. This also means that migrating to C++ can be done gradually, starting with C and working in new language features at your own pace. Although this is not the best way to reap the benefits of object-oriented design, it minimizes short term risk and provides a basis for iterative changes to a working system.

Front end features - a free lunch

Many of the features of C++ are strictly front-end issues. They have no effect on code generation. The benefits conferred by these features are therefore free of cost at runtime.

Default arguments to functions are an example of a cost-free front end feature. The compiler inserts default arguments to a function call where none are specified by the source.

A less obvious front end feature is 'function name overloading'. Function name overloading is made possible by a remarkably simple compile time mechanism. The mechanism is commonly called 'name mangling', but has also been termed 'name decoration'. Anyone who has seen a linker error about the absence of `?my_function@@YAHH@Z` knows which term is more appropriate.

Name mangling modifies the label generated for a function using the types of the function arguments, or function signature. So a call to a function void my_function(int) generates a label like `?my_function@@YAXH@Z` and a call to a function void `my_function(my_class*)` generates a label like `?my_function@@YAXPAUmy_class@@@Z`. Name mangling ensures that functions are

not called with the wrong argument types and it also allows the same name to be used for different functions provided their argument types are different.

Listing 1 shows a C++ code fragment with function name overloading. There are two functions called `my_function`, one taking an `int` argument, the other taking a `char const*` argument.

```
// C++ function name overload example
void my_function(int i) {
  // ...
}

void my_function(char const* s) {
  // ...
}

int main() {
   my_function(1);
   my_function("Hello world");
   return 0;
}
```

Listing 1: Function name overloading

Listing 2 shows how this would be implemented in C. Function names are altered to add argument types, so that the two functions have different names.

```
/* C substitute for function name overload */

void my_function_int(int i) {
    /* ... */
}

void my_function_charconststar(char const* s) {
    /* ... */
}
int main() {
    my_function_int(1);
    my_function_charconststar ("Hello world");
    return 0;
}
```

Listing 2: Function name overloading in C

References

A reference in C++ is physically identical to a pointer. Only the syntax is different. References are safer than pointers because they can't be null, they can't be uninitialized, and they can't be changed to point to something else. The closest thing to a reference in C is a const pointer. Note that this is not a pointer to a const value, but a pointer that can't be modified. **Listing 3** shows a C++ code fragment with a reference.

```
// C++ reference example
void accumulate(int& i, int j) {
    i += j;
}
```

Listing 3: C++ reference

Listing 4 shows how this would be implemented in C.

```
/* C substitute for reference example */
void accumulate(int* const i_ptr, int j) {
    *i_ptr += j;
}
```

Listing 4: Reference in C

Classes, member functions and objects

Classes and member functions are the most important new concept in C++. Unfortunately, they are usually introduced without explanation of how they are implemented, which tends to disorient C programmers from the start. In the subsequent struggle to come to terms with object-oriented design, hope of understanding code generation quickly recedes.

But a class is almost the same as a C `struct`. Indeed, in C++, a `struct` is defined to be a class whose members are public by default. A member function is a function that takes a pointer to an object of its class as an implicit parameter. So a C++ class with a member function is equivalent, in terms of code generation, to a C struct and a function that takes that `struct` as an argument.

Listing 5 shows a trivial class A with one member variable x and one member function f().

```
// A trivial class

class A {
private:
    int x;
```

```
public:
    void f();
};

void A::f() {
    x = 0;
}
```

Listing 5: A trivial class with member function

Parts of a class are declared as private, protected, or public. This allows the programmer to prevent misuse of interfaces. There is no physical difference between private, protected, and public members. These specifiers allow the programmer to prevent misuse of data or interfaces through compiler enforced restrictions.

Listing 6 shows the C substitute for Listing 5. `Struct A` has the same member variable as class A and the member function `A::f()` is replaced with a function `f_A(struct A*)`. Note that the name of the argument of `f_A(struct A*)` has been chosen as "this", which is a keyword in C++, but not in C. The choice is made deliberately to highlight the point that in C++, an object pointer named this is implicitly passed to a member function.

```
/* C substitute for trivial class A */

struct A {
    int x;
};

void f_A(struct A* this) {
    this->x = 0;
}
```

Listing 6: C substitute for trivial class with member function

An object in C++ is simply a variable whose type is a C++ class. It corresponds to a variable in C whose type is a struct. A class is little more than the group of member functions that operate on objects belonging to the class. When an object-oriented application written in C++ is compiled, data is mostly made up of objects and code is mostly made up of class member functions.

Clearly, arranging code into classes and data into objects is a powerful organizing principle. Clearly also, dealing in classes and objects is inherently no less efficient than dealing with functions and data.

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