C++ on Embedded Systems

Matt Kline
matt@bitbashing.io

February 26, 2017

This October, my team at work switched from C to C++ for embedded firmware development. Many of its features, including classes, automatic resource cleanup, parametric polymorphism, and additional type safety are just as useful on an RTOS or bare metal as they are on a desktop running a general-purpose OS. Using C++ lets us write safer, more expressive firmware.

C++'s automagic is a double-edged sword, however. Some language features depend on system facilities that we don't want to provide in embedded environments.[*] Wrangling the toolchain can also be difficult. We don't want to completely discard `libgcc` and `libstdc++` since they provide vital facilities like `memcpy`, atomic operations, and hardware-specific floating-point functions, but we must avoid certain parts of them.

This guide is a short attempt to codify what we've learned while moving our firmware to C++. Hopefully it provides a solid primer.

Setup

Building a toolchain

The good news is that GCC works very well as a cross-compiler for all sorts of targets, including the ARM systems we usually use for embedded development.[†] While some variant can usually be installed from your Linux distribution's package manager, it is *highly* recommended that teams build and use their own cross-compiler. This has several advantages:

- By having the entire team use the same version of the same toolchain, everyone should get identical builds. This is quite helpful for debugging and testing.
- The pace of compiler development has increased with the recent updates to C++, and newer versions offer significantly improved code generation in some cases.[‡] During the development of a previous project, we also ran into compiler bugs on older (4.8.x) versions which caused our system to crash.

Building an entire cross-compiler toolchain is usually an arduous task, but we've had good success using crosstool-NG. It helps you configure your toolchain using an interface similar to Linux's `make nconfig`, manages and downloads dependencies, and performs the builds for you. Recent versions also allow you to provide arbitrary GCC sources, such as its latest release. The resulting binaries can be statically linked, so deployment becomes as simple as tarballing the toolchain and placing it somewhere accessible. Projects using the toolchain can then use a short script to pull it down, extract it, and run it.

[*]Dynamic memory allocation is the simplest and most prevalent example. We usually want to avoid it—at least after startup—on a real time embedded system, but exception handling and many other C++ features demand it.
[†]Clang also seems to provide solid tools for cross-compilation, but to date, it hasn't been tried for our firmware.
[‡]For an example, see `https://gcc.gnu.org/bugzilla/show_bug.cgi?id=69878`.

Linking C code

Embedded projects often have plenty of C dependencies, such as manufacturer-provided drivers and the RTOS. Build them using `gcc` and wrap any of their headers you `#include` with `extern "C" { }`. Similarly, any C++ functions that you want to call from a non-C++ environment—e.g., RTOS functions or startup assembly—must be tagged with `extern "C"`. This instructs the compiler not to mangle symbol names as it normally would.

Compiler flags

Exception handling and RTTI are difficult to provide without dynamic memory allocation (much more on that below), so you likely want to disable them with `-fno-exceptions`, `-fno-non-call-exceptions`, and `-fno-rtti`. And while it may reboot, firmware never exits in the same sense as a userspace program does. Teardown code (including global destructors) can be omitted with `-fno-use-cxa-atexit`. Other useful flags for embedded development include:

`-ffreestanding`, which indicates that your program exists in an environment where standard library facilities may be absent and where your program may not begin at `main()`.

`-fstack-protector-strong`, which is discussed later.

`-fno-common`, which ensures that each global variable is only declared once, in a single object. This may improve performance on some targets.

`-ffunction-sections` and `-fdata-sections`, which split functions and data into their own ELF sections. This allows the linker to eliminate additional unused code when passed `--gc-sections`.

These aren't specific to C++, but are worth mentioning here.

Enabling language features

As mentioned above, several useful C++ features require underlying system support. In a bare metal or RTOS environment, we'll need to provide it ourselves.

Disclaimer: Most of this is, of course, implementation-specific. Everything that follows is based on our experiences using ARM Cortex-M4 boards with GCC 6. Hopefully this provides a useful starting point, even if details change.

Global object initialization

Global objects can be quite useful for defining interfaces to hardware resources in an embedded system, and these objects might have constructors. The C++ runtime normally guarantees that all global (or file-local) objects are constructed before entering `main()`, but in an embedded environment, we must call the constructors ourselves. GCC groups them into an array of function pointers under the symbol name `.init_array`. After adding an entry to the linker script resembling:

```
. = ALIGN(4);
.init_array :
{
    __init_array_start = .;
    KEEP (*(.init_array*))
    __init_array_end = .;
} > FLASH
```

we can call the functions like so:

```
static void callConstructors()
{
    // Start and end points of the constructor list,
    // defined by the linker script.
    extern void (*__init_array_start)();
    extern void (*__init_array_end)();

    // Call each function in the list.
    // We have to take the address of the symbols, as __init_array_start *is*
    // the first function pointer, not the address of it.
    for (void (**p)() = &__init_array_start; p < &__init_array_end; ++p) {
        (*p)();
    }
}
```

When to perform this step is another question. Should it be after hardware initialization? After RTOS setup, but before your tasks begin executing? In the first RTOS task? Depending on what you decide, it may be prudent to ensure these constructors don't make OS calls or modify hardware state—besides RAM, of course.

Inheritance

Judicious use of inheritance and run time polymorphism can be quite useful on embedded systems. However, `operator delete` is required whenever we give a base class a virtual destructor—as is standard practice—even if we never heap-allocate an object of that class.* The `libgcc` versions assume a Unix-like userspace, so we should define our own. If we're trying to avoid dynamic memory allocation, a call to `delete` is probably a serious bug, and we should likely panic.

```
void operator delete(void* p)
{
    DIE("delete called on pointer %p (was an object heap-allocated?)", p);
}

// Same as above, just a C++14 specialization.
// (See http://en.cppreference.com/w/cpp/memory/new/operator_delete)
void operator delete(void* p, size_t t)
{
    DIE("delete called on pointer %p, size %zu", p, t);
}
```

*See http://stackoverflow.com/q/31686508 and http://eli.thegreenplace.net/2015/c-deleting-destructors-and-virtual-operator-delete/ for details. Kind readers have pointed out that we should avoid virtual destructors altogether if we don't need polymorphic deletion. Adding one increases the size of the class, since a vtable is needed.

If objects with virtual destructors are stack-allocated, a version of the destructor without `operator delete` is used.

Unrecommended language features

Scoped, static objects

Consider some function with a `static` variable:

```
void foo()
{
    static Bar someObject;
    // Do some work with someObject here.
}
```

If the object can be trivially initialized via placement in `.data` or `.bss` sections, there is no problem here. The trouble arises if a constructor must be called at run time to initialize `someObject`. C++11 guarantees that the construction of local static objects is race-free. That is, if multiple threads call `foo()` at once, the compiler must provide some locking mechanism[*] to ensure that the object's initial value is decided by a single thread.

Since we might call functions in our system *before* the OS is running and can provide meaningful locking, and since we generally want to do all initialization in an embedded system at startup to make subsequent code as deterministic as possible, it's instead recommended that any static objects are placed at the file level. Alternatively, one can compile with `-fno-threadsafe-statics` if they are *positive* that functions with static objects will not be called concurrently.

Exceptions

Modern exception-handling mechanisms are complex,[†] and most implementations—including `glibc`'s—requires dynamic memory allocation and other facilities we don't have. Third-party solutions like `libunwind` also assume they sit atop some Unix-like userspace. Because of these complexities, we haven't attempted using exceptions for our embedded projects.

If you are interested in overcoming these hurdles, work Rian Quinn presented at CppCon 2016[‡] seems to be a good starting point. In order to run C++ code in the Linux kernel, he built his own stack unwinding library, which can be found at `https://github.com/Bareflank/hypervisor/tree/master/bfunwind`.

Miscellaneous tools and notes

Stack overrun detection

Consider a function that allocates some storage on the stack:

---

*See `http://preshing.com/20130930/double-checked-locking-is-fixed-in-cpp11/` for the usual approach.
†See `https://mentorembedded.github.io/cxx-abi/abi-eh.html` for the full spec used by GCC and Clang.
‡The presentation can be found at `https://www.youtube.com/watch?v=uQSQy-7lveQ`.

```
void bar()
{
    char arrayOnStack[10];
    // ...read and write to the local array...
}
```

If we walk off the end of the array, we can corrupt the memory following the current stack frame, enter Undefined Behavior land, and can't reason about the state of our program anymore. Fun is not had. Fortunately, the compiler can provide a slight safety net at the cheap cost of one write and one read per function. When given one of the flags below, GCC transforms the function above into something resembling the following:

```
void bar()
{
    // Assuming the stack grows down, out-of-bounds writes to arrayOnStack
    // may clobber _canary.
    uintptr_t _canary = __stack_chk_guard;

    char arrayOnStack[10];
    // ...read and write to the local array...

    if (_canary != __stack_chk_guard) {
        // We have done terrible things to our stack. Panic.
        __stack_chk_fail();
    }
}
```

Different flags offer control over how often the compiler adds these checks. To save you a trip to the man pages,

`-fstack-protector` emits guards for "functions that call `alloca`, and functions with [character] buffers larger than 8 bytes."

`-fstack-protector-strong` emits guards for "[functions] that have local array definitions, or have references to local frame addresses." This is the generally recommended setting.

`-fstack-protector-all` emits guards for every function. This is probably overkill and too expensive for an embedded system.

`-fstack-protector-explicit` emits guards for functions marked by a `stack_protect` attribute. This is probably too tedious to use effectively.

To use these guards, we must provide both of the symbols shown in the example above. One is the canary, and the other is a function to call when stack corruption is detected. Panicking or rebooting is probably the only sane recourse here.

```
extern "C" {

// The canary value
extern const uintptr_t __stack_chk_guard = 0xdeadbeef;

// Called if the check fails
[[noreturn]]
void __stack_chk_fail()
{
    DIE("Stack overrun!");
}
} // end extern "C"
```

The compiler handles the rest. It's worth noting that if we are particularly unlucky and manage to read or write past the current stack frame *without* modifying the canary, this system won't notice the corruption. Life is hard sometimes.

On inlining and optimization

In embedded systems, we're often under pressure to keep our code as small as possible. Inlining may seem antithetical to that goal, but this isn't always the case. Placing trivial code in headers as `inline` functions allows modern C++ compilers to generate incredibly small and efficient output. See Jason Turner's *Rich Code for Tiny Computers* talk from CppCon 2016* for an extreme example.

If you're building firmware with `-Os` (i.e., optimize for size), consider adding

`-finline-small-functions`, which inlines functions whenever the compiler thinks the function body is smaller than the size of the call sites.

`-findirect-inlining`, which runs additional inlining passes. For example, if `main()` calls `a()` and `a()` calls `b()`, this compiler could fold the body of `a()` into `main()`, then notice that `b()` is also a good candidate for inlining and fold it in. Serious improvements can be gained through such second-order effects.

Current projects are built using `-O2`, which provides "nearly all supported optimizations that do not involve a space-speed tradeoff." As always, test, test, and test some more! ARM and other RISC architectures produce particularly readable disassembly—examine it to see what the compiler generates when given different options. For quick experiments, try the Godbolt compiler explorer, which colors the disassembly to show which lines of code generated it.

Good luck and godspeed!

*See `https://www.youtube.com/watch?v=zBkNBP00wJE`