C++ in Deeply-Embedded Systems
Modern Code on Tiny Chips

Dr. Michael von Tessin, Sonova

Most deeply-embedded systems are implemented in C. In this paper, we explain why, and why those systems could profit from using C++. We present how this can be achieved successfully. To this end, we also report from our own experience in converting a large, productive, deeply-embedded code base from C to C++.

Introduction
Embedded and deeply-embedded systems are often implemented in C instead of C++. Therefore, software developers forgo the C++ benefits of a strong type system, template metaprogramming (TMP) and object-oriented programming (OOP). One reason for this is the lack of support for modern C++ standard revisions by compilers used in embedded systems. Another reason for shying away from using C++ in embedded systems is insufficient expertise in how to use C++ in this realm, and the folklore about C++ having an inherent overhead, compared to C.

In this paper, we show how C++ can be used successfully in deeply-embedded systems. For the most popular C++ language features, we analyze whether or not they incur an overhead in the executable code, and if yes, how large it is. The outcome is a useful subset of C++ language features that is suitable for use in deeply-embedded systems, and also covers mixed-memory systems (ROM/RAM/NVM). Along the way, we validate our claims by referring to our own experience in converting a large, productive, deeply-embedded code base from C to C++, gradually over the course of three years.

Can you use C++?
Company policy, regulations and the like might prevent you from using C++ for your embedded project. Overcoming this hurdle is out of scope of his paper. Also, there might not be a compiler with adequate C++ support for the target system you might be locked-in to. However, at least for ARM-based systems, the situation has improved considerably. ARM recently introduced version 6 of its compiler toolchain [1], which is based on Clang/LLVM and therefore fully supports C++14. Since it follows Clang's evolvement, it will also be supporting newer C++ standard revisions.

Our experience and measurements of overhead are based on using that toolchain. Our application (hearing-device firmware developed according to IEC 62304 [2]), runs on an ASIC with the following properties: ARM Cortex-M0 CPU with 128 KiB ROM and 144 KiB RAM.

Why use C++?
Many C developers ask themselves why they should start using C++ for their embedded system. One reason is to be able to use **OOP**. In fact, many C code bases already try to mimic OOP by using structs to contain "class members", and using functions (taking a pointer to that struct as first argument) to act as "member functions". Using proper basic OOP in C++ gets you the following additional benefits:

- easier-to-use syntax
- public/private declarations for variables/functions, that are checked by the compiler
- controlled initialization/destruction of member variables via constructors/destructors

In addition, more advanced OOP concepts (involving `virtual` functions), such as inheritance and polymorphism, are hard to mimic in C in a maintainable way.

Another benefit of using C++ is its support for **compile-time optimizations**, e.g. TMP [4] and compile-time evaluation of `constexpr` functions. In its simplest application, we already benefit from the following:

- C macro "constants" can be replaced by properly typed `constexpr` variables.
- C macro "functions" can be replaced by templated functions. This not only increases type safety, but also allows the same function definition to be used not only at compile-time, but also at runtime, if needed.
- Scoped enums (`enum class`) provide additional type safety compared to regular C enums.
- Type casts (which are often unavoidable in embedded systems) can be hidden behind a controlled, small library of type-safe templated utility functions. This allows you to ban all type casts from (non-library) embedded application code.

Templates can also be used for **code size optimizations** that, if implemented in C, would make the code less maintainable. For example, consider the following function:

```
void ThrowException(EXCEPTION_T exception);
```

It is called from many locations in the code, always with a constant enum value as argument. For every call, the compiler has to generate code to load the constant into R0 before calling the function. In C++, we can now write the following templated wrapper function and call it instead:

```
template<EXCEPTION_T exception>
void ThrowException() { ThrowException(exception); }
```

At first, the executable code will now additionally contain one instantiation of this function for every exception argument that is used in the code. However, the overcall code size still decreases because in every calling location, the compiler can omit loading a constant into R0. In order to do this optimization in C, all those generated functions would need to be programmed out by hand in advance, considering all the exceptions arguments currently in use in the code base. Such an optimization is therefore only possible in C++ in a maintainable way.

How much of C++ can you use?

Our analysis of overhead in executable code is based on the following assumptions:

- ARM compiler 6 toolchain [1]
- exception handling is disabled (`--no_exceptions`)
- RTTI (run-time type information) is disabled (`--no_rtti_data`)

The following simple but very useful C++ language features do not incur any overhead in the executable code:

- namespaces
- `constexpr`
- `static_assert`
- `auto` specifier

- scoped enums (`enum class`)
- default function arguments
- overloading of functions and operators

In the OOP area, we have verified that using basic OOP concepts in C++ does not incur any overhead in code size compared to a mimicked C approach.

Advanced OOP concepts (involving `virtual` functions) incur an overhead due to the required vtables [6]: Each virtual class takes 16 bytes for a basic vtable, and each virtual member function in that class increases the size of the vtable by 4 bytes. Each instance of that class takes 4 bytes in addition for the vtable pointer. However, a similar overhead is often also required in mimicked approaches in C in order to implement the same indirection functionality.

Unfortunately, using the C++ STL (standard template library) has turned out to be prohibitive in most cases, for the following reasons:

- Using the STL often pulls in large amounts of library code, e.g. for exception handling, even though exceptions are disabled.
- Large parts of the STL (e.g. containers) require dynamic memory allocation, which is often not desired in deeply-embedded systems.

Having to avoid the STL turned out to be less painful than expected. We mostly used one of the following alternatives:

- *intrusive containers* (containers without dynamic memory allocation), which are offered by the boost library [3]
- custom-made containers from a small library that we wrote ourselves

Custom-made containers are especially useful for systems with very specific trade-offs, e.g. if the run-time overhead does not matter that much, but every byte of code counts (image size). Or if it can be accepted that e.g. a map has a not-too-large maximum capacity, which allows for a simple array-based implementation that is free of dynamic allocation and pointer linking. And in all those cases, the container can still be used like any other container (e.g. with an iterator).

C++ for mixed-memory systems

If your system has a mixed-memory architecture (e.g. ROM/RAM/NVM), C++ can be used to implement *ROM patching*: Imagine you have a class `MyLibClass` that is compiled and put in ROM. You also have a class `MyUserClass` that uses `MyLibClass` and is executed from RAM, e.g. after having been loaded from NVM when booting. After the ROM has been taped out, or even after the product based on this ROM has been released, you discover a bug in `MyLibClass`, which you want to fix (patch) for your product.

This is possible if the functions in `MyLibClass` are `virtual`. You can now derive a class `MyLibClassPatched` from `MyLibClass` and override the buggy function. `MyLibClassPatched` obviously has to be placed into RAM (because the ROM cannot be changed anymore). Now you only have to replace the use of `MyLibClass` with `MyLibClassPatched` in `MyUserClass`. This fixes the bug, without having to reimplement the entire `MyLibClass` in RAM.

How to transition a code base from C to C++?
Such a transition is best made in four steps:

(1) Tell the compiler to interpret the source code as C++ instead of C.
For some compilers, just renaming the source files from ".c" to ".cpp" is sufficient. For other compilers, a specific command-line argument is needed additionally. Since C is almost a subset of C++, this step normally requires no source code changes. However, exceptions do exist [5]! It is important that this step is done first for the entire code base. Otherwise, it might not be possible to perform the coming steps for certain modules, e.g. if they are used by other modules that are still ".c". Also, this step introduces name mangling for the generated linker symbols. Thus, in order to link ".c" and ".cpp" modules, `extern "C"` is necessary. You can avoid this by performing this step for all modules at the same time.

(2) Modify the code to use simple C++ language features.
In order to profit from better syntax and increased type-safety, the former C code can now be improved as follows:

- use namespaces instead of variable/function name prefixes
- use scoped enums instead of regular C enums
- if you used a custom-defined C bool, use the C++ bool instead
- use `static_assert` instead of run-time asserts (if possible)
- use `constexpr` instead of "constant" macros, and (if possible) instead of `const`
- use templated functions instead of "function" macros

(3) Convert obvious mimicked C "OOP" code to real OOP code.
For example, if there is a struct called `INSTANCE_T`, and a pointer to it is passed to various functions operating on it, convert it to a proper class and make those functions class members. Do not forget to specify `public/private` appropriately. If there is an `Init` function, consider converting it into the class' constructor.

In those three first steps, the source code has become more readable/maintainable and type safe. The executable code size, in our experience, does not change at all in most cases. We were even able to perform steps (1) and (2) on most modules that are in ROM, because recompiling them did not change a single byte in the executable code.

(4) Start using advanced OOP and other C++ language features.
This last step should be taken with care. For example, the developer team should be provided with guidelines of which C++ language features should be used, and in which way. For example, class member functions should not be unnecessarily defined as `virtual`, and the STL shall not be used.

While new modules would be written according to these guidelines from the start, the effort of applying this step to the entire existing code base is often prohibitively large. Therefore, it is a good strategy to perform this last step selectively per module. For example, it would make sense to tackle it when the module has to undergo a refactoring anyway. Over time, the code base will transform away from C-like to more advanced C++ code. Our code base has been in this transformation for about three years now. During that time, dozens of products have been released from this code base. Certain old and very stable modules (e.g. device drivers) are still around in C style, and will be for a long time. We never felt that this mix of styles in the same code base is a problem.

Conclusion
There is a powerful subset of C++ that can be used in the implementation of deeply-embedded systems. It enables the developers to write safer and easier to maintain code, and to implement optimizations that would not be possible in C. Unnecessary overhead in executable code can be avoided by staying in the aforementioned subset. Our proposed four-step approach enables transitioning existing, productive code bases from C to C++ over a flexible period of time.

References
[1] ARM compiler 6
https://developer.arm.com/products/software-development-tools/compilers/arm-compiler

[2] IEC 62304 medical device software standard
https://en.wikipedia.org/wiki/IEC_62304

[3] Boost C++ libraries / intrusive containers
https://www.boost.org/doc/libs/1_68_0/doc/html/intrusive.html

[4] Template metaprogramming (TMP)
https://en.wikipedia.org/wiki/Template_metaprogramming

[5] Compatibility of C and C++
https://en.wikipedia.org/wiki/Compatibility_of_C_and_C%2B%2B

[6] vtable (virtual function/method table)
https://en.wikipedia.org/wiki/Virtual_method_table

Author
Dr. Michael von Tessin is a software architect and developer, working at Sonova on deeply-embedded systems, such as hearing devices, where every byte and cycle counts. In the past, he has worked on formal verification of high-assurance multi-processor microkernels, which are used, for example, by defense and space agencies. He has completed a MSc in CS at ETH Zürich, and a PhD at UNSW Australia.

Contact
Email: michael.vontessin@sonova.com