Top 10 Reasons to Use C++ for Embedded DSP

ESC-3015

Matt Liberty

Jetperch LLC

Boston, September 2012

The C language has long been the heavyweight in embedded programming. C offers an abstraction from the physical hardware while allowing direct access to resources including memory and peripherals. Although C++ is a mature language that has been around for over 25 years, C remains the dominant embedded programming language. The adoption of C++ has been hindered by limited microprocessor support, poor compiler implementations and developer confusion over resource costs associated with C++ programming constructs. C++ compilers have continued to advance, and the C++ language is now viable for embedded programming for many targets including microcontroller applications running on bare metal. One particularly viable application is digital signal processing (DSP) which benefits greatly from the C++ language constructs. This paper examines the top 10 reasons for selecting C++ over alternative languages, especially C, for embedded DSP applications.

C++ Compared to What?

DSP algorithms can be implemented in many different languages. Some common alternatives include:

1. **Assembly** – the lowest level language that is particular to a single microprocessor or digital signal processor. Although hand-tuning can yield excellent performance, the cost is a significant increase in design time, reduced maintainability and limited portability.
2. **C 89 (ANSI C) and C 99** – the most common language choice for embedded microcontrollers and offers good portability and robustness. C lacks the higher level constructs found in most modern programming languages.
3. **C++** – an extension of C that adds object oriented constructs and templates. As a proper superset of C, C++ was designed to extend C's capabilities while only incurring resource costs for the features that are used.
4. **Embedded Coder for Matlab/Simulink** – Matlab® and Simulink® from The MathWorks® are popular languages for DSP algorithm development. Embedded Coder can translate Matlab and Simulink programs into C.
5. **Labview** – Labview® from National Instruments is another popular development language. The primary target for Labview is National Instruments equipment which limits its usefulness for embedded development.
6. **PC: Matlab / Octave / Sage** – This collection of products provide a programming language focused around matrix math. Although data exploration and algorithm development can be accelerated using these languages, they consume more resources that typically available for embedded deployments.
7. **PC: MathCAD / Maple / Mathematica** – This collection of products provide mathematical tools for developing algorithms, but they are not suitable for embedded deployment.

8. **PC: Python with scipy & numpy** – Python is a general purpose interpreted language that includes C/C++ libraries for matrix math. Although lacking the language specialization for matrix math, Python has powerful matrix and DSP support while also providing a full general-purpose programming language. C and C++ libraries can be easily accessed from Python.

Embedded software means different things to different people. Embedded can mean 128 bytes of RAM on an 8051 processor or a quad-core ARM® system on a chip. Generally, embedded applications are characterized by some real-time component and limited resources. For portable embedded devices, power utilization is critical. Small embedded systems often run on the bare metal without an operating system and may have no dynamic memory allocation. Embedded engineers must balance these demanding resource constraints against quality and time to market. Tool selection can be critical in enabling embedded projects to meet the objectives on time and under budget. C++ is one potential tool that should be considered for embedded projects.

The Top 10

The following list captures 10 top reasons for using C++ for embedded DSP applications. This list is certainly not exhaustive, and item priority can easily vary depending upon the application. The list provides a starting point for considering C++ in your next project.

10. Inlined functions

Inlined functions are a language construct that provides functionality similar to C macros except with the full type checking present in functions. When inlined, the function's body is place directly in the caller's body so that no function call overhead occurs. Typical function call overhead includes saving the register state on the stack, branching to the function, returning from the function, and restoring state. For small functions, this overhead can exceed the size of the function body. Inlined functions provide a mechanism for the developer to provide a compiler hint. This functionality is also present in C99 compilers, but many developers often still use macros in C.

C	C++

`#define ADD_TWO(a) ((a) + 2)`	`inline int addTwo(int a) {`
	`    return a + 2;`
	`}`

9. Memory management

Like C, C++ uses the stack for local storage and the heap for persistent storage. While C provides malloc and free, C++ provides new and delete. Beyond a simple name change, the C++ memory allocation provides finer control over memory management compared to C. C++ allows user-defined global new / delete handlers similar to custom malloc / free implementations in C, but it also allows for per-class customizations. One useful DSP application is creating a fixed memory pool for class instances. This pool can be placed on the top-level stack or in reserved memory segment without requiring dynamic memory allocation.

C++ also has a special syntax for placement new, which is similar to casting an address pointer in C except that initialization happens automatically. Placement new is especially useful for device drivers that can automatically initialize and finalize the device. A good introduction for placement new and the C++ new operator can be found in [Ben11].

8. Scalability

For DSP application code, one significant aspect of scalability is the ability to gracefully perform across a wide variety of platforms. When run on less resourced constrained systems, the DSP application could yield potentially higher quality output, more flexibility and dynamic configuration. Even for DSP applications targeting a single, highly resource constrained platform, developers will often want debug functionality not available on the embedded target. From a single code base, a C++ application can implement customized new/delete operators with the factory design pattern to produce the most applicable instances depending upon the target platform. One example is static memory allocation as discussed in item 9. Other scalability options include floating pointer versus fixed point, test point logging and filter selection.

7. Templates

C++ templates provide a full Turing-complete programming language often called template metaprogramming (TMP). Although TMP can be an extremely useful tool, careful use is required for highly resource constrained embedded systems. Many embedded compilers have limited template support making extensive use of template features challenging. However, DSP has a very useful template application that is generally well-supported by compliers. In DSP, we often wish to implement the same function across multiple data types including float, double, integer and fixed point. In C, we either implement individual functions for each type or use type-unsafe macros to generate the functions for each type. C++ templates offer much better compile-time checking and type support. More importantly, the allow designers to practice “Don’t repeat yourself” (DRY) [Hun00].

C	C++

`#define SQUARE(T) \`
`    T square(T a) {\`
`        return a * a; \`
`    }`
`SQUARE(float)`
`SQUARE(double)`
`SQUARE(int)`
	`template <typename T>`
	`T square(T a) {`
	`    return a * a;`
	`}`

In practice, the C example above will usually need to be split into two parts, the .h header which contains the function declaration and the .c file which contains the function implementation. This split further compounds the maintenance problem for the C code.

6. Better offline code

When compared with C, C++ offers a simplified syntax and more powerful features that enable more rapid development and testing. DSP designers often create a wealth of offline analysis tools during the development of the algorithm. In many projects, this investment does not transfer between the theoretical algorithm development and the embedded implementation. Finding meaningful ways to bridge this divide can reduce the testing gap in the embedded implementation. Most DSP algorithm developers do not work in C or C++ as their first language and instead opt for Matlab or Python. Tying C or C++ algorithms back into Matlab through mex files is a tedious process. However, Boost.Python provides almost automatic wrapping for C++ objects into Python objects. Creating algorithm level tests in Python is often much faster than test creation in C. These tests can then be extended to run on the native PC, a processor simulator or hardware in the loop. GNU Radio and OpenCV are two examples of large C/C++ libraries that contain Python bindings to accelerate configuration and testing.

Even if integration with Matlab or Python is not desired, modern C++ code can greatly reduce the tedium of offline code development. Modern C++ often contains no new/delete even though it uses the heap. Classes instead use smart pointers which implement reference counting, a simplified form of automatic garbage collection. By eliminating memory management tedium where it is not required in the offline code, a significant percentage of C / C++ hassles and development time disappear. The Standard Template Library (STL), while a concern for many highly constrained embedded deployments, can greatly accelerate offline code development. Newer features including “auto” variable types and concise loop syntax further improve productivity.

5. Structure initialization

In C, structure declaration does not perform any automatic initialization. Programmers are left with two choices: create an initialization function or require every code point to correctly initialize every field. In the latter case, adding new fields or modifying default values is challenging. In both cases, the code using the structure has an additional step that can be missed. Even if the original developers are tuned to these steps, the code is often turned over to other engineers for bug fixes and future feature improvements. During the lifecycle of a project, these steps may be missed introducing subsequent bugs.

C++ provides a concise syntax for implementing automatic structure initialization, constructors. Every structure is also a class in C++. The only different between a struct and a class is the access permission. Structures default to public while classes default to private. By defining a constructor, all code that declares the structure will include the automatic initializer.

C

```
struct my_struct {
    int32_t a;
    int32_t b;
};
struct my_struct *
my_init(struct my_struct * s) {
    s->a = 1;
    s->b = 2;
    return s;
}
my_struct s;
my_init(&s);
```

C++

```
struct my_struct2 {
    my_struct2() : a(1), b(2) {}
    int32_t a;
    int32_t b;
};
my_struct2 s2;
```

4. Operator overloading

C++ includes a feature called operator overloading which allows user-defined code to assign type-dependent functionality to the built-in operators including '+', '-', '*' and '/'. Much early C++ code misused this feature for operations that did not match the normal mathematical definitions, and many computer scientists shied away from operator overloading. However, DSP is a perfect arena where operator overloading makes both mathematical and common sense. Two common cases are matrix math and saturation arithmetic. Both cases use mathematical operators in their specific domain, but the default C implementations do not support either. With C++, this default restriction is removed. One major advantage is that the code becomes portable between types. The application can contain the operator '+' which works when the type is both double and a saturated int32_t. The application code then has the ability to easily switch between optimized and non-optimized implementations using compiler flags.

C

```
inline int32_t sadd32(int32_t a, int32_t b) {
    // Add intrinsics or inline asm here
    int32_t c = a + b;
    if (a > 0 && b > 0 && c < 0) {
        return INT32_MAX;
    } else if (a < 0 && b < 0 && c > 0) {
        return INT32_MIN;
    }
    return c;
}

int32_t c = sadd32(a, b);
```

C++

```
struct sint32_t {
    sint32_t(int32_t value) : value_(value) {}
    inline sint32_t& operator=(int32_t value) {
        this->value_ = value;
            return *this;
     }
    int32_t value_;
};
inline int32_t operator + (sint32_t a, sint32_t
b) {
    return sadd32(a.value_, b.value_);
}

sint32_t c = a + b;
```

3. Fixed point math

Migrating from floating point to fixed point (or integer) is a common DSP optimization. Floating point computations have inherently more computational complexity. Even processors that include a floating point unit consume more power when performing floating point arithmetic when compared to integer arithmetic. Even with the cost reduction of floating point math with recent processors, power constraints in portable devices may dictate the conversion to fixed point. This optimization is often tedious and error-prone. Although higher level tools do exist, they can be complicated to configure or produce unmaintainable code. C++ allows the option to automatically manage the decimal place, add type-aware logging, and greatly simplify the conversion from fixed point to floating point and back again while keeping maintainable code throughout the process. Numerous potential starting-points exist for embedded projects including SystemC. A good starting point is to search the web for "c++ fixed point". A typical implementation looks something like:

```
// Q: The decimal point is just after bit Q (0 is the same as
RepresentationT)
// RepresentationT: The internal representation, such as int32_t
// MacT: The multiply/accumulation type, often twice the width of
//              RepresentationT to maintain precision.
template <int32_t Q, typename RepresentationT, typename MacT>
struct FixedPointT {
    static const int_fast8_t Q_ = Q;
    FixedPointT(int32_t v) : rep_(v << Q_) {}
    RepresentationT rep_;
};
```

And can be used like:

```
FixedPointT<2, int32_t, int64_t> a = 42;
FixedPointT<4, int32_t, int64_t> b = 12;
FixedPointT<0, int32_t, int64_t> c = a * b;
```

Since the FixedPointT uses operator overloading, the conversion from fixed point back to floating point becomes trivial. The FixedPointT class simply needs a compile option that implements a trivial template that typedefs FixedPointT to double. This allows simplified regression testing of optimized code to isolate structural errors from fixed point mathematical errors.

2. Object oriented design

One of the primary purposes of C++ was to extend C to include objects. The original name of C++ was C with Classes [Str01]! Many DSP constructs including filters, signal sources and signal sinks map naturally to classes as they contain an interface coupled with hidden, persistent data. When filters are implemented as objects, one filter can be easily interchanged with another by changing only a line or two of code. Although C can support object oriented programming, the syntax is much clearer and easier to maintain in C++.

C	C++

`struct myObj {`
`    int32_t a;`
`};`
`struct myObj *`
`myObj_init(struct myObj * self) {`
`    self->a = 0;`
`    return self;`
`}`
`int32_t myObj_f1(struct myObj * self,`
`int32_t b) {`
`    return self->a + b;`
`}`
`myObj obj1;`
`myObj_init(&obj1);`
`int32_t c = myObj_f1(&obj1, 4);`
	`class MyObjCpp {`
	`public:`
	`    MyObjCpp() : a(0) {}`
	`    int32_t f1(int32_t b) {`
	`        return this->a + b;`
	`    }`
	`    int32_t a;`
	`};`
	`MyObjCpp x;`
	`int32_t y = x.f1(4);`

1. Higher level of abstraction

The most compelling reason to select a tool is that it allows the engineers to get the job done correctly, enjoyably, on time and under budget. Programming languages continue to evolve, and the right tool depends highly upon the project constraints. For embedded systems with limited resources, interpreted languages are often not an option. For many products, operating systems command too much of a price premium and developers write for bare metal or very simple tasking libraries. For these targets, engineers are required to exert tight control over memory and code size. This domain has long been commanded by C. With careful use, C++ allows the same tight control while eliminating repetitive boilerplate code and simplifying the resulting application language. These abstractions produce code that is shorter, easier to read, easier to review and easier to maintain. This assertion is notoriously hard to quantify [Ken07]. Since C++ is a strict superset of C, teams can restrict their C++ use to the parts of C++ that offer provable benefit.

C++ Concerns

C++ was designed to be as efficient as C, and like C, the design philosophy of C++ is to only pay for what you use. However, neither C nor C++ attempts to protect the program from programmer error. Both rely on the programmer being intelligent and do not question programmer intent. For embedded programming in C++, this combination is dangerous for novice programmers who do not approach the language carefully. With greater expressive power and larger standard libraries, programmers can more easily create a program that exceeds their design resources. If printf exists in C, why is it not a valid option for bare metal debug? The same argument goes for C++, except many more "standard" features are available. "C makes it easy to shoot yourself in the foot; C++ makes it harder, but when you do, it blows away your whole leg." [Str01] (the creator of C++).

Embedded C++ (EC++) was originally created with the intent of creating a limited, safe subset of embedded C++. Unfortunately, a number of the "safe" improvements including templates, namespaces and casts were really to enable simpler compliers rather than safer code. However, embedded programmers should be particularly aware of the tradeoffs for using:

- **Run-Time Type Information (RTTI)**: Dynamically inspecting run-time properties of classes including the operations typeid and dynamic_cast.
- **Multiple Inheritance**: Allowing a class to have more than one base class.
- **Exceptions**: Allowing an alternate "return path" through every function.
- **Standard Template Library (STL)**: Additional capabilities including collections that can create significant performance issues (new/delete) when used carelessly in embedded systems.

Object oriented programming can also cause significant code bloat when used incorrectly. Inheritance should be used sparingly to mode "is-a" relationships rather than "has-a" relationships. The "has-a" relationships are often best implemented through delegation to an instance that is a member. Multiple inheritance should be used even more sparingly. Virtual methods are another cost that need only be paid when needed. One caveat is that virtual methods cannot be inlined in any statically compiled language, including C++.

As with any language, functionality comes at a cost. Embedded developers coming from a C background who are particularly concerned about resources should profile the features that they need and build up a list of "approved" language features. Unlike many virtual machine based or interpreted languages, C++ allows developers to only pay for the features needed.

Conclusion

Choosing the right tool is a challenging task. If C gives you enough rope to hang yourself, then C++ gives enough rope for you to hang yourself and your project team. As with most tools, some level of expertise is required to wield C++ correctly. The additional expressiveness of C++ allows for tighter, cleaner and more easily maintained application code compared to C. In some ways, C++ offers even tighter control of memory than C. C++ is not the right language for every application, but as this list indicates, it has some features particularly suited for embedded DSP applications.

As with any resource-constrained development, profiling the target application on the target platform provides the best cost metrics. Developers integrating features for a resource-constrained platform must be aware of the tradeoffs when using any language, and C++ is no different. C++ fortunately allows C developers to explore the language carefully and only add language features as needed. The cost/benefit of C++ is different for every application. Profiling a few items in this list can be a quick exercise. From there, developers can weigh the features best suited for their particular application.

References

[Ben11] The many faces of operator new in C++, Eli Bendersky, 17 Feb 2011, Retrieved 23 Feb 2012.

[Hun00] The Pragmatic Programmer, Andrew Hunt & David Thomas. Addison-Wesley, 2000.

[Str01] Bjarne Stroustrup's FAQ, Bjarne Stroustrup, Retrieved 24 Feb 2012.

[Ken07] Defining and Measuring the Productivity of Programming Languages, Ken Kennedy, Charles Koelbel, 2007, Retrieved 24 Feb 2012.