WHY C++ ?

INTRODUCTION

For the past ten or fifteen years, the choice of programming languages for embedded systems has been between assembler and a small variety of higher-level languages. For many (but obviously not all) programmers, the higher-level language of choice has been C. In recent years, programmers have been considering, and actually even using, C++ for programming embedded systems.

Just as C was designed to appeal to assembler programmers, C++ was designed to appeal to C programmers. Along the way, C++ seems to have attracted programmers who have been using other languages. However, since most of the interest in C++ seems to be coming from C programmers, most of this presentation focuses on contrasting C++ with C.

I won't presume to tell you that C++ is right for you or your application. It may not be. C may be just fine for your needs, and there are other high-level languages, notably Ada, that merit consideration for some projects. My objective is simply to provide you with background information that will help you decide for yourself if C++ is worth a try.

WHY USE C FOR EMBEDDED PROGRAMMING?

When evaluating a new language to replace the one you're already using, you should remember those things that your current language does well. If the new language can't handle some task that the old one could, you should be sure that the task is no longer important to you. In this particular case, it's worth noting what C does well so that we can determine if C++ does those same things just as well.

What makes C suitable for embedded programming?

- C is a general-purpose programming language, so it can meet a broad range of needs.
- C is a "low-level" high-level language. (Some people think of it as a "high-level" assembler.) C provides machine-level data objects, such as bits, characters, and integers with different sizes. C also provides many machine-level operators such as increment, decrement, shift, bitwise-and, and bitwise-or, many of which map directly into individual machine instructions.
- C programs can be highly portable across platforms. Even if an entire C program is not portable, significant portions of it can be.

In the *Rationale for the C Standard* (Hudson [1989]), the ANSI C standards committee characterized the "spirit" of C as:

- Trust the programmer.
- Don't prevent the programmer from doing what needs to be done.
- Keep the language small and simple.
- Make it fast, even if it's not guaranteed to be portable.

C was not designed to teach good habits; it assumes that you, the programmer, already have them. It provides modest type checking to catch common errors, but you can bypass the checks when necessary, typically by employing casts.

As is the case with any language, C attempts to balance safety and portability against speed and space efficiency. C tends to favor efficiency. C sometimes defines operations as "whatever the target machine's hardware does" rather than by some architecture-independent rule. For example, when shifting a signed integer with negative value to the right, a C program may propagate the sign bit, or simply shift in zeros, depending on what the underlying hardware wants to do.

WHY LOOK TO C++?

Processors keep getting faster and memory keeps getting cheaper. Data paths are wider than they were years ago. Consequently, embedded applications have been getting larger and more ambitious. Larger projects require larger teams and more powerful tools. Thus priorities have shifted. For an increasing number of embedded programming projects, project control and maintainability are becoming more important than efficiency.

This shift in priorities makes object-oriented languages and techniques more appealing. Data abstraction (the principal object-oriented technique) goes a long way toward improving the quality of designs and programs. For C programmers in particular, C++ has the following advantages:

- C++ is a "better" C. Since it is essentially a superset of C, C programmers can start using C++ as a "better" C with little, if any, loss of productivity. C++ is "better" in the sense that it applies stricter translation-time checking than C does, so C++ compilers can catch more errors early in the development process.
- C++ supports data abstraction. C++ programs can use classes and objects to partition systems into simpler components, yielding designs that are easier to maintain. Classes and objects can enforce the semantics of component interfaces better than function libraries.
- C++ supports object-oriented programming. Class hierarchies can model system components (such as devices, files, and tasks) very naturally. Virtual functions can capture subtle variations in behavior among closely related components without complicating component interfaces.

- C++ partitions the name space better than C, reducing the chance of global name conflicts.

Contrary to much of the early hype, you need not learn a completely new way of programming to benefit from C++. You can use it as a procedural language and grow into data abstraction and object-oriented programming at your own pace.

C++ captures essentially the same “spirit” as C. C++ is also a general-purpose language with the same low-level programming capabilities as C. In some ways, C++ places even more trust in the programmer than C does. Bjarne Stroustrup, the inventor of C++, characterized the “spirit” of C++ (Stroustrup [1989]) as:

- C++ is an engineering compromise. It exists to solve real problems, and was driven more by constraints than by principles.
- What you don’t use you don’t pay for.
- Don’t leave room for a lower level language. If C++ can’t do the whole job, people will start writing some parts of the program in a lower-level language.
- Aim at radical portability – of the language and of the libraries.
- Serve users first and compiler writers second. Rely on compiler technology, minimize run-time support, and distrust elaborate mechanisms.
- Don’t break C compatibility, unless you really have to.

In short, C++ was designed so that C programmers could exploit more powerful programming methodologies without sacrificing that which is good about C.

Of course, C++ has its drawbacks:

- C++ is a far more complex language than C. Compiler diagnostics can be very cryptic. Run-time bugs can be very subtle. C++ places higher demands on tools such as linkers and debuggers.
- Hidden run-time costs can creep into C++ programs. Virtual functions increase object sizes and slow down function calls. Constructors and destructors can add overhead to object creation and destruction and complicate exception handling. Temporary objects can appear at surprising times with surprising costs in time and space. Classes can have hidden fields, making them harder to map into existing data structures.

Fortunately, you can overcome most of the drawbacks with training and experience.

ADDRESSING THE CONCERNS

C++ adds quite a number of syntactic features to C. Many of the features appear to have additional speed or space penalties relative to C. The fact is that some do and some don’t. Part of learning to use C++ for embedded applications is learning to gauge the costs of the features you use. Let’s look at a few C++ features to gain a sense of how to do this.

The principal feature of C++ is classes. A class defines a data type with associated operations on that type. The operations are expressed as member and friend functions. For example, a class definition such as:

```
class port
    {
public:
    int open(port_regs *p);
    int close();
    int get();
    int put(int c);
private:
    port_regs *regs;
    int errno;
    };
```

defines a type **`port`** with member functions **`open`**, **`close`**, **`get`**, and **`put`**.

A common concern for embedded programmers is that member functions appear to occupy space in each object, and therefore increase the application's memory requirements. In fact...

- Ordinary member functions *do not* increase object sizes.

For example, a **`port`** object occupies the same storage as if the **`port`** type had been defined as just:

```
struct port
    {
    port_regs *regs;
    int errno;
    };
```

In a sense, the difference between non-member functions and ordinary member functions is merely notational. Classes and member functions provide an explicit notation for data abstraction:

- A class encapsulates data with the functions that operate on that data.
- A class uses *access specifiers* (the keywords **`public`** and **`private`**) to grant or deny access to class members from outside the class.

This notation depends only on run-time facilities already available in C.

Well, you ask, if it doesn't provide any additional run-time capabilities, then why bother? The class notation provides the opportunity for:

- better compile-time enforcement of the semantics of component interfaces, and

- fewer global names, thereby reducing the possibility of global name conflicts that occur when two or more components use the same global identifier for different purposes.

Notice that it is *ordinary* member functions that do not increase object sizes. What about member functions that aren't ordinary? Well, the reality is that...

- *Virtual* member functions *do* increase object sizes.

For example, if you change some of the member functions of **`port`** into virtual functions, as in:

```
class port
    {
public:
    int open(port_regs *p);
    virtual int close();
    virtual int get();
    virtual int put(int c);
private:
    port_regs *regs;
    int errno;
    };
```

then you increase the size of each **`port`** object by the size of one data pointer. A **`port`** would then occupy the same storage as if the **`port`** type were:

```
struct port
    {
    port_regs *regs;
    int errno;
    void *vptr;
    };
```

OK, so some C++ features have extra run-time costs and others do not. How can you tell which is which? As a general rule, the spirit of C++ is "What you don't use you don't pay for." In other words,

- C++ does not make you pay for a feature unless you use it.

C++ adheres to this rule fairly well, but not perfectly. For example, you must request the virtual call mechanism for a function by using the keyword **`virtual`** in the member function declaration. You get more functionality than with ordinary member functions, but you also pay a little for it. On the other hand,

- C++ makes you ask for certain optimizations, such as inline function calls.

For example, if a function has an extremely short body, the execution cost (run time and/or code space) of actually jumping to and from the function body may be greater than the cost of the function body itself. In that case, it makes sense to expand the function body in line at the point of each call.

In C, you write an inline function as a macro. In C++, you write it as a function but you declare it **`inline`**, as in:

```
inline int port::put(int c)
    {
    // something very short
    }
```

Inserting the keyword **`inline`**, though not automatic, is generally less work than converting a function to a macro.

The "pay for only what you use" philosophy clearly applies to the C part of C++. With most development systems that support both C and C++, rebuilding a C program as a C++ program produces essentially the same object program. Here are the caveats:

- You will probably need to massage your C source a little to get it to compile as C++.
- C++ compilers may add a little extra code for program initiation and termination, but the speed and space penalty is usually marginal.
- With most implementations, the C++ features supporting exception handling and run-time type information may spawn additional static data. Exception handling may introduce extra overhead on each function call and return. However, C++ implementations usually offer compile and link options to fine tune these features or disable them completely.

Another related concern regarding member functions is that they execute more slowly than non-member function calls. The reality is that...

- Ordinary member function calls are just as fast as non-member function calls.

A member function call such as:

```
p.put('x');
```

applies a **`put`** operation to object **`p`**. **`p`** is called the *receiver object* of the call, and the compiler generates code which passes a pointer to **`p`** as an additional argument to **`put`**. For example, C++ translates the member function declaration:

```
int port::put(char c);
```

into the equivalent of a C function that looks something like:

```
int port_put(port *const this, char c);
```

and translates the call:

```
p.put('x');
```

into something like:

```
port_put(&p, 'x');
```

Is passing the address of the receiver object an extra cost? Not really. The vast majority of C functions that operate on an object of struct type pass the address of that object as an explicit argument. When you redesign this code using classes, that struct object becomes the receiver object in a member function call. The number of actual arguments remains the same.

What about the efficiency of virtual function calls? Are they slower than ordinary function calls? Yes.

- On most architectures, a virtual function call executes 2 to 4 more instructions than a non-virtual function call.

Again, you only pay for this if you ask for it. Moreover, there is a notation by which you can turn off the virtual call mechanism for an individual call to a virtual function (thus saving the added cost) without turning off the virtual call mechanism for any other calls.

OTHER LOW-COST C++ FEATURES

C++ offers C programmers several other attractive features that incur little or no run-time cost. Some of these features can be very useful in improving the clarity of programs. Here’s a sampling.

Reference types offer an alternative to pointers as a way of referring to objects indirectly. For example, if you write (in either C or C++):

```
port p;
...
port *pp = &p;    /* pp is a pointer to a port */
```

then **pp** is the address of **p**, and ***pp** refers to **p** itself. In C++, you can also write:

```
port &pr = p;    /* pr is a reference to a port */
```

so that **&pr** is the address of **p**, and **pr** refers to **p** itself.

Believe it or not, references have many notational advantages. For example, passing a large object by value, as in

```
void f(large_object lo);
```

can be expensive. Each call such as **f(x)** must copy **x** into **lo** in its entirety. In C, you can reduce the cost of each call by declaring **f** to pass its parameter by address, as in:

```
void f(large_object const *lo);
```

But then you must write all the calls as **f(&x)**, which is cheaper than passing by value, but less readable.

In C++, you can declare **f** to pass its parameter by reference:

```
void f(large_object const &lo);
```

and then you can still write all the calls as **f(x)**, which looks like passing by value, but has the efficiency of passing by address.

Not everyone likes references. Some beginners find the notation deceptive and confusing. Most learn to appreciate it. You must decide for yourself.

Another feature that can add readability to C++ programs with no run-time cost is *function name overloading*. In C, every function must have a unique name. Thus, C libraries are filled with names of closely related functions with slightly different spellings, such as **sqrt**, **fsqrt**, and **ldsqrt**. In C++, a program can declare different functions with the same name as long as each function has a sufficiently distinct parameter list:

```
      float sqrt(float);
     double sqrt(double);
long double sqrt(long double);
```

For a given call to an overloaded function, the compiler selects (at compile time) the function whose formal parameters are the best match for the actual arguments in the call. For instance:

```
d = sqrt(10.0);          /* calls sqrt(double) */
f = sqrt(10.0f);         /* calls sqrt(float) */
```

Compilers can distinguish these calls because **`10.0`** has type **`double`** while **`10.0f`** has type **`float`**.

A compiler will reject a call that does not have a unique best match:

```
x = sqrt(10);            /* error: ambiguous */
```

This is ambiguous because **`10`** has type **`int`**, which converts equally well to either **`float`** or **`double`**.

Function name overloading, used judiciously, can help you implement libraries and program components with simpler, more intuitive, interfaces.

An important side-effect of function name overloading is *type-safe linkage*, which guarantees (at link time) that a function declared in one translation unit and defined in another has the same parameter types in both places. For example, C++ can catch this error:

```
/* file 1 */                    /* file 2 */
void f(int)                     void f(long int);
    {                           ...
    ...                             {
    }                               f(0L);    /* link error */
                                    }
```

C cannot. Thus, C++ eliminates yet another source of bugs.

MIGRATING FROM C TO C++

Converting C source to C++ requires only a few, largely mechanical, changes to the C source. Even if you are not converting source code, you must make small changes in your coding style to use C++ instead of C. For example:

- C++ has more reserved words, such as **`class`**, **`public`**, **`private`**, **`inline`**, **`virtual`**, **`new`**, and **`delete`**. You must avoid using these words as identifiers.

- C++ is less lenient about type conversions. You may have to insert a cast here or there. Eventually, you may want to rethink the way you use types to reduce the number of casts.

For the most part, using C++ to compile C imposes no real restrictions on the expressiveness of C. Rather, it forces you to abandon poor coding practices, most of which are already considered obsolete by the C standard. Translating C directly into C++ typically yields code that's no less efficient than C, but is a little safer (from more rigorous static checking).

THE DOWN SIDE

So what are some of the genuine concerns regarding the use of C++ for embedded programming? Here are some examples...

- The effects of constructors and destructors can add surprising run-time overhead to programs.

A *constructor* for a class **`X`** is a special member function that automatically initializes **`X`** objects at run time. A *destructor* for class **`X`** is a special member function that releases resources used by an **`X`** object that's about to be destroyed. For the most part, constructors and destructors are good things. They make programs more reliable. They do work the program should do anyway, but you may be surprised at *when* it gets done.

Here's an example of a possible surprise. If **`string`** is a class type with a constructor and destructor, then:

```
for (i = 0; i < N; ++i)
    {
    string s;
    // ... use s ...
    }
```

creates and destroys **`s`** on each iteration, invoking both a constructor and destructor each time around. In this case, avoiding the overhead is easy – move the declaration for **`s`** outside the loop. Not all such problems have such simple solutions.

- C++ programs create and destroy temporary objects in places you might not expect, again producing surprising run-time overhead.

For example, the standard **`string`** class defines the **`+`** operator as concatenation so that, for strings **`s`**, **`t`**, and **`u`**,

```
u = s + t;
```

concatenates **`s`** and **`t`** and assigns the result to **`u`**. Most compilers produce a program that creates a temporary **`string`** object (using a constructor) to hold the result of **`s + t`**. The program destroys the temporary after copying it to **`u`**. In this example, you can avoid creating the temporary (and the constructor and destructor calls) by rewriting the code as:

```
u = s;
u += t;
```

Again, correcting the problem is not always that simple.

- C++ compiler diagnostics can be *very* cryptic.

The syntax of C++ is much more complicated than that of C. C++ compilers have more difficulty pinpointing errors. Templates compound the problem.

- The Standard C++ Library has a very large footprint.

Even the smallest C++ programs may drag in tens of thousands of bytes of code from the standard library (see Plauger [1996]). Thus, C++ executable images can be surprisingly large. You may need to avoid certain library facilities, or "roll your own" simplified versions of library components. See Green [1996] for an example of a simplified i/o library.

DEVELOPMENT TOOLS

Tools for embedded programming with C++ are essentially the same as the tools you need for C, with a few additional requirements. Here are some things you should look for.

- Diagnostic messages and debugger symbols should appear as they do in the source.

Many C++ implementations use a form of "name mangling" to preserve compile-time type information for use at link time. The linker uses the "mangled" names to enforce type-safe linkage. For example, the **`sqrt`** functions:

```
float sqrt(float d);
double sqrt(double d);
complex sqrt(complex c);
```

might be mangled as:

```
sqrt__Ff
sqrt__Fd
sqrt__F7complex
```

In early C++ implementations, these "mangled" names appeared in diagnostic messages and link maps. You should expect more recent C++ development tools to display the function declarations as they appear in the source.

- Debuggers should be able to set breakpoints in inline function bodies and template instances.

What's the debugger to do when you ask it to set a breakpoint in an inline function body? When the compiler expands a function body inline, there's no one copy of the function body where the debugger can set breakpoints. The copies are scattered all over the executable code. Templates may also generate different instances of the executable code from a single copy of the source code. This poses similar problems for debuggers. Many C++ compilers disable inline expansions when you enable symbolic debugging.

- Class browsers can be helpful tools for navigating through class inheritance hierarchies.

When dealing with class hierarchies, sometimes you can't tell which member came from which source file or header file. A good graphical browser can quickly locate a program component no matter which file or header it actually resides in.

STRATEGIC RECOMMENDATIONS

The proper mind set is to think of object-oriented approaches as evolutionary, not revolutionary. Think of C++ the same way.

My experience is that, if projects run into trouble in using C++, it is because they underestimate the time it takes to transition to C++. Technical evangelists can create undue pressures by overstating the benefits of object-oriented techniques and C++.

A safe, sensible strategy is to:

- Adopt C++ incrementally.

That is, use only a small subset of C++ for your first project, use a larger subset on your next project, and so on. Plum and Saks [1991] and Plauger [1993] recommend the following incremental subsets of C++:

1. *Typesafe C*: the common subset of Standard C and C++.
2. *Enhanced C*: C++ features that improve the safety and convenience of procedural programming.
3. *Object-Based C++*: C++ with classes and templates but not derivation.
4. *Object-Oriented C++*: the entire C++ language.

Progressing through levels such as these reduces technical risks by innovating in smaller steps. It also eases tension within programming teams caused by disparities in ability. Let's look at these levels in greater detail.

Typesafe C is essentially Standard C with a few restrictions (some of which were mentioned earlier). The primary restrictions are that you must avoid C++ keywords, use casts a bit more often, and avoid old-style (non-prototype) function headings.

Enhanced C is Typesafe C plus features such as:

- inline functions (in place of macros)
- reference types (for passing arguments)
- function name overloading (for more intuitive user interfaces)
- operator overloading (for notational convenience)
- operators **`new`** and **`delete`** (for better storage management)

You should use Enhanced C only if you can completely abandon Standard C. The resulting code should be more readable, but no less efficient, than it would be in C.

Object-Based C++ is Enhanced C plus:

- classes and access control (for data abstraction and encapsulation)
- friend functions
- templates (for generic classes and functions)
- exception handling (for orderly error recovery)
- namespaces (for managing the global namespace)

It excludes derived classes and virtual functions. This dialect:

- partitions the name space very effectively
- enforces the semantics of devices, APIs, etc.
- can still be as efficient as Standard C
- avoids the worst complexities of object-oriented programming

Object-Based C++ dialect offers the largest payoff for embedded systems that have serious performance requirements.

EMBEDDED C++

In 1996, a consortium of Japanese semiconductor manufacturers agreed that C++ had gotten too complicated and expensive for programming embedded systems. They specified a subset of C++ for embedded systems that they called *EC++* (Embedded C++).

EC++ omits the most complicated and expensive parts of C++. The EC++ specification is a list of changes to (mostly deletions from) the C++ standard. That specification is available at **`http://www.caravan.net/ec2plus`**. Specifically, EC++ omits from C++:

- exception handling
- multiple inheritance
- run-time type information (RTTI)

because they incur speed and space penalties that are often unacceptable in embedded systems. It omits:

- namespaces and using-declarations
- templates

because they were the least stable parts of the draft C++ standard, and remain so with current compilers. Templates can also be responsible for producing surprising large code. EC++ also omits:

- mutable class members
- new-style casts

purportedly because their utility does not justify the complexity they add to C++. The exact combination of features in EC++ is still the subject of discussion.

The EC++ Library is a subset of the full C++ Library. The EC++ Library components are smaller and less interconnected than their C++ Library counterparts. Therefore, EC++ programs tend to have a smaller footprint than comparable C++ programs. That is, a C++ program that uses only EC++ language and library features should yield a smaller executable image when compiled and linked as EC++.

Several C++ compiler and library vendors are already supporting EC++. It may prove to be a viable alternative for many embedded applications.

SUMMARY

C++ is a viable language for many, but not all, embedded programming applications. Smaller systems with demanding resource requirements may never see a benefit, and could possibly suffer, from using C++ instead of C. However, most large systems should see at least some benefit. Some will experience significant improvements in software quality. Some may eventually see a reduction in development costs.

There’s no doubt that C++ is a large language and takes time to master. The best way to find out if it’s right for you is to try it out in small doses.

ACKNOWLEDGMENT

Thanks to P.J. Plauger for sharing his notes on this material with me.

REFERENCES

- Green [1996]. Curtis Green, “Shrinking Iostreams for Embedded Developers”, *Embedded Systems Programming*, June 1996. Miller Freeman.
- Hudson [1988]. R. Hudson, ed., *Rationale for Draft Proposed American National Standard for Information Processing Systems, Programming Language C*. American National Standards Institute.
- Plauger [1993]. P.J. Plauger, “Embedded Programming in C++”, *Proceedings of the Embedded Systems Conference East*, April 1993. Miller Freeman.
- Plauger [1996]. P.J. Plauger, “Too Much of a Good Thing”, *Embedded Systems Programming*, July 1996. Miller Freeman.
- Plum and Saks [1991]. Thomas Plum and Dan Saks, *C++ Programming Guidelines*. Plum Hall.
- Stroustrup [1989]. Bjarne Stroustrup, *Keynote Address: Organizational Meeting of ANSI X3J16*. December 1989.