Using C++ Efficiently
In
Embedded Applications

César A Quiroz
Mentor Graphics, Microtec Division

Abstract. Moving to C++ presents opportunities for higher programmer productivity. The requirements of embedded systems, however, demand that the adoption of C++ be carefully measured for the performance impact of run-time costs present in C++, but not in C. This talk suggests strategies for developers who are starting their acquaintance with C++.

This talk is aimed at embedded systems developers who are considering the adoption of C++[BS97] in their projects. The material assumes professional acquaintance with embedded development, and with a language in the general class of ANSI/ISO C[C90]. On the other hand, not much knowledge of C++ is presumed. This talk is meant, therefore, for engineers tackling their first embedded system development projects in C++.

There are good reasons for using C++ to implement an embedded system, for instance:

- C++ compares favorably with C (the leading alternative) in matters of syntactic convenience, compile-time type-checking, memory allocation and initialization, and code reuse through derivation,
- The newly approved Draft Standard for C++ ends a long period of uncertainty about the ultimate shape of the language. Conforming implementations are gradually turning up. Within a couple of years, we should find that most implementations are fully conforming.

On the other hand, there are corresponding reasons for caution, too:

- C++ (plus its libraries) is a larger, more complex, language than C (its libraries included). A developer who has a good understanding of the consequences of employing this or that C feature may be misled when evaluating outwardly similar features in C++,
- The long gestation period of the C++ standard has promoted a drift in implementation. The meaning of a C++ program (never mind its performance) does depend on what compiler it was meant for.

I assume here that the decision of adopting C++ has been made already, and that the development team is evaluating *which* C++ features will be appropriate for the project at hand.

Feature Cost Evaluation

If we could give for sure advice of the form "Embedded developers should never use feature XYZ" then this talk would not be needed. Such generalities are normally obvious (obviously right or obviously wrong), but unlikely to be correct enough to use in real situations.

Evaluating a "feature", whatever we mean by that[1], has to be always a qualified proposition. Using a feature may be a clear win for some projects, but not for others; it will depend on the cost of the feature relative to the project's needs and constraints.

This sort of evaluation is uncertain because it depends on how much the feature buys us, how much it takes away from our budget, and (not the least) because different implementations cost differently.

For instance, consider the implementation of potentially unaligned accesses on processors (like most RISC cpus) where unaligned access triggers an exception[PPC].

An example could be this function:

```
int fetch_from( void *p )
{
    return *(int *)p;             // what if 'p' points to an odd address?
}
```

where we may be requested to obtain an integer from a non-int-aligned address. This could be needed, for one case, if we had to extract information from a message format that had been designed without consideration for this problem.

The implementation could deal with this sort of access in a number of ways; two extreme ones are worth comparing. One is pessimistic (and normally inefficient), the other optimistic (and sometimes fast).

The implementation may decide that such accesses need to be performed by fetching bytes one at a time to ensure correct alignment. In effect, the **return** statement would mutate into something like:

```
{
        int     tmp;
        char    *dst = (char *)&tmp;
        char    *src = (char *)p;

        while (dst < (char *)&tmp + sizeof(int)) *dst++ = *src++;
        return tmp;
}
```

This, although legal, is likely to be inefficient always, as it dooms aligned accesses to the same treatment.

On the other hand, the implementation could be optimistic, in whose case the fetch through the argument pointer would be implemented directly as a word load, betting that the pointer points to a well-aligned integer. Because this implementation would trigger exceptions whenever it loses its bet, it would be part of its duty to supply an exception handler to correct the trapped accesses transparently.

Deciding between these two decisions isn't trivial. If you assume that most of your potential misaligned accesses will actually be misaligned, then fetching one byte at a time always is a good idea. You will be very slow in a few cases (when the access is actually aligned) but you will have bounded the cost of the misaligned ones. Otherwise, if most accesses are actually well aligned, then the exception handler is the way to go. The heavy cost of the context switch will be paid only by the few accesses that fail to be properly aligned.

Another consideration could be that you (the applications designer or implementor) don't control the exception vectors, and can't guarantee the installation of the needed handler.

Naturally, there are other possible implementations. The above tends to indicate simply that we can't say "you should not depend on potentially misaligned accesses"; as often, it just depends.

---

[1] For concreteness, think of "using virtual functions", "using multiple inheritance", "using **stdio**", as examples of adopting a feature. An example of a positive evaluation could be "we'll use **stdio** instead of writing our own I/O system, because it only takes 600K of storage, and we can have it today". A negative evaluation would sound like "we can't afford 600K of code for **stdio** alone, as we only output strings anyway--we'll write our own string I/O functions". These examples are not a claim that **stdio** takes (as much as, as little as) 600K in any given system, of course.

First Principle[2] of Feature Evaluation

Although feature evaluation must be relative to specific requirements and constraints, there is some room for general criteria. We can at least get a first level of evaluation, based on the observation that things done *before* run-time are cheaper than things done *at* run-time.

This observation leads to a cost evaluation principle:

> Features that can be implemented at compile (or assembly, or linkage) time are cheaper than features that require making decisions after the program has been loaded in memory.

Consider, for emphasis, these declarations:

```
int        i      = 0;                 // (d1)
int        *ip    = &i;                // (d2)
extern int ei;                         // (d3)
int        *eip   = &ei;               // (d4)
```

In (d1) we initialize a file-local variable. Some component of the development toolkit will know where the variable **i** will be located, and will be able to arrange for a zero to be copied into it. We can imagine that either the compiler or the assembler are likely to set up this initialization completely, leaving no work to be done at run-time.

Declaration (d2) may look the same as (d1), especially for absolute loading, but there is a difference in the general case. It is true that the toolkit will have early access to all the needed information (where the variable is, what the initial value should be), but it is likely that some final adjustment is needed, either by the linker, or by the run-time support (if the code is position-independent, for instance). There isn't a whole lot more to do in (d2) than in (d1), but there is some.

Finally, (d4) is like (d2), only more so. Even in absolute loads, the final address won't be known until the linker is done with the program; indeed, as a quality of implementation issue, the linker may not resolve the address completely, but could just create a little code stub to do the initialization at run-time.

Using the principle

The rest of this talk will cover uses of this principle for C++ features that the beginner designer may wish to consider. For each of the examples that we'll go through we'll try to determine what sort of run-time impact a feature has on code that uses it.

Limitations of the principle

The most important limitation of the principle is that it is only reliable when giving *negative* information. A feature that has to be expensive (defined by work that cannot be done before run-time) will be expensive always, but a feature that *could* be cheap may not be cheap at all in a given implementation.

Consequently, *trust but verify*. Before deciding that Feature X is so cheap that you can use it in every other line, it doesn't hurt to test examples of the intended use against your development toolkit. If the generated code is not as cheap as you hoped for, you have the choice of either taking the issue to your toolkit suppliers, or of not using the feature that way. If you wait until the code is ready, you may box yourself.

Quality-of-implementation isn't always predictable from prior experience. Techniques improve over time, and some times need to be replaced. Therefore, the advice you get from considering a given implementation may be incorrect after only a few releases of your preferred toolkit. Periodic re-evaluation (when considering new toolkits, for instance) is needed to make sure you don't build up dangerous assumptions.

---

[2] There is, so to speak, a *zeroth principle*: Do What You Must. From here on we assume that you have satisfied all mandatory constraints, and are exploring the feature space for good matches to your requirements.

A well-known example of the need for periodic assessment is the use of the **register** keyword in old C[KR1]. Although that was never guaranteed by the language, most implementations considered for register allocation first the variables declared with the **register** storage class, in their order of declaration. Current implementations prefer to ignore the declaration altogether, and do their own register allocation from scratch. Therefore, applications based on the concept that "the **register** feature should be used in all inner loops" may find surprising results when running on modern implementations that disregard the hint. In those implementations, applying the **register** keyword to a variable only indicates that you want an error message if you ever take that variable's address.

Application to C++ as of Early 1998

As we examine a number of examples below, we have to keep in mind that some of the comparisons are based on actual experience, while others are based on an educated guess. With the advent of a standard for C++, implementors can finally concentrate on implementing one language well, instead of spreading themselves thin by implementing many dialects. The guess is that the implementation of some of the new features in C++ (notably, templates) will be cheaper soon than is current practice now. This is so because the techniques needed to implement templates cheaply are well understood, and the need will be pressing as soon as the standard C++ library becomes commonplace.

Therefore, dangerous as it may be, the advice given below is on a "best-effort assumed" basis. Claims that a feature is expensive are generally stable if they are based on provable run-time needs. Claims to the contrary, on the other hand, are vulnerable to changes in the implementation.

C++: Cheap features

In this section I consider a number of features whose implementation is cheap, because it can be done entirely inside the compiler. In practice, moreover, it is done that way; there is little or no potential for surprises here.

Initialized Static Consts (vs Preprocessor Defines)

A stated goal of C++'s design is to render the C preprocessor obsolescent. To achieve that goal one needs to supply the functionality provided by both object-like and function-like macros [C90, §6.8.3].

C++'s replacement for object-like macros (at least, most of the interesting ones) is in the form of static consts initialized to literals. The idea is to replace

```
#define EOF (-1)
```

with

```
const int EOF = -1;              // static by default
```

The usual concern about this replacement is that the defined variable will take space at run-time. However, if the address of the variable is never taken, the compiler is very likely to "fold" the literal into its uses (this assumes that the value is, indeed, constant-foldable: typically this demands a scalar type).

Arbitrary Placement of Declarations

C requires a specific sequence: each block starts with declarations, then statements. C++ permits declarations interspersed with the statements, allowing a declaration to happen nearer the point of actual usage (often, also allowing a context-dependent initialization to happen after preliminary elaboration). This is just syntactic convenience, and does not mean that a C++ implementation "opens" and "closes" scopes at run-time.

This concern may come from consideration of very old C compilers, which started each block by subtracting from the stack pointer to allocate frame space. That stopped being the case very long ago.

References

References (and reference types) are just syntactic sugar for pointers. The compiler doesn't need to generate special code to keep track of where the reference points to, and in some cases the reference itself does not exist at run time (the variable referred to does, of course). Passing by reference is just passing a pointer, and leaving it to the compiler to figure out when that pointer needs dereferencing.

Namespaces

Code reuse sometimes has the disadvantage that you end up with name collisions. Think of those "sort" or "display" functions (named with that generic lack of imagination) that appear in most code. Those collisions sometimes can be helped by hiding things in classes, but not everything can be construed as a class member, and C libraries don't lend themselves easily to such wrapping.

Namespaces take care of (most of) this problem. Any names appearing on your code (variables, functions, enumerators, etc.) are resolved to one namespace or another, including the unnamed global one. There is no run-time penalty to using namespaces.

New/Delete

The combination of **new** and constructors takes care of allocating and initializing a heap-based object. It is no more expensive than what you could do by hand with **malloc** and initialization functions in C. This is clearly cheap, and less prone to accidents than the **malloc** route.

Similarly, combining **delete** and destructors gives us the same functionality as **free** would in C. Perhaps there is a certain run-time cost: most C structs don't get "destructed" in the C++ sense. However, the default, compiler-provided, destructor in C++ is empty and normally costs nothing.

So, there is no reason to **malloc** and **free** things yourself.

Now, this is not the whole story. The **new** and **delete** facilities can throw exceptions; we'll address that below.

Finally, let's evaluate array-**new** and array-**delete**. These operators incur a certain run-time cost: someplace there has to be a way to tell, from just the address of an array, how many elements it has. This cost cannot be less than **sizeof(size_t)** , if we hide the array size at a fixed offset from the heap object that is allocated to the array. It could grow perhaps to **sizeof(size_t)+K * sizeof(void *)**, if the mapping from address to size is some sort of hash table (the coefficient **K** has to do with the hash table implementation in use).

Typically, this added cost is negligible: maintaining the mapping is trivial in comparison to allocating/initializing the array; and the memory used up this way is much less than the memory used up in the array storage itself.

However, this little cost could be important if you find yourself generating a lot of *small* dynamic arrays, as in this example:

```
double *point = new double[3];    // and then more of the same
. . .
delete[] point;
```

You should consider keeping those small arrays (especially if they are all of the same size) inside classes, for the sake of decreasing the potential cost of keeping track of their sizes at run-time.

Overloading

Again, this is largely an issue of naming. The compiler needs to determine, just by looking at a call, which of several functions identically named (but with different parameter types) most closely matches the types of the arguments of the call.

A resolved call to an overloaded function does not look at all different (from a run-time perspective) than a call to a non-overloaded function. Therefore, this is a trivially cheap feature when run-time space and time consumption are considered.

Two warnings are in order, though. The first has nothing to do with run-time efficiency: there is an intellectual cost in tracking which of a set of overloaded functions is meant. This is especially so when overloaded operators are used. Overloading may attract beginner developers even when inappropriate (no feature worth anything is free from this). When **x+y** happens to mean "open file **x** and append structure **y** as a record, then close **x** again", then any amount of complexity may be hiding behind the most innocent-looking code.

The second warning has to do with relying on overloading resolution to convert (via constructors) the arguments to a call. For instance, in

```
struct S {
    int            i;
    S( int ii ) : i ( ii ) { };
};

extern void f( S );

int main( )
{
    f( 1 );                                 // actually, f( S( 1 ) );
    return 0;
}
```

it may be hard to see that the call to **f(1)** requires a temporary location to construct **S(1)** first. If that sort of expression occurs frequently in the same function, its stack requirements may grow surprisingly.

The example above worries us because of the conversion, not of the overloading. Indeed, it can be fixed by using overloading:

```
// add this line somewhere above
void f( int j ) { f( S( j ) ); }
```

makes sure that only one temporary is constructed on the stack, no matter how many times expressions of the form **f(** *integer* **)** appear in the code.

Yet, it is useful to mention this sort of waste in the context of overloading (instead of in the context of construction/conversion) because some times the surprise comes from a conversion that made the overloading resolvable. For instance:

```
class S;
class T;
extern void f( S );
extern void f( T );
. . .
    f( 1 );
```

the call is unambiguous if one of **S** or **T** (but not the other) has a constructor that converts integers into objects of the proper class. You normally won't get a message in this case, but you will get the temporary variable on the stack.

C++: Maybe-cheap features

Here I consider features that *could* be cheap. There is no substantial reason for these features to generate code (or take space) that couldn't be matched with something you would have to write yourself in C. On the other hand, the state of the art may no be quite there; verification by challenging your chosen compiler is mandatory.

Initialization/Conversion/Termination

I have already mentioned cases where implicit conversion may add unexpected costs.

Beyond that, there is a risk of code space waste. The compiler must generate special members (constructors, destructors) if you don't provide them yourself. Regrettably, the compiler may not know that some of those were not provided because they weren't needed. Proper dead-code elimination at link time should take care of this, but you should make sure that your compiler doesn't leave useless special members around.

Sometimes one hears the complaint that the run-time is calling constructors that the code doesn't even mention. The usual reason is implicit conversion; you may not be aware that, for instance, passing an argument by value has necessitated a copy construction. Obviously, you don't want the compiler to eliminate the special members in that case.

Chances are, these special members by themselves are not a performance problem. Their usefulness repays whatever expense is incurred.

Function In-lining

In-line expansion of functions is another issue altogether. If your compiler takes the **inline** hint as a command to inline, you may be in trouble.

The reason is that in-lining can be very expensive in code space. Modern processors make function calling protocols quite cheaper than was usual in the CISC era; it may not pay off to in-line even very small functions, especially in the presence of interprocedural register allocation.

A better attitude to take is that **inline** hints the compiler to consider the functions so marked before anything else is inlined, but to feel free to ignore the hint (as with **register**).

If you verify that your compiler in-lines excessively you should consider looking for options that limit the in-lining, or even building your application with **-Dinline=** to suppress the hint.

Multiple Inheritance

Single inheritance allows you to refine (add behavior) to code written by others. Perhaps you received a **class IOPort**, and refined it to a **class SerialPort**. Multiple inheritance allows you to combine (add among themselves, and then add behavior) several classes. For instance, from your **class SerialPort** and someone's **class IP**, you may be able to derive your own **class SLIP**.

Therefore, multiple inheritance is a good thing, but not much better than single inheritance. Nevertheless, multiple inheritance requires larger virtual function tables.

Of course, your toolkit should make every effort to reduce the number of copies of the virtual function tables it needs. It should also give you the option to promise not to use multiple inheritance, so that the compiler can use thinner virtual function tables. You have matter for verification here, especially if you want to use the full power of C++.

Using templates

Templates are very compact descriptions of parametric code. For instance, you may have types that differ only on some count (how many widgets in a gadget, as in "reserve a queue for N characters, otherwise you are a serial port") or on a type (as in "you are an IP stack, using a SLIP/PPP/Ethernet/… link layer").

We used to do this with pre-processor macros, as in

```
#define NEW(T) ((T *)malloc(sizeof(T)))
#define BUFFER(NAME,T,N)\
    T       NAME[N];
```

Templates are completely converted into normal, non-parametric code, before[3] run-time (we say that they are *instantiated*). Therefore, they should be *very* cheap. Regrettably, the mechanism that instantiates templates may, conservatively, instantiate too much.

This is especially worrisome for the C++ standard library, as it depends heavily on templates. All toolkit implementors are likely to optimize this aspect at least somewhat; it is only a matter of time before all viable implementations of C++ impose no, or almost no, code space overhead for using the standard libraries. This is something you need to verify too: to what extent does your compiler produce unneeded instantiations, and what plans there are to reduce the waste.

Your own templates should not be any more expensive to use than standard library templates, of course. That is worth verifying, too.

C++: The expensive stuff

Finally, I consider those features that have some unavoidable run-time impact. I am not recommending, of course, that you *never* use them. Instead, I would say that beginner users of the language should think long and hard before these features take a significant part in their designs. There is some complexity involved, and no amount of optimization will remove all of it.

The characteristic common to all these expensive features is the need to maintain data structures at run-time.

Run-Time Type Identification

Run-Time Type Identification (RTTI) provides answers to this question: "given that I have a pointer to type T, is the object pointed to *also* of type S?". Generally, this is not true for randomly chosen types S and T. However, if T is derived from S, it is always true; if S is derived from T, it is *sometimes* true. The practical form of this test is to ask (by casting) for a pointer to the object of type S that corresponds to our pointer to type T.

Never mind how useful this is, it requires keeping at run-time a representation of the type hierarchy, and tracking the ultimate provenance of each object.

RTTI introduces two costs, then. In terms of space, it requires a data structure that will provide the type correspondences, even across several steps of derivation. It also consumes time, because that data structure needs to be maintained.

As this feature was introduced relatively late in C++'s development, it is common to find options to refuse RTTI support. It does make sense (if you don't have a good use for it) to refuse this feature; notice that you should also refuse Exception Handling in that case. The beginning C++ developer has other reasons to decline the favor of Exception Handling anyway.

Exception Handling

Exception handling (EH) is, perhaps, the single most expensive feature an embedded design may consider. On stylistic grounds it is a far superior way to capture aberrant executions than the traditional return codes; it counts as one of the major reasons why C++ is better than C for development in large scale.

The problem is for embedded system developers. For starters, EH demands an RTTI implementation, or something very close to it. Second, a **throw** (a non-recoverable bad thing being forwarded upwards in the call chain in hopes that someone will know what to do with it) requires considerable run-time effort: for the current scope, already built objects need to be destructed. Then, for each consecutive scope around the one that threw, we need to find if it is capable of handling the problem (by comparing the types it is willing to **catch** with the type of the object being thrown). If so, we need to reactivate that scope with the appropriate

[3] I didn't say "at compile time". This is more interesting (i.e., better left for another occasion) than it looks at first sight.

context, else we need to act as if this scope had initiated the throwing (therefore, destructing everything constructed so far, etc.).

This is way too much help.

The typical implementation has to decide between pessimistic or optimistic approaches. In the pessimistic approach some work will be incurred in each scope (in case it is involved in the unwinding of a throw). In the optimistic approach, the preparation work will be postponed until the time of the throw (which is hoped never to happen). Most implementations opt for building (at compile time) potentially large tables that describe, for a throw within a range of addresses:

- What objects are fully constructed and need destruction,
- Where to find the types of surrounding catchers.

Discussion of the merits and costs of EH would require a talk by itself. Suffice here to recommend against casual use of EH in embedded applications.

This is not as straightforward as it seems. First, a C++ standard system will throw under some circumstances; you have to decide what to do with those (for instance, on memory exhaustion **new** will throw an object of type **bad_alloc**). For some of these cases, the language offers portable ways to avoid the exception (**new(nothrow)…** depends on the old method of returning a null pointer on failures).

Second, code you are reusing may depend on exceptions for error handling.

This is a suitable attitude for beginners (and, as such, can be refined as one gets confidence on the language):

> Don't use exceptions of your own. If you must, let them be catastrophic (that is, cause a reset). In particular, don't use exceptions for communication, for event-driven programming, or for any common situation that requires a repair (rather than a system reset).

It is possible to implement EH without RTTI (program meanings are slightly different). Indeed, there have been catch-and-throw implementations even for Old C. So it could appear practical to have EH but no RTTI. I discourage that thought: the cost of RTTI once you have EH is not so large that you should prefer a departure from the standard.

Conclusions

Anticipating the performance impact of those features of C++ that distinguish it from C can be difficult. Not only it may prove hard to estimate the run-time consequences of some of them, it just happens that each development toolkit's choice of implementation techniques may make comparisons unpredictable.

However, even developers new to C++ can benefit from an analysis based on two pillars:

- Features that require run-time decisions should be expensive; therefore, consider them expensive.
- Features that are implementable entirely "in the compiler's head" should be cheap. However, testing your chosen toolkit early about this assumption is easy, and can be a life-saver. Consider them cheap only after verification.

I have tried to give above my view of what is cheap and what is not for embedded systems development. This advice comes from the perspective of early 1998; it is, therefore, to be revised as the state of the art advances. The aim has been to help those developers who are still making the first acquaintance of C++. I hope that these comparisons will be useful, at least as a starting point for argument, but they do not replace doing your own analysis based on the requirements and constraints of your project.

References

[BS94] *The Design and Evolution of C++*. Bjarne Stroustrup. Addison-Wesley Publishing Company, 1994. This is the definitive document as to the *why* of C++'s features.

[BS97] *The C++ Programming Language*. Bjarne Stroustrup. Third edition, Addison-Wesley Publishing Company, 1997. The language will actually be defined by an international standard, for which a final draft has already been approved as of this writing. This reference is likely to remain authoritative even after publication of the international standard.

[C90] ISO/IEC 9899-1990. The international standard for C. It echoes substantially the *American National Standard for Programming Languages—C*, from 1989. A number of *technical corrigenda* amend or clarify its text.

[KR1] *The C Programming Language*. Brian W. Kernighan, Dennis M. Ritchie. First edition, Prentice-Hall Inc., 1978.

[PPC] *PowerPC™ Microprocessor Family: The Programming Environments*. IBM/Motorola, 1997. This is just a good example of a RISC processor; others abound nowadays.