Modern C++ in embedded systems â€“ Part 1: Myth and Reality

Dominic Herity - February 17, 2015

In 1998, I wrote an article for Embedded Systems Programming called **C++ in Embedded Systems â€“ Myth and Reality.** The article was intended to inform C programmers concerned about adopting C++ in embedded systems programming.

A lot has changed since 1998. Many of the myths have been dispelled, and C++ is used a lot more in embedded systems. There are many factors that may contribute to this, including more powerful processors, more challenging applications, and more familiarity with object-oriented languages.

C99 (an informal name for ISO/IEC 9899:1999) adopted some C++ features including const qualification and inline functions. C++ has also changed. C++11 and C++14 have added some cool features (how did I manage without the auto type specifier?) and some challenges, like deciding when to use constexpr functions.

But C++ has not displaced C, as I thought it would in 1998. C is alive and well in the Linux kernel, and there is a body of opinion implacably opposed to C++ in that environment.

The suspicion lingers that C++ is somehow unsuitable for use in small embedded systems. For 8- and 16-bit processors lacking a C++ compiler, that may be a concern, but there are now 32-bit microcontrollers available for under a dollar supported by mature C++ compilers. As this article series will make clear, with the continued improvements in the language most C++ features have no impact on code size or on speed. Others have a small impact that is generally worth paying for. To use C++ effectively in embedded systems, you need to be aware of what is going on at the machine code level, just as in C. Armed with that knowledge, the embedded systems programmer can produce code that is smaller, faster and safer than is possible without C++.

My history with C++

When I started a new microcontroller project a few years ago, I had to choose a tool-chain for the project. The MCU used (NXP LPC2458) was a 72MHz ARM7 with 512KB FLASH and 64KB RAM. Some toolchain vendors were surprised to be asked about the memory footprint of C++ libraries. When one vendor was pressed on the issue of a bloated library component, they said not many people are using C++ in such resource-constrained devices and itâ€™s hard to justify the cost of improving the library. Bear in mind that this â€œresource-constrained deviceâ€ was somewhat more powerful than the DOS platform that ran commercial software written in C++ in the 90s.

So in 2015, it seems that thereâ€™s still a need to de-mystify C++ for software engineers who are expert in embedded systems and in C, but wary of C++. If youâ€™re not familiar with C++, if you find that not many people are using it for applications like yours and if itâ€™s considered unsuitable for the Linux kernel, this wariness is understandable.

This is a revised version of the 1998 article addressing this issue. Less attention is given to features present in C99, since C programmers are likely to be familiar with them. The reader is assumed to be familiar with C99, which is used in the C code examples. The reader is also assumed to understand the C++ language features discussed, but doesnâ€™t need to be a C++ expert. A reader that is unfamiliar with some language features can still get value from this article by skipping over those features. The intended use of C++ language features and why they might be preferable to alternatives is also beyond

the scope of this article.

This article aims to provide a detailed understanding of what C++ code does at the machine code level, so that readers can evaluate for themselves the speed and size of C++ code as naturally as they do for C code.

To examine the nuts and bolts of C++ code generation, we will discuss the major features of the language and how they are implemented in practice. Implementations will be illustrated by showing pieces of C++ code followed by the equivalent (or near equivalent) C code. We will then discuss some pitfalls specific to embedded systems and how to avoid them.

We will not discuss the uses and subtleties of the C++ language or object-oriented design, as these topics have been well covered elsewhere. See http://en.cppreference.com/w/ for explanations of specific C++ language features.

C++11 and C++14 features are discussed separately in sections towards the end. The bulk of the article applies to the C++03 version of the language. C++11 is backward compatible with C++03 and C++14 is backward compatible with C++11. This helps the reader to ignore advanced features on a first reading and come back to them later.

Myths about C++. Some of the perceptions that discourage the use of C++ in embedded systems are:

- C++ is slow.
- C++ produces bloated machine code.
- Objects are large.
- Virtual functions are slow.
- C++ isnâ€™t ROMable.
- Class libraries make large binaries.
- Abstraction leads to inefficiency.

Most of these ideas are wrong. When the details of C++ code generation are examined in detail, hopefully it will be clear what the reality behind these myths is.

Anything C does, C++ can do. One property of C++ is so obvious that it is often overlooked. This property is that C++ is almost exactly a superset of C. If you write a code fragment (or an entire source file) in the C subset, the compiler will usually act like a C compiler and the machine code generated will be what you would get from a C compiler. (See **Compatibility of C and C++** for information about C constructs that wonâ€™t compile as C++)

Because of this simple fact, anything that can be done in C can also be done in C++. Existing C code can typically be re-compiled as C++ with about the same amount of difficulty that adopting a new C compiler entails. This also means that migrating to C++ can be done gradually, starting with C and working in new language features at your own pace. Although this is not the best way to reap the benefits of object-oriented design, it minimizes short term risk and provides a basis for iterative changes to a working system.

Front end features - a free lunch

Many of the features of C++ are strictly front-end issues. They have no effect on code generation. The benefits conferred by these features are therefore free of cost at runtime.

Default arguments to functions are an example of a cost-free front end feature. The compiler inserts default arguments to a function call where none are specified by the source.

A less obvious front end feature is â€˜function name overloadingâ€™. Function name overloading is made possible by a remarkably simple compile time mechanism. The mechanism is commonly called â€˜name manglingâ€™, but has also been termed â€˜name decorationâ€™. Anyone who has seen a linker error about the absence of `?my_function@@YAHH@Z` knows which term is more appropriate.

Name mangling modifies the label generated for a function using the types of the function arguments, or function signature. So a call to a function void my_function(int) generates a label like `?my_function@@YAXH@Z` and a call to a function void `my_function(my_class*)` generates a label like `?my_function@@YAXPAUmy_class@@@Z`. Name mangling ensures that functions are not called with the wrong argument types and it also allows the same name to be used for different functions provided their argument types are different.

Listing 1 shows a C++ code fragment with function name overloading. There are two functions called `my_function`, one taking an `int` argument, the other taking a `char const*` argument.

```
// C++ function name overload example
void my_function(int i) {
  // ...
}

void my_function(char const* s) {
  // ...
}

int main() {
   my_function(1);
   my_function("Hello world");
   return 0;
}
```

Listing 1: Function name overloading

Listing 2 shows how this would be implemented in C. Function names are altered to add argument types, so that the two functions have different names.

```
/* C substitute for function name overload */

void my_function_int(int i) {
    /* ... */
}

void my_function_charconststar(char const* s) {
    /* ... */
}
int main() {
    my_function_int(1);
    my_function_charconststar ("Hello world");
    return 0;
}
```

Listing 2: Function name overloading in C

References
A reference in C++ is physically identical to a pointer. Only the syntax is different. References are safer than pointers because they canâ€™t be null, they canâ€™t be uninitialized, and they canâ€™t be changed to point to something else. The closest thing to a reference in C is a const pointer. Note that this is not a pointer to a const value, but a pointer that canâ€™t be modified. **Listing 3** shows a C++ code fragment with a reference.

```
// C++ reference example
void accumulate(int& i, int j) {
    i += j;
}
```

Listing 3: C++ reference

Listing 4 shows how this would be implemented in C.

```
/* C substitute for reference example */
void accumulate(int* const i_ptr, int j) {
    *i_ptr += j;
}
```

Listing 4: Reference in C

Classes, member functions and objects
Classes and member functions are the most important new concept in C++. Unfortunately, they are usually introduced without explanation of how they are implemented, which tends to disorient C programmers from the start. In the subsequent struggle to come to terms with object-oriented design, hope of understanding code generation quickly recedes.

But a class is almost the same as a C `struct`. Indeed, in C++, a `struct` is defined to be a class whose members are public by default. A member function is a function that takes a pointer to an object of its class as an implicit parameter. So a C++ class with a member function is equivalent, in terms of code generation, to a C struct and a function that takes that `struct` as an argument.

Listing 5 shows a trivial class A with one member variable x and one member function f().

```
// A trivial class

class A {
private:
    int x;
public:
    void f();
};

void A::f() {
    x = 0;
}
```

Listing 5: A trivial class with member function

Parts of a class are declared as private, protected, or public. This allows the programmer to prevent

misuse of interfaces. There is no physical difference between private, protected, and public members. These specifiers allow the programmer to prevent misuse of data or interfaces through compiler enforced restrictions.

Listing 6 shows the C substitute for Listing 5. `Struct A` has the same member variable as class A and the member function `A::f()` is replaced with a function `f_A(struct A*)`. Note that the name of the argument of `f_A(struct A*)` has been chosen as â€œthisâ€, which is a keyword in C++, but not in C. The choice is made deliberately to highlight the point that in C++, an object pointer named this is implicitly passed to a member function.

```
/* C substitute for trivial class A */

struct A {
    int x;
};

void f_A(struct A* this) {
    this->x = 0;
}
```

Listing 6: C substitute for trivial class with member function

An object in C++ is simply a variable whose type is a C++ class. It corresponds to a variable in C whose type is a struct. A class is little more than the group of member functions that operate on objects belonging to the class. When an object-oriented application written in C++ is compiled, data is mostly made up of objects and code is mostly made up of class member functions.

Clearly, arranging code into classes and data into objects is a powerful organizing principle. Clearly also, dealing in classes and objects is inherently no less efficient than dealing with functions and data.
Title-1
Constructors and destructors
In C++, a constructor is a member function that is guaranteed to be called when an object is instantiated or created. This typically means that the compiler generates a constructor call at the point where the object is declared. Similarly, a destructor is guaranteed to be called when an object goes out of scope. So a constructor typically contains any initialization that an object needs and a destructor does any tidying up needed when an object is no longer needed.

The insertion of constructor and destructor calls by the compiler outside the control of the programmer is something that makes the C programmer uneasy at first. Indeed, programming practices to avoid excessive creation and destruction of so-called temporary objects are a preoccupation of C++ programmers in general. However, the guarantee that constructors and destructors provide - that objects are always initialized and are always tidied up - is generally worth the sacrifice. In C, where no such guarantees are provided, consequences include frequent initialization bugs and resource leakage.

Namespaces
C++ namespaces allow the same name to be used in different contexts. The compiler adds the namespace to the definition and to name references at compile time. This means that names don’t have to be unique in the application, just in the namespace in which they are declared. This means that we can use short, descriptive names for functions, global variables, classes, etc. without having to keep them unique in the entre application. **Listing 7** shows an example using the same function name in two namespaces.

```
// Namespace example
namespace n1 {
    void f() {
    }
    void g() {
        f(); // Calls n1::f() implicitly
    }
};

namespace n2 {
    void f() {
    }
    void g() {
        f(); // Calls n2::f() implicitly
    }
};

int main() {
    n1::f();
    n2::f();
    return 0;
}
```

Listing 7: Namespaces

When large applications are written in C, which lacks namespaces, this is often achieved by adding prefixes to names to ensure uniqueness. See **Listing 8.**

```
/* C substitute for namespace */

void n1_f() {
}

void n1_g() {
    n1_f();
}

void n2_f() {
}

void n2_g() {
    n2_f();
}

int main() {
    n1_f();
    n2_f();
    return 0;
}
```

Listing 8: C substitute for namespaces using prefixes

Inline functions
Inline functions are available in C99, but tend to be used more in C++ because they help achieve abstraction without a performance penalty.

Indiscriminate use of inline functions can lead to bloated code. Novice C++ programmers are often cautioned on this point, but appropriate use of inline functions can significantly improve both size and speed.

To estimate the code size impact of an inline function, estimate how many bytes of code it takes to implement it and compare that to the number of bytes needed to do the corresponding function call. Also consider that compiler optimization can tilt the balance dramatically in favor of the inline function. If you conduct actual comparisons studying generated code with optimization turned on, you may be surprised by how complex an inline function can profitably be. The breakeven point is often far beyond what can be expressed in a legible C macro.

Operator overloading
A C++ compiler substitutes a function call when it encounters an overloaded operator in the source. Operators can be overloaded with member functions or with regular, global functions. So the expression `x+y` results in a call to `operator+(x, y)` or `x.operator+(y)` if one of these is declared. Operator overloading is a front end issue and can be viewed as a function call for the purposes of code generation.

New and delete
In C++, `new` and `delete` do the same job as `malloc()` and `free()` in C, except that they add constructor and destructor calls, eliminating a source of bugs.

Simplified container class
To illustrate the implementation of a class with the features we have discussed, let us consider an example of a simplified C++ class and its C alternative.

Listing 9 shows a (not very useful) container class for integers featuring a constructor and destructor, operator overloading, new and delete. It makes a copy of an int array and provides access to array values using the `operator[]`, returning 0 for an out of bounds index. It uses the `(nothrow)` variant of new to make it easier to compare to the C alternative.

```
#include <iostream>
#include <new>

class int_container {
public:
    int_container(int const* data_in, unsigned len_in) {
        data = new(std::nothrow) int[len_in];
        len = data == 0? 0: len_in;
        for (unsigned n = 0; n < len; ++n)
            data[n] = data_in[n];
    }

    ~int_container() {
        delete [] data;
    }

    int operator[](int index) const {
        return index >= 0 && ((unsigned)index) < len?
```

```
data[index]: 0;
        }
    private:
        int* data;
        unsigned len;
    };

    int main() {
        int my_data[4] = {0, 1, 2, 3};
        int_container container(my_data, 4);
        std::cout << container[2] << "\n";
    }
```

Listing 9: A simple integer container class featuring constructor, destructor, operator overloading, new and delete

Listing 10 is a C substitute for the class in Listing 9. Operator overload `int_container::operator[](int)` is replaced with function `int_container_value(…)`. The constructor and destructor are replaced with `int_container_create(…)` and `int_container_destroy(…)`. These must be called by the user of the class, rather than calls being added automatically by the compiler.

```
    #include <stdio.h>
    #include <stdlib.h>

    struct int_container {
        int* data;
        unsigned len;
    };

    void int_container_create(struct int_container* this, int
const* data_in, unsigned len_in) {
        this->data = malloc(len_in * sizeof(int));
        this->len = this->data == 0? 0: len_in;
        for (unsigned n = 0; n < len_in; ++n)
            this->data[n] = data_in[n];
    }

    void int_container_destroy(struct int_container* this) {
        free(this->data);
    }

    int int_container_value(struct int_container const* this, int
index) {
        return index >= 0 && index < this->len? this->data[index]:
0;
    }

    int main() {
      int my_data[4] = {0, 1, 2, 3};
      struct int_container container;
      int_container_create(&container, my_data, 4);
```

```
    printf("%d\n", int_container_value(&container, 2));
    int_container_destroy(&container);
}
```

Listing 10: C substitute for simple string class

Note how much easier to read `main()` is in Listing 9 than in Listing 10. It is also safer, more coherent, more maintainable, and just as fast. Consider which version of `main()` is more likely to contain bugs. Consider how much bigger the difference would be for a more realistic container class. This is why C++ and the object paradigm are safer than C and the procedural paradigm for partitioning applications.

All C++ features so far discussed confer substantial benefits at no runtime cost.

Inheritance

In discussing how C++ implements inheritance, we will limit our discussion to the simple case of single, non-virtual inheritance. Multiple inheritance and virtual inheritance are more complex and their use is rare by comparison.

Let us consider the case where class B inherits from class A. (We can also say that B is derived from A or that A is a base class of B.)

We know from the previous discussion what the internal structure of an A is. But what is the internal structure of a B? We learn in object-oriented design (OOD) that inheritance models an ‘is a’ relationship – that we should use inheritance when we can say that a B ‘is a’ A. So if we inherit Circle from Shape, we’re probably on the right track, but if we inherit Shape from Color, there’s something wrong.

What we don’t usually learn in OOD is that the ‘is a’ relationship in C++ has a physical as well as a conceptual basis. In C++, an object of derived class B is made up of an object of base class A, with the member data of B tacked on at the end. The result is the same as if the B contains an A as its first member. So any pointer to a B is also a pointer to an A. Any member functions of class A called on an object of class B will work properly. When an object of class B is constructed, the class A constructor is called before the class B constructor and the reverse happens with destructors.

Listing 11 shows an example of inheritance. Class B inherits from class A and adds the member `function B::g()` and the member `variable B::secondValue.`

```
// Simple example of inheritance

class A {
public:
    A();
    int f();
private:
    int value;
};

A::A() {
    value = 1;
}

int A::f() {
```

```
    return value;
}

class B: public A {
private:
    int secondValue;
public:
    B();
    int g();
};

B::B() {
    secondValue = 2;
}

int B::g() {
    return secondValue;
}

int main() {
    B b;
    b.f();
    b.g();
    return 0;
}
```

Listing 11: Inheritance

Listing 12 shows how this would be achieved in C. `Struct B` contains a `struct A` as its first member, to which it adds a variable `secondValue`. The function `BConstructor(struct B*)` calls `AConstructor` to ensure initialization of its 'base class'. Where the function `main()` calls `b.f()` in Listing 11, `f_A(struct A*)` is called in Listing 12 with a cast.

```
/* C Substitute for inheritance */

struct A {
    int value;
};

void AConstructor(struct A* this) {
    this->value = 1;
}

int f_A(struct A* this) {
    return this->value;
}

struct B {
    struct A a;
    int secondValue;
};
```

```
void BConstructor(struct B* this) {
    AConstructor(&this->a);
    this->secondValue = 2;
}

int g_B(struct B* this) {
    return this->secondValue;
}

int main() {
    struct B b;
    BConstructor(&b);
    f_A ((struct A*)&b);
    g_B (&b);
    return 0;
}
```

Listing 12: C Substitute for inheritance

It is startling to discover that the rather abstract concept of inheritance corresponds to such a straightforward mechanism. The result is that well-designed inheritance relationships have no runtime cost in terms of size or speed.

Inappropriate inheritance, however, can make objects larger than necessary. This can arise in class hierarchies, where a typical class has several layers of base class, each with its own member variables, possibly with redundant information.

Title-1
Virtual functions
Virtual member functions allow us to derive class B from class A and override a virtual member function of A with one in B and have the new function called by code that knows only about class A. Virtual member functions provide polymorphism, which is a key feature of object-oriented design.

A class with at least one virtual function is referred to as a ‘polymorphic’ class. The distinction between a polymorphic and a non-polymorphic class is significant because they have different trade-offs in runtime cost and functionality.

Virtual functions have been controversial because they exact a price for the benefit of polymorphism. Let us see, then, how they work and what the price is.

Virtual functions are implemented using an array of function pointers, called a vtable, for each class that has virtual functions. Each object of such a class contains a pointer to that class’s vtable. This pointer is put there by the compiler and is used by the generated code, but it is not available to the programmer and it cannot be referred to in the source code. But inspecting an object with a low level debugger will reveal the vtable pointer.

When a virtual member function is called on an object, the generated code uses the object’s vtable pointer to access the vtable for that class and extract the correct function pointer. That pointer is then called.

Listing 13 shows an example using virtual member functions. Class A has a virtual member function f(), which is overridden in class B. Class A has a constructor and a member variable, which are

actually redundant, but are included to show what happens to vtables during object construction.

```
// Classes with virtual functions

class A {
private:
    int value;
public:
    A();
    virtual int f();
};

A::A() {
    value = 0;
}

int A::f() {
    return 0;
}

class B: public A {
public:
    B();
    virtual int f();
};

B::B() {
}

int B::f() {
    return 1;
}

int main() {
    B b;
    A* aPtr = &b;
    aPtr->f();
    return 0;
}
```

Listing 13: Virtual Functions

Listing 14 shows what a C substitute would look like. The second last line in `main()` is a dangerous combination of casting and function pointer usage.

```
/* C substitute for virtual functions */

struct A {
    void **vTable;
    int value;
};
```

```
int f_A(struct A* this);

void* vTable_A[] = {
    (void*) &f_A
};

void AConstructor(struct A* this) {
    this->vTable = vTable_A;
    this->value = 1;
}

int f_A(struct A* this) {
    return 0;
}

struct B {
    struct A a;
};

int f_B(struct B* this);

void* vTable_B[] = {
    (void*) &f_B
};

void BConstructor(struct B* this) {
    AConstructor((struct A*) this);
    this->a.vTable = vTable_B;
}

int f_B(struct B* this) {
    return 1;
}

int main() {
    struct B b;
    struct A* aPtr;

    BConstructor(&b);
    typedef void (*f_A_Type)(struct A*);

    aPtr = (struct A*) &b;
    ((f_A_Type)aPtr->vTable[0]) (aPtr);
    return 0;
}
```

Listing 14: C substitute for virtual functions

This is the first language feature we have seen that entails a runtime cost. So let us quantify the costs

of virtual functions.

The first cost is that it makes objects bigger. Every object of a class with virtual member functions contains a vtable pointer. So each object is one pointer bigger than it would be otherwise. If a class inherits from a class that already has virtual functions, the objects already contain vtable pointers, so there is no additional cost. But adding a virtual function can have a disproportionate effect on a small object. An object can be as small as one byte and if a virtual function is added and the compiler enforces four-byte alignment, the size of the object becomes eight bytes. But for objects that contain a few member variables, the cost in size of a vtable pointer is marginal.

The second cost of using virtual functions is the one that generates most controversy. That is the cost of the vtable lookup for a function call, rather than a direct one. The cost is a memory read before every call to a virtual function (to get the object's vtable pointer) and a second memory read (to get the function pointer from the vtable). This cost has been the subject of heated debate and it is hard to believe that the cost is typically less than that of adding an extra parameter to a function. We hear no arguments about the performance impact of additional function arguments because it is generally unimportant, just as the cost of a virtual function call is generally unimportant.

A less discussed, but more significant, cost of virtual functions is their impact on code size. When an application is linked after compilation, the linker can identify regular, non-virtual functions that are never called and remove them from the memory footprint. But because each class with virtual functions has a vtable containing pointers to all its virtual functions, the pointers in this vtable must be resolved by the linker. This means that all virtual functions of all classes used in a system are linked. Therefore, if a virtual function is added to a class, the chances are that it will be linked, even if it is never called.

So virtual functions have little impact on speed, but their effects on code size and data size should be considered. Because they involve overheads, virtual functions are not mandatory in C++ as they are in other object-oriented languages. So if, for a given class, you find the costs outweigh the benefits, you can choose not to use virtual functions.

Templates

C++ templates are powerful, as shown by their use in the Standard C++ Library. A class template is rather like a macro that produces an entire class as its expansion. Because a class can be produced from a single statement of source code, careless use of templates can have a devastating effect on code size. Older compilers will expand a templated class every time it is encountered, producing a different expansion of the class in each source file where it is used. Newer compilers and linkers, however, find duplicates and produce at most one expansion of a given template with a given parameter class.

Used appropriately, templates can save a lot of effort at little or no cost. After all, it's a lot easier and probably more efficient to use complex<float> from the Standard C++ Library, rather than write your own class.

Listing 15 shows a simple template class A<T>. An object of class A<T> has a member variable of type T, a constructor to initialize and a member function `A::f()` to retrieve it.

```
// Sample template class

template<typename T> class A {
private:
    T value;
public:
    A(T);
```

```
    T f();
};

template<typename T> A<T>::A(T initial) {
    value = initial;
}

template<typename T> T A<T>::f() {
    return value;
}

int main() {
    A<int> a(1);
    a.f();
    return 0;
}
```

Listing 15: A C++ template

The macro `A(T)` in **Listing 16** approximates a template class in C. It expands to a `struct` declaration and function definitions for functions corresponding to the constructor and the member function. We can see that although it is possible to approximate templates in C, it is impractical for any significant functionality.

```
/* C approximation of template class */

#define A(T)                                                    \
    struct A_##T {                                              \
        T value;                                                \
    };                                                          \
                                                                \
    void AConstructor_##T(struct A_##T* this, T initial) {      \
        (this)->value = initial;                                \
    }                                                           \
                                                                \
    T A_f_##T(struct A_##T* this) {                             \
        return (this)->value;                                   \
    }

A(int) /* Macro expands to ‘class’ A_int */

int main() {
    struct A_int a;
    AConstructor_int(&a, 1);
    A_f_int(&a);
    return 0;
}
```

Listing 16: A C ‘template’

Exceptions

Exceptions are to `setjmp()` and `longjmp()` what structured programming is to `goto`. They

impose strong typing, guarantee that destructors are called, and prevent jumping to a disused stack frame.

Exceptions are intended to handle conditions that do not normally occur, so implementations are tailored to optimize performance for the case where no exceptions are thrown. With modern compilers, exception support results in no runtime cost unless an exception is thrown. The time taken to throw an exception is unpredictable and may be long due to two factors. The first is that the emphasis on performance in the normal case is at the expense of performance in the abnormal case. The second factor is the runtime of destructor calls between an exception being thrown and being caught.

The use of exceptions also causes a set of tables to be added to the memory footprint. These tables are used to control the calling of destructors and entry to the correct catch block when the exception is thrown.

For detailed information on the costs of exceptions with different compilers, see **Effective C++ in an Embedded Environment.**

Because of the cost of exception support, some compilers have a 'no exceptions' option, which eliminates exception support and its associated costs.

```
// C++ Exception example

#include <iostream>
using namespace std;

int factorial(int n) throw(const char*) {
    if (n<0)
        throw "Negative Argument to factorial";
    if (n>0)
        return n*factorial(n-1);
    return 1;
}

int main() {
   try {
         int n = factorial(10);
         cout << "factorial(10)=" << n;
    } catch (const char* s) {
         cout << "factorial threw exception: " << s << "\n";
    }
    return 0;
}
```

Listing 17: A C++ exception example

Listing 17 above shows an example of an exception and **Listing 18** below shows a C substitute that has several shortcomings. It uses global variables. It allows `longjmp(ConstCharStarException)` to be called either before it is initialized by `setjmp(ConstCharStarException)` or after `main()` has returned. In addition, substitutes for destructor calls must be done by the programmer before a `longjmp()`. There is no mechanism to ensure that these calls are made.

```
/* C approximation of exception handling */

#include <stdio.h>
#include <setjmp.h>

jmp_buf ConstCharStarException;
const char* ConstCharStarExceptionValue;

int factorial(int n) {
    if (n<0) {
    ConstCharStarExceptionValue = "Negative Argument to
factorial";
        longjmp(ConstCharStarException, 0);
    }
    if (n>0)
    return n*factorial(n-1);
    return 1;
}

int main() {
    if (setjmp(ConstCharStarException)==0) {
        int n = factorial(10);
        printf("factorial(10)=%d", n);
    } else {
        printf("factorial threw exception: %s\n",
ConstCharStarExceptionValue);
    }
    return 0;
}
```

Listing 18: A C ‘exception’ example

For language features discussed up to this point, it has been possible to entertain the possibility of a C substitute as a practical proposition. In this case of exceptions, however, the additional complexity and opportunities for error make a C substitute impractical. So if you’re writing C, the merits of exception-safe programming are a moot point.

Runtime type information

The term ‘runtime type information’ suggests an association with purer object-oriented languages like Smalltalk. This association raises concerns that efficiency may be compromised. This is not so. The runtime cost is limited to the addition of a type_info object for each polymorphic class and type_info objects aren’t large.

To measure the memory footprint of a type_info object, the code in **Listing 19** was compiled to assembly. The output was put through the name demangler at www.demangler.com and the result was annotated to highlight the size of the type_info object and the class name string. The result was 30 bytes. This is about the cost of adding a one line member function to each class.

Many compilers have an option to disable runtime type information, which avoids this cost for an application that does not use type_info objects.

```
// type_info test classes ////////////////
```

```
      class Base {
      public:
        virtual ~Base() {}
      };
      class Derived: public Base {};
      class MoreDerived: public Derived {};

/* type_info for more_derived generated by g++ 4.5.3 for cygwin.
Total 30 bytes */
_typeinfo for MoreDerived:
    .long    _vtable for __cxxabiv1::__si_class_type_info 8
/* 4 bytes */
    .long    _typeinfo name for MoreDerived
/* 4 bytes */
    .long    _typeinfo for Derived
/* 4 bytes */
/* ... */
_vtable for MoreDerived:
    .long    0
    .long    _typeinfo for MoreDerived
/* 4 bytes */
    .long    _MoreDerived::~MoreDerived()
    .long    _MoreDerived::~MoreDerived()
/* ... */
_typeinfo name for MoreDerived:
    .ascii "11MoreDerived\0"
/*14 bytes */
```

Listing 19: type_info Memory Footprint Measurement

Part 2: Modern C++ in Embedded Systems: Evaluating C++

Dominic Herity *is a Principal Software Engineer at* ***Faz Technology Ltd****. He has 30 years' experience writing software for platforms from 8 bit to 64 bit with full life cycle experience in several commercially successful products. He served as Technology Leader with Silicon & Software Systems and Task Group Chair in the Network Processing Forum. He has contributed to research into Distributed Operating Systems and High Availability at Trinity College Dublin. He has publications on various aspects of Embedded Systems design and has presented at several conferences in Europe and North America. He can be contacted at* ***dherity@gmail.com****.*