C++ in Embedded Systems: Myth and Reality

by Dominic Herity

Not that we want to foster a religious war, but according to this author, C++ is superior to C for embedded systems programming. Here's why.

I was surprised a while ago by what I heard in a coffee break conversation during a C++ course. The attendees were speculating on the relevance of C++ to embedded systems, and the misconceptions that surfaced drove me ultimately to write this article.

Some perceive that C++ has overheads and costs that render it somehow unsuited to embedded systems programming, that it lacks the control and brevity of C, or that, while it may be suited to some niche applications, it will never displace C as the language of choice for embedded systems.

These perceptions are wrong. Where compilers and other tools are adequate, C++ is always preferable to C as an implementation language for embedded systems. While doing everything that C does, it offers greater opportunities for expression, encapsulation, re-use, and it even allows size and speed improvements that are impractical in C.

Why, then, do these perceptions persist? The main reason is that when people form their opinions, they know a lot more about C than about C++. They have read some books, written some code, and are competent at using the features of C++, but they lack the knowledge of what is happening under the hood, the familiarity that allows one to visualize the disassembly while typing source or even while formulating a design. It is to these people that this article is addressed.

This article aims to replace exaggerated claims and misapplied generalizations with informed comment. It supports the view that C++, used appropriately, is superior to C for embedded systems programming.

It aims to provide detailed understanding of what C++ code does at the machine 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, I 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.

I will then discuss some pitfalls specific to embedded systems and how to avoid them. I won't discuss the uses and subtleties of C++ or object-oriented (OO) design, as these topics have been well covered elsewhere.

The Myths

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

- C++ is slow
- C++ produces large binaries
- Abstraction leads to inefficiency
- Objects are large
- Virtual functions are slow
- C++ isn't ROMable
- Class libraries make large binaries

Some of these perceptions are exaggerated concerns. Others are just wrong. When the details of C++ code generation are examined in detail, it will be clear what the reality behind these myths is.

Anything C can do, C++ can do better

The most obvious property of C++ is so obvious that it's often overlooked: C++ is a superset of C. If you write a code fragment (or an entire source file) in the C subset, the compiler will act like a C compiler and the machine code generated will be exactly what you would expect from a C compiler.

This simple point invalidates any claims that a system can be implemented in C, but not in C++. In practice, 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 isn't the best way to reap the benefits of OO design, it minimises short term risk and it provides a basis for iterative changes to a working system.

Front end features: a free lunch

Many features of C++ are strictly front end issues and have no effect on code generation. The benefits conferred by these features are therefore free of cost at run time. These features include use of the keywords **const**, **private**, **protected**, and **public**, which allow the programmer to prevent misuse of interfaces. No physical difference exists between **private**, **protected**, and **public** members. Neither is there a difference between **const** and **non-const** data. These specifiers allow the programmer to prevent misuse of data or interfaces through compiler-enforced restrictions.

Default arguments to functions are another neat, 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 function name overloading. Function name overloading is made possible by a remarkably simple compile-time mechanism. The mechanism is commonly called name mangling, but has been called name decoration by those who have nothing better to do than sanitize perfectly good terms. Anyone who has seen a linker protesting the absence of ?foo@@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 **int foo(int)** generates a reference to a label such as ?foo@@YAHH@Z, while a call to a function **void foo(int)** generates a label like ?foo@@YAXH@Z and a call to a function **void foo(bar*)** generates a label like ?foo@@YAXPAU bar@@@Z. Name mangling ensures that functions aren't 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 foo, one taking an int argument, the other taking a **char*** argument.

```
Listing 1

Function name overloading.


// C++ function name overload example

void foo(int i)

{

        // ...

}

void foo(char* s)

{
```

```
        // ...
}
void main()
{
        foo(1);
        foo("Hello world");
}
```

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.

```
Listing 2
Function name overloading in C.

/* C substitute for */
/* function name overload */
void foo_int(int i)
{
        /* ... */
}
void foo_charstar(char* s)
{
        /* ... */
}
void main()
{
        foo_int(1);
        foo_charstar("Hello world");
}
```

In C++, name mangling is automatic, but in a C substitute, it would be the responsibility of the programmer.

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 OO design, hope of understanding code generation quickly recedes.

Behind the protection and scoping, 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 3 shows a trivial class **foo** with one member variable **x** and one member function **bar()**.

```
Listing 3

A trivial class with member function.


// A trivial class

class foo

{

private:

        int x;

public:

        void bar();

};

void foo::bar()

{

        x = 0;

}
```

Listing 4 shows the C substitute for Listing 3. Struct **foo** has the same member variable as class **foo** and the member function **foo::bar()** is replaced with a function **bar_foo(struct foo*)**. Note the name of the argument of **bar_foo(struct foo*)** has been chosen as **this**, 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.

```
Listing 4

C substitute for trivial class with member function.


/* C substitute for trivial class foo */

struct foo

{

        int x;

};

void bar_foo(struct foo* this)

{

        this->x = 0;

}
```

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. At the machine code level, data is mostly made up of objects and code is mostly made up of classes.

Clearly, arranging code into classes and data into objects is a powerful organizing principle. Dealing with classes and objects is inherently just as efficient as dealing with functions and data.

Constructors and Destructors

In C++, a constructor is a member function guaranteed to be called when an object is instantiated or created. This typically means 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, the consequence is frequent initialization bugs and resource leakage.

Inline Functions

Inline functions are a safer and more powerful substitute for C macros in many situations. It is rare to see a macro that has local variables, and rightly so, because macros rapidly become illegible. Inline functions, by contrast, have the legibility and safety of ordinary functions. Clearly, indiscriminate use of inline functions can lead to bloated code, and novice C++ programmers are invariably cautioned on this point.

However, appropriate use of inline functions can 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 break-even 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 or global functions. So foo+bar is evaluated to be operator+(foo, bar) or foo.operator+ (bar), which in terms of code generation amounts to the same thing. 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 guarantee constructor and destructor calls. They also tend to be better-suited to frequent use with small quantities of memory than older implementations of **malloc()** and **free()**.

Simplified String Class

To illustrate the implementation of a class with the features we've discussed, let's consider an example of a simplified C++ class and its C alternative.

Listing 5 shows a string class featuring a constructor and destructor, operator overloading , new and delete, and an inline function.

```
Listing 5

A simple string class featuring constructors, destructors, operator overloading, new,
and delete.



// A simplified string class

#include

#include

using namespace std;



class String {

private:

  char* data;

  unsigned len;

public:

  String();

  ~String();
```

```
  unsigned length() const;
  String& operator=(const char* s);
};
inline unsigned String::length() const
{
  return len;
};
String::String() {
  len = 0;
  data = 0;
}
String::~String() {
  if (data != 0)
    delete [] data;
}
String& String::operator=(const char* s) {
  len = strlen(s);
  data = new char [ len+1 ];
  if (data == 0)
    len = 0;
  else
    strcpy(data, s);
  return *this;
}


void main() {
  String s;
  s = "Hello world";
  cout << s.length(); }
```

Listing 6 is a C substitute for the string class shown in Listing 5. The inline function **String::length()** is replaced by the macro **length_String()**. Operator overloads **String::operator=(const char*)** and **operator<< (ostream&, int)** are replaced with function calls **operatorEquals_String_const_ char_star(String*, const char*)** and **operator**

ShiftLeft_ostream_unsigned (ostream*, int) respectively. The constructor and destructor must then be called by the user of the class, rather than being added automatically by the compiler.

Listing 6

C substitute for simple string class.

```
/* C substitute for simplified string class */
#include
#include
#include

struct String {
  char* data;
  unsigned len;
};
#define length_String(s) ((s)->len)
void StringConstructor(String* this) {
  this->len = 0;
  this->data = 0;
}
void StringDestructor(String* this) {
  if (this->data != 0)
    free(this->data);
}
String operatorEquals_String_const_char_star(
  String* this, const char* s) {
  this->len = strlen(s);
  this->data = (char*) malloc(
    (this->len+1) * sizeof(char));
  /* If char had a constructor, */
  /* it would be have to be */
```

```
  /* called here. */
  if (this->data == 0)
    this->len = 0;
  else
    strcpy(this->data, s);
  return *this;
}
FILE*
  operatorShiftLeft_ostream_unsigned(FILE*,
    unsigned);

void main() {
  String s;
  StringConstructor(&s);
  operatorEquals_String_const_char_star(
    &s, "Hello world");
  operatorShiftLeft_ostream_unsigned(stdout,
    length_String(&s));
  StringDestructor(&s);
}
```

See how much harder to read the function **main()** is in Listing 6 than in Listing 5 and consider how much more danger there is of a bug occurring. Consider how much worse the problem would be for a more realistic string class. This is why C++ and the object paradigm are so much superior to C and the procedural paradigm for partitioning complex problems.

Next, consider that the code and data generated by Listing 5 is just as small and just as fast as that generated by Listing 6. It is also safer, more coherent, more readable, and more maintainable. Of the C++ features discussed so far, all confer substantial benefit at no run-time 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's consider the case in which 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 now know what the internal structure of an A is. What is the internal structure of a B? An object of class B is made up of an object of class A, with the member data of B tacked on at the end. In fact, the result is the same as if the B contains an A as its first member. Therefore, 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 first and the reverse happens with destructors.

Listing 7 shows an example of inheritance. Class B inherits from class A and adds the member function **B::g()** and the member variable **B::secondValue**.

```
Listing 7
Inheritance.

// 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;

}


void main() {

  B b;

  b.f();

  b.g();

}
```

Listing 8 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 initialisation of its "base class." Where the function **main()** calls **b.f()** in Listing 7, **f_A(struct A*)** is called in Listing 8 with a cast.

```
Listing 8

C substitute for inheritance.


/* 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;
}

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

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

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

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 OO design.

Virtual functions have been controversial. It would seem that they exact a price for the benefit of polymorphism. Let's 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 the class's vtable. This pointer is put there by the compiler and is used by the generated code, but it isn't available to the programmer and it can't be referred to in the source code. Inspection of objects with a low-level debugger will reveal the vtable pointer, if the reader is interested. Of course, the vtable pointer is kept from the programmer for good reasons, and using it directly is an excellent way to prevent your code being ported and to make its maintenance exciting!

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

Listing 9 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.

```
Listing 9
Virtual functions.

// 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;
}
```

```
void main() {
  B b;
  A* aPtr = &b;
  a->f();
}
```

Listing 10 shows what a C substitute would look like. The result is both extremely ugly and hazardous. In the last line of **main()**, we see the virtual function call, which, after all the casting, uses the object's vtable pointer to look up the vtable of its class for the function pointer.

```
Listing 10
C substitute for virtual functions.

/* 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 {
```

```
  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;
}


void 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);
}
```

Let's quantify the costs of virtual functions, in order of priority. 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 the 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 to 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 cost of virtual functions is their impact on code size. 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 isn't used in a particular system.

So the bottom line on virtual functions is that they have little impact on speed, but be aware of their effects on code size and data size. Virtual functions are not mandatory in C++, unlike in other OO languages.

Templates

C++ templates are powerful, as shown by their use in the Standard C++ Library. A class template is rather like a macro which produces an entire class as its expansion. Because a class can be produced from a single statement of source code, misuse 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's 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 generally more efficient to use **complex** from the Standard C++ Library, rather than write your own class.

Listing 11 shows a simple template class **A**. An object of class **A** has a member variable of class **T**, a constructor to initialise and a member function **A::f()** to retrieve it.

```
Listing 11
A C++ template.


// Sample template class
template class A {
private:
  T value;
public:
  A(T);
  T f();
};
template A::A(T initial) {
  value = initial;
}
template T A::f() {
  return value;
}


void main() {
```

```
  A a(1);
  a.f();
}
```

The macro **A(T)** in Listing 12 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's possible to approximate templates in C, it is impractical for any significant functionality.

```
Listing 12
A C "template."

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

A(int) /* Macro expands to Œclass' A_int */
```

```
void main() {
  A_int a;
  AConstructor_int(&a, 1);
  A_f_int(&a);
}
```

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 unused stack frame.

Exceptions are intended to handle conditions that don't normally occur, so implementations are tailored to optimize performance for the case when no exceptions are thrown. Support for exceptions results in a small performance penalty for each function call. (This is to record information to ensure destructor calls are made when 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 somewhat at the expense of performance in the abnormal case. The second factor is the run time of destructor calls between an exception being thrown and being caught.

Because of the performance penalty in the no exceptions case, many compilers have a "no exceptions" option, which eliminates exception support and its associated performance cost.

Listing 13 shows an example of an exception and Listing 14 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.

```
Listing 13
A C++ exception example.


/ C++ Exception example
#include
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;
}
```

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

Listing 14

A C "exception" example.

```
/* C approximation of exception handling */
#include
#include
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;
}

void main()
{
```

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

Run-time Type Information

Run-time type information is a recent addition to C++. Its name suggests an association with purer OO languages like Smalltalk. This association causes anxiety among the performance-conscious that efficiency has been compromised for purity. This is not so. Run-time type information exploits the vtable pointer in an object that has one and provides sensible defaults for an object that does not. If you don't use run-time type information, the only run-time cost is that classes are a little larger. If you use a compiler option to disable run-time type information, that cost is avoided.

Memory Considerations

Having discussed the implementation of the major C++ language features, we can now evaluate C++ in terms of the machine code it generates. Embedded systems programmers are particularly concerned about code and data size, so we need to discuss C++ in these terms.

How big is a class?

In C++, most code is in class member functions and most data is in objects belonging to these classes. C++ classes tend to have many more member functions than a C programmer would expect to use. This is because properly designed classes are complete and contain member functions to do anything with objects belonging to the class that might legitimately be needed. For a well conceptualized class, this number will be reasonably small, but nevertheless larger than what the C programmer is accustomed to.

When calculating code size, bear in mind that modern linkers designed for C++ extract from object files only those functions that are actually called, not the entire object files. In essence, they treat object files like libraries. This means that non-virtual class member functions that are unused have no code size penalty. So a class that seems to have a lot of baggage in terms of unused member functions may be quite economical in practice.

In the case of virtual functions, it is reasonable to assume that all virtual functions of all classes used in a system will be linked into the binary. But class completeness in itself does not lead to code bloat.

How big is an object?

The size of an object can be calculated by examining its class (and all its base classes). Ignore member functions and treat the data the same as for a struct. Then add the size of a pointer if there are any virtual functions in the class or base classes. You can confirm your result by using the **sizeof** operator. It will become apparent that the combined size of objects in a system need be no greater than the size of data in a C-based procedural model. This is because the same amount of state is needed to model a system regardless of whether it is organized into objects.

C++ and the Heap

Heap usage is much more common in C++ than in C. This is because of encapsulation. In C, where a function requires an unknown amount of memory, it is common to externalize the memory as an input parameter and leave the user with the problem. This is at least safer than **malloc**ing an area and relying on the user to free it. But C++, with its encapsulation and destructors, gives class designers the option (and responsibility) of managing the class heap usage.

This difference in philosophy is evident in the difference between C strings and a C++ string class. In C, you get a **char** array. You have to decide in advance how long your string can be and you have to continuously make sure it doesn't get any bigger. A C++ string class, however, uses new and delete to allow a string to be any size and to grow if necessary. It also makes sure that the heap is restored when the string is no longer needed.

The consequence of all this is that you can scrape by in an embedded system written in C without implementing **malloc** and **free**, but you won't get far in C++ without implementing **new** and **delete**.

ROMable Objects

Linkers for embedded systems allow const static data to be kept in ROM. For a system written in C, this means that all the non-varying data known at compile time can be specified by static initializers, compiled to be stored in ROM and left there.

In C++, we can do the same, but we tend not to. In well designed C++ code, most data is encapsulated in objects. Objects belong to classes and most classes have constructors. The natural OO equivalent to const initialized data is a const object. A const static object that has a constructor must be stored in RAM for its constructor to initialize it. So while in C a const static object occupies cheap and plentiful ROM, its natural heir in C++ occupies expensive and scarce RAM. Initialization is performed by start-up code that calls static constructors with parameters specified in declarations. This start-up code occupies more ROM than the static initializer would have.

So if a system includes a lot of data that can be kept in ROM, special attention to class design is needed to ensure that the relevant objects are ROMable. For an object to be ROMable, it must be capable of initialization by a static initializer like a C struct. Although the easy way to do this is to make it a simple C struct (without member functions), it is possible to make such a class a bit more object-oriented.

The criteria for a static initializer to be allowed for a class are:

- The class must have no base classes
- It must have no constructor
- It must have no virtual functions
- It must have no private or protected members
- Any classes it contains must obey the same rules

In addition, we should also require that all member functions of a ROMable class be const. A C **struct** meets these criteria, but so does a class that has member functions.

Although this solves the ROMability problem and enhances the C **struct** with member functions, it falls far short of the OO ideal of a class that is easy to use correctly and difficult to use incorrectly. The unwary class user can, for example, declare and use a non-**const**, uninitialized instance of the class that is beyond the control of the class designer.

To let us sleep securely in our OO beds, something more is needed. That "something" is class nesting. In C++, we can declare classes within classes. We can take our dubious class that is open to misuse, and put it in the private segment of another class. We can also make the const static instances of the dubious class **private** static members of the

encapsulating class. This outer class is subject to none of the restrictions that the ROMable class is, so we can put in it a proper restricted interface to our const static data.

To illustrate this discussion, let's consider a simplified example of a hand-held electronic multi-language dictionary. To keep it simple, the translator handles translation to German or French and it has a vocabulary of two words, "yes" and "no." Obviously, these dictionaries must be held on ROM. A C solution would be something like Listing 15.

```
Listing 15
A C ROMable dictionary.


/* A C ROMable dictionary */
#include
typedef struct
{
  const char* englishWord;
  const char* foreignWord;
} DictEntry;
const static DictEntry germanDict[] =
{
  {"yes", "ja"},
  {"no", "nein"},
  {NULL, NULL}
};
const static DictEntry frenchDict[] =
{
  {"yes", "oui"},
  {"no", "non"},
  {NULL, NULL}
};
const char* FromEnglish(
  const DictEntry* dict,
  const char* english);
const char* ToEnglish(
  const DictEntry* dict,
```

```
  const char* foreign);
/* ... */


void main()
{
  puts(FromEnglish(frenchDict, "yes"));
}
```

A **Dict** is an array of **DictEntry**. A **DictEntry** is a pair of const **char*** pointers, the first to the English word, the second to the foreign word. The end of a **Dict** is marked by a **DictEntry** containing a pair of NULL strings.

To complete the design, we add a pair of functions which perform translation from and to English using a dictionary. This is a simple design. The two dictionaries and the strings to which they point reside in ROM.

Let's now consider what happens if we produce a naïve OO design in C++. Looking at Listing 15 through OO glasses, we identify a class **Dict** with two member functions **const char* Dict::fromEnglish(const char*)**, and **const char* Dict::toEnglish(const char*)**. We have a clean and simple interface. Unfortunately, Listing 16 won't compile. The static initializers for **frenchDict** and **germanDict** try to access private members of the objects.

```
Listing 16
A C++ ROMable dictionary NOT!


// NOT a ROMable dictionary in C++
#include
using namespace std;


class Dict
{
public:
  Dict();
  const char* fromEnglish(
    const char* english) const;
  const char* toEnglish(
    const char* foreign) const;
private:
  enum { DictSize = 3 };
```

```
  struct
  {
    const char* english;
    const char* foreign;
  } table[DictSize];
};
// *** Following won't compile ***
const static Dict germanDict =
{
  {
    {"yes", "ja"},
    {"no", "nein"},
    {NULL, NULL}
  }
};
// *** Following won't compile ***
const static Dict frenchDict =
{
  {
    {"yes", "oui"},
    {"no", "non"},
    {NULL, NULL}
  }
};
// ...

void main()
{
  cout << germanDict.fromEnglish("yes"); }
```

If we make these members public and eliminate the constructor as in Listing 17, the class will meet the criteria for static initializers and the code will compile, but we've broken encapsulation. Users can see the internal implementation of the class and bypass the intended access functions. Even worse, they can create their own (con-const) instances of **Dict** whose internal state is outside our control.

```
Listing 17
A C++ ROMable corruptable dictionary.

// A ROMable dictionary in C++,
// but with poor encapsulation
#include
using namespace std;

class Dict
{
public:
  const char* fromEnglish(
    const char* english) const;
  const char* toEnglish(
    const char* foreign) const;
// PLEASE don't access anything in the class
// below this comment.
// PLEASE don't create your own instances
// of this class.
  enum { DictSize = 3 };
  struct
  {
    const char* english;
    const char* foreign;
  } table[DictSize];
```

```
};
const static Dict germanDict =
{
  {
    {"yes", "ja"},
    {"no", "nein"},
    {NULL, NULL}
  }
};
const static Dict frenchDict =
{
  {
    {"yes", "oui"},
    {"no", "non"},
    {NULL, NULL}
  }
};
// ...


void main()
{
  cout << germanDict.fromEnglish("yes"); }
```

Now, let's do it right. In Listing 18, the class **Dict** in Listing 17 becomes **DictTable**, which is nested privately within the new class **Dict**. Class **Dict** also contains, as a static member, an array of **DictTables**, which we can initialize statically. The function **main()** shows use of this class **Dict**, which has a clean interface.

```

Listing 18
A clean C++ ROMable dictionary.


#include
using namespace std;
class Dict
```

```
{
public:
  typedef enum
  {
    german,
    french
  } Language;
  Dict(Language lang);
  const char* fromEnglish(
    const char* english) const;
  const char* toEnglish(
    const char* foreign) const;
private:
  class DictTable
  {
  public:
    const char* fromEnglish(
      const char* english) const;
    const char* toEnglish(
      const char* foreign) const;
    enum { DictSize = 3 };
    struct
    {
      const char* english;
      const char* foreign;
    } table[DictSize];
  };
  const static DictTable
    DictTables[];
  Language myLanguage;
};
```

```
const Dict::DictTable Dict::DictTables[]=
{
  {
        {
      {"yes", "ja"},
      {"no", "nein"},
      {NULL, NULL}
        }
  },
  {
        {
      {"yes", "oui"},
      {"no", "non"},
      {NULL, NULL}
        }
  }
};
// ...
void main()
{
  Dict germanDict (Dict::german);
  cout << germanDict.fromEnglish("yes"); }
```

So to make the best use of OO design for data on ROM, special class design is needed, which is quite unlike the casual approach typical of C.

EC++

As we've seen, some C++ features have costs that may be undesirable in embedded systems. Could a subset of C++ be more cost-effective in embedded systems? Which subset? Programmers who use ad hoc subsets are frustrated in importing libraries, an ironic consequence of adopting a language which offers such promise for libraries. Wouldn't a widely recognised subset be better? Class library vendors could supply products written in the subset. Compiler vendors could enforce it and take advantage in code generation of the known absence of troublesome features. This was the motivation behind Embedded C++ (EC++), a de facto standard subset of C++. Embedded C++ is the same as C++ except that it prohibits multiple inheritance, templates, exceptions, namespaces, and run-time type identification. Because it's a strict subset of C++ with no extensions, it can be compiled with an existing C++ compiler.

EC++ shouldn't be an automatic choice for embedded systems programming. While most of its restrictions are easy to

accept, forswearing templates and the Standard C++ Library shouldn't be done lightly. There is no reason why the full language can't be used in embedded systems. But if you do decide to use a subset, EC++ would be better supported than an ad hoc subset.

For more information about EC++, see P.J Plauger's articles on the subject ("Embedded C++: An Overview," *ESP*, December 1997, p. 40; and "Embedded C++," *C/C++ User's Journal*, February 1997, p. 35).

Class Libraries and Embedded Systems

A major benefit of using C++ is the availability of class libraries and the productivity gains they promise. OO design makes class libraries much easier to use (and harder to misuse) then their procedural predecessors. But a suspicion exists that class libraries are large and of little use, and that while this may be tolerable on a desktop PC, class libraries are generally unsuited to embedded systems.

This is a misplaced generalization. If a judgement is made on the basis of so-called application framework libraries, such as Microsoft Foundation Classes (MFC), it is easy to see how this opinion is formed. Application frameworks are designed to do as much as possible of an application, allowing the programmer to concentrate on the specifics of his or her project. This is usually done by inheriting from classes in the library to produce specialized behavior. Most of the functionality of the classes in the framework are not the concern of the programmer using them.

While this type of class library is well suited to its intended use, small code size isn't its best feature, and its narrow focus makes it essentially useless outside its target application area. An embedded system needs a more general purpose class library that can help with a variety of programming problems. (Indeed, the same could be said of any system whose implementation isn't dominated by the needs addressed by a framework.) The characteristics of a class library suitable for embedded systems are as follows.

It should include classes such as strings and container classes like lists, vectors, hash tables, maps and sets, and more. Such a class library can dramatically shrink and improve code that in C manipulates **char***s and arrays.

Classes in the library should be independent. This implies limited use of inheritance. This characteristic is in contrast with a framework in which most classes are in an inheritance hierarchy. Independence allows the programmer to treat the library like an a la carte menu, while a framework is more like a dietary regime.

Several general purpose class libraries exist, though none has been used as much as MFC. They include the draft Standard C++ Library, gnu's libg++, and Rogue Wave Tools.h++. None are tailored to embedded systems, but all are better models of a class library for embedded systems than a framework library. Class libraries specifically designed for embedded systems are starting to emerge, most notably a revision of the Standard C++ Library for embedded systems.

Compilers and Tools for Embedded Systems

Most C cross compiler vendors for 32-bit targets now offer C++ compilers as well. The older compilers among them are based on AT&T's cfront. Cfront was the original C++ compiler and it produces C code output, which is then fed to a C compiler. Naturally, cfront was an easy option for established C compiler vendors who were looking for a C++ offering. But cfront has a poor reputation in terms of efficiency and optimizations, and it is understandable that a compiler whose mission it was to prototype the language should suffer from these limitations. Unfortunately for the acceptance of C++, opinions have been formed on the basis of experience with this compiler, which compared badly with mature, optimizing C compilers.

The Free Software Foundation's g++, which is well established as a native compiler that generates excellent machine code on several platforms, has been used successfully as a cross compiler.

Several compiler vendors now offer improved C++ compilers in a package with source-level debugging and other necessities of modern embedded systems development. Unfortunately, we are not yet at the stage where a C++ cross compiler from an established vendor can be assumed to produce good code quality and be supported by a good robust tool set, so it is necessary to be cautious about tool investments.

Reality Check

Having minutely examining the costs of C++ features, it is only fair to see how C stands up to the same degree of scrutiny. Consider the C code in Listing 19. One line contains a floating-point value, which will pull in parts of the floating-point library and have a disproportionate effect on code size, if floating point isn't needed. The next line will have a similar effect, if printf had been avoided elsewhere. The next line calls **strlen(s)** strlen(s) times, rather than once, which has a serious impact on run time.

```
Listing 19
C reality check.


/* Reality check - */
/* some things to avoid in C */
#include
#include
void main()
{
  char s[] = "Hello world";
  unsigned i;
  int var =1.0;
  printf(s);
  for (i=0; i
```

But who would argue that these abuses are reasons not to use C in embedded systems? Similarly, it is wrong to brand C++ as unsuitable for embedded systems because it can be misused.

Bigger than a bread box?

There is no need for a system implemented in C++ to be larger than a C implementation. Most C++ features have no impact on code size or on speed. Some C++ features have a minimal impact in these areas, and they have been discussed out of proportion to their significance. Using C++ effectively in embedded systems requires that you 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 gain great benefits from using C++, while instinctively avoiding the pitfalls that intimidate the novice.

Dominic Herity is a senior software engineer with Silicon and Software Systems, an engineering design services company located in Dublin, Ireland. He has over 10 years experience developing a wide variety of embedded systems. He welcomes your comments at dominic@s3dub.ie.

Return to Embedded.com

Send comments to: Webmaster