# Why are you Still Using C?

According to a poll on Embedded.com, 68% of the respondents are using C for developing their embedded software. Why do embedded developers choose C over C++? Sure, there are some practical reasons to avoid C++, such as the availability of tools for your embedded processor. But another possibility is that embedded programmers do not know the advantages that an Object Oriented programming language can bring them. They may not know what the tradeoffs are when choosing to use C++ over C. This article describes some of the key reasons to use the OO features of C++ in your embedded applications and how to evaluate the cost tradeoffs.

## A True Story

I was involved in the development of a second generation product. An existing system had become obsolete. It was a communications system, written in C. The system was part of a very successful product. It was built on traditional circuit switched technology. Because of advances in network technology and the need for additional features that the network technology could facilitate, the company decided they needed to advance their architecture to eliminate the circuit switched implementation and replace it with an IP based switching implementation.

The existing product was in many ways the spec. The systems engineer responsible for defining how the system had to behave was one of the lead engineers for the original product build some 15 years ago. As you might guess, the first cut of the spec was: “The system must do everything the last system does plus…”

The product’s code had evolved over these years to become unmaintainable. What was really needed was a system that did what the old one did except that used up to date technology and had a few more features. Why did the company feel that they had to re-engineer the product? Special cases, changing requirements, and hardware changes turned the C code into a mass of conditional logic -- error prone and difficult to understand.

What if the system had been designed and implemented to isolate the features of the product from the hardware implementation? If the core features could be made and kept independent of the hardware, much of this code base would still be usable. The application had a job to do; switch calls in this communication system. Many of the rules had nothing to do with being circuit switched, but they we implemented with knowledge of the underlying hardware. What if the separation between application rules (i.e. when




james@wingman-sw.com

is it OK to connect a call) was kept separate from how to connect the calls (i.e. close relays K81/82, K97/98, open relay K24/25).

It's possible to keep this separation in C, but it is much easier in an OO language like C++.

## C++ Is C

When Bjarne Stroustrup developed C++ he designed it to continue to be useable as a low-level programming language[STROU]. The zero-overhead rule states that "What you don't use, you don't pay for". So that means that if you got your C to compile under C++ and did not use any of the OO features of C++, it would not cost you anything except the cost of the new compiler. Hey, I'm not going to argue for you to use C++ as C. C++ does not have what Stroustrup calls "distributed fat". The lack of distributed fat means that as an engineer, as I choose to use some feature of C++ I can know the cost of using that feature. If I don't use it, I won't pay for it.

There are very powerful features of C++ that can be used at a known cost and benefit. If I choose to use some C++ feature, I can make a cost benefit tradeoff. You're an engineer, you're paid to make tradeoffs. I'd urge you to make the tradeoff based on fact rather than rumor.

## C++ Has a Bad Reputation

We've all heard the stories of the C++ program that runs horribly slow, uses tons of memory, or leaks memory like a sieve. C++ is not the problem; it's the use of C++ that is the problem. C++ is a big, complex and powerful language. With that power comes the responsibility to wield the power safely and effectively. C++ does not cause bad programs, bad programming does.

## The Problem to Address

The problem in my true story was that the dependencies were not managed. Separation between functional behavior and hardware implementation were not made. Granted, the designers and evolvers of the system did not have the tools to effectively manage the changes that pummeled that system into its formless state.

## Object Oriented Features

C++ is a big language. It provides many features that may have a place in your embedded software: encapsulation, inheritance, polymorphism, exceptions, templates, and the




james@wingman-sw.com

standard library. These features have costs. I am only going to limit my discourse to the features of C++ that support Object Oriented design and programming.

## Encapsulation

Encapsulation is the packaging of hidden data and the functions that operate on that data. You can practice encapsulation in C, but few people do. Encapsulation in C can be achieved by using file scope to hide data structures and to allow access to those data structures by functions defined within the same file scope. The language does little to encourage this behavior.

With C++ encapsulation is part of the language. The basic building block in C++ is the class. A class is a user defined data type and the allowed operations for that data type. C ++ supports the encapsulation through the definition of classes. So, if a class defines a new data type, what is an object? An object is an instance of a class.

Think of classes as data structures, with hidden data and functions that operate on that hidden data. Encapsulation in C++ allows the software engineer to build self-contained objects. C++ does not assure this, but the features of the language support it. C++ and the Single Responsibility Principle[MARTIN] give you the tools and the guidance to create cohesive objects.

Encapsulation has a cost. It is not generally regarded as high, but it does have a cost. Each call to a member function has a hidden additional parameter, the pointer to the encapsulated data structure (a.k.a. the 'this' pointer). You will pay this cost for member function calls, except in-line function calls -- defined later. Keep in mind that you are probably already paying this cost when you pass the pointer to a data structure around.

There is a memory-space and execution-time cost to having a decent design. A free-for-all type of design, where any function can touch globally accessible data, might be able to be made to have a smaller footprint. If your footprint is your biggest concern, maybe a good design is too expensive. You're the engineer, you make the tradeoff. You are buying into a huge maintenance nightmare, so please don't get hooked on this approach if you don't have to.

## Object Oriented Design

The hype around Object Oriented Design and Programming lists modeling the real world as the big advantage to software designers. The hype goes something like this: if objects represent real world entities, software would be intuitive to understand, anyone could develop software and life would be wonderful. Frankly, this is mostly boloney. Some of

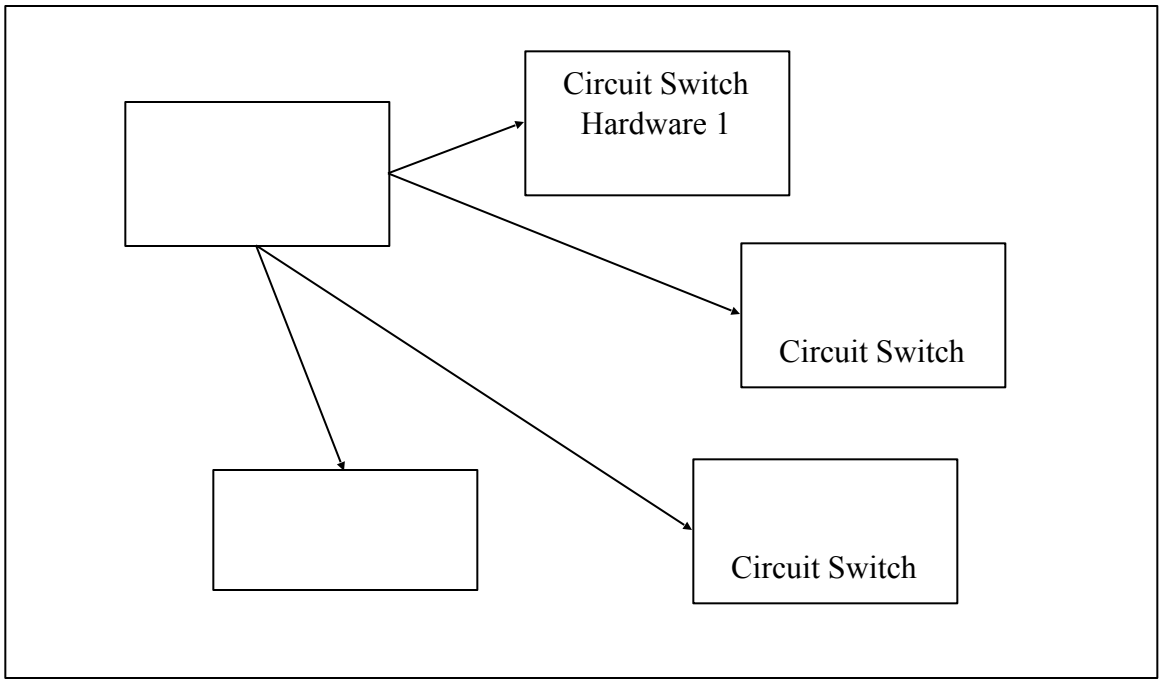



james@wingman-sw.com

the objects in your designs will have real world analogs and that will be helpful. But, that is not the biggest benefit for a software engineer.

OO is an engineering tool. It allows the software engineer to create a loosely coupled software system. The idea of reducing coupling has been talked about for decades in the computer science field. But what does it actually mean? Maybe an analogy would help. As a driver I am decoupled from the car that I control. I can sit down in just about any car in the world, turn it on, buckle my seat belt put it in gear and go. It's possible to for me to drive any car in the world because the car has, more or less, a standard interface. When I want to make the car go faster I don't have to concern myself with the kind of fuel the car uses. I don't say to the carburetor (or fuel injector) burn more gasoline. I operate the car's accelerate function by pressing on the accelerator.

I can build software the same way. I can create interfaces to parts of the system as a mechanism to decouple one part of the system from another. Then those parts can be varied independently. Remember my true story above? The designers of the communications system only had procedural programming at their disposal. As the system evolved, the hardware changed. The software had to stay backwards compatible so the design used conditional logic to deal with the variation. Procedural programming results in a dependency structure where the core logic depends on implementation details (figure-1). We've all seen C code like this where every function has a mass of conditional logic to deal with all the variation.







james@wingman-sw.com

Object oriented programming provides an alternative. The core logic is our intellectual property. We can preserve our investment in that property by isolating it from things that are likely to change. In this case it is the hardware that is changing, but it could be other forms of change as well.
What if the designers of the original communications system had employed interfaces to decouple the system? They may have defined an interface to the call switch that was not hardware implementation dependent (figure-2). The core communications system logic (complexity of that subsystem is not represented above) would use the CallSwitch interface to connect the parties that need to be connected. The original hardware control software is implemented to that interface. Later the IP Based Switch Implementation,

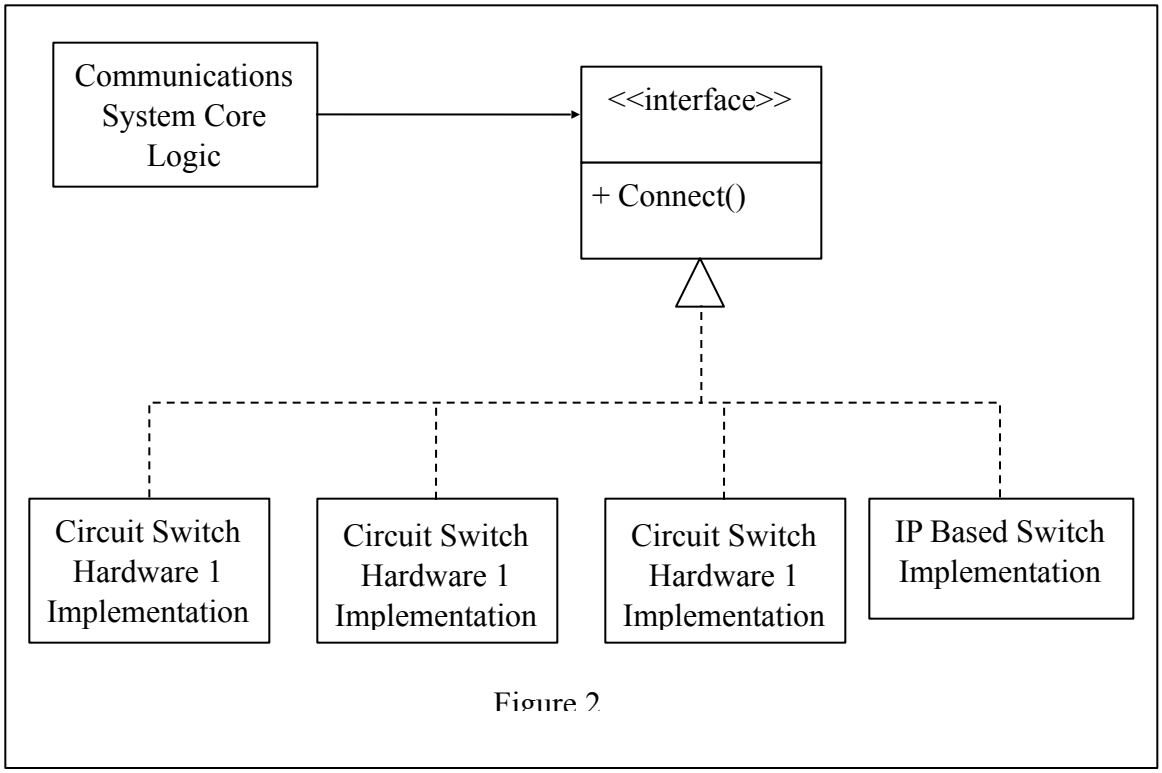



Figure 2

and all the versions in between, could also be built to the same interface. The interface is likely to evolved, but the core logic investment could be better protected by not intertwining it with the specific hardware implementation.

Notice the direction of the dependencies. Both the core logic and the specific implementations depend on the CallSwitch interface class. We are using the dependency inversion principle. [MARTIN] Instead of having high level logic depending on details, both the details and high level logic depend on the interfaces.




james@wingman-sw.com

This design also adheres to the Open-Closed Principle[MEYER]. This principle states that *Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*.[MARTIN] The Communication System Core Logic can be extended but does not need to be modified for different kinds of CallSwitches. We can add the IP Based Switch Implementation without changing the Communication System Core Logic.

## How is Variation Dealt with Now?

If you develop embedded software it's no news to you that hardware is ever changing. Parts become obsolete. Cost reduction efforts result in new designs. Customers want new features that result in a revision of the hardware design.

What do you do now? Well, if 68% of you are still using C, you probably have a mass of conditional logic (either at run time or compile time) that deals with the variations. You have big switch-case statements, if-else chains, and nested ifs. It is possible in C to design to interfaces, but again the language gives you no real help.

Hardware will change. User requirements will change. Features will be added. Get used to it. Don't complain. Protect your self from that inevitability. Get good at dealing with the change.

## Object Oriented Programming in C++

Remember, C++ is C and more. C++ supports object oriented programming. A member function can be declared as virtual in C++. This means that the binding to the function is dynamic. It is determined at run-time.

An interface class in C++ is a class that has only pure virtual member functions. An interface class defines the member function signatures of the interface methods. A class implements that interface by inheriting the interface class, and then implements each of the pure virtual member functions. For example:

```
//CallSwitch.h
class CallSwitch
{
     public:
          virtual void Connect(User* a, User* b) = 0;
};

// CircuitSwitchImplementation.h
class CircuitSwitchImplementation : public CallSwitch
{
```




james@wingman-sw.com

```
        public:
             virtual void Connect(User* a, User* b);
             //implemented in the Cpp file
};
```

In the interface, all methods are shown to be virtual and not implemented. The "= 0" notation tells the compiler that there is no implementation for Connect in the CallSwitch class. It is an interface specification.

A group of classes that inherit the same interface class and follow the same interaction protocol are substitutable. The Communications System Core Logic can use any class that follows the CallSwitch protocol.

How does this work? The Communication System Core Logic accesses the Connect method through a pointer to an object through a pointer to a CallSwitch. The pointer to a CallSwitch can point to any class that inherits from CallSwitch. That is the OO decoupling mechanism that is built into C++. This simple and powerful feature is called polymorphism. Calling a function through a virtual function is called a polymorphic dispatch.

## Cost of Polymorphism

When you call a function in C, the compiler and linker determine the address of the function to be called. This is a direct call. When you call a virtual function in C++ the linker does not know the address. Addresses are bound at runtime. The address is stored in a table called the virtual table. Each object (an instance of a class) has an associated virtual table. There is an entry for each of the class's virtual functions in the virtual table. Calling a virtual function means getting the address of the virtual table, then using an offset to access the called function's virtual table entry and calling that function. This is an indirect call.

When you write this code:

```
CallSwitch sw = new CircuitSwitchImplementation();
sw->Connect(u1, u2);
```

The compiler essentially writes this code under the hood:

```
(sw->vtable[CONNECT_OFFSET])(sw, u1, u2);
```

To summarize the costs of using virtual functions:




james@wingman-sw.com

- Execution time overhead for the polymorphic call
- A virtual table is needed for each class that has virtual methods
- There is a slot in the virtual table for each virtual function
- A virtual function call takes more bytes than a direct call

You get to choose when to incur these costs. If you do not specify a member function as virtual, then the linker can figure out what method to call and there are no polymorphic dispatch costs. If a class has no virtual functions then there is no virtual table.

Earlier we discussed the alternative to using OO to deal with variations. That alternative is to have conditional logic in the form of if-else chains and switch-case statements. These constructs also have a cost in memory usage and execution time. Often the conditional logic is repeated in many places.

## Avoiding Shooting Yourself in the Foot with C++

C++ suffers from a bad reputation because it is easy to do some very costly operations. For example, objects can not only be passed by pointer, they can also be passed by value. So if you happen to forget declare a parameter type as a pointer, the compiler won't complain. The compiler gladly copies the object onto the stack and calls the function. For a class with a lot of member variable storage, this can be costly in execution time and In stack space.

It turns out that the savvy C++ designer can prevent this. In C++ to be able to pass a parameter by value the compiler needs what is called a copy constructor. The copy constructor's roll is to initialize a new object from an existing object. If your application performs some operation that needs a copy constructor, the compiler will write one for you and call it unless you provide your own. It's very presumptuous of the compiler to write this function for you, so let's not let it. You can prevent the compiler from writing the copy constructor by declaring the copy constructor in the private area of your class, and then not provide an implementation. It is private so that no one can call it (except the class itself), and it is not implemented to make sure it uses no space and cannot even be called by the containing class.

The same problem exists for returning an object by value. The compiler will then need a copy constructor and an assignment operator. What will happen if you accidentally forget the pointer designation for a return value? You guessed it, the compiler writes one for you. You can use the same technique above to declare a private undefined assignment operator and prevent the whole problem.

Every time I create a class in C++ I always start with a class that looks like this:




james@wingman-sw.com

```
class ClassName
{
     public:
          ClassName(); //the default constructor

     private:
          //Degenerate copy and assignment
          ClassName(const Classname&);
          ClassName& operator=(const ClassName&)
};
```

Following this simple technique prevents the compiler from writing this code for you and prevents calling the code as well.

## Other Tradeoffs

You have many other choices in C++. To name a few more: exceptions, in-line functions, template functions and template classes.

Exceptions have both a memory cost and an execution cost. This varies by compiler but some measurements put the memory overhead between 3-13% with an execution overhead between 4-6%.[LIPPMAN] Exceptions can be turned off (zero-overhead principle).

In-line functions might just seem like the classic speed versus space tradeoff. With this feature you may in some instances be able to have your cake and eat it too. In OO programming there is a tendency to make member functions smaller and more focused. This means more parameters are passed around, more calls done, using both space and time. Private member functions often come into existence when a function gets too big, and helper functions are extracted to make the code more understandable. Many of these small methods end up being private and only called from one place. In this case using in-line is both smaller and faster than a non-in-line call.

Other uses of in-line are basically a storage space versus execution time tradeoff. You're the engineer, you decide.

Templates suffer from a bad reputation for space usage. A template is just a way for you to get the compiler to write code for you that you otherwise would write for yourself. Templates are not inherently evil. They are inherently useful. Using certain template libraries may cause code bloat, but it's not the fault of templates.




james@wingman-sw.com

There are other gotchas in C++ worth looking at. I'd recommend that the serious embedded C++ programmer look at Scott Meyer's books: *Effective C++* and *More Effective C++*.

## Conclusions

Embedded programmers could build much better systems using C++ rather than C. There are extreme cases where memory and execution constraints may prevent using these techniques. Is this the exception rather than the rule? I think so.

Encapsulation alone can lead to better, more resilient designs. Huge improvements in flexibility and decoupling are possible when you add to that the use of interfaces and polymorphism. The costs are not huge. They are measurable and deterministic. Know what the compiler is doing for you. Learn how to make the tradeoffs.

[STROU] Stroustrup, Bjarne. *The Design and Evolution of C++*, 1994, AT&T Bell Labs, P.121

[MARTIN] Martin, Robert C.. *Agile Software Development; Principles, Patterns, and Practices*. Pearson Education, Inc. 2003, P.95

[MARTIN]ibid, P.127

[LIPPMAN]Lippman, Stanley B. *Inside the C++ Object Model*. Addison-Wesley Publishing. 1996




james@wingman-sw.com