# Using C++ to Create Better Device Software

Günter Obiltschnig
Applied Informatics Software Engineering GmbH
St. Peter 33
9184 St. Jakob im Rosental
Austria
guenter.obiltschnig@appinf.com

**Abstract.** Software for embedded systems is becoming ever more complex. This requires a radical re-thinking in the embedded software development community, comparable to the Assembler-to-C transition years ago. Object-oriented software development is a proven solution for taming software complexity. While, at least to a certain degree, object-oriented principles can also be applied to C programming, a programming language with inherent support for object-oriented programming brings many advantages. But support for object-oriented programming is just one feature of C++. C++ has many features that make writing reliable and robust code easier than in C. This paper introduces two ANSI C++ techniques that can be used to write more reliable and robust code for embedded systems. These are the RAII (Resource Acquisition Is Initialization) idiom for resource management and exceptions for error handling. The paper is targeted at developers having a basic knowledge of C++, and who might consider moving to C++ for their next project.

## 1 Introduction

Software for embedded systems is becoming ever more complex. This requires a radical re-thinking in the embedded software development community, comparable to the Assembler-to-C transition years ago. Even in many smaller embedded system development projects, the days of the one-engineer “software team” are counted, if not all over. Object-oriented software development is a proven solution for taming software complexity. While, at least to a certain degree, object-oriented principles can also be applied to C programming, a programming language with inherent support for object-oriented programming brings many advantages. This is comparable to the fact that structured programming can somehow be done in Assembly language. However, doing structured programming in a high-level language like C is much more efficient. But support for object-oriented programming is just one feature of C++. C++ has many features that make writing reliable and robust code easier than in C.

C++ is thus slowly but steadily replacing C as the programming language of choice for embedded or device software development. While C++ (or at least some of its features) has often (falsely[1]) been considered prohibitively resource intensive for embedded devices, especially with today's available powerful embedded hardware and mature compiler and build system technology, this is certainly no longer the case. Under certain circumstances the code generated by a state-of-the-art C++ compiler may even be more efficient than the code produced by a C compiler. What makes C++ unique among programming languages is that it

[1] There seem to be two causes for this: a missing understanding about the internals of C++, and bad experiences with early compilers, which really produced inefficient (or even wrong) code in some cases.

covers the full range from low-level programming (interrupt service routines, direct hardware access) to high-level programming (classes, generic programming) in one language, without the need to connect both ends via awkward artificial interfacing mechanisms. With the software for embedded systems becoming ever more sophisticated and complex, the high level abstractions of C++ are certainly needed in addition to its suitability for low-level programming.

This paper introduces two ANSI C++ [1] techniques that can be used to write more reliable and robust code for embedded systems. A special focus will be on the performance implications of certain language features, by taking a detailed look at what happens "behind the scenes". The topics that will be covered include:

- **The RAII (Resource Acquisition Is Initialization) paradigm for automatic resource management.** RAII is a powerful paradigm that frees the developer from certain complexities of manual resource management. Prominent examples of RAII are scoped locks and "smart" pointers. A scoped lock is a class that acquires a mutex at the beginning of a block and automatically releases it when the block is left, no matter how. Smart pointers help programmers manage the lifetime of objects on the heap, by automatically deleting an object and releasing its memory when it is no longer referenced.
- **C++ exceptions for better error handling.** Proper error handling is hard to do in C. C++ exceptions provide a powerful facility for reliable error handling at the cost of some minimal overhead. However, unless the code using exceptions is exception-safe, exceptions may be the cause of resource leaks, which are fatal for embedded systems. The second part of this paper discusses the three levels of exception-safety in C++ code, and how to achieve them.

## 2 The C++ Object Lifecycle

In C++, every class has one or more constructors, as well as exactly one destructor. The C++ compiler automatically takes care that for every object that is constructed, no matter whether on the heap, on the stack, or statically, a constructor is called. Furthermore it ensures that for every object that is destroyed, the object's destructor is called. Even a class (or struct) for which the programmer has not implemented a constructor or destructor will have one. In this case, the compiler automatically implements the missing constructor or destructor[2]. However, especially for classes containing pointer member variables, the constructor and destructor generated by the compiler might not do the right thing. Therefore, it's a good practice to always provide a constructor and destructor, even if their implementation might be empty.

There are actually two constructors that are required for every class. The first one is the default constructor, a constructor taking no arguments. The second one is the copy constructor, which is used for constructing an object by copying another argument.

C++ also supports object assignment. For this to work, a class must implement the assignment operator. Again, if a class does not implement an assignment operator, the C++ compiler will provide one (which, however, might not do the right thing, especially if the class has pointer member variables).

In C++, one can actually distinguish two kinds of classes. Classes that support value semantics, and classes that do not. Value semantics means that a class supports a default constructor, a copy constructor and an assignment operator, and optionally some or all relational operators (==, !=, <, <=, >, >=). These classes can be used in the same way as the build-in data types of C++. An example for such a class is the string class in the standard

[2] In the trivial case, the constructor might be empty and not do anything, and no additional code (and thus no overhead) will result.

library (`std::string`). Classes that do not support value semantics do not have a copy constructor and an assignment operator. In such a case, to prevent the compiler from generating them, they are declared as private in the class definition.

A class supporting value semantics basically looks like this:

```
class Value
{
public:
    Value();
        // default constructor

    Value(const Value& value);
        // copy constructor

    ~Value();
        // destructor

    Value& operator = (const Value& value);
        // assignment operator

    bool operator == (const Value& value);
    bool operator != (const Value& value);

    // ...

private:
    // ...
};
```

A class not supporting value semantics in contrast looks like this:

```
class NonValue
{
public:
    NonValue();
        // default constructor

    NonValue(int someArg, bool someOtherArg);
        // construct from some other values/objects

    ~NonValue();
        // destructor

private:
    NonValue(const NonValue&);
    NonValue& operator = (const NonValue&);

};
```

A private copy constructor and a private assignment operator are a sign to the compiler that it does not have to implement them, and, in fact, nobody is going to implement them as they are never going to be called.

Especially if the standard template library (STL) is used in a C++ project, it makes sense to add another member function (as well as a freestanding function) to each class supporting value semantics. For efficiency, many algorithms in the STL use a swap operation to exchange the values of two objects. For objects that do not support a swap operation themselves, the C++ standard library provides a default implementation:

```
// namespace std
template <typename C>
void swap(C& c1, C& c2)
{
```

```
    C temp(c1);
    c1 = c2;
    c2 = temp;
}
```

This works for every class supporting value semantics (copy construction and assignment operator), although at some cost. For many classes, a swap operation can be implemented much more efficiently, not involving the (potentially costly) construction of a temporary object.

As an example, we look at the implementation of a very primitive string class. To keep the example short, we only show the implementation of the copy constructor and the assignment operator.

```
class SimpleString
{
public:
    SimpleString();
        // creates an empty string

    SimpleString(const char* str);
        // creates a SimpleString from a C string.

    SimpleString(const SimpleString& str);
        // copies another string

    ~SimpleString();
        // destroys the string

    SimpleString& operator = (const SimpleString& other);
        // assignment operator

    std::size_t length() const;
        // returns the length of the string

private:
    char*       _pData;
    std::size_t _length;
};


SimpleString::SimpleString(const SimpleString& str):
    _pData(new char[str._length]),
    _length(str._length)
{
    memcpy(_pData, str._pData, _length);
}


SimpleString& SimpleString::operator = (const SimpleString& str)
{
    if (&str != this)
    {
        delete _pData;
        _length = str._length;
        _pData  = new char[_length];
        memcpy(_pData, str._pData, _length);
    }
    return *this;
}
```

Now, if we want to exchange the value of two `SimpleString` objects, the template instantiation of the standard swap function for `SimpleString` becomes:

```
void swap(SimpleString& s1, SimpleString& s2)
```

```
{
    SimpleString tmp(s1);
    s1 = s2;
    s2 = tmp;
}
```

This leads to the creation of a temporary string object, which all the necessary overhead like allocation of memory on the heap, etc. To make things more efficient, what we have to do is to implement a swap operation for `SimpleString`:

```
void SimpleString::swap(SimpleString& str)
{
    using std::swap;
    swap(_pData, str._pData);
    swap(_length, str._length);
}
```

We also have to provide a freestanding `swap` function for `SimpleString`:

```
void swap(SimpleString& s1, SimpleString& s2)
{
    s1.swap(s2);
}
```

However, for use with the standard library, the freestanding swap should rather be implemented as a specialization of the `std::swap` template:[3]

```
namespace std
{
    template <>
    void swap<SimpleString>(SimpleString& s1, SimpleString& s2)
    {
        s1.swap(s2);
    }
}
```

As we have seen, the implementation of the assignment operator is surprisingly complicated:

```
SimpleString& SimpleString::operator = (const SimpleString& str)
{
    if (&str != this)
    {
        delete _pData;
        _length = str._length;
        _pData  = new char[_length];
        memcpy(_pData, str._pData, _length);
    }
    return *this;
}
```

First, we have to guard against self assignment. Then we have to get rid of the old data, allocate space for the new data, and copy the data.

Using our swap implementation, we can come up with a much simpler implementation of the assignment operator:

```
SimpleString& SimpleString::operator = (const SimpleString& str)
{
    SimpleString tmp(str);
    swap(tmp);
    return *this;
```

[3] This can only be done for non-template classes, as C++ does not allow template specialization for another template. In the case of a template class, a freestanding swap function must be implemented. Argument-dependent lookup ensures that the correct swap function will be called. For more information, see [2].

```
}
```

We no longer have to guard against self assignment, and this new implementation is actually a lot more robust (exception safe) than the original one.

In our original implementation, we use the `new` operator to allocate memory for the copied data, after the old data has already been deleted. Now, if we run out of memory, the new operator will throw an exception, and we'll end up with a corrupted `SimpleString` object (the new length has already been assigned, but `_pData` still points to freed memory). Of course, we could fix this by allocating the new memory before releasing the old (this is left as an exercise for the reader), but the implementation using the swap operation does this in a much more elegant way. It is also a whole lot shorter, and as every developer knows, fewer lines of code mean fewer bugs.

Now one might argue that the new implementation using swap is not as efficient as the old one. That's not necessarily so. The only case where the new implementation is less efficient than the old one is a self assignment. Self assignment, however, is very rare. Instead, in the normal case, we got rid of a conditional branch (the test for self assignment), which will positively affect performance by avoiding a possible pipeline stall in the processor.

## 3 Resource Acquisition Is Initialization

The C++ compiler takes care that constructors and destructors of objects are always appropriately called. We can use this to our advantage, even in not so obvious ways.

Let's consider the following example. In multithreaded code, access to a shared resource has to take place inside a critical section. A critical section is a block of code guarded by a synchronization primitive (a mutex or semaphore) that ensures that at most one thread at a time can execute the block. Assuming we have a mutex[4] class with the following interface:

```
class Mutex
{
public:
    void lock();
        // acquire the mutex; wait if another thread holds the mutex

    void unlock();
        // release the mutex

    // ...
};
```

a critical section can be written as follows:

```
static Mutex mutex;
mutex.lock();
// code in critical section
// access and manipulate shared resource
mutex.unlock();
```

This code might seem fine at first. However, the code in the critical section might under certain circumstances throw an exception. In this case, the mutex will never be released, and a later deadlock is guaranteed. Or some developer might decide to optimize some code path inside the critical section and adds a `return` or `break` statement which also prevents `unlock()` from being called.

[4] A mutex has two operations. The first operation is usually called *lock* or *acquire*, and will acquire ownership of the mutex. A mutex can have at most one owner, so if there is already another thread owning the mutex, the operation will block the thread until the mutex becomes available. The second operation, *unlock* or *release*, will give up ownership of the mutex, and allow another thread waiting for the mutex to continue execution.

Against the first case, we can defend our code using a `try ... catch` block:

```
static Mutex mutex;
mutex.lock();
try
{
    // code in critical section
    mutex.unlock();
}
catch (...)
{
    mutex.unlock();
    throw;
}
```

However, this is error prone (it is easy to forget one call to `unlock()`), unelegant, and it does defend us against a `return` or `break` in the critical section.

Using our knowledge about constructors and destructors, we can come up with a far better way to implement a critical section:

```
class ScopedLock
{
public:
    ScopedLock(Mutex& mutex):
        _mutex(mutex)
    {
        _mutex.lock();
    }

    ~ScopedLock()
    {
        _mutex.unlock();
    }

private:
    Mutex& _mutex;

    ScopedLock();
    ScopedLock(const ScopedLock& lock);
    ScopedLock& operator = (const ScopedLock& lock);
};
```

We create a new class `ScopedLock` that just consists of a constructor and a destructor, and stores a reference to a `Mutex` object. In the constructor, the `Mutex` is locked, and in the destructor the `Mutex` object is unlocked.

We use the new `ScopedLock` class as follows:

```
static Mutex mutex;
{
    ScopedLock lock(mutex);
    // code in critical section
}
```

Now, no matter how we exit the block enclosing the critical section – normally, via an exception, or a `return` or `break` statement, or even a `goto`, the mutex will always be unlocked, since the compiler ensures that the destructor will be called for every automatic variable when leaving its scope.

What we have done in our `ScopedLock` class is referred to as the *Resource Acquisition Is Initialization* idiom (or RAII, for short). The acquisition of a resource (in our case, a mutex) is bound to the construction (initialization) of an object, whereas the release of the resource is bound to the destruction of the object. For an automatic variable (stored on the stack), the

C++ compiler guarantees us that the destructor is called when leaving its scope, no matter how.

Our `ScopedLock` class has no negative effect on the performance of the resulting program. The C++ compiler's ability to *inline* the code of the constructor and the destructor means that there is no actual function call to the constructor or destructor taking place in the code. Instead, the code of the constructor, or destructor, respectively, is directly placed (inlined) at the call site. In fact, the resulting code will be equivalent to the one using a `try ... catch` block, but probably more efficient, as the compiler's optimizer can do a better job.

## 4 Smart Pointers

Another area where RAII can be put to good use is memory management. Getting the management of memory on the heap right is on of the hardest things to do in both C and C++. But while C leaves the developer out in the cold in this regard, C++ can give you almost the same comfort as a garbage-collected language like Java, with the additional bonus of determinism. Consider the following implementation of a trivial "smart" pointer:

```
template <typename C>
class AutoPtr
{
public:
    AutoPtr(): _ptr(0)
    {
    }

    AutoPtr(C* ptr): _ptr(ptr)
    {
    }

    ~AutoPtr()
    {
        delete ptr;
    }

    C* operator -> ()
    {
        return _ptr;
    }

private:
    C* _ptr;

    AutoPtr(const AutoPtr&);
    AutoPtr& operator = (const AutoPtr&);
};
```

We can use this `AutoPtr` class in the following way:

```
void testAutoPtr()
{
    AutoPtr<std::string> pString(new std::string("Hello, world!"));
    std::cout << pString->length() << std::endl;
}
```

In this example, the destructor of the `AutoPtr` ensures that the `std::string` object we have allocated on the heap will be deleted when we return from the function, no matter in which way. Note that how overloading the arrow (`->`) operator makes the use of `AutoPtr` fully transparent – we can use `AutoPtr` almost like a plain pointer.

An “industrial strength” `AutoPtr` class will require a bit more work, though. We will at least have to handle assignments and copying (the hardest part), overload the dereferencing operator (`*`), and possibly implement the relational operators.

The main obstacle when implementing copy and assignment for a smart pointer is finding out who is responsible for deleting the object on the heap. “The last one switches off the light” seems like a good strategy, but how can this be implemented? We need to keep track of the number of smart pointers referencing a single object, using a reference counter. Whenever a smart pointer gets a reference to the object, the reference counter is incremented. Whenever a smart pointer releases its reference to the object (because another object is assigned, or because the smart pointer is destroyed), the reference counter is decremented. When the reference counter reaches zero, the object on the heap is deleted.

There are two strategies for implementing such a smart pointer. The first strategy is to make the reference count a part of the object to be managed by the smart pointer, as shown in Figure 1. This is easier to implement, but restricts the smart pointer to objects providing a reference counter (and member functions to manage it).

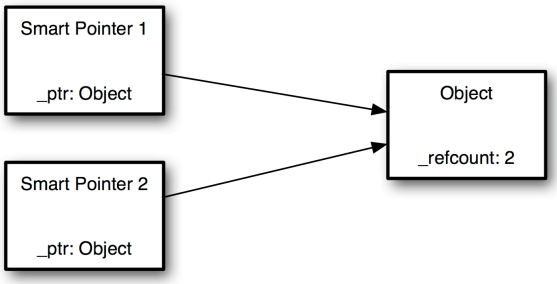


*Figure 1: Smart pointer storing the reference counter in the object.*

The second strategy is to store the reference count separately from the object, as shown in Figure 2.

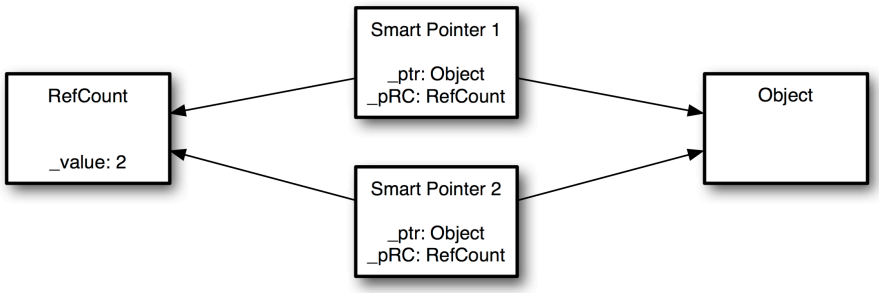


*Figure 2: Smart pointer storing the reference counter separately from the object.*

A smart pointer using a separate reference counter is commonly referred to as a shared pointer – pointers share ownership of an object, and the last one deletes it.

A trivial implementation of such a shared pointer could look like this:

```
template <typename C>
class SharedPtr
{
public:
    SharedPtr():
        _ptr(0),
        _pRC(new int(1))
    {
    }
```

```
    SharedPtr(C* ptr):
        _ptr(ptr),
        _pRC(new int(1))
    {
    }

    SharedPtr(const SharedPtr& other):
        _ptr(other._ptr),
        _pRC(other._pRC)
    {
        ++*_pRC;
    }

    ~SharedPtr()
    {
        if (--*_pRC == 0)
        {
            delete _ptr;
            delete _pRC;
        }
    }

    void swap(SharedPtr& other)
    {
        using std::swap;
        swap(_ptr, other._ptr);
        swap(_pRC, other._pRC);
    }

    SharedPtr& operator = (const SharedPtr& other)
    {
        SharedPtr tmp(other);
        swap(tmp);
        return *this;
    }

    SharedPtr& operator = (C* ptr)
    {
        SharedPtr tmp(ptr);
        swap(tmp);
        return *this;
    }

    C* operator -> ()
    {
        return _ptr;
    }

    // ...

private:
    C*   _ptr;
    int* _pRC;
};
```

This example implementation shows how a shared pointer basically works. A real world implementation would be more complicated. For example, manipulation of the reference counter must be done in a thread safe way, and assignment of different, but related instantiations of `SharedPtr` must be handled (e.g., assignment of `SharedPtr<D>` to `SharedPtr<B>`, where `D` is a subclass of `B`). However, only few programmers will probably ever have to implement such a `SharedPtr` themselves, as there are already a lot of high-quality implementations available in open source libraries, for example POCO [3] and Boost

[4]. Also, the next revision of the C++ standard will include a shared pointer as part of the C++ standard library.

## 5 Exceptions

Exceptions are C++'s mechanism for error handling. In C, error handling is done in one of two ways. Either a global (or thread-local) variable is used for storing an error number or error flag, or a function communicates an error to the caller via its return value (or an output parameter). A combination of both is also possible, as it is done by the C standard library.

Error handling in C is a constant pain. Basically, the result of every function call has to be checked, leading to endless if/else chains[5]. Cleaning up the state of data structures if an error occurs in the middle of an operation is hard to get right. To make things even more complicated, testing all possible error scenarios and the resulting code paths is almost impossible.

All of this can be avoided in C++, using exceptions for error handling. Error handling code (in the form of `try ... catch` blocks) can be concentrated on a few places in a program, and with clever programming (RAII), the compiler takes care of cleaning up the mess after an error occurs.

There are, however, some pitfalls when using exceptions. First, an exception in the wrong place can leave objects or data structures in an invalid state, causing later trouble. Second, exceptions have a certain overhead (both memory and run-time) that has to be taken into account.

The first issue can be dealt with by writing exception safe code. We can distinguish three levels of exception safety that an operation can guarantee [5]:

- **the basic guarantee**: an operation always leaves objects in a valid state and does not leak resources.
- **the strong guarantee**: in addition to the basic guarantee, the operation either succeeds or leaves all objects unchanged.
- **the nothrow guarantee**: the operation never throws an exception.

Using the RAII idiom, the basic guarantee is relatively easy to implement, and sufficient for most cases. The strong guarantee can be quite hard to implement, but nevertheless may be required for certain operations (for example, the `push_back` member function of a `std::vector` either succeeds, or leaves the vector unchanged). Examples where the nothrow guarantee is used are simple swap implementations, or the `pop_back` operation of `std::vector`. Also, while theoretically possible, a destructor of an object should never throw an exception.[6]

The use of exceptions results in a certain overhead, both in memory and run-time. This is, however, a minimal price to pay for the many advantages that exceptions can deliver [6]. Nevertheless, exceptions cannot be used under certain circumstances. These are interrupt service routines, and real-time code[7].

---

[5] In fact, error handling in C is such a pain, that in certain cases a `goto` statement can actually make the code clearer – this is the only acceptable use for the `goto` statement in C.

[6] Destructors are called during stack unwinding when an exception is being handled. Throwing an exception, while another exception is currently being handled will result in the program being terminated.

[7] It may be possible to use exceptions in real-time code, but this requires a detailed knowledge of the exception implementation of the used C++ compiler (which may not be available), as well as extensive time analysis of all possible code paths.

## 6 Conclusion

This paper has shown that C++ offers far more than object-oriented programming. Two techniques have been shown that make writing robust software in C++ a lot easier than in C. RAII and exceptions are two techniques that perfectly complement each other.

## References


[1] ISO/IEC 14882 International Standard: Programming Languages – C++, Second Edition, ISO/IEC, 2003

[2] Scott Meyers, Effective C++, Third Edition, Addison Wesley, 2005

[3] POCO – The C++ Portable Components http://poco.appinf.com

[4] Boost C++ Libraries http://www.boost.org

[5] David Abrahams, Exception-Safety in Generic Components, http://www.boost.org/more/generic_exception_safety.html

[6] Technical Report on C++ Performance (ISO/IEC TR 18015:2006) http://open-std.org/jtc1/sc22/wg21/docs/TR18015.pdf