Five Popular Myths about C++

Bjarne Stroustrup

Morgan Stanley, Columbia University, Texas A&M University

1. Introduction

Here, I will explore, and debunk, five popular myths about C++:

1. "To understand C++, you must first learn C"
2. "C++ is an Object-Oriented Language"
3. "For reliable software, you need Garbage Collection"
4. "For efficiency, you must write low-level code"
5. "C++ is for large, complicated, programs only"

If you believe in any of these myths, or have colleagues who perpetuate them, this short article is for you. Several of these myths have been true for someone, for some task, at some time. However, with today's C++, using widely available up-to date ISO C++ 2011 compilers, and tools, they are mere myths.

I deem these myths "popular" because I hear them often. Occasionally, they are supported by reasons, but more often they are simply stated as obvious, as if needing no support. Sometimes, they are used to dismiss C++ from consideration for some use.

Each myth requires a long paper or even a book to completely debunk, but my aim here is simply to raise the issues and to *briefly* state my reasons.

2. Myth 1: "To understand C++, you must first learn C"

No. Learning basic programming using C++ is far easier than with C.

C is almost a subset of C++, but it is not the best subset to learn first because C lacks the notational support, the type safety, and the easier-to-use standard library offered by C++ to simplify simple tasks. Consider a trivial function to compose an email address:

```
string compose(const string& name, const string& domain)
{
        return name+'@'+domain;
}
```

It can be used like this

```
string addr = compose("gre","research.att.com");
```

Naturally, in a real program, not all arguments will be literal strings.

The C version requires explicit manipulation of characters and explicit memory management:

```
char* compose(const char* name, const char* domain)
{
        char* res = malloc(strlen(name)+strlen(domain)+2); // space for strings, '@', and 0
        char* p = strcpy(res,name);
        p += strlen(name);
        *p = '@';
        strcpy(p+1,domain);
        return res;
}
```

It can be used like this

```
char* addr = compose("gre","research.att.com");
// …
free(addr);        // release memory when done
```

Which version would you rather teach? Which version is easier to use? Did I really get the C version right? Are you sure? Why?

Finally, which **compose()** is likely to be the most efficient? Yes, the C++ version, because it does not have to count the argument characters and does not use the free store (dynamic memory) for short argument strings.

2.1 Learning C++

This is not an odd isolated example. I consider it typical. So why do so many teachers insist on the "C first" approach?

- Because that's what they have done for ages.
- Because that's what the curriculum requires.
- Because that's the way the teachers learned it in their youth.
- Because C is smaller than C++ it is assumed to be simpler to use.
- Because the students have to learn C (or the C subset of C++) sooner or later anyway.

However, C is not the easiest or most useful subset of C++ to learn first. Furthermore, once you know a reasonable amount of C++, the C subset is easily learned. Learning C before C++ implies suffering errors that are easily avoided in C++ and learning techniques for mitigating them.

For a modern approach to teaching C++, see my *Programming: Principles and Practice Using C++* [13]. It even has a chapter at the end showing how to use C. It has been used, reasonably successfully, with tens of thousands of beginning students in several universities. Its second edition uses C++11 and C++14 facilities to ease learning.

With C++11 [11-12], C++ has become more approachable for novices. For example, here is standard-library **vector** initialized with a sequence of elements:

```
vector<int> v = {1,2,3,5,8,13};
```

In C++98, we could only initialize arrays with lists. In C++11, we can define a constructor to accept a **{}** initializer list for any type for which we want one.

We could traverse that vector with a range-for loop:

```
For (int x : v) test(x);
```

This will call **test()** once for each element of **v**.

A range-for loop can traverse any sequence, so we could have simplified that example by using the initializer list directly:

```
for (int x : {1,2,3,5,8,13}) test(x);
```

One of the aims of C++11 was to make simple things simple. Naturally, this is done without adding performance penalties.

3. Myth 2: "C++ is an Object-Oriented Language"

No. C++ supports OOP and other programming styles, but is deliberately not limited to any narrow view of "Object Oriented." It supports a synthesis of programming techniques including object-oriented and generic programming. More often than not, the best solution to a problem involves more than one style ("paradigm"). By "best," I mean shortest, most comprehensible, most efficient, most maintainable, etc.

The "C++ is an OOPL" myth leads people to consider C++ unnecessary (when compared to C) unless you need large class hierarchies with many virtual (run-time polymorphic) functions – and for many people and for many problems, such use is inappropriate. Believing this myth leads others to condemn C++ for not being purely OO; after all, if you equate "good" and "object-oriented," C++ obviously contains much that is not OO and must therefore be deemed "not good." In either case, this myth provides a good excuse for not learning C++.

Consider an example:

```
void rotate_and_draw(vector<Shape*>& vs, int r)
{
        for_each(vs.begin(),vs.end(), [](Shape* p) { p->rotate(r); });   // rotate all elements of vs
        for (Shape* p : vs) p->draw();                                   // draw all elements of vs
}
```

Is this object-oriented? Of course it is; it relies critically on a class hierarchy with virtual functions. It is generic? Of course it is; it relies critically on a parameterized container (**vector**) and the generic function

for_each. Is this functional? Sort of; it uses a lambda (the **[]** construct). So what is it? It is modern C++: C++11.

I used both the range-**for** loop and the standard-library algorithm **for_each** just to show off features. In real code, I would have use only one loop, which I could have written either way.

3.1 Generic Programming

Would you like this code more generic? After all, it works only for **vector**s of pointers to **Shape**s. How about lists and built-in arrays? What about "smart pointers" (resource-management pointers), such as **shared_ptr** and **unique_ptr**? What about objects that are not called **Shape** that you can **draw()** and **rotate()**? Consider:

```
template<typename Iter>
void rotate_and_draw(Iter first, Iter last, int r)
{
        for_each(first,last,[](auto p) { p->rotate(r); });      // rotate all elements of [first:last)
        for (auto p = first; p!=last; ++p) p->draw();           // draw all elements of [first:last)
}
```

This works for any sequence you can iterate through from **first** to **last**. That's the style of the C++ standard-library algorithms. I used **auto** to avoid having to name the type of the interface to "shape-like objects." That's a C++11 feature meaning "use the type of the expression used as initializer," so for the **for**-loop **p**'s type is deduced to be whatever type **first** is. The use of **auto** to denote the argument type of a lambda is a C++14 feature, but already in use.

Consider:

```
void user(list<unique_ptr<Shape>>& lus, Container<Blob>& vb)
{
        rotate_and_draw(lus.begin(),lus.end());
        rotate_and_draw(begin(vb),end(vb));
}
```

Here, I assume that **Blob** is some graphical type with operations **draw()** and **rotate()** and that **Container** is some container type. The standard-library list (**std::list**) has member functions **begin()** and **end()** to help the user traverse its sequence of elements. That's nice and classical OOP. But what if **Container** is something that does not support the C++ standard library's notion of iterating over a half-open sequence, [**b**:**e**)? Something that does not have **begin()** and **end()** members? Well, I have never seen something container-like, that I couldn't traverse, so we can define free-standing **begin()** and **end()** with appropriate semantics. The standard library provides that for C-style arrays, so if **Container** is a C-style array, the problem is solved – and C-style arrays are still very common.

3.2 Adaptation

Consider a harder case: What if **Container** holds pointers to objects and has a different model for access and traversal? For example, assume that you are supposed to access a **Container** like this

```
for (auto p = c.first(); p!=nullptr; p=c.next()) { /* do something with *p */}
```

This style is not uncommon. We can map it to a [**b**:**e**) sequence like this

```
template<typename T> struct Iter {
        T* current;
        Container<T>& c;
};

template<typename T> Iter<T> begin(Container<T>& c) { return Iter<T>{c.first(),c}; }
template<typename T> Iter<T> end(Container<T>& c) { return Iter<T>{nullptr,c}; }
template<typename T> Iter<T> operator++(Iter<T> p) { p.current = p.c.next(); return *this; }
template<typename T> T* operator*(Iter<T> p) { return p.current; }
```

Note that this is modification is nonintrusive: I did not have to make changes to **Container** or some **Container** class hierarchy to map **Container** into the model of traversal supported by the C++ standard library. It is a form of adaptation, rather than a form of refactoring.

I chose this example to show that these generic programming techniques are not restricted to the standard library (in which they are pervasive). Also, for most common definitions of "object oriented," they are not object-oriented.

The idea that C++ code must be object-oriented (meaning use hierarchies and virtual functions everywhere) can be seriously damaging to performance. That view of OOP is great if you need run-time resolution of a set of types. I use it often for that. However, it is relatively rigid (not every related type fits into a hierarchy) and a virtual function call inhibits inlining (and that can cost you a factor of 50 in speed in simple and important cases).

4 Myth 3: "For reliable software, you need Garbage Collection"

Garbage collection does a good, but not perfect, job at reclaiming unused memory. It is not a panacea. Memory can be retained indirectly and many resources are not plain memory. Consider:

```
class Filter { // take input from file iname and produce output on file oname
pubilic:
        Filter(const string& iname, const string& oname);    // constructor
        ~Filter();                                           // destructor
        // …
private:
        ifstream is;
        ofstream os;
        // …
};
```

This **Filter**'s constructor opens two files. That done, the **Filter** performs some task on input from its input file producing output on its output file. The task could be hardwired into **Filter**, supplied as a lambda, or provided as a function that could be provided by a derived class overriding a virtual function. Those details are not important in a discussion of resource management. We can create **Filter**s like this:

```
void user()
{
        Filter flt {“books”,”authors”};
        Filter* p = new Filter{“novels”,”favorites”};
        // use flt and *p
        delete p;
}
```

From a resource management point of view, the problem here is how to guarantee that the files are closed and the resources associated with the two streams are properly reclaimed for potential re-use.

The conventional solution in languages and systems relying on garbage collection is to eliminate the **delete** (which is easily forgotten, leading to leaks) and the destructor (because garbage collected languages rarely have destructors and “finalizers” are best avoided because they can be logically tricky and often damage performance). A garbage collector can reclaim all memory, but we need user actions (code) to close the files and to release any non-memory resources (such as locks) associated with the streams. Thus memory is automatically (and in this case perfectly) reclaimed, but the management of other resources is manual and therefore open to errors and leaks.

The common and recommended C++ approach is to rely on destructors to ensure that resources are reclaimed. Typically, such resources are acquired in a constructor leading to the awkward name “Resource Acquisition Is Initialization” (RAII) for this simple and general technique. In **user()**, the destructor for **flt** implicitly calls the destructors for the streams **is** and **os**. These destructors in turn close the files and release the resources associated with the streams. The **delete** would do the same for ***p**.

Experienced users of modern C++ will have noticed that **user()** is rather clumsy and unnecessarily error-prone. This would be better:

```
void user2()
{
        Filter flt {“books”,”authors”};
        unique_ptr<Filter> p {new Filter{“novels”,”favorites”}};
        // use flt and *p
}
```

Now ***p** will be implicitly released whenever **user()** is exited. The programmer cannot forget to do so. The **unique_ptr** is a standard-library class designed to ensure resource release without runtime or space overheads compared to the use of built-in “naked” pointers.

However, we can still see the **new**, this solution is a bit verbose (the type **Filter** is repeated), and separating the construction of the ordinary pointer (using **new**) and the smart pointer (here, **unique_ptr**) inhibits some significant optimizations. We can improve this by using a C++14 helper function **make_unique** that constructs an object of a specified type and returns a **unique_ptr** to it:

```
void user3()
{
        Filter flt {“books”,”authors”};
```

```
        auto p = make_unique<Filter>("novels","favorites");
        // use flt and *p
}
```

Unless we really needed the second **Filter** to have pointer semantics (which is unlikely) this would be better still:

```
void user4()
{
        Filter flt {"books","authors"};
        Filter flt2 {"novels","favorites"};
        // use flt and flt2
}
```

This last version is shorter, simpler, clearer, and faster than the original.

But what does **Filter**'s destructor do? It releases the resources owned by a **Filter**; that is, it closes the files (by invoking their destructors). In fact, that is done implicitly, so unless something else is needed for **Filter**, we could eliminate the explicit mention of the **Filter** destructor and let the compiler handle it all. So, what I would have written was just:

```
class Filter { // take input from file iname and produce output on file oname
public:
        Filter(const string& iname, const string& oname);
        // …
private:
        ifstream is;
        ofstream os;
        // …
};

void user3()
{
        Filter flt {"books","authors"};
        Filter flt2 {"novels","favorites"};
        // use flt and flt2
}
```

This happens to be simpler than what you would write in most garbage collected languages (e.g., Java or C#) and it is not open to leaks caused by forgetful programmers. It is also faster than the obvious alternatives (no spurious use of the free/dynamic store and no need to run a garbage collector). Typically, RAII also decreases the resource retention time relative to manual approaches.

This is my ideal for resource management. It handles not just memory, but general (non-memory) resources, such as file handles, thread handles, and locks. But is it really general? How about objects that needs to be passed around from function to function? What about objects that don't have an obvious single owner?

4.1 Transferring Ownership: move

Let us first consider the problem of moving objects around from scope to scope. The critical question is how to get a lot of information out of a scope without serious overhead from copying or error-prone pointer use. The traditional approach is to use a pointer:

```
X* make_X()
{
        X* p = new X:
        // … fill X …
        return p;
}

void user()
{
        X* q = make_X();
        // … use *q …
        delete q;
}
```

Now who is responsible for deleting the object? In this simple case, obviously the caller of **make_X()** is, but in general the answer is not obvious. What if **make_X()** keeps a cache of objects to minimize allocation overhead? What if **user()** passed the pointer to some **other_user()**? The potential for confusion is large and leaks are not uncommon in this style of program.

I could use a **shared_ptr** or a **unique_ptr** to be explicit about the ownership of the created object. For example:

```
unique_ptr<X> make_X();
```

But why use a pointer (smart or not) at all? Often, I don't want a pointer and often a pointer would distract from the conventional use of an object. For example, a **Matrix** addition function creates a new object (the sum) from two arguments, but returning a pointer would lead to seriously odd code:

```
unique_ptr<Matrix> operator+(const Matrix& a, const Matrix& b);
Matrix res = *(a+b);
```

That ***** is needed to get the sum, rather than a pointer to it. What I really want in many cases is an object, rather than a pointer to an object. Most often, I can easily get that. In particular, small objects are cheap to copy and I wouldn't dream of using a pointer:

```
double sqrt(double);      // a square root function
double s2 = sqrt(2);      // get the square root of 2
```

On the other hand, objects holding lots of data are typically handles to most of that data. Consider **istream**, **string**, **vector**, **list**, and **thread**. They are all just a few words of data ensuring proper access to potentially large amounts of data. Consider again the **Matrix** addition. What we want is

```
Matrix operator+(const Matrix& a, const Matrix& b); // return the sum of a and b
Matrix r = x+y;
```

We can easily get that.

```
Matrix operator+(const Matrix& a, const Matrix& b)
{
        Matrix res;
        // … fill res with element sums …
        return res;
}
```

By default, this copies the elements of **res** into **r**, but since **res** is just about to be destroyed and the memory holding its elements is to be freed, there is no need to copy: we can "steal" the elements. Anybody could have done that since the first days of C++, and many did, but it was tricky to implement and the technique was not widely understood. C++11 directly supports "stealing the representation" from a handle in the form of move operations that transfer ownership. Consider a simple 2-D **Matrix** of doubles:

```
class Matrix {
        double* elem;    // pointer to elements
        int nrow;        // number of rows
        int ncol;        // number of columns
public:
        Matrix(int nr, int nc)                    // constructor: allocate elements
                :elem{new double[nr*nc]}, nrow{nr}, ncol{nc}
        {
                for(int i=0; i<nr*nc; ++i) elem[i]=0; // initialize elements
        }

        Matrix(const Matrix&);                    // copy constructor
        Matrix operator=(const Matrix&);          // copy assignment

        Matrix(Matrix&&);                         // move constructor
        Matrix operator=(Matrix&&);               // move assignment

        ~Matrix() { delete[] elem; }              // destructor: free the elements

        // …
};
```

A copy operation is recognized by its reference (**&**) argument. Similarly, a move operation is recognized by its rvalue reference (**&&**) argument. A move operation is supposed to "steal" the representation and leave an "empty object" behind. For **Matrix**, that means something like this:

```
Matrix::Matrix(Matrix&& a)                        // move constructor
        :nrow{a.nrow}, ncol{a.ncol}, elem{a.elem} // "steal" the representation
{
        a.elem = nullptr;                         // leave "nothing" behind
}
```

That's it! When the compiler sees the **return res;** it realizes that **res** is soon to be destroyed. That is, **res** will not be used after the return. Therefore it applies the move constructor, rather than the copy constructor to transfer the return value. In particular, for

```
Matrix r = a+b;
```

the **res** inside **operator+()** becomes empty – giving the destructor a trivial task – and **res**'s elements are now owned by **r**. We have managed to get the elements of the result – potentially megabytes of memory – out of the function (**operator+()**) and into the caller's variable. We have done that at a minimal cost (probably four word assignments).

Expert C++ users have pointed out that there are cases where a good compiler can eliminate the copy on **return** completely (in this case saving the four word moves and the destructor call). However, that is implementation dependent, and I don't like the performance of my basic programming techniques to depend on the degree of cleverness of individual compilers. Furthermore, a compiler that can eliminate the copy, can as easily eliminate the move. What we have here is a simple, reliable, and general way of eliminating complexity and cost of moving a lot of information from one scope to another.

Also, move semantics applies to assignment also, so for

```
r = a+b;
```

we get the move optimization from the move assignment operator. Optimizing assignment is far harder for an optimizer to do without language/programmer support than optimizing initialization.

Often, we don't even need to define all those copy and move operations. If a class is composed out of members that behave as desired, we can simply rely on the operations generated by default. Consider:

```
class Matrix {
        vector<double> elem;      // elements
        int nrow;                 // number of rows
        int ncol;                 // number of columns
public:
        Matrix(int nr, int nc)                    // constructor: allocate elements
                :elem(nr*nc), nrow{nr}, ncol{nc}
        { }

        // …
};
```

This version of **Matrix** behaves like the version above except that it copes slightly better with errors and has a slightly larger representation (a **vector** is usually three words).

What about objects that are not handles? If they are small, like an **int** or a **complex<double>**, don't worry. Otherwise, make them handles or return them using "smart" pointers, such as **unique_ptr** and **shared_ptr**. Don't mess with "naked" **new** and **delete** operations.

Unfortunately, a **Matrix** like the one I used in the example is not part of the ISO C++ standard library, but several are available (open source and commercial). For example, search the Web for "Origin Matrix

Sutton" and see Chapter 29 of my *The C++ Programming Language (Fourth Edition)* [11] for a discussion of the design of such a matrix.

4.2 Shared Ownership: shared_ptr

In discussions about garbage collection it is often observed that not every object has a unique owner. That means that we have to be able ensure that an object is destroyed/freed when the last reference to it disappears. In the model here, we have to have a mechanism to ensure that an object is destroyed when its last owner is destroyed. That is, we need a form of shared ownership. Say, we have a synchronized queue, a **sync_queue**, used to communicate between tasks. A producer and a consumer are each given a pointer to the **sync_queu**e:

```
void startup()
{
        sync_queue* p  = new sync_queue{200};    // trouble ahead!
        thread t1 {task1,iqueue,p};          // task1 reads from *iqueue and writes to *p
        thread t2 {task2,p,oqueue};          // task2 reads from *p and writes to *oqueue
        t1.detach();
        t2.detach();
}
```

I assume that **task1**, **task2**, **iqueue**, and **oqueue** have been suitably defined elsewhere and apologize for letting the thread outlive the scope in which they were created (using **detatch()**). Also, you may imagine pipelines with many more tasks and **sync_queue**s. However, here I am only interested in one question: "Who deletes the **sync_queue** created in **startup()**?" As written, there is only one good answer: "Whoever is the last to use the **sync_queue**." This is a classic motivating case for garbage collection. The original form of garbage collection was counted pointers: maintain a use count for the object and when the count is about to go to zero delete the object. Many languages today rely on a variant of this idea and C++11 supports it in the form of **shared_ptr**. The example becomes:

```
void startup()
{
        auto p = make_shared<sync_queue>(200); // make a sync_queue and return a stared_ptr to it
        thread t1 {task1,iqueue,p};          // task1 reads from *iqueue and writes to *p
        thread t2 {task2,p,oqueue};          // task2 reads from *p and writes to *oqueue
        t1.detach();
        t2.detach();
}
```

Now the destructors for **task1** and **task2** can destroy their **shared_ptr**s (and will do so implicitly in most good designs) and the last task to do so will destroy the **sync_queue**.

This is simple and reasonably efficient. It does not imply a complicated run-time system with a garbage collector. Importantly, it does not *just* reclaim the memory associated with the **sync_queue**. It reclaims the synchronization object (mutex, lock, or whatever) embedded in the **sync_queue** to mange the synchronization of the two threads running the two tasks. What we have here is again not just memory

management, it is general resource management. That "hidden" synchronization object is handled exactly as the file handles and stream buffers were handled in the earlier example.

We could try to eliminate the use of **shared_ptr** by introducing a unique owner in some scope that encloses the tasks, but doing so is not always simple, so C++11 provides both **unique_ptr** (for unique ownership) and **shared_ptr** (for shared ownership).

4.3 Type safety

Here, I have only addressed garbage collection in connection with resource management. It also has a role to play in type safety. As long as we have an explicit **delete** operation, it can be misused. For example:

```
X* p = new X;
X* q = p;
delete p;
// …
q->do_something();        // the memory that held *p may have been re-used
```

Don't do that. Naked **delete**s are dangerous – and unnecessary in general/user code. Leave **delete**s inside resource management classes, such as **string**, **ostream**, **thread**, **unique_ptr**, and **shared_ptr**. There, **delete**s are carefully matched with **new**s and harmless.

4.4 Summary: Resource Management Ideals

For resource management, I consider garbage collection a last choice, rather than "the solution" or an ideal:

1. Use appropriate abstractions that recursively and implicitly handle their own resources. Prefer such objects to be scoped variables.
2. When you need pointer/reference semantics, use "smart pointers" such as **unique_ptr** and **shared_ptr** to represent ownership.
3. If everything else fails (e.g., because your code is part of a program using a mess of pointers without a language supported strategy for resource management and error handling), try to handle non-memory resources "by hand" and plug in a conservative garbage collector to handle the almost inevitable memory leaks.

Is this strategy perfect? No, but it is general and simple. Traditional garbage-collection based strategies are not perfect either, and they don't directly address non-memory resources.

5. Myth 4: "For efficiency, you must write low-level code"

Many people seem to believe that efficient code must be low level. Some even seem to believe that low-level code is inherently efficient ("If it's that ugly, it ***must*** be fast! Someone must have spent a lot of time and ingenuity to write that!"). You can, of course, write efficient code using low-level facilities only, and some code has to be low-level to deal directly with machine resources. However, do measure to see if your efforts were worthwhile; modern C++ compilers are very effective and modern machine

architectures are very tricky. If needed, such low-level code is typically best hidden behind an interface designed to allow more convenient use. Often, hiding the low level code behind a higher-level interface also enables better optimizations (e.g., by insulating the low-level code from "insane" uses). Where efficiency matters, first try to achieve it by expressing the desired solution at a high level, don't dash for bits and pointers.

5.1 C's qsort()

Consider a simple example. If you want to sort a set of floating-point numbers in decreasing order, you could write a piece of code to do so. However, unless you have extreme requirements (e.g., have more numbers than would fit in memory), doing so would be most naïve. For decades, we have had library sort algorithms with acceptable performance characteristics. My least favorite is the ISO standard C library **qsort()**:

```
int greater(const void* p, const void* q)      // three-way compare
{
        double x = *(double*)p;    // get the double value stored at the address p
        double y = *(double*)q;
        if (x>y) return 1;
        if (x<y) return -1;
        return 0;
}

void do_my_sort(double* p, unsigned int n)
{
        qsort(p,n,sizeof(*p),greater);
}

int main()
{
        double a[500000];
        // … fill a …
        do_my_sort(a,sizeof(a)/sizeof(*a));          // pass pointer and number of elements
        // …
}
```

If you are not a C programmer or if you have not used **qsort** recently, this may require some explanation; **qsort** takes four arguments

- A pointer to a sequence of bytes
- The number of elements
- The size of an element stored in those bytes
- A function comparing two elements passed as pointers to their first bytes

Note that this interface throws away information. We are not really sorting bytes. We are sorting **double**s, but **qsort** doesn't know that so that we have to supply information about how to compare **double**s and the number of bytes used to hold a **double**. Of course, the compiler already knows such information perfectly well. However, **qsort**'s low-level interface prevents the compiler from taking

advantage of type information. Having to state simple information explicitly is also an opportunity for errors. Did I swap **qsort()**'s two integer arguments? If I did, the compiler wouldn't notice. Did my **compare()** follow the conventions for a C three-way compare?

If you look at an industrial strength implementation of **qsort** (please do), you will notice that it works hard to compensate for the lack of information. For example, swapping elements expressed as a number of bytes takes work to do as efficiently as a swap of a pair of **double**s. The expensive indirect calls to the comparison function can only be eliminated if the compiler does constant propagation for pointers to functions.

5.2 C++'s sort()

Compare **qsort()** to its C++ equivalent, **sort()**:

```
void do_my_sort(vector<double>& v)
{
        sort(v,[](double x, double y) { return x>y; });          // sort v in decreasing order
}

int main()
{
        vector<double> vd;
        // … fill vd …
        do_my_sort(v);
        // …
}
```

Less explanation is needed here. A **vector** knows its size, so we don't have to explicitly pass the number of elements. We never "lose" the type of elements, so we don't have to deal with element sizes. By default, **sort()** sorts in increasing order, so I have to specify the comparison criteria, just as I did for **qsort()**. Here, I passed it as a lambda expression comparing two doubles using **>**. As it happens, that lambda is trivially inlined by all C++ compilers I know of, so the comparison really becomes just a greater-than machine operation; there is no (inefficient) indirect function call.

I used a container version of **sort()** to avoid being explicit about the iterators. That is, to avoid having to write:

```
std::sort(v.begin(),v.end(),[](double x, double y) { return x>y; });
```

I could go further and use a C++14 comparison object:

```
sort(v,greater<>());          // sort v in decreasing order
```

Which version is faster? You can compile the **qsort** version as C or C++ without any performance difference, so this is really a comparison of programming styles, rather than of languages. The library implementations seem always to use the same algorithm for **sort** and **qsort**, so it is a comparison of programming styles, rather than of different algorithms. Different compilers and library implementations give different results, of course, but for each implementation we have a reasonable reflection of the effects of different levels of abstraction.

I recently ran the examples and found the **sort()** version 2.5 times faster than the **qsort()** version. Your mileage will vary from compiler to compiler and from machine to machine, but I have never seen **qsort** beat **sort**. I have seen **sort** run 10 times faster than **qsort**. How come? The C++ standard-library **sort** is clearly at a higher level than **qsort** as well as more general and flexible. It is type safe and parameterized over the storage type, element type, and sorting criteria. There isn't a pointer, cast, size, or a byte in sight. The C++ standard library STL, of which **sort** is a part, tries very hard not to throw away information. This makes for excellent inlining and good optimizations.

Generality and high-level code can beat low-level code. It doesn't always, of course, but the **sort**/**qsort** comparison is not an isolated example. Always start out with a higher-level, precise, and type safe version of the solution. Optimize (only) if needed.

6. Myth 5: "C++ is for large, complicated, programs only"

C++ is a big language. The size of its definition is very similar to those of C# and Java. But that does not imply that you have to know every detail to use it or use every feature directly in every program. Consider an example using only foundational components from the standard library:

```
set<string> get_addresses(istream& is)
{
        set<string> addr;
        regex pat { R"((\w+([.-]\w+)*)@(\w+([.-]\w+)*))"};   // email address pattern
        smatch m;
        for (string s; getline(is,s); )                      // read a line
                if (regex_search(s, m, pat))                 // look for the pattern
                        addr.insert(m[0]);                   // save address in set
        return addr;
}
```

I assume you know regular expressions. If not, now may be a good time to read up on them. Note that I rely on move semantics to simply and efficiently return a potentially large set of strings. All standard-library containers provide move constructors, so there is no need to mess around with **new**.

For this to work, I need to include the appropriate standard library components:

```
#include<string>
#include<set>
#include<iostream>
#include<sstream>
#include<regex>
using namespace std;
```

Let's test it:

```
istringstream test {        // a stream initialized to a sting containing some addresses
        "asasasa\n"
        "bs@foo.com\n"
        "ms@foo.bar.com$aaa\n"
        "ms@foo.bar.com aaa\n"
```

```
        "asdf bs.ms@x\n"
        "$$bs.ms@x$$goo\n"
        "cft foo-bar.ff@ss-tt.vv@yy asas"
        "qwert\n"
};

int main()
{
        auto addr = get_addresses(test);        // get the email addresses
        for (auto& s : addr)                    // write out the addresses
                cout << s << '\n';
}
```

This is just an example. It is easy to modify **get_addresses()** to take the **regex** pattern as an argument, so that it could find URLs or whatever. It is easy to modify **get_addresses()** to recognize more than one occurrence of a pattern in a line. After all, C++ is designed for flexibility and generality, but not every program has to be a complete library or application framework. However, the point here is that the task of extracting email addresses from a stream is simply expressed and easily tested.

6.1 Libraries

In any language, writing a program using only the built-in language features (such as **if**, **for**, and **+**) is quite tedious. Conversely, given suitable libraries (such as graphics, route planning, and database) just about any task can be accomplished with a reasonable amount of effort.

The ISO C++ standard library is relatively small (compared to commercial libraries), but there are plenty of open-source and commercial libraries "out there." For example, using (open source or proprietary) libraries, such as Boost [3], POCO [2], AMP [4], TBB [5], Cinder [6], vxWidgets [7], and CGAL [8], many common and more-specialized tasks become simple. As an example, let's modify the program above to read URLs from a web page. First, we generalize **get_addresses()** to find any string that matches a pattern:

```
set<string> get_strings(istream& is, regex pat)
{
        set<string> res;
        smatch m;
        for (string s; getline(is,s); )         // read a line
        if (regex_search(s, m, pat))
                res.insert(m[0]);               // save match in set
        return res;
}
```

That is just a simplification. Next, we have to figure out how to go out onto the Web to read a file. Boost has a library, **asio**, for communicating over the Web:

```
#include "boost/asio.hpp"       // get boost.asio
```

Talking to a web server is a bit involved:

```
int main()
try {
        string server = "www.stroustrup.com";
        boost::asio::ip::tcp::iostream s {server,"http"};  // make a connection
        connect_to_file(s,server,"C++.html");              // check and open file

        regex pat {R"((http://)?www([./#\+-]\w*)+)"}; // URL
        for (auto x : get_strings(s,pat))                    // look for URLs
                cout << x << '\n';
}
catch (std::exception& e) {
        std::cout << "Exception: " << e.what() << "\n";
        return 1;
}
```

Looking in **www.stroustrup.com**'s file **C++.html**, this gave:

http://www-h.eng.cam.ac.uk/help/tpl/languages/C++.html
http://www.accu.org
http://www.artima.co/cppsource
http://www.boost.org
…

I used a **set**, so the URLs are printed in lexicographical order.

I sneakily, but not altogether unrealistically, "hid" the checking and HTTP connection management in a function (**connect_to_file()**):

```
void connect_to_file(iostream& s, const string& server, const string& file)
        // open a connection to server and open an attach file to s
        // skip headers
{
        if (!s)
                throw runtime_error{"can't connect\n"};

        // Request to read the file from the server:
        s << "GET " << "http://"+server+"/"+file << " HTTP/1.0\r\n";
        s << "Host: " << server << "\r\n";
        s << "Accept: */*\r\n";
        s << "Connection: close\r\n\r\n";

        // Check that the response is OK:
        string http_version;
        unsigned int status_code;
        s >> http_version >> status_code;

        string status_message;
        getline(s,status_message);
```

```
        if (!s || http_version.substr(0, 5) != "HTTP/")
                throw runtime_error{ "Invalid response\n" };

        if (status_code!=200)
                throw runtime_error{ "Response returned with status code" };

        // Discard the response headers, which are terminated by a blank line:
        string header;
        while (getline(s,header) && header!="\r")
                ;
}
```

As is most common, I did not start from scratch. The HTTP connection management was mostly copied from Christopher Kohlhoff's **asio** documentation [9].

6.2 Hello, World!

C++ is a compiled language designed with the primary aim of delivering good, maintainable code where performance and reliability matters (e.g., infrastructure [10]). It is not meant to directly compete with interpreted or minimally-compiled "scripting" languages for really tiny programs. Indeed, such languages (e.g. JavaScript) – and others (e.g., Java) – are often implemented in C++. However, there are many useful C++ programs that are just a few dozen or a few hundred lines long.

The C++ library writers could help here. Instead of (just) focusing on the clever and advanced parts of a library, provide easy-to-try "Hello, World" examples. Have a trivial-to-install minimal version of the library and have a max-one-page "Hello, World!" example of what the library can do. We are all novices at some time or other. Incidentally, my version of "Hello, World!" for C++ is:

```
#include<iostream>

int main()
{
        std::cout << "Hello, World\n";
}
```

I find longer and more complicated versions less than amusing when used to illustrate ISO C++ and its standard library.

7 The Many Uses of Myths

Myths sometimes have a basis in reality. For each of these myths there have been times and situations where someone could reasonably believe them based on evidence. For today, I consider them flat-out wrong, simple misunderstandings, however honestly acquired. One problem is that myths always serve a purpose – or they would have died out. These five myths have served and serve in a variety of roles:

- They can offer comfort: No change is needed; no reevaluation of assumptions is needed. What is familiar feels good. Change can be unsettling, so it would be nice if the new was not viable.

- They can save time getting started with a new project: If you (think you) know what C++ is, you don't have to spend time learning something new. You don't have to experiment with new techniques. You don't have to measure for potential performance snags. You don't have to train new programmers.
- They can save you from having to learn C++: If those myths were true, why on earth would you want to spend time learning C++?
- They can help promote alternative languages and techniques: If those myths were true, alternatives are obviously necessary.

But these myths are not true, so intellectually honest promotion of status quo, alternatives to C++, or avoidance of modern C++ programming styles cannot rely on them. Cruising along with an older view of C++ (with familiar language subsets and techniques) may be comfortable, but the state of software is such that change is necessary. We can do much better than with C, "C with Classes", C++98, etc.

Sticking to the old-and-true is not cost free. Maintenance cost is often higher than for more modern code. Older compilers and tool chains deliver less performance and worse analysis than modern tools relying on more structured modern code. Good programmers often choose not to work on "antique" code.

Modern C++ (C++11, C++14) and the programming techniques it supports are different and far better than "common, popular myths" would indicate.

If you believe one of these myths, don't just take my word for it being false. Try it. Test it. Measure "the old way" and the alternatives for some problem you care about. Try to get a real hold on the time needed to learn the new facilities and techniques, the time to write code the new way, the runtime of the modern code. Don't forget to compare the likely maintenance cost to the cost of sticking with "the old way." The only perfect debunking of a myth is to present evidence. Here, I have presented only examples and arguments.

And no, this is not an argument that C++ is perfect. C++ is not perfect; it is not the best language for everything and for everybody. Neither is any other language. Take C++ for what it is, rather than what it was 20 years ago or what someone promoting an alternative claims it to be. To make a rational choice, get some solid information and – as far as time allows – try for yourself to see how current C++ works for the kind of problems you face.

8 Summary

Don't believe "common knowledge" about C++ or its use without evidence. This article takes on five frequently expressed opinions about C++ and argues that they are "mere myths:"

1. "To understand C++, you must first learn C"
2. "C++ is an Object-Oriented Language"
3. "For reliable software, you need Garbage Collection"
4. "For efficiency, you must write low-level code"

5. "C++ is for large, complicated, programs only"

They do harm.

9 Feedback

Not convinced? Tell me why. What other myths have you encountered? Why are they myths rather than valid experiences? What evidence do you have that might debunk a myth?

10 References

1. ISO/IEC 14882:2011 Programming Language C++
2. POCO libraries: http://pocoproject.org/
3. Boost libraries: http://www.boost.org/
4. AMP: C++ Accelerated Massive Parallelism. http://msdn.microsoft.com/en-us/library/hh265137.aspx
5. TBB: Intel Threading Building Blocks. www.threadingbuildingblocks.org/
6. Cinder: A *library* for professional-quality creative coding. http://libcinder.org/
7. vxWidgets: A Cross-Platform GUI Library. *www.wxwidgets.org*
8. Cgal - *Computational Geometry Algorithms Library*. www.cgal.org
9. Christopher Kohlhoff: Boost.Asio documentation. http://www.boost.org/doc/libs/1_55_0/doc/html/boost_asio.html
10. B. Stroustrup: *Software Development for Infrastructure*. Computer, vol. 45, no. 1, pp. 47-58, Jan. 2012, doi:10.1109/MC.2011.353.
11. Bjarne Stroustrup: *The C++ Programming Language (4th Edition)*. Addison-Wesley. ISBN 978-0321563842. May 2013.
12. Bjarne Stroustrup: *A Tour of C++*. Addison Wesley. ISBN 978-0321958310. September 2013.
13. B. Stroustrup: *Programming: Principles and Practice using C++ (2nd edition)*. Addison-Wesley. ISBN 978-0321992789. May 2014.

Postscript

The 10 sections of this article was posted on isocpp.org in three installments and reposted widely. It attracted a quite varied set of comments. I have now fixed a few typos that were reported. Thanks.

This postscript is my observations on some of the comments posted.

The comments prove – yet again – that the "Myths" paper was needed. People keep repeating the old hairy rationalizations. Unfortunately, many programmers don't read long papers and dismiss short ones for being incomplete. The unwillingness of many programmers to read a long paper was the reason I released this paper in three separate parts.

This paper is not a research paper carefully outlining every alternative and carefully documenting every detail. I said that right up front:

> Each myth requires a long paper or even a book to completely debunk, but my aim here is simply to raise the issues and to *briefly* state my reasons.

However, many seems to confuse the examples used to illustrate a point with the point itself. Many tried to "debunk the debunking" by changing examples, by changing the constraints on the examples, or by deeming the examples trivial. The examples are small – they have to be to fit into a short paper – but they are not unrepresentative of code found as part of real-world programs.

Many commenters quickly did a shift from the C++11/C++14 that I base my arguments on to some older version. C++14 is ***not*** the C++ of the 1980s. It is ***not*** what most people were first taught. It is ***not*** the C++ that people is presented with in most beginning C++ courses today. It is ***not*** what most people see when they look at a large code base. I want to change that. Not being able to do some example that I present in an antique version of C++ or with an outdated compiler is unfortunate, but better versions of the major compiler ship (typically for free) today. I showed no examples of "bleeding edge" code.

The problem of code in older styles is one that every successful programming language must face, so please don't judge C++ exclusively based on 20-year-old techniques and 10-year old compilers. Look at modern C++ and find ways of getting it into use – many has already. You almost certainly used a program today that was written using C++11. There are C++11 in many steps of the chain between the computer on which I write this and the computer on which you read it.

Quite a few comments were along the lines of "Language X has a feature that does exactly that" and "Library Y in language X does exactly that." Obviously, if you have a language that provides a simpler solution to the best you can do in C++, with acceptable performance, portability, and tool-chain constraints for what you want to do, use it. However, no language and library is perfect for everything and everybody.

I present examples of general problems and general techniques. Matching a single example in some way is not necessarily significant. My points are general and the examples only simple illustrations. Given a sufficiently good library, just about any programming task can be simple and pleasant. Given a sufficiently constrained task, we can always design a specialized language to be more elegant than a general-purpose one. For example, the asio library I used in §6.1 is a flexible, efficient, general-purpose networking library. For any one task, I could wrap it in a far simpler function (or small set of functions) to make that task significantly more convenient. The code I showed would then be the implementation. My key point in §6.2 is that the C++ library development community could do many programmers a favor by spending a little more tome making simple things simple. For example 99% of the time I prefer **sort(v)** to **sort(v.begin(),v.end())**.

Performance

My comments about performance caused quite a stir in places. Many people tried to dismiss them with arguments or mere counter-assertions. I don't accept performance arguments unsupported by measurements. My comments have been validated by real performance measures in many contexts over years. Many can be found in the literature. My main performance points hold over a wide range of similar examples and scale.

Please note that I assume a modern, standards-conforming C++ implementation. For example, when I talk about the performance of the short-string optimization, I don't mean a pre-C++11 standard library without that optimization. Also, I take no notice of comments to the effect that C++ facilities such as **std::sort()** or **std::string** are slow if you don't use an optimizer – of course they are, but talking about performance of unoptimized code is silly. If you use GCC or Clang use –O2; for Microsoft, use release mode.

C

I know C and its standard library pretty well. I wrote considerable amounts of C before many of today's students were even born and contributed significantly to the C language: function prototypes, **const**, **inline**, declarations in **for**-statement, declarations as statements, and more came from my work. I have followed its development and the evolution programming styles in C.

Yes, the C versions of **compose()** fails to check **malloc()**'s return value. I did ask if you thought I got it right. I did not present production-quality code, and I knew that. Failing to check results is a major source of errors, so my "mistake" failing to check the result of **malloc()** was deliberate illustrates a real problem. As in this case, exceptions can often help.

Yes, you could write the C version of **compose()** differently using less well known standard-library functions, and yes, you can avoid the use of free store if you let the caller supply a buffer allocated on the stack and let the caller deal with the problem of string arguments that would overflow the buffer. However, such alternatives completely miss the point: it is harder to write such code than the C++ version, and far harder to get it right. Novices get the C++ version right the first time, but not any of the C versions, especially not the C versions that rely on standard-library function not commonly taught to novices.

C++ use

C++ has been used for demanding embedded systems and critical systems for years, examples are The Mars Rovers (scene analysis and autonomous operations), The F-35s and F-16s (flight controls), and many, many more: http://www.stroustrup.com/applications.html. And, yes, the Orion space capsule is programmed in C++.

Libraries

Yes, libraries vary in quality and it can be extremely hard to choose from the large selection of libraries beyond the standard. This is a major problem. However, such libraries exist and researching them is often more productive than simply barging ahead and reinventing yet-another wheel.

Unfortunately, C++ Libraries are often not designed for interoperability with other libraries.

Unfortunately, there is not a single place to go to look for C++ Libraries.

Teaching

I have observed students being taught by the "C first" approach for many years and seen the programs written by such students for decades. I have taught C++ as the first programming language to thousands

of students over several years. My claims about the teachability of C++ are based on significant experience, rather than introspection.

C++ is easier to teach to beginners than C because of a better type system and more notational support. There are also fewer tricks and workarounds to learn. Just imagine how you would teach the styles of programming you use in C using C++; C++'s support for those is better.

I would never dream of giving a beginner's C++ course that

- didn't include a thorough grounding in memory management, pointers, etc.
- didn't give the students a look at "plain C"' and some idea of how to use it
- didn't present a rationale for the major features
- tried to teach all of C++ and every C++ technique

Similarly, good C teachers do not try to teach all of C and all C techniques to beginners.

http://www.stroustrup.com/programming.html is my answer to the question "How would you teach C++ to beginners?" It works.

For a rather old paper comparing aspects of teachability of C and C++, see B. Stroustrup: *Learning Standard C++ as a New Language*. C/C++ Users Journal. pp 43-54. May 1999 (www.stroustrup.com/papers.html). Today, I could write the C version a bit better and the C++ version quite a bit better. The examples reflect common styles of the time (and were reviewed by expert C and C++ programmers).

C++ today is ISO Standard C++14, rather than what I described 30 years ago or what your teacher may have taught you 20 years ago. Learn C++11/C++14 as supported by current mainstream compilers and get used to it. It is a far better tool than earlier versions of C++. Similarly, C today is ISO Standard C11, rather than K&R C (though I am not sure if the C compilers today are as close to C11 as the C++ compilers are close to C++14).

I am appalled by much that is taught as "good C++."

C++ is not (and never were meant to be) an "OOP" language. It is a language that supports OOP, other programming techniques, and combinations of such techniques.

If you are an experienced programmer, I recommend *A Tour of C++* [12] as a quick overview of modern C++.