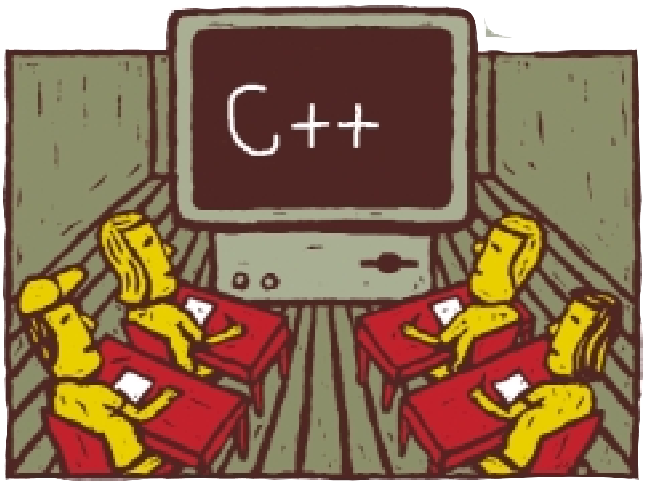
Modern C++ is a more expressive, simpler language than C, and a language in its own right, so why do so many people insist on teaching it historically? **Kevlin Henney** appeals for a reform of the C++ education system

The miseducation of C++

Kevlin Henney is an independent software development consultant and trainer. He can be reached at **www.curbralan.com**

LEARNING LATIN IN ORDER TO learn French is a roundabout route that might satisfy linguists, historians and hobbyists with time on their hands, but it's unlikely to appeal to the practically-minded. Nevertheless, it's still the approach taken by many C++ books and courses. They start by assuming or teaching C and C idioms (with perhaps a basic garnish of C++), and then add classes and other language features according to taste. Templates and the standard library are left until the end – for some reason they are often considered advanced topics simply because they're more recent. The informed and more up-to-date thinking on this topic reflects a quite different perspective[1,2,3].



The historical approach may well recall the way that C++ evolved, and often the way that course and book authors learned the language, but for C++ novices (the majority of whom are no longer C programmers), this is more a tour of the twisted back streets of C++, with many important and interesting sights overlooked. Learning C++ is no holiday, but it takes longer this way than by treating it as a new language. Programmers taught in the old way have less effective C++ skills than those taught in the new way, because they are blind to the expressiveness that contemporary idioms and the standard library offer them. They are often taught C++ within a constrained, object-oriented orthodoxy, ignoring the language's support for other styles, such as generic programming.

These programmers are not to blame. It is typically the fault of training companies, universities, authors, publishers and colleagues who are still rooted in a former, more mechanically minded C++ era.

The three ages of C++

C++ has evolved through three reasonably distinct phases, each corresponding roughly with editions of *The C++ Programming Language*[4,5,6]. Each era can be regarded as a separate language, with different features and supporting different styles:

- Early C++: The first public incarnation of the language formed part of a growing interest in the use of object-oriented development in industry. Early C++ was a minimal language, extending C with a stronger type system, and the notion of classes with public and private access, single inheritance and operator overloading.
- Classic C++: This is the C++ that many long-time C++ programmers know, and many new C++ programmers are still effectively being taught. It brought abstract classes, multiple inheritance and further refinements into the language. Classic C++ also introduced templates and exceptions, although they took longer to mature than other, more obviously evolutionary features. Where templates are used, it's normally to articulate container classes and smart pointers. Classic C++ saw the greatest proliferation in dialects as it was in this period that the language underwent standardisation. Changes ranged from the trivial to the major, from `bool` to `namespace`, from the proper incorporation of the standard C library to the internationalisation overhaul of I/O. Many projects still bear the scars of this portability interregnum.

FACTS AT A GLANCE

- **Most C++ teaching is stuck in a bygone era of thinking in C and then adding objects.**
- **The majority of programmers now learning C++ have never used C.**
- **Modern C++ should be taught as a language in its own right.**
- **C++ supports many programming styles, not just C and basic OO.**
- **Standard C++ should be taught at a high level rather than a low level.**
- **Generic programming complements traditional object-oriented programming.**

● Modern C++: This is where we are now. The language was standardised in 1998[7], providing a fixed target for vendors and a consolidation in thinking and practice. Exception safety is now a well understood topic and the Standard Template Library (STL) part of the library has introduced the full power of generic programming to the language, raising templates from being merely a way to express containers to an integral part of design expression in C++. However, before it all starts sounding too good to be true, compilers still need to do some catching up, and parts of the library demonstrate the excesses possible through design by committee. For instance, allocators, locales and strings all have interfaces and semantics to die from, although strings at least have the merit of being usable. Inside std::basic_string is a small class struggling to get out.

These three eras aren't separated by crisp boundaries. As with any evolving language, each era fades into the next, and the seeds of subsequent styles and demand for features are often sown in the previous. For instance, although STL was made available in 1994 and then adopted into the draft standard, its full impact on design style wasn't felt and understood until later.

From bottom to top

There's too much C++ being written. This isn't about language preferences, but a quantity judgement on the code being written to express higher level designs. Much C++ suffers from being low-level and repetitive. Whether it's the manual management of strings and arrays or the tedious transcription of similar loops, there's a lot of repetition and little abstraction of small housekeeping tasks – in short, error-prone code and a failure to reuse what exists already.

The sentiment has been growing for some time that teaching C++ from scratch shouldn't involve a descent into memory management to achieve simple tasks. Allan Vermeulen made the distinction that C++ was effectively two languages, one for building components and the other for using them – Bottom C++ and Top C++[8]. Using an efficient library that wraps up low-level or common facilities is a different prospect from writing one, and each does not require the same type or level of knowledge. This separation of views between clients and suppliers has been preached for a long time in OO thinking.

Text manipulation and good old-fashioned I/O have come back into fashion, thanks to HTML, XML and ecommerce in general, so it's worth seeing the difference between introducing C++ with and without the standard library in this context. Consider the common task of concatenating two given strings with a space between them, such as a person's first and last name. This task isn't exactly rocket science, but with a nuts and bolts view of C++, this looks C-like and something like the following (assuming using namespace std for brevity):

```
char *full_name(const char *first, const char *last)
{
    const size_t length = strlen(first) + strlen(last) + 1;
    char *result = new char[length + 1];
    strcpy(result, first);
    strcat(result, " ");
    strcat(result, last);
    return result;
}
```

It's possible to write this function in a single return statement, but it's unlikely anyone would thank you for it. Using this function also demands a certain amount of hoop jumping:

```
{
    char *name = full_name(first, last);
    ... // perform tasks using name
    delete[] name;
}
```

This isn't exception safe, though, so perhaps the following:

```
{
    char *name = full_name(first, last);
    try
    {
        ... // perform tasks using name
    }
    catch(...)
    {
        delete[] name;
        throw; // repropagate to caller
    }
}
```

Contrast this with the library-based approach:

```
string full_name(const string &first, const string &last)
{
    return first + " " + last;
}
```

And in use, including exception safety:

```
{
    string name = full_name(first, last);
    ... // perform tasks using name
}
```

It beats me why anyone would consider the first approach better for learning, but it still seems to be popular. Is it a philosophy of 'no pain, no gain'? Perhaps 'no pain, no pain' is better. The nitty-gritty internal workings can be learned more easily when you're comfortable with the usage – you don't have to be able to design a string class before you can use one. This is, after all, the whole point of encapsulation.

For the same reason, templated container classes should be introduced early on. Collections of data are so common in programs that they can hardly be considered advanced, and therefore something to save until last. It's far easier to explain that vector<int> can be read as "vector of int", where vector is a resizable array type, than to explain all the details behind int *, the relationship between arrays and pointers, and how to use new[] and delete[] correctly.

Hello, worlds

A modern approach to teaching C++ isn't simply a concession to using std::string and std::vector. It includes embracing generic programming. Algorithmic abstraction in generic programming goes beyond either the procedural view or the conventional object-oriented view.

To get a feel for the difference, along with an appreciation of the STL as more than just a container library, consider the task of printing out a sequence of strings. Assume, first, that the source of strings is fixed:

```
const char *const source[] =
{
    "Mercury", "Venus", "Earth", "Mars",
    "Jupiter", "Saturn", "Uranus", "Neptune", "Pluto"
};
```

To print it, you can resort to the common-or-garden for loop:

```
for(std::size_t at = 0; at != 9; ++at)
   std::cout << source[at] << std::endl;
```

Aside from some nasty hard-wiring of a constant, this doesn't seem to be crying out for improvement. Let's move the goalposts a little. What if the input sequence isn't fixed at compile time? How would you accommodate an arbitrary input set that could be piped or typed in at runtime, such as "Mother Very Easily Makes Jam Sandwiches Using No Peanuts", and process the sequence in some way before printing it out? Without self-managing containers and strings, this task turns into a delicate but long-winded demonstration of memory management minutiae. With them, it looks much simpler:

```
std::vector<std::string> source;
std::string input;
while(std::cin >> input)
   source.push_back(input);
... // process source
for(std::size_t at = 0; at != source.size(); ++at)
   std::cout << source[at] << std::endl;
```

Let's say that you want to alphabetically order the input. Keeping things simple, let's assume the English alphabet and an ASCII-based character set. The standard multiset container uses default *less-than* ordering, and default *less-than* ordering for a string is just what we want. All the input is ordered as it comes in and, because it's a multiset rather than a set, duplicate strings will be retained:

```
std::multiset<std::string> source;
std::string input;
while(std::cin >> input)
   source.insert(input);
... // output
```

The subscript operator can't be used for output because associative containers, such as multiset, don't support random access. Iterators offer a means of decoupling the traversal of a container from its underlying representation. STL iterators follow pointer notation and semantics, representing an abstracted level of indirection to each element:

```
... // input
std::multiset<std::string>::iterator at = source.begin();
while(at != source.end())
   std::cout << *at++ << std::endl;
```

Although the condition and body of the loop are quite straightforward, there is a lot of syntactic baggage in the iterator declaration. This load can be lightened a little with the aid of a typedef, an oft-neglected feature in a class-based world:

```
typedef std::multiset<std::string> strings;
strings source;
std::string input;
while(std::cin >> input)
   source.insert(input);
strings::iterator at = source.begin();
while(at != source.end())
   std::cout << *at++ << std::endl;
```

This is about as far as you can go with the "STL is a container library" point of view. Iterators are the key to opening up the rest of the STL through its algorithms. Let's switch back to using vector so that we can accommodate general manipulation of ordering:

```
typedef std::vector<std::string> strings;
strings source;
std::string input;
while(std::cin >> input)
   source.push_back(input);
... // process source
strings::iterator at = source.begin();
while(at != source.end())
   std::cout << *at++ << std::endl;
```

If we want to sort the input, as with the previous example, we can use the standard sort algorithm that operates on iterator ranges, completely independent of the underlying container type:

```
std::sort(source.begin(), source.end());
```

An iterator range is defined as a half-open range that includes its first element but excludes its last. The independence means that you can use this algorithm with any range that supports random access iterators, including raw C-style arrays and any of your own application-specific containers.

If your sorting criteria are slightly different – say you want to sort on string length – you can define a function that performs the desired ordering comparison:

```
bool shorter(const std::string &lhs, const std::string &rhs)
{
   return lhs.size() < rhs.size();
}
```

And then use it with std::sort:

```
std::sort(source.begin(), source.end(), shorter);
```

The concept of an iterator range isn't restricted to containers. You can also wrap I/O streams with iterators. To replace the input loop, we can use the istream_iterator adaptor on std::cin:

```
std::istream_iterator<std::string> begin(std::cin), end;
std::copy(begin, end, std::back_inserter(source));
```

A defaulted istream_iterator represents the end of file. back_inserter returns an adapted iterator that performs push_back on source for every string copied. It is also possible to initialise source directly from the input, without worrying about an intermediate copying stage. Dispensing with temporary variables and introducing other typedefs, reverting to default sort ordering and abstracting the output in terms of copying between iterator ranges, gives the following script-like code:

```
typedef std::istream_iterator<std::string> in;
typedef std::ostream_iterator<std::string> out;
std::vector<std::string> source(in(std::cin), in());
std::sort(source.begin(), source.end());
std::copy(source.begin(), source.end(), out(std::cout, "\n"));
```

Another change in requirements allows us to see a different approach to tackling the output. Imagine you want the output within quotes, so you can see any trailing spaces. The following function prints a string out this way:

```
void quoted(const std::string &arg)
{
   std::cout << '"' << arg << '"' << std::endl;
}
```

We can still abstract the loop by using one of the other standard algorithms, std::for_each, which applies its third argument to each element in its given iterator range:

```
std::for_each(source.begin(), source.end(), quoted);
```

What if we wish to parameterise the function so it can take streams other than std::cout, such as a named file? std::for_each expects to execute its third argument as a unary function, so we don't have the available bandwidth to introduce another argument. However, we can define simple, lightweight classes whose instances, to all intents and purposes, behave like functions:

```
class quoted
{
public:
    quoted(std::ostream &chosen_out)
      : out(chosen_out)
    {
    }
    void operator()(const std::string &arg)
    {
        out << '"' << arg << '"' << std::endl;
    }
private:
    std::ostream &out;
};
```

Function objects can retain individual state, such as the chosen output stream, but still execute using the conventional function call syntax. This allows them to slot right into many standard algorithms:

```
std::for_each(source.begin(), source.end(), quoted(cout));
```

The quoted class seems to be quite useful, and possibly something you might want to reuse in another context. However, it is currently tied to std::string. You might wish to use it with other string types, numbers or your own application-specific classes. It's tempting to make the whole class a template:

```
template<typename argument_type>
class quoted
{
public:
    quoted(std::ostream &chosen_out = std::cout)
      : out(chosen_out)
    {
    }
    void operator()(const argument_type &arg)
    {
        out << '"' << arg << '"' << std::endl;
    }
private:
    std::ostream &out;
};
```

But this makes it a little more awkward to use:

```
std::for_each(
    source.begin(), source.end(),
    quoted<std::string>());
```

In truth, the only feature that is dependent on the argument type is operator() itself. A member template can capture this variability:

```
class quoted
{
public:
    quoted(std::ostream &chosen_out = std::cout)
      : out(chosen_out)
    {
    }
    template<typename argument_type>
    void operator()(const argument_type &arg)
    {
        out << '"' << arg << '"' << std::endl;
    }
private:
    std::ostream &out;
};
```

Leaving the usage as before:

```
std::for_each(source.begin(), source.end(), quoted());
```

A new way of learning

If you're unfamiliar with generic programming, this idiomatic approach may look daunting. However, it is just that – a lack of familiarity. Much language teaching is simply how to write procedural code in different languages in different styles. For C++, such teaching can be characterised as how to write procedural code with curly brackets, pointers and objects. Generic programming is different from the procedural model of OO in that it embraces a more value-based, functional programming style that is practical and efficient. (In execution, it can be faster than C.) The code we have just developed isn't necessarily something you would hit a programmer with at the beginning of a course or a book, but something to work towards – from loops for novices to loops for practitioners.

C++ is by no means a trivial language, but modern C++ offers a different set of features and styles from its earlier incarnations, supporting simpler and more expressive idioms. Rather than learning it historically as a class-based extension of C, it can be understood directly and effectively as a quite distinct language.

It is far more compelling to teach the language by means of evolving designs – a set of differences, each of which demonstrates particular features and techniques – than through a set of differences from C. Many programmers, and therefore many companies, are quite simply missing out on a more effective set of skills by being taught the wrong language.

So, next time you're looking – for yourself or others – for a course or book that purports to teach C++, look to see when strings and simple container use are introduced. If it's over a quarter of the way through, be suspicious. If it's over halfway through, don't touch it, except for morbid or historical interest. ■

References

1. Stanley B Lippman, *Essential C++*, Addison-Wesley, 2000
2. Andrew Koenig and Barbara E Moo, *Accelerated C++*, Addison-Wesley, 2000
3. Bjarne Stroustrup, Adding Classes to the C Language: An Exercise in Language Evolution, *Software – Practice and Experience*, Volume 13, Wiley, 1983
4. Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1986
5. Bjarne Stroustrup, *The C++ Programming Language*, Second edition, Addison-Wesley, 1991
6. Bjarne Stroustrup, *The C++ Programming Language*, Third edition, Addison-Wesley, 1997
7. *International Standard: Programming Language – C++*, ISO/IEC 14882:1998(E), 1998
8. Allan Vermeulen, Top C++, *Object Magazine*, SIGS, June 1996