Why reinvent the wheel when the work has already been done in the STL? **Kevlin Henney** dons his toolbelt for a whirlwind tour of the intricacies of containers and iterators

If I had a hammer...

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

HUMANS ARE TOOL USERS AND THAT INCLUDES software developers. Flicking through the pages of *Application Development Advisor* reveals the limitless range of software tools on which you can spend your budget. And if you believe many of the productivity promises – based largely on the prevention of thinking, it seems – the combined effect of half a dozen of these tools should have your project completed single-handedly before its start date.

In addition to the flashier, more expensive power tools of the trade, we shouldn't forget the smaller utilities – the hammers and screwdrivers of the software development world. C++ comes with a number of language and library features that simplify certain common or overlooked development tasks, allowing attention to detail without the need for intensive coding.

So, what's the standard C++ equivalent of the hammer or the screwdriver? For my money, there's no doubt what the hammer is – it's containers. If one part of the standard library has been embraced above all others, it's the container class templates – most commonly `vector`, with `map` following a close second. When many C++ programmers think of the STL, this is what comes to mind. And as the saying goes, when the only tool you have is a hammer, everything else looks like a nail.

Contrary to popular myth, however, the point of the STL is that it's not really a container library: it's a specification of a library of algorithms and the things algorithms work on. And what do algorithms work on? Iterators, not containers. The only reason that containers exist is to give certain iterators something to chew on.

Does this distinction matter? Is the reversal of perspective anything more than an academic exercise? Well, from the nail's point of view, perspective is everything... especially if the nail concerned is actually a screw.

Group the loop

The common experience of iterators is that they are nested within the scope of container class templates, so it's often assumed that they're secondary abstractions – that is, that they're less important and merely following in the wake of containers. Although the syntax for using iterators is simple enough – a subset of pointer notation – the syntax for declaring them almost seems to discourage their use. Consider a list of strings, `words`, and a function, `action`, that is applied to each string in the list:

```
for(std::list<std::string>::iterator at = words.begin();
   at != words.end();
   ++at)
{
   action(*at);
}
```

Not even the most ardent supporter of generic programming will, hand on heart, describe the full type declaration as either convenient or elegant. A *using directive* or *using declaration* isn't the solution. At best, it reduces type names by the amount of `std::`. At worst, it negates the benefit of having namespaces in the first place.

But a simple tool exists in the language for simplifying this syntax: `typedef`. Although not fully redundant, `typedef` is used less in C++ than in C. It's often regarded as the poor cousin of the class because it only creates type aliases rather than proper types with their own distinct behaviour. And this is precisely why it's useful as a local abbreviation facility:

```
typedef std::list<std::string>::iterator input;
for(input at = words.begin(); at != words.end(); ++at)
{
   action(*at);
}
```

`typedef`s are a zero-overhead feature. That is, they cost nothing in the generated code, their strength lying in making code clearer and more expressive. Although `typedef`s are commonly used at global or class scopes, they can appear in any scope you want. The more

FACTS AT A GLANCE

- **The C++ standard library offers a number of fine-grained, tactical tools for programming.**
- **Developers often look to the STL for containers, but these are almost an incidental feature of generic programming.**
- **Generic programming is built on algorithmic abstraction, which means that the algorithm functions and iterators precede containers in importance.**
- **Common looping tasks can be greatly simplified by taking such a view of the STL.**

local their scope, the simpler you can make their name. In the example shown, the iterator type is named after its category, which is one of *input*, *output*, *forward*, *bidirectional* or *random access*. Only input iterator operations – operators !=, ++ and reading through * – are required for the loop, and the chosen type name reflects this. This simple naming also leaves the loop code unaffected should the underlying representation change, for example to a vector or a deque.

Although the second example is neater than the first, we can still do better. Why should we have to know the type at all? And what's the most important feature of this code – the fact that action is being applied to each element in words, or the loop's housekeeping details? Hopefully the former, but looking at the code suggests that if real estate is anything to go by, it's the latter. Design is about capturing and reflecting intent:

```
std::for_each(words.begin(), words.end(), action);
```

The fact that loops repeat doesn't mean you have to keep rewriting them. The std::for_each function template, found in the <algorithm> header, is perhaps the simplest algorithm in the STL, but it serves to demonstrate the basic idiom of generic programming – the difference between loops for novices and loops for experts[1]. Most of the time, you shouldn't need to worry about how complex an iterator type name is because you never see it: let the compiler do the work of deducing it. This approach isn't rocket science, but it's certainly different from the container-centric view.

Customer-facing code

Forms. Our life is full of them and, thanks to the web, it looks set to remain that way for some time. Form processing is perhaps the most common area of basic string manipulation encountered by application programmers. However, there are times when you feel that functional correctness and usability could be brought a little closer together. A common experience is to fill in some kind of number – such as dates, credit card numbers, telephone numbers and ISBNs – with what you might consider natural spacing or separators, only to have them spat back at you.

The standard library offers a range of algorithms to make software slightly more customer-facing with the minimum of fuss. You can look in the std::basic_string class template interface to resolve your string manipulation tasks, but the interface is large and confused. It's neither complete enough to be considered fully general-purpose, nor consistent enough to be considered cohesive[2]. While many basic string processing operations are supported directly within basic_string, the STL offers a more minimal, extensible and consistent approach. std::basic_string satisfies STL container requirements, including the possession of iterators. Consider replacing dashes in a piece of text with spaces, for example from "7221-7691-7160-9691" to "7221 7691 7160 9691":

```
std::replace(text.begin(), text.end(), '-', ' ');
```

Allowing spaces for convenience is another formatting concession that's worth offering the user. I recall once filling out a 30-digit licence code for a piece of software and having it rejected, without explanation, because I had included spaces every five digits, formatted just like the licence code printed in the documentation.

Expecting users to type in 30 digits without spacing is a little inhumane and does little to inspire confidence in the product you're going to so much trouble to install. A colleague suggested that this wasn't a programming problem, and that the user could type the code with spaces and then manually remove them before pressing "OK". I can't think of many users who would agree that such an approach was "OK".

To handle the removal of characters, the std::remove algorithm seems like the obvious candidate. It's part of the solution, but it comes with a twist. The following code almost does what we want:

```
std::remove(text.begin(), text.end(), ' ');
```

The twist is that, in spite of its name, the std::remove algorithm doesn't remove elements from the target container itself. Remember that algorithms operate on iterators rather than containers. The remove algorithm does effectively remove the given value from the range, but it does not – and cannot – change the range itself. Its return value is the end of the resulting range once all the values have been shunted up, which means that the given string needs to be resized to eliminate the leftovers:

```
std::string::iterator new_end =
  std::remove(text.begin(), text.end(), ' ');
text.erase(new_end, text.end());
```

Or, more concisely, following the principle outlined earlier for letting the compiler do the type deduction work:

```
text.erase(
  std::remove(text.begin(), text.end(), ' '),
  text.end());
```

The standard doesn't specify what's in the junk left at the end of the range after removal, so ignoring or erasing it is about the only reasonable thing you can do. This feature of remove highlights a terminology issue in the standard. It's not so much an issue of the name of the algorithm itself, although that's perhaps also an issue in this case, but of the use of the term *algorithm*. Although the standard refers to them as *algorithms*, the main header is called <algorithm> and all the STL literature refers to them as *algorithms*, function templates like remove are not algorithms.

An algorithm is a specification of control flow. Quicksort is an algorithm that specifies a particular way to sort elements in a sequence by recursive partitioning of that sequence. In addition to detailing the intent and the performance characteristics, Quicksort is a specification of the steps involved in performing this kind of sort, as opposed to any other sorting algorithm. By contrast, the std::sort function template doesn't specify an algorithm – only the requirements, both functional and non-functional, that are placed on any algorithm used to implement it.

Making code count

Another demonstration of the ability of iterators and (so-called) algorithms to reduce the amount of code involved in certain tasks is to consider the humble word counter. Under Unix, or with a set of Unix tools like Cygwin[3], you can quickly count the number of words, characters or lines in a particular text file, such as a source file with wc. In environments that are poor in simple tools, such as Windows, the alternative is to fire up a word processor and hunt down the menu option to do this for you. There's a significant mismatch between the size of the tool and the size of the problem – a sledgehammer to tap in a nail.

How difficult is it to write a word counter? If we restrict ourselves to counting space-separated words coming in on the standard input, so that to count the words in a file we redirect it on the command line, all we need is a single executable statement:

```
#include <iostream>
#include <iterator>
#include <string>

int main()
{
   typedef std::istream_iterator<std::string> in;
   std::cout << std::distance(in(std::cin), in())
             << std::endl;
   return 0;
}
```

An istream_iterator takes an input stream object, such as cin, and treats it as a sequence of values of its parameter type, in this case std::string. The values read in are space separated, and the end of the stream is indicated by a default constructed iterator. Note again the use of the typedef to simplify the local use of an otherwise long-winded name. The only other thing you need to know is that the std::distance function template measures the distance from the beginning to the end of the iterator range; that is, the number of iterations needed to get from one to the other.

To count the number of characters on the input stream, only a slight modification is required:

```
#include <iostream>
#include <iterator>

int main()
{
   typedef std::istreambuf_iterator<char> in;
   std::cout << std::distance(in(std::cin.rdbuf()), in())
             << std::endl;
   return 0;
}
```

The difference here is that an istreambuf_iterator iterates over the raw characters in the underlying stream buffer used by a formatting stream object. The rdbuf member function returns this buffer for cin, and distance now counts individual characters rather than words.

To count lines, it's not the distance from the beginning to the end of the stream that matters, but the number of newline characters:

```
#include <algorithm>
#include <iostream>
#include <iterator>

int main()
{
   typedef std::istreambuf_iterator<char> in;
   std::cout << std::count(in(std::cin.rdbuf()), in(), '\n')
             << std::endl;
   return 0;
}
```

What if, rather than taking input from the standard input, you wanted the user to list files explicitly on the command line? A first cut suggests surrounding the relevant counting code with an argument handling loop as follows:

```
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>

int main(int argc, char *argv[])
{
   typedef std::istreambuf_iterator<char> in;
   for(int arg = 1; arg < argc; ++arg)
   {
      std::ifstream file(argv[arg]);
      std::cout << argv[arg] << '\t'
                << std::count(in(file.rdbuf()), in(), '\n')
                << std::endl;
   }
   return 0;
}
```

With a little extra decomposition, however, we can do better than this. What's the intent of the code? For each named file, the lines in that file are counted and reported back to the user, so we can break up the solution along the following lines:

```
#include <algorithm>
#include <fstream>
#include <iostream>
#include <iterator>

void count_file(const char *filename)
{
   typedef std::istreambuf_iterator<char> in;
   std::ifstream file(filename);
   std::cout << filename << '\t'
             << std::count(in(file.rdbuf()), in(), '\n')
             << std::endl;
}

int main(int argc, char *argv[])
{
   std::for_each(argv + 1, argv + argc, count_file);
   return 0;
}
```

It's left as an exercise for the reader to improve the error checking and to default the input to the standard input stream in the absence of any arguments.

Less code, more software

To make a genuine claim that you're using the STL requires more than just the use of a few container types. Iterators and so-called algorithms are the screws and screwdrivers of generic programming, while the standard library, with containers (and function objects) plays the supporting role, rather than vice versa.

The generic programming approach allows us to be less distracted by mechanism and to focus more on the intent of the small tasks in a program. The basic message is a minimalist one – less code, more software. We can achieve a greater functional effect for code if we complement the traditional, coarse granularity of OO design methods – which often lead to a lot of wasted effort if followed through to a more detailed level – with the finer abstractions of generic programming. ■

References

1. Henney, Kevlin, "The Miseducation of C++", *Application Development Advisor* 5(3), April 2001
2. Henney, Kevlin, "Distinctly Qualified", *C/C++ Users Journal* online, www.cuj.com/experts/1905/henney.htm, May 2001
3. Cygwin, www.cygwin.com