THE EXPERT'S VOICE® IN C++

Using the C++ Standard Template Libraries

Ivor Horton

Apress®

Using the C++ Standard Template Libraries

Ivor Horton

Apress®

Using the C++ Standard Template Libraries

ISBN-13 (pbk): 978-1-4842-0005-6

ISBN-13 (electronic): 978-1-4842-0004-9

Managing Director: Welmoed Spahr
Lead Editor: Steve Anglin
Technical Reviewer: Marc Gregoire
Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Louise Corrigan, Morgan Ertel, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh
Coordinating Editor: Mark Powers
Copy Editor: Karen Jameson
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

This book is for my dear wife, Eve.

Contents at a Glance

Contents

About the Author

Ivor Horton graduated as a mathematician and was lured into information technology with promises of great rewards for very little work. In spite of the reality being a great deal of work for relatively modest rewards, he has continued to work with computers to the present day. He has been engaged at various times in programming, systems design, consultancy, and the management and implementation of projects of considerable complexity.

Ivor has many years of experience in designing and implementing systems for engineering design and manufacturing control. He has developed occasionally useful applications in a wide variety of programming languages, and has taught primarily scientists and engineers to do likewise. His currently published works include tutorials on C, C++, and Java. At the present time, when he is not writing programming books or providing advice to others, he spends his time fishing, traveling, and enjoying life in general.

About the Technical Reviewer

Marc Gregoire is a software engineer from Belgium. He graduated from the Catholic University of Leuven, Belgium, with a degree in "Burgerlijk ingenieur in de computer wetenschappen" (equivalent to master of science in engineering in computer science). The year after, he received the cum laude degree of master in artificial intelligence at the same university. After his studies, Marc started working for a software consultancy company called Ordina Belgium. As a consultant, he worked for Siemens and Nokia Siemens Networks on critical 2G and 3G software running on Solaris for telecom operators. This required working in international teams spanning from South America and the United States to Europe, the Middle East, Africa, and Asia. Now, Marc is working for Nikon Metrology on 3D laser scanning software.

His main expertise is C/C++, specifically Microsoft VC++ and the MFC framework. Next to C/C++, Marc also likes C# and uses PHP for creating web pages. In addition to his main interest of Windows development, he also has experience in developing C++ programs running 24/7 on Linux platforms (e.g., EIB home automation software).

Since April 2007, he has received the yearly Microsoft MVP (Most Valuable Professional) award for his Visual C++ expertise.

Marc is the founder of the Belgian C++ Users Group (www.becpp.org) and an active member on the CodeGuru forum (as Marc G). He also creates freeware and shareware programs that are distributed through his web site at www.nuonsoft.com, and maintains a blog on www.nuonsoft.com/blog/.

Acknowledgments

I'd like to thank Mark Powers and the rest of the Apress editorial and production teams for their help and support throughout. I would especially like to thank Marc Gregoire for his usual outstanding technical review. His many comments and suggestions have undoubtedly made the book much better that it otherwise would be.

Introduction

Welcome to *Using the C++ Standard Template Libraries*. This book is a tutorial on the class and function templates that are contained within a subset of the header files that make up the C++ Standard Library. These are generic programming tools that offer vast capability, are easy to use, and make many things simple to implement that would otherwise be difficult. The code they generate is usually more efficient and reliable than you could write yourself.

I'm usually unhappy with explanations of just what things *do*, without an elaboration of what things are *for*. It's often difficult to guess the latter from the former. My approach therefore, is not just to explain the functionality of the class and function templates, but as far as possible to show how you apply them in a practical context. This leads to some sizeable chunks of code at some points, but I believe you'll think that it's worth it.

The collection of header files from the C++ Standard Library that are the subject of this book have often been referred to in the past as the *Standard Template Library* or simply the *STL*. I'll use "the STL" in the book as a convenient shorthand to mean the set of headers containing templates that I discuss in the book. Of course, there's really no such thing as the STL - the C++ Language Standard doesn't mention it so formally it doesn't exist. In spite of the fact that it is undefined, most C++ programmers know roughly what is meant by the STL. It's been around in various guises for a long time.

The idea of generic programming that is embodied in the STL originated with Alexander Stepanov back in 1979 - long before there was a standard for the C++ language. The first implementation of the STL for C++ was originated by Stepanov and others around 1989 working at Hewlett Packard, and this STL implementation was complementary to the libraries that were provided with C++ compilers at that time. The capability offered by the STL was first considered for incorporation into the first proposed C++ language standard in the early 1990s, and the essentials of the STL made it into the first language standard for C++ that was published in 1998. Since then the generic programming facilities that the STL represents have been improved and extended, and templates are to be found in many header files that are not part of what could be called the STL. All the material in the book relates to the most recently approved language standard at the time of writing, which is C++ 14.

The STL is not a precise concept and this book doesn't cover all the templates in the C++ Standard Library. Overall, the book describes and demonstrates the templates from the Standard Library that I think should be a first choice for C++ programmers to understand, especially those who are relatively new to C++. The primary Standard Library header files that are discussed in depth include:

For data containers:	`<array>`, `<vector>`, `<deque>`, `<stack>`, `<queue>`, `<list>`, `<forward_list>`, `<set>`, `<unordered_set>`, `<map>`, `<unordered_map>`
For iterators:	`<iterator>`
For algorithms:	`<algorithm>`
For random numbers and statistics:	`<random>`
For processing numerical data:	`<valarray>`, `<numeric>`
For time and timing:	`<ratio>`, `<chrono>`
For complex numbers:	`<complex>`

Templates from other headers such as `<pair>`, `<tuple>`, `<functional>`, and `<memory>` also get dragged in to the book at various points. The templates for *data containers* are fundamental; these will be useful in the majority of applications. Iterators are a basic tool for working with containers so they are included also. *Algorithms* are function templates that operate on data stored in containers. These are powerful tools that you can also apply to arrays and they are described and illustrated with working examples. I have included a chapter that explains the templates that relate to random number generation and statistics. While some of these are quite specialized, many are widely applicable in simulations, modeling, and games programs. The templates for compute-intensive numerical data processing are discussed, and those relating to time and timing. Finally, there's a brief introduction to the class templates for working with complex numbers.

Prerequisites for Using the Book

To understand the contents of this book you need to have a basic working knowledge of the C++ language. This book complements my *Beginning C++* book, so if you have worked through that successfully, you're in good shape to tackle this. A basic understanding of what class templates and function templates are, and how they work is essential, and I have included an overview of the basics of these in Chapter 1. If you are not used to using templates, the syntax can give the impression that they are lot more complicated than they really are. Once you get used to the notation, you'll find them relatively easy to work with. Lambda expressions are also used frequently with the STL so you need to be comfortable with those too.

You'll need a C++ 14-compliant compiler and of course a text editor suitable for working with program code. There has been quite a renaissance in C++ compiler development in recent years, and there are several excellent compilers out there that are largely in conformance with C++ 14, in spite of it being a recently approved standard. There are at least three available that you can use without charge:

- GCC is the GNU compiler collection that supports C++, C, and Fortran as well as other languages. GCC supports all the C++ 14 features used in this book. You can download GCC from gcc.gnu.org. The GCC compiler collection works with GNU and Linux, but there's a version for Microsoft Windows that you can download from `www.mingw.org`.
- The *ideaone* online compiler supports C++ 14 and is accessible through *ideaone.com*. The compiler it uses for C++ 14 is GCC 5.1 at the time of writing. *ideaone.com* also supports a wide range of other programming languages, including C, Fortran, and Java.
- The *Microsoft Visual Studio 2015 Community Edition* runs under the Microsoft Windows operating system. It supports C++ as well as several other programming languages and comes with a complete development environment.

Using the Book

For the most part, I have organized the material in this book to be read sequentially, so the best way to use the book is to start at the beginning and keep going until you reach the end. Generally, I try not to use capabilities before I have explained them. Once I have explained something, I plug it in to subsequent material whenever it makes sense to do so, which is why I recommend going through the chapters sequentially. There are a few topics that require some understanding of the underlying mathematics, and I have included the maths in these instances. If you are not comfortable with the maths, you can skip these without limiting your ability to understand what follows.

No one ever learned programming by just reading a book. You'll only learn how to use the STL by writing code. I strongly recommend that you key in all the examples – don't just copy them from the download files – and compile and execute the code that you've keyed in. This might seem tedious at times, but it's surprising how much just typing in program statements will help your understanding, especially when you may feel you're struggling with some of the ideas. It will help you remember stuff, too. If an example doesn't work, resist the temptation to go straight back to the book to see why. Try to figure out from your code what is wrong.

Throughout the chapters there are code fragments that are executable for the most part if the appropriate header files are included. Generally you can execute them and get some output if you put them in the `main( )` function. I suggest you set up a program project for this purpose. You can copy the code into an empty definition for `main( )` and just add further `#include` directives for the header files that are required as you go along. You'll need to delete previous code fragments most of the time to prevent name conflicts.

Making your own mistakes is a fundamental part of the learning process and the exercises should provide you with ample opportunity for that. The more mistakes that you make and that you are able to find and fix, the greater the insight you'll have into what can and does go wrong using the templates. Make sure you complete all the exercises that you can, and don't look at the solutions until you're sure that you can't work it out yourself. Many of these exercises just involve a direct application of what's covered in a chapter – they're just practice, in other words – but some also require a bit of thought or maybe even inspiration.

I wish you every success with the STL. Above all, enjoy it!

—Ivor Horton

CHAPTER 1

Introducing the Standard Template Library

This chapter explains the fundamental ideas behind the Standard Template Library (STL). This is to give you an overall grasp of how the various types of entities in the STL hang together. You'll see more in-depth examples and discussion of everything that I introduce in this chapter in the book. In this chapter you'll learn the following:

- What is in the STL
- How *templates* are defined and used
- What a *container* is
- What an *iterator* is and how it is used
- The importance of *smart pointers* and their use with containers
- What *algorithms* are and how you apply them
- What is provided by the *numerics* library
- What a *function object* is
- How *lambda expressions* are defined and used

Besides introducing the basic ideas behind the STL, this chapter provides brief reminders of some C++ language features that you need to be comfortable with because they will be used frequently in subsequent chapters. You can skip any of these sections if you are already familiar with the topic.

Basic Ideas

The STL is an extensive and powerful set of tools for organizing and processing data. These tools are all defined by *templates* so the data can be of any type that meets a few minimum requirements. I'm assuming that you are reasonably familiar with how class templates and function templates can be defined and how they are used, but I'll remind you of the essentials of these in the next section. The STL can be subdivided into four conceptual libraries:

- *The Containers Library* defines containers for storing and managing data. The templates for this library are defined within the following header files: `array`, `vector`, `stack`, `queue`, `deque`, `list`, `forward_list`, `set`, `unordered_set`, `map`, and `unordered_map`.
- *The Iterators Library* defines iterators, which are objects that behave like pointers and are used to reference sequences of objects in a container. The library is defined within a single header file, `iterator`.
- *The Algorithms Library* defines a wide range of algorithms that can be applied to a set of elements stored in a container. The templates for this library are defined in the `algorithm` header file.
- *The Numerics Library* defines a wide range of numerical functions, including numerical processing of sets of elements in a container. The library also includes advanced functions for random number generation. The templates for this library are defined in the headers `complex`, `cmath`, `valarray`, `numeric`, `random`, `ratio`, and `cfenv`. The `cmath` header has been around for a while, but it has been extended in the C++ 11 standard and is included here because it contains many mathematical functions.

Many complex and difficult tasks can be achieved very easily with remarkably few lines of code using the STL. For instance, without explanation, here's the code to read an arbitrary number of floating-point values from the standard input stream and calculate and output the average:

```
std::vector<double> values;
std::cout << "Enter values separated by one or more spaces. Enter Ctrl+Z to end:\n ";
values.insert(std::begin(values), std::istream_iterator<double>(std::cin),
                                        std::istream_iterator<double>());
std::cout << "The average is "
          << (std::accumulate(std::begin(values), std::end(values), 0.0)/values.size())
          << std::endl;
```

It requires only four statements! Long lines admittedly, but no loops are required; it's all taken care of by the STL. This code can be easily modified to do the same job with data from a file. Because of the power and wide applicability of the STL, it's a *must* for any C++ programmer's toolbox. All STL names are in the `std` namespace so I won't always qualify STL names explicitly with `std` in the text. Of course, in any code I will qualify names where necessary.

Templates

A *template* is a parametric specification of a set of functions or classes. The compiler can use a template to generate a specific function or class definition when necessary, which will be when you use the function template or class template type in your code. You can also define templates for parameterized type aliases. Thus a template is not executable code – it is a blueprint or recipe for creating code. A template that is

never used in a program is ignored by the compiler so no code results from it. A template that is not used can contain programming errors, and the program that contains it will still compile and execute; errors in a template will not be identified until the template is used to create code that is then compiled.

A function or class definition that is generated from a template is an *instance* or an *instantiation* of the template. Template parameter values are usually data types, so a function or class definition can be generated for a parameter value of type `int`, for example, and another definition with a parameter value of type `string`. Parameter arguments are not necessarily types; a parameter specification can be an integer type that requires an integer argument. Here's an example of a very simple function template:

```
template <typename T> T& larger(T& a, T& b)
{
    return a > b ? a : b;
}
```

This is a template for functions that return the larger of the two arguments. The only limitation on the use of the template is that the type of the arguments must allow a `>` comparison to be executed. The type parameter `T` determines the specific instance of the template to be created. The compiler can deduce this from the arguments you supply when you use `larger()`, although you can supply it explicitly. For example:

```
std::string first {"To be or not to be"};
std::string second {"That is the question."};
std::cout << larger(first, second) << std::endl;
```

This code requires the `string` header to be included. The compiler will deduce the argument for `T` as type `string`. If you want to specify it, you would write `larger<std::string>(first, second)`. You would need to specify the template type argument when the function arguments differ in type. If you wrote `larger(2, 3.5)`, for example, the compiler cannot deduce `T` because it is ambiguous – it could be type `int` or type `double`. This usage will result in an error message. Writing `larger<double>(2, 3.5)` will fix the problem.

Here's an example of a class template:

```
template <typename T> class Array
{
private:
  T* elements;                                  // Array of type T
  size_t count;                                 // Number of array elements

public:
  explicit Array(size_t arraySize);             // Constructor
  Array(const Array& other);                    // Copy Constructor
  Array(Array&& other);                         // Move Constructor
  virtual ~Array();                             // Destructor
  T& operator[](size_t index);                  // Subscript operator
  const T& operator[](size_t index) const;      // Subscript operator-const arrays
  Array& operator=(const Array& rhs);           // Assignment operator
  Array& operator=(Array&& rhs);                // Move assignment operator
  size_t size() { return count; }               // Accessor for count
};
```

The `size_t` type alias is defined in the `cstddef` header and represents an unsigned integer type. This code defines a simple template for an array of elements of type `T`. Where `Array` appears in the template definition `Array<T>` is implied and you could write this if you wish. Outside the body of the template – in

an external function member definition, you must write `Array<T>`. The assignment operator allows one `Array<T>` object to be assigned to another, which is something you can't do with ordinary arrays. If you wanted to inhibit this capability, you would still need to declare the `operator=()` function as a member of the template. If you don't, the compiler will create a public default assignment operator when necessary for a template instance. To prevent use of the assignment operator, you should specify it as deleted – like this:

```
Array& operator=(const Array& rhs)=delete;     // No assignment operator
```

In general, if you need to define any of a copy or move constructor, a copy or move assignment operator, or a destructor, you should define all five class members, or specify the ones you don't want as deleted.

■ **Note** A class that implements a move constructor and a move assignment operator is said to have *move semantics*.

The `size()` member is implemented within the class template so it's `inline` by default and no external definition is necessary. External definitions for function members of a class template are themselves templates that you put in a header file – usually the same header file as the class template. This is true even if a function member has no dependence on the type parameter `T`, so `size()` would need a template definition if it was not defined inside the class template. The type parameter list for a template that defines a function member must be identical to that of the class template. Here's how the definition of the constructor might look:

```
template <typename T>                          // This is a function template with parameter T
Array<T>::Array(size_t arraySize) try : elements {new T[arraySize]}, count {arraySize}
{}
catch(const std::exception& e)
{
  st::cerr << "Memory allocation failure in Array constructor." << std::endl;
  rethrow e;
}
```

The memory allocation for elements could throw an exception so the constructor is a function `try` block. This allows the exception to be caught and responded to but the exception must be rethrown – if you don't `rethrow` the exception in the `catch` block, it will be rethrown anyway. The template type parameter is essential in the qualification of the constructor name because it ties the function template definition to the class template. Note that you *don't* use the `typename` keyword in the qualifier for the member name; it's only used in the template parameter list.

Of course, you can specify an external template for a function member of a class template as `inline` – for example, here's how the copy constructor for the `Array` template might be defined:

```
template <typename T>
inline Array<T>::Array(const Array& other)
try : elements {new T[other.count]}, count {other.count}
{
  for (size_t i {}; i < count; ++i)
    elements[i] = other.elements[i];
}
catch (std::bad_alloc&)
{
  std::cerr << "memory allocation failed for Array object copy." << std:: endl;
}
```

This assumes that the assignment operator works for type T. Without seeing the code for a template before you use it, you may not realize the dependency on the assignment operator. This demonstrates how important it is to *always* define the assignment operator along with the other members I mentioned earlier for classes that allocate memory dynamically.

■ **Note** The `class` and `typename` keywords are interchangeable when specifying template parameters so you can write either `template<typename T>` or `template<class T>` when defining a template. Because `T` is not necessarily a class type, I prefer to use `typename` because I feel this is more expressive of the possibility that a template type argument can be a fundamental type as well as a class type.

The compiler instantiates a class template as a result of a definition of an object that has a type produced by the template. Here's an example:

```
Array<int> data {40};
```

An argument for each class template type parameter is always required, unless there is a default argument. When this statement is compiled, three things happen: the definition for the `Array<int>` class is created so that the type is identified, the constructor definition is generated because it must be called to create the object, and the destructor is created because it's needed to destroy the object. That's all the compiler needs to create and destroy the `data` object so this is the only code that it generates from the templates at this point. The class definition is generated by substituting `int` in place of `T` in the template definition, but there's one subtlety. The compiler *only* compiles the member functions that the program *uses*, so you don't necessarily get the entire class that would result from a simple substitution of the argument for the template parameter. On the basis of the definition for the `data` object, the class will be defined as:

```
class Array<int>
{
private:
  int* elements;
  size_t count;

public:
  explicit Array(size_t arraySize);
  virtual ~Array();
};
```

You can see that the only function members are the constructor and the destructor. The compiler won't create instances of anything that isn't required to create the object, and it won't include parts of the template that aren't needed in the program.

You can define templates for type aliases. This can be useful when you are working with the STL. Here's an example of a template for a type alias:

```
template<typename T> using ptr = std::shared_ptr<T>;
```

This template defines `ptr<T>` to be an alias for the smart pointer template type `std::shared_ptr<T>`. With this template in effect you can use `ptr<std::string>` in your code instead of `std::shared_ptr<std::string>`. It's clearly less verbose and easier to read. The following `using` directive will simplify it further:

```
using std::string;
```

Now you can use `ptr<string>` in your code instead of `std::shared_ptr<std::string>`. Templates for type aliases can make your code easier to understand and much easier to type.

The Containers

Containers are the bedrock of the STL capabilities because most of the rest of the STL relates to them. A container is an object that stores and organizes other objects in a particular way. When you use a container you'll inevitably be using *iterators* to access that data so you'll need a good understanding of those too. The STL provides several categories of container:

- *Sequence containers* store objects in a linear organization, similar to an array, but not necessarily in contiguous memory. You can access the objects in a sequence by calling a function member or through an iterator; in some cases you can also use the subscript operator with an index.
- *Associative containers* store objects together with associated keys. You retrieve an object from an associative container by supplying its associated key. You can also retrieve the objects in an associative container using an iterator.
- *Container adapters* are adapter class templates that provide alternative mechanisms for accessing data stored in an underlying sequence container, or associative container.

It's important to appreciate that unless the objects are rvalues – temporary objects – of a type that has move semantics, all the STL containers store *copies* of the objects that you store in them. The STL also requires that the move constructor and assignment operator must be specified as `noexcept`, which indicates they do not throw exceptions. If you add an object of a type that does not have move semantics to a container and modify the original, the original and the object in the container will be different. However, when you retrieve an object, you get a reference to the object in the container so you can modify stored objects. The copies that are stored are created using the copy constructor for the type of object. For some objects, copying can be a process that carries a lot of overhead. In this case, it will be better to either store pointers to the objects in the container, or to move objects into the container assuming that move semantics have been implemented for the type.

■ **Caution** Don't store derived class objects in a container that stores elements of a base class type. This will cause slicing of the derived class objects. If you want to access derived class objects in a container with a view to obtaining polymorphic behavior, store pointers to the objects in a container that stores base class pointers – or better still – smart pointers to the base type.

Containers store the objects they hold on the heap and manage the space they occupy automatically. The allocation of space in a container storing objects of type `T` is managed by an *allocator*, and the type of the allocator is specified by a template parameter. The default type argument is `std::allocator<T>`, and an object of this type is an allocator that allocates heap memory for objects of type `T`. This provides the possibility for you to supply your own allocator. You might want to do this for performance reasons, but this is rarely necessary and most of the time the default allocator is fine. Defining an allocator is an advanced subject and

I won't be discussing it further in this book. I'll therefore omit the last template parameter for template types when it represents an allocator. The `std::vector<typename T, typename Allocator>` template has a default value for `Allocator` specified as `std::allocator<T>` so I'll write this as `std::vector<typename T>`. This explanation is just so you'll know the option to provide an allocator is there.

A type `T` must meet certain requirements if `T` objects are to be stored in a container, and these requirements ultimately depend on the operations you need to perform on the elements. A container will usually need to copy elements and may need to move and interchange elements. The bare minimum for type `T` objects to be stored in a container in this case looks like this:

```
class T
{
  public:
   T();                                    // default constructor
   T(const T& t);                          // Copy constructor
   ~T();                                   // Destructor
   T& operator=(const T& t);               // Assignment operator
};
```

Considering that the compiler provides default implementations for all the members above in many circumstances, most class types should meet these requirements. Note that `operator<()` hasn't been included in the definition for `T`, but objects of a type without `operator<()` defined will not be usable as keys in any of the associative containers such as `map` and `set`, and the ordering algorithms such as `sort()` and `merge()` cannot be applied to sequences where the elements do not support the less-than operation.

■ **Note** If the type of your objects does not meet the requirements of a container that you are using, or you misuse the container template in some other way, you will often get compiler error messages relating to code that is deep in a Standard Library header file. When this occurs, don't rush to report errors in the Standard Library. Look for errors in your code that is using the STL!

Iterators

An *iterator* is an object of a class template type that behaves like a pointer. As long as an iterator, `iter`, points to a valid object you can dereference it to obtain a reference to the object by writing `*iter`. If `iter` points to a class object you can access a member, `member`, of the object by writing `iter->member`. Thus you use iterators just like pointers.

You use iterators to access the elements in a container when you are processing them in some way, and in particular when you are applying an STL algorithm. Thus iterators connect algorithms to the elements in a container regardless of the type of the container. Iterators decouple the algorithm from the data source; an algorithm has no knowledge of the container from which the data originates. Iterators are instances of template types that are defined in the `iterator` header, but this header is included by all of the headers that define containers.

You typically use a pair of iterators to define a *range* of elements; the elements can be objects in a container, elements in a standard array, characters in a `string` object, or elements in any other type of object that supports iterators. A *range* is a sequence of elements that is specified by a *begin iterator* that points to the first element in the range, and an *end iterator* that points to *one past the last* element. Even when the sequence is a subset of the elements in a container, the second iterator still points to *one past the last element* in the sequence – *not* the last element in the range. The end iterator for a range that represents all the

elements in a container will not point to anything and therefore cannot be dereferenced. Iterators provide a standard mechanism for identifying a range of elements in the STL, and elsewhere. The specification of a range of elements is independent from where the elements originate so a given algorithm can be applied to a range of elements from *any* source as long as the iterators meet the requirements of the algorithm. I'll have more to say about the characteristics of different kinds of iterators later.

Once you understand how iterators work, it's easy to define your own template functions to process data sequences that are specified by iterators as arguments. Instances of your function templates can then be applied to data from any source that can be defined as a range; the code will work just as well with data from an array as it does with data from a vector container. You'll see examples of this in action later in the book.

Obtaining Iterators

You can obtain iterators from a container by calling the `begin()` and `end()` function members of the container object; these return iterators that point to the first element and one past the last element respectively. The iterator that the `end()` member of a container returns does not point to a valid element so you can't dereference it or increment it. The string classes such as `std::string` also have these function members so you can obtain iterators for these, too. You can obtain the same iterators as those returned by the `begin()` and `end()` function members of a container by calling the global functions `std::begin()` and `std::end()` with the container object as the argument; these are defined by templates in the `iterator` header. The global `begin()` and `end()` functions work with an ordinary array or a `string` object as the argument and therefore offer a uniform way of obtaining iterators.

Iterators allow you to step through the elements in a range by incrementing the begin iterator to move from one object to the next, as shown in Figure 1-1; 'container' in the figure implies a `string` object or an array, as well as an STL container. By comparing the incremented `begin` iterator with the `end` iterator you can determine when the last element has been reached. There are other operations you can apply to iterators, but this depends on the type of iterator, which in turn depends on the kind of container you are using. There are global `cbegin()` and `cend()` functions that return `const` iterators for array, containers, or `string` objects. Remember—a `const` iterator points to something that is constant and you can still modify the iterator itself. I'll introduce other global functions that return other kinds of iterators later in this section.

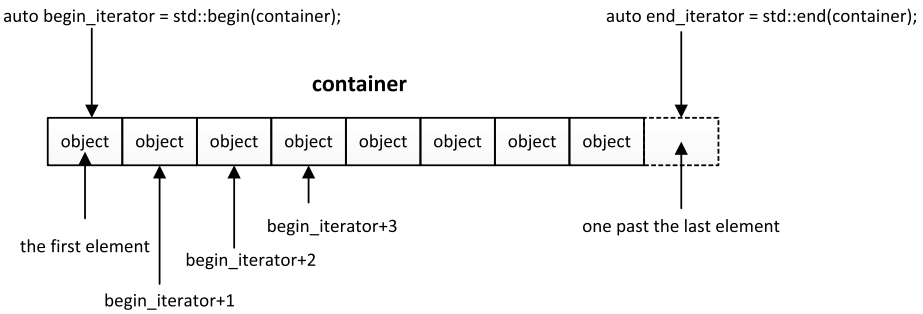

Figure 1-1. *Operation of iterators*

Iterator Categories

All iterator types must have a copy constructor, a copy assignment operator, and a destructor. The objects that an iterator point to must be *swappable*; I'll explain what this implies further in the next chapter. There are five *categories* of iterators that reflect different levels of capability. Different algorithms may require different levels of capability for the iterators that identify the range of elements they are to operate on. The categories are not new iterator template types; the category that an iterator type supports is identified by an argument value for a type parameter for the `iterator` template. I'll explain more about this a little later in this section.

The category of the iterators you get for a container depends on the type of the container. The categories enable an algorithm to determine the capabilities of the iterators that you pass to it. An algorithm can use the category of an iterator argument in two ways: first, it can establish that the minimum functional requirements for the operation are met; and second, if the minimum requirement for iterators is exceeded, the algorithm may use the extended capability to carry out the operation more efficiently. Of course, algorithms can only be applied to elements in containers that provide iterators with the required level of capability.

The iterator categories are as follows, ordered from the simplest to the most complex:

1. *Input iterators* have read access to objects. If `iter` is an input iterator, it must support the expression `*iter` to produce a reference to the value to which `iter` points. Input iterators are single use only, which means that once an iterator has been incremented, to access the previous element that it pointed to you need a new iterator. Each time you want to read a sequence, you must create a new iterator. The operations that you can apply to input iterators are: `++iter` or `iter++`; `iter1==iter2` and `iter1!=iter2`; and `*iter` Note the absence of the decrement operator. You can use the expression `iter->member` for input iterators.

2. *Output iterators* have write access to objects. If `iter` is an output iterator, it allows a new value to be assigned so `*iter=new_value` is supported. Output iterators are single use only. Each time you want to write a sequence, you must create a new iterator. The operations that you can apply to output iterators are: `++iter` or `iter++`; and `*iter` Note the absence of the decrement operator. You only get write access with output iterators. You *cannot* use the expression `iter->member` for output iterators.

3. *Forward iterators* combine the capabilities of input and output iterators and add the capability to be used more than once. Therefore you can reuse a forward iterator to read or write an element as many times as necessary. The operation to be performed determines when forward iterators are required. The `replace()` algorithm that searches a range and replaces elements requires the capability of a forward iterator, for example, because the iterator that points to an element that is to be replaced is reused to overwrite it.

4. *Bidirectional iterators* provide the same capabilities as forward iterators but allow traversal through a sequence backward as well as forward. Therefore in addition to incrementing these iterators to move to the next element, you can apply the prefix and postfix decrement operators, `--iter` and `iter--`, to move to the previous element.

5. *Random access iterators* provide the same capabilities as bidirectional iterators but also allow elements to be accessed at random. In addition to the operations permitted for bidirectional iterators, these support the following operations:

   - Incrementing and decrementing by an integer: `iter+n` or `iter-n` and `iter+=n` or `iter-=n`
   - Indexing by an integer: `iter[n]`, which is equivalent to `*(iter+n)`
   - The difference between two iterators: `iter1-iter2`, which results in an integer specifying the number of elements.
   - Comparing iterators: `iter1<iter2`, `iter1>iter2`, `iter1<=iter2`, and `iter1>=iter2`.

   Sorting a range of elements will require the range to be specified by random access iterators.
   You can use the subscript operator with random access iterators. Given an iterator, `first`, the expression `first[3]` is equivalent to `*(first+3)` so it accesses the fourth element. In general, in the expression `iter[n]` with an iterator, `iter`, `n` is an offset and the expression returns a reference to the element at offset `n` from `iter`. Note that the index you use with the subscript operator applied to an iterator is not checked. There is nothing to prevent the use of index values outside the legal range.

Each iterator category is identified by an empty class called an *iterator tag class* that is used as a type argument to the `iterator` template. The sole purpose of the iterator tag classes is to specify what a particular iterator type can do so they are used as an `iterator` template type argument. The standard iterator tag classes are:

```
input_iterator_tag
output_iterator_tag
forward_iterator_tag which is derived from input_iterator_tag
bidirectional_iterator_tag which is derived from forward_iterator_tag
random_access_iterator_tag which is derived from bidirectional_iterator_tag
```

The inheritance structure for these classes reflects the cumulative nature of the iterator categories. When an `iterator` template instance is created, the first template type argument will be one of the iterator tag classes, which will determine the capabilities of the iterator. In chapter 2 I'll explain how you can define your own iterators and how you specify their category.

If an algorithm requires an iterator of a given category, then you can't use an inferior iterator; however you can always use a superior iterator. The forward, bidirectional, and random access iterators can also be *constant* or *mutable*, depending on whether dereferencing the iterator produces a reference, or a `const` reference. Obviously you can't use the result of dereferencing a `const` iterator on the left of an assignment.

The characteristics of the iterators that you get for a container depend on the container type. For example, `vector` and `deque` containers provide random access iterators; this reflects the fact that the elements in these containers can be accessed randomly. On the other hand the `list` and `map` containers always supply bidirectional iterators; these containers don't support random access to elements. Input and output iterator and forward iterator types are typically used to specify parameters for algorithms to reflect the minimum level of capability required by the algorithm. I'll be explaining iterators further with working examples in the context of applying algorithms to the contents of containers later in the book – they