IN THIS CHAPTER

- Working with container functions
- Performing random access with iterator functions
- Working with algorithms and utilities
- Creating random numbers with functions
- Creating temporary buffers with allocators

Chapter 1

Exploring the Standard Library Further

The Standard Library is one of the most important parts of the C++ developer's toolkit because it contains a host of interesting functions that let you write great applications. The Standard Library originally started as the Standard Template Library (STL), and a number of companies, including Silicon Graphics, Inc. (SGI) and IBM, distributed it for everyone to use. The International Standards Organization (ISO) eventually took over STL, made a few minor changes to it, added some additional features, and renamed it the Standard Library.

REMEMBER

The STL and the Standard Library are two separate entities. As the Standard Library has grown, it has also become more different from the STL. Consequently, when you work with C++ today, you likely work with the Standard Library and, to avoid confusion, shouldn't refer to it as the STL. In fact, the Standard Library and the STL use different headers — again, to avoid confusion (you can see a heading listing at `https://en.cppreference.com/w/cpp/header`). The site at `https://www.tutorialspoint.com/What-s-the-difference-between-STL-and-Cplusplus-Standard-Library` provides a short discussion of the differences between the Standard Library and STL that includes additional details.

GETTING A COPY OF THE STANDARD LIBRARY DOCUMENTATION

The Standard Library is incredibly large, so this book doesn't document it completely. The Code::Blocks product doesn't come with a Standard Library reference either. However, to really use the Standard Library, you really do need a copy of the documentation.

You can join ISO for a bazillion bucks and get a copy of its document for free, or you can purchase a copy of it from `https://www.iso.org/standard/68564.html`. Note that the price shown is in Swiss Francs, so you need to consider the exchange rate. As an alternative, you can buy a copy of the Standard Library documentation from an ISO member, such as the American National Standards Institute (ANSI). Check it out at `https://webstore.ansi.org/` (simply type ISO/IEC 14882:2017 in the search field).

Because many common STL elements and the Standard Library are relatively close, you have a third alternative: Use an STL resource. One of the best written and easiest-to-use resources is from SGI at `http://www.martinbroadhurst.com/sgi-stl-documentation.html`. The downside to using the STL documentation is that it doesn't contain information about newer features found only in the Standard Library.

In addition to the resources mentioned so far, you'll want to check out Bjarne Stroustrup's website at `http://www.stroustrup.com/#standard`. Just in case you don't know, he's the guy who designed and originally implemented C++.

This chapter offers an overview of the Standard Library and shows you some examples of how to use it. However, if you don't see what you want here, don't worry; later chapters have additional examples, and you can always refer to the Standard Library documentation for even more examples. Before the chapter moves on to any examples, however, you need to know what the Standard Library contains, so the first section of this chapter gives you a list of Standard Library function categories.

REMEMBER

You don't have to type the source code for this chapter manually. In fact, using the downloadable source is a lot easier. You can find the source for this chapter in the `\CPP_AIO4\BookVII\Chapter01` folder of the downloadable source. See the Introduction for details on how to find these source files.

Considering the Standard Library Categories

The Standard Library documentation uses a formal approach that you're going to find difficult to read and even harder to understand; it must have been put together by lawyers more interested in the precise meaning of words rather than the usability of the document. This immense tome (1,300+ pages) requires quite a bit of time to review. Fortunately, you don't have to wade through all that legal jargon mixed indiscriminately with computer jargon and the occasional bit of English. This chapter provides the overview you need to get going quickly.

The best way to begin is to break the Standard Library into smaller pieces. You can categorize the Standard Library functions in a number of ways. One of the most common approaches is to use the following categories:

Algorithms	Atomic Operations (C++ 11 and above)	C Compatibility
Concepts (C++ 20 and above)	Containers	Coroutines (C++ 20 and above)
Filesystem (C++ 17 and above)	Input/Output	Iterators
Localization	Numerics	Ranges (C++ 20 and above)
Regular Expressions (C++ 11 and above)	Strings	Thread Support (C++ 11 and above)
Utilities		

Note that this table doesn't include the *Standard Library Extensions*, additions that add specialized (non-general) functionality, which appear at `https://en.cppreference.com/w/cpp/experimental/lib_extensions_2`. The following sections provide a brief description of each of these categories and tell what you can expect to find in them. Knowing the category can help you locate the function you need quickly on websites that use these relatively standard category names.

Algorithms

Algorithms perform data manipulations such as replacing, locating, or sorting information. You've already seen some algorithms used in the book because it's hard to create a substantial application without using one. There aren't any types in the Algorithms category. The following is a list of common algorithm functions

(functions removed since C++ 11 and above don't appear in the list even if you can use them in an older version of C++):

adjacent_find	all_of (C++ 11 and above)	any_of (C++ 11 and above)
binary_search	clamp (C++ 17 and above)	copy
copy_backward	copy_if (C++ 11 and above)	copy_n (Updated C++ 11)
count	count_if	equal
equal_range	fill	fill_n
find	find_end	find_first_of
find_if	find_if_not (C++ 11 and above)	for_each
for_each_n (C++17 and above)	generate	generate_n
includes	inplace_merge	is_heap (Updated C++ 11)
is_heap_until (C++ 11 and above)	is_partitioned (C++ 11 and above)	is_permutation (C++ 11 and above)
is_sorted (Updated C++ 11)	is_sorted_until (C++ 11 and above)	iter_swap
lexicographical_compare	lexicographical_compare_three_way (C++ 20 and above)	lower_bound
make_heap	max	max_element
merge	min	min_element
minmax (C++ 11 and above)	minmax_element (C++ 11 and above)	mismatch
move (C++ 11 and above)	move_backward (C++ 11 and above)	next_permutation
none_of (C++ 11 and above)	nth_element	partial_sort
partial_sort_copy	partition	partition_copy (C++ 11 and above)
partition_point (C++ 11 and above)	pop_heap	prev_permutation

push_heap	remove	remove_copy
remove_copy_if	remove_if	replace
replace_copy	replace_copy_if	replace_if
reverse	reverse_copy	rotate
rotate_copy	sample (C++ 17 and above)	search
search_n	set_difference	set_intersection
set_symmetric_ difference	set_union	shift_left (C++ 20 and above)
shift_right (C++ 20 and above)	shuffle (C++11 and above)	sort
sort_heap	stable_partition	stable_sort
swap	swap_ranges	transform
unique	unique_copy	upper_bound

In addition to the <algorithm> header entries found in the previous table, C++ 17 and above users have access to the <execution> header entries in the table that follows. This functionality is still part of the Algorithms category but appears in a different header. An *execution policy* determines whether your code executes in sequence (the normal approach) or in parallel. Executing code in parallel, whenever possible, makes your application run significantly faster.

is_execution_policy	parallel_policy	parallel_unsequenced_ policy
sequenced_policy	unsequenced_policy (C++ 20 and above)	

Atomic operations

An *atomic operation* is a code block that executes as a single concurrent entity without the use of locking mechanisms. There are several benefits to using atomic operations:

- Because the atomic operation is indivisible, it's free of *data races* where:
  - Two or more threads in a single process access the same memory location concurrently

- At least one of the thread accesses is for writing
- The threads don't use any exclusive locks to control their accesses to that memory

» which results in nondeterministic behavior. The changes made by the threads vary run-by-run.

» Application development is significantly easier because you don't have to manage locks.

» There is a smaller risk of data-related errors.

You must have C++ 11 or above to use this feature. The following table contains the Atomic Operations category functions.

`atomic_compare_exchange_strong`	`atomic_compare_exchange_strong_explicit`	`atomic_compare_exchange_weak`
`atomic_compare_exchange_weak_explicit`	`atomic_exchange`	`atomic_exchange_explicit`
`atomic_fetch_add`	`atomic_fetch_add_explicit`	`atomic_fetch_and`
`atomic_fetch_and_explicit`	`atomic_fetch_or`	`atomic_fetch_or_explicit`
`atomic_fetch_sub`	`atomic_fetch_sub_explicit`	`atomic_fetch_xor`
`atomic_fetch_xor_explicit`	`atomic_flag_clear`	`atomic_flag_clear_explicit`
`atomic_flag_notify_all` (C++ 20 and above)	`atomic_flag_notify_one` (C++ 20 and above)	`atomic_flag_test` (C++ 20 and above)
`atomic_flag_test_and_set`	`atomic_flag_test_and_set_explicit`	`atomic_flag_test_explicit` (C++ 20 and above)
`atomic_flag_wait` (C++ 20 and above)	`atomic_flag_wait_explicit` (C++ 20 and above)	`atomic_init` (Deprecated in C++ 20)
`atomic_is_lock_free`	`atomic_load`	`atomic_load_explicit`

atomic_notify_all (C++ 20 and above)	atomic_notify_one (C++ 20 and above)	atomic_signal_fence
atomic_store	atomic_store_explicit	atomic_thread_fence
atomic_wait (C++ 20 and above)	atomic_wait_explicit (C++ 20 and above)	kill_dependency

C Compatibility

A C compatibility header provides you with access to functionality that came with the original C language. For example, you find special math functions like `pow()` (raises a number to the given power) in the `<math.h>` header.

WARNING

All the C compatibility headers are deprecated at this point, which means you can still use them, but not for long. You should instead use the Utilities category equivalents. For example, the `<ctime>` header replaces the `<time.h>` header (note that the `<ctime>` header lacks the `.h` file extension).

Concepts

C++ 20 adds the capability to provide predicates that express a generic algorithm's expectations through *concepts*. You use a concept to formally document the constraints on a template to enforce certain behaviors. In addition, because the compiler knows the constraints at the outset, it can usually compile your application faster. The article at `https://isocpp.org/blog/2016/02/a-bit-of-background-for-concepts-and-cpp17-bjarne-stroustrup` provides more details about the potential for concepts. You could also read the fuller discussion at `https://www.stroustrup.com/good_concepts.pdf`. The following table provides a listing of Concepts category functions.

assignable_from	common_reference_with	common_with
constructible_from	convertible_to	copy_constructible
copyable	default_initializable	derived_from
destructible	equality_comparable	equality_comparable_with
equivalence_relation	floating_point	integral
invocable	movable	move_constructible

predicate	regular	regular_invocable
relation	same_as	semiregular
signed_integral	strict_weak_order	swappable
swappable_with	totally_ordered	totally_ordered_with
unsigned_integral		

Containers

Containers work just like the containers in your home — they hold something. You've already seen containers at work in other areas of this book. For example, both queues and deques are kinds of containers. The Containers category doesn't contain any functions, but it does contain a number of types including those in the following table (types are removed because C++ 11 and above don't appear in the list even if you can use them in an older version of C++).

array (C++ 11 and above)	deque	forward_list (C++ 11 and above)
list	map	queue
set	span (C++ 20 and above)	stack
unordered_map (C++ 11 and above)	unordered_set (C++ 11 and above)	vector

Coroutines

A *coroutine* is a new feature in C++ 20 that allows a function to suspend execution and resume its task later. The function stores the data needed to allow task resumption separately, rather than on the stack. This feature helps support sequential code that executes asynchronously, such as nonblocking I/O, without requiring use of callbacks. You can find an example of a coroutine at https://blog.panicsoftware.com/your-first-coroutine/. The following table contains the Coroutines category classes.

coroutine_handle	coroutine_traits	noop_coroutine_handle
noop_coroutine_promise	std::hash<std::coroutine_handle>	suspend_always
suspend_never		

Filesystem

Before C++ 17, C++ lacked the ability to perform some basic file system tasks, such as determining the existence of a path. The Filesystem category provides functionality needed to work with file systems on a local system. You can see an example of the `<filesystem>` header in use in the "Using the filesystem Library" sidebar in Book 6, Chapter 4. The article at `https://www.codingame.com/playgrounds/5659/c17-filesystem` provides additional examples. The following table lists the Filesystem category classes.

`copy_options`	`directory_entry`	`directory_iterator`
`directory_options`	`file_status`	`file_time_type`
`file_type`	`filesystem_error`	`path`
`perm_options`	`perms`	`recursive_directory_iterator`
`space_info`		

Input/Output

The Input/Output category is an old friend in this book because you see it used in every example. Not every heading in this category appears in the book, but most do in some form. The following table provides a listing of the Input/Output category headers, which make it possible to access various forms of I/O.

`cstdio`	`fstream`	`iomanip`
`ios`	`iosfwd`	`iostream`
`istream`	`ostream`	`sstream`
`streambuf`	`strstream` (Deprecated in C++ 98)	`syncstream` (C++ 20 and above)

Iterators

Iterators enumerate something. When you create a list of items and then go through that list checking items off, you're enumerating the list. Using iterators helps you create lists of items and manipulate them in specific ways. The kind of iterator you create is important because some iterators let you go forward only, some can go in either direction, and some can choose items at random. Each kind of iterator has its specific purpose.

The Iterators category includes a number of classes. These classes determine the kind of iterator you create in your code and the capabilities of that iterator. The following is a list of the iterator classes (classes removed since C++ 11 and above don't appear in the list even if you can use them in an older version of C++):

back_insert_iterator	bidirectional_iterator (handled as concept in C++ 20 and above)	bidirectional_iterator_tag
common_iterator (C++ 20 and above)	contiguous_iterator_tag (C++ 20 and above)	counted_iterator (C++ 20 and above)
default_sentinel (C++ 20 and above)	forward_iterator (handled as concept in C++ 20 and above)	forward_iterator_tag
front_insert_iterator	incremental_traits (C++ 20 and above)	indirect_result_t (C++ 20 and above)
indirectly_readable_traits (C++ 20 and above)	input_iterator (handled as concept in C++ 20 and above)	input_iterator_tag
insert_iterator	istream_iterator	istreambuf_iterator
iter_common_reference_t (C++ 20 and above)	iter_difference_t (C++ 20 and above)	iter_move (C++ 20 and above)
iter_reference_t (C++ 20 and above)	iter_rvalue_reference_t (C++ 20 and above)	iter_swap (C++ 20 and above)
iter_value_t (C++ 20 and above)	iterator (Deprecated C++ 17)	iterator_traits
move_iterator (C++ 11 and above)	move_sentinel (C++ 20 and above)	ostream_iterator
ostreambuf_iterator	output_iterator (handled as concept in C++ 20 and above)	output_iterator_tag
projected (C++ 20 and above)	random_access_iterator (handled as concept in C++ 20 and above)	random_access_iterator_tag
reverse_iterator	unreachable_sentinel_t (C++ 20 and above)	

Localization

When you write applications for multiple languages, the application needs to know how to handle these languages correctly. The Localization category classes won't automatically convert your text to the other language. You need to perform any needed translation yourself. However, it does help you perform these tasks:

- Character classification
- String collation
- Numeric, monetary, and date/time formatting and parsing
- Message retrieval

REMEMBER

For most applications today, you use the `<locale>` header. If you have older applications that rely on the C language localization functionality that used to appear in `local.h`, you use `<clocale>` instead. C++ 11 introduced the `<codecvt>` header for converting Unicode character sets. This functionality is deprecated in C++ 17 — use the `codecvt` class in the `<locale>` header instead. The following table shows the `<local>` header classes.

`codecvt`	`codecvt_base`	`codecvt_byname`
`collate`	`collate_byname`	`ctype`
`ctype_base`	`ctype_byname`	`ctype<char>`
`locale`	`messages`	`messages_base`
`messages_byname`	`money_base`	`money_get`
`money_put`	`moneypunct`	`moneypunct_byname`
`num_get`	`num_put`	`numpunct`
`numpunct_byname`	`time_base`	`time_get`
`time_get_byname`	`time_put`	`time_put_byname`
`wbuffer_convert` (Added C++ 11, deprecated C++ 17)	`wstring_convert` (Added C++ 11, deprecated C++ 17)	

Numerics

The Numerics category is immense, as you might imagine. It provides access to all sorts of functions to perform math-related tasks. The most basic of the associated headers is `<cmath>`, which contains basic math functionality, such as obtaining

the absolute value of a number using the `abs` function. The following table lists the Numerics category headers.

Header	C++ version	Short description
`<bit>`	C++ 20 and above	Bit manipulation
`<cfenv>`	C++ 11 and above	Floating point environment access
`<cmath>`		Common math functions
`<complex>`		Complex number operations
`<numbers>`	C++ 20 and above	Math constants
`<numeric>`		Operations on values in ranges
`<random>`	C++ 11 and above	Random number generation
`<ratio>`	C++ 11 and above	Compile-time rational math
`<valarray>`		Interacts with arrays of values

Ranges

Being able to work efficiently with ranges of values is important in reducing the amount of code you write and ensuring that the code you do write is easy to understand. Most modern languages provide shortcuts for working with ranges of values, and the C++ 20 standard adds this functionality to C++.

TIP

Unfortunately, as of this writing, no major compilers or libraries actually implement this functionality, so you need to download and install the range-v3 library (`https://github.com/ericniebler/range-v3/`) to actually use it. This library is the basis for ranges support in C++ 20. (Using this library is outside the scope of this book; however, you can find documentation for it at `https://ericniebler.github.io/range-v3/`.)

This particular category is designed to work with *views*, which describe what you want to see as output. For example, if you want to sort a range in reverse order, you provide a view that describes this need, such as `views::reverse(v)`, where `v` is a vector containing the range you want to interact with. A sort might then look like `ranges::sort(views::reverse(v));`.

Ranges also work with concepts, described in the "Concepts" section of this chapter. A concept defines the sort of range you work with. The following table lists range concepts.

bidirectional_range	common_range	contiguous_range
forward_range	input_range	output_range
random_access_range	range	sized_range
view	viewable_range	

After you have a range, an idea of how you want to see it, and a task in mind, you can use the various <ranges> header classes to perform work. The following table shows the classes associated with the standard for this header.

borrowed_iterator_t	borrowed_subrange_t	dangling
iterator_t	range_difference_t	range_reference_t
range_rvalue_reference_t	range_size_t	range_value_t
ref_view	sentinel_t	subrange
view_interface	views::all	views::all_t
views::common (common_view)	views::counted	views::empty (empty_view)
views::filter (filter_view)	views::iota (iota_view)	views::join (join_view)
views::reverse (reverse_view)	views::single (single_view)	views::split (split_view)
views::take (take_view)	views::transform (transform_view)	

You can also customize the manner in which ranges work using customization point objects. The following table lists these objects found in the std::ranges namespace.

ranges::begin	ranges::data	ranges::empty
ranges::end	ranges::rbegin	ranges::rend
ranges::size		

Regular Expressions

Regular expression support appears in C++ 11 and above. It helps you look for patterns in strings. For example, you can ensure that email addresses and telephone numbers are in the right format before someone enters them into a database, reducing a few data errors in the process. You can also use regular expressions to perform search-and-replace operations. The following table contains the Regular Expression category classes.

`basic_regex`	`match_results`	`regex_error`
`regex_iterator`	`regex_token_iterator`	`regex_traits`
`sub_match`		

Strings

The Strings category provides a wide range of support for strings in C++. You have seen many examples of the `<string>` header, the most commonly used header, in use in this book. Humans understand strings quite well, but computers handle characters and, therefore, strings as numbers. To make strings easier to use, you need library support. The following table lists the String category headers and their purpose.

Header	C++ Version	Short Description
`<cctype>`		Determines the category of narrow (`char`) characters.
`<charconv>`	C++ 17 and above	Conversion to and from characters.
`<cstring>`		Narrow character string handling functions.
`<cuchar>`	C++ 11 and above	C-style Unicode character conversion functions.
`<cwchar>`		Wide and multibyte character string-handling functions.
`<cwctype>`		Determines the category of wide (`wchar_t`) characters.
`<format>`	C++ 20 and above	String formatting functionality.
`<string_view>`	C++ 17 and above	Basic string view handling class.
`<string>`		Basic string handling class.

Thread Support

Multithreaded applications allow an application to apparently perform more than one task at a time. Obviously, the actual simultaneous execution of tasks on a system relies on the number of processors or cores it contains, but multithreading enables you to share processors in a manner that lets you perform tasks efficiently. Parallel and threaded execution of tasks falls into a category of development called *concurrency*, which isn't covered in this book, but you can find a basic article on it at `https://isocpp.org/wiki/faq/cpp11-library-concurrency`.

Utilities

Utilities are functions and types that perform small service tasks within the Standard Library. The functions include `min()`, `max()`, and the relational operators. The types include `chart_traits` (the traits of characters used in other Standard Library features, such as `basic_string`) and `pair` (a pairing of two heterogeneous values). The following table lists the various essential headers provided as part of the Utilities category, their associated C++ version, and a short description. If no C++ version is supplied, you can use the header in all current versions of C++.

Header	C++ version	Short description
`<any>`	C++ 17 and above	Provides support for the `any` class for objects that hold instances of any `CopyConstructible` type.
`<bitset>`		Implements constant-length bit arrays.
`<chrono>`	C++ 11 and above	C++ time utilities.
`<compare>`	C++ 20 and above	Supports the three-way (spaceship) operator.
`<csetjmp>`		Passes control to a particular execution context.
`<csignal>`		Passes signals (messages) between various application elements.
`<cstdarg>`		Handles variable-length argument lists.
`<cstddef>`		Standard macros and typedefs.
`<cstdlib>`		General-purpose utilities for program control, dynamic memory allocation, random numbers, sort, and search.
`<ctime>`		C-style time and date utilities.

Header	C++ version	Short description
<functional>		Provides function objects, function invocations, bind operations, and reference wrappers.
<initializer_list>	C++ 11 and above	Provides the means to initialize containers other than array, such as vector, list, and map.
<optional>	C++ 17 and above	A wrapper for a variable that may or may not contain an object.
<source_location>	C++ 20 and above	Identifies the location of source code.
<tuple>	C++ 11 and above	Allows creation of tuples.
<type_traits>	C++ 11 and above	Obtains compile-time type information.
<typeindex>	C++ 11 and above	Provides a wrapper around a type_info object for use as an index in associative and unordered associative containers.
<typeinfo>		Obtains runtime information.
<utility>		Basic utility functions.
<variant>	C++ 17 and above	Provides support for variant type variables.
<version>	C++ 20 and above	Supplies implementation-dependent library information.

Parsing Strings Using a Hash

Hashes are an important security requirement for applications today. A *hash* creates a unique numeric equivalent of any string you feed it. Theoretically, you can't duplicate the number that the hash creates by using another string. A hash isn't reversible — it isn't the same as encryption and decryption.

A common use for hashes is to send passwords from a client to a server. The client converts the user's password into a numeric hash and sends that number to the server. The number varies daily depending on some formula that both client and server know. The server verifies the number, not the password. Even if people are listening in, they have no way to ascertain the password from the number;

therefore, they can't steal the password for use with the target application. Other examples of hash use are to:

- Verify a file's hash to a previously saved hash value to ensure no one has modified the file.
- Compare two files to ensure they're most likely the same.
- Make dictionary searches fast.

Code::Blocks provides excellent support for hashes. However, in order to use it, you must enable support for C++ 11 extensions using the technique found in the "Working with ranges" section of Book 1, Chapter 5. After you enable the required support, you can create the `HashingStrings` example shown here to demonstrate the use of hashes.

```
#include <iostream>
#include <unordered_map>

using namespace std;

int main() {
  hash<const char*> MyHash;
  cout << "The hash of \"Hello World\" is:" << endl;
  cout << MyHash("Hello World") << endl;
  cout << "while the hash of \"Goodbye Cruel World\" is:"
    << endl;
  cout << MyHash("Goodbye Cruel World") << endl;
  return 0;
}
```

The example begins by creating a hash function object, `MyHash`. You use this function object to convert input text to a hash value. The function object works just like any other function, so you might provide the input text as `MyHash("Hello World")`. Hashes always output precisely the same value given a particular input. Consequently, you should see the following output from this example.

```
The hash of "Hello World" is:
4952133
while the hash of "Goodbye Cruel World" is:
4952192
```

REMEMBER

Hashes have uses other than security requirements. For example, you can create a container that relies on a hash to make locating a particular value easier. In this case, you use a key/value pair in a *hash map* using `unordered_map<>`. The `HashMap` example, shown next, illustrates how to create a hash map:

```
#include <iostream>
#include <unordered_map>
#include <string.h>

using namespace std;

struct eqstr {
  bool operator()(const char* s1, const char* s2) const
  {
    return strcmp(s1, s2) == 0;
  }
};

int main() {
  unordered_map<const char*, int,
    hash<const char*>, eqstr> Colors;
  Colors["Blue"] = 1;
  Colors["Green"] = 2;
  Colors["Teal"] = 3;
  Colors["Brick"] = 4;
  Colors["Purple"] = 5;
  Colors["Brown"] = 6;
  Colors["LightGray"] = 7;
  cout << "Brown = " << Colors["Brown"] << endl;
  cout << "Brick = " << Colors["Brick"] << endl;
  // This key isn't in the hash map, so it returns a
  // value of 0.
  cout << "Red = " << Colors["Red"] << endl;
}
```

An unordered (hash) map requires four inputs:

- Key type
- Data type
- Hashing function
- Equality key

The first three inputs are straightforward. In this case, the code uses a string as a key type, an integer value as a data type, and `hash<const char*>` as the hashing function. You already know how the hashing function works from the previous example in this section.

The equality key class is a little more complex. You must provide the hash map with a means of determining equality. In this case, the code compares the input string with the string stored as the key. The `eqstr` structure performs the task of comparing the input string to the key. The structure must return a Boolean value, so the code compares the `strcmp` function to `0`. When the two are equal, meaning that the strings are equal, `eqstr` returns `true`.


REMEMBER

The example goes on to check for three colors, only two of which appear in the hash map `Colors`. In the first two cases, you see the expected value. In the third case, you see `0`, which indicates that `Colors` doesn't contain the desired key. Always reserve `0` as an error indicator when using a hash map, because the hash map will always return a value, even if it doesn't contain the desired key. The output from this example is

```
Brown = 6
Brick = 4
Red = 0
```


Obtaining Information Using a Random Access Iterator

Most containers let you perform random access of data they contain. For example, the `RandomAccess` example shows that you can create an `iterator` and then add to or subtract from the current offset to obtain values within the container that `iterator` supports:

```
#include <iostream>
#include <vector>

using namespace std;

int main() {
  vector<string> Words;
  Words.push_back("Blue");
  Words.push_back("Green");
```

```
  Words.push_back("Teal");
  Words.push_back("Brick");
  Words.push_back("Purple");
  Words.push_back("Brown");
  Words.push_back("LightGray");
  // Define a random iterator.
  vector<string>::iterator Iter = Words.begin();
  // Access random points.
  Iter += 5;
  cout << *Iter << endl;
  Iter -= 2;
  cout << *Iter << endl;
  return 0;
}
```

In this case, the `vector`, `Words`, contains a list of seven items. The code creates an `iterator` for `Words` named `Iter`. It then adds to or subtracts from the `iterator` offset and displays the output onscreen. Here is what you see when you run this example:

```
Brown
Brick
```

Sometimes you need to perform a special task using a random-access `iterator`. For example, you might want to create a special function to summate the members of `vector` or just a range of members within `vector`. In this case, you must create a specialized function to perform the task as follows because the Standard Library doesn't include any functions to do it for you, as shown in the `RandomAccess2` example:

```
#include <iostream>
#include <vector>

using namespace std;

template <class RandomAccessIterator>
float AddIt(RandomAccessIterator begin,
            RandomAccessIterator end) {
  float Sum = 0;
  RandomAccessIterator Index;
  // Make sure that the values are in the correct order.
  if (begin > end)
  {
    RandomAccessIterator temp;
```

```
    temp = begin;
    begin = end;
    end = temp;
  }
  for (Index = begin; Index != end; Index++)
    Sum += *Index;
  return Sum;
}

int main() {
  vector<float> Numbers;
  Numbers.push_back(1.0);
  Numbers.push_back(2.5);
  Numbers.push_back(3.75);
  Numbers.push_back(1.26);
  Numbers.push_back(9.101);
  Numbers.push_back(11.3);
  Numbers.push_back(1.52);

  // Sum the individual members.
  float Sum;
  Sum = AddIt(Numbers.begin(), Numbers.end());
  cout << Sum << endl;
  Sum = AddIt(Numbers.end(), Numbers.begin());
  cout << Sum << endl;

  // Sum a range.
  vector<float>::iterator Iter = Numbers.begin();
  Iter += 5;
  Sum = AddIt(Iter, Numbers.end());
  cout << Sum << endl;
  return 0;
}
```

This example builds on the previous example. You still create a `vector`, `Numbers`, and fill it with data. However, in this case, you create an output variable, `Sum`, that contains the summation of the elements contained in `Numbers`.

`AddIt()` is a special function that accepts two `RandomAccessIterator` values as input. These two inputs represent a range within the `vector` that you want to manipulate in some way. The example simply adds them, but you can perform any task you want. The output is a `float` that contains the summation.

AddIt() works as you expect. You call it as you would any other function and provide a beginning point and an end point within vector. The first two calls to AddIt sum the entire vector, and the third creates an iterator, changes its offset, and then sums a range within vector. Here is the output from this example:

```
30.431
30.431
12.82
```


REMEMBER

A random-access iterator can go in either direction. In addition, you can work with individual members within the container supplied to iterator. As a result, the functions you create for iterator must be able to work with the inputs in any order. How you handle this requirement depends on the kind of function you create.

Locating Values Using the Find Algorithm

The Standard Library contains a number of functions to find something you need within a container. Locating what you need as efficiently as possible is always a good idea. The four common find() algorithms are

- find()
- find_end()
- find_first_of()
- find_if()

The algorithm you use depends on what you want to find and where you expect to find it. You'll likely use the plain find() algorithm most often. The FindString example shows how to locate a particular string within vector. You can use the same approach to locate something in any container type:

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

int main() {
  vector<string> Words;
  Words.push_back("Blue");
```

```
  Words.push_back("Green");
  Words.push_back("Teal");
  Words.push_back("Brick");
  Words.push_back("Purple");
  Words.push_back("Brown");
  Words.push_back("LightGray");

  vector<string>::iterator Result =
    find(Words.begin(), Words.end(), "LightGray");
  if (Result != Words.end())
    cout << *Result << endl;
  else
    cout << "Value not found!" << endl;

  Result = find(Words.begin(), Words.end(), "Black");
  if (Result != Words.end())
    cout << *Result << endl;
  else
    cout << "Value not found!" << endl;
}
```

The example starts with a `vector` containing color strings. In both cases, the code attempts to locate a particular color within `vector`. The first time the code is successful because `LightGray` is one of the colors listed in `Words`. However, the second attempt is thwarted because `Black` isn't one of the colors in `Words`. Here's the output from this example:

```
LightGray
Value not found!
```

WARNING

Never assume that the code will find a particular value. Always assume that someone is going to provide a value that doesn't exist and then make sure you provide a means of handling the nonexistent value. In this example, you simply see a message stating that the value wasn't found. However, in real-world code, you often must react to situations in which the value isn't found by

- Indicating an error condition
- Adding the value to the container
- Substituting a standard value
- Defining an alternative action based on invalid input

TIP

You can use the `find()` algorithm for external and internal requirements. Even though the example shows how you can locate information in an internal `vector`, you can also use `find()` for external containers, such as disk drives.

Using the Random Number Generator

Random number generators fulfill a number of purposes. Everything from games to simulations require a random number generator to work properly. Randomness finds its way into business what-if scenarios as well. In short, you need to add random output to your application in many situations. Creating a random number isn't hard. All you need to do is call a random number function, as shown in the `RandomNumberGenerator` example:

```
#include <iostream>
#include <time.h>
#include <stdlib.h>

using namespace std;

int main() {
  // Always set a seed value.
  srand((unsigned int)time(NULL));
  int RandomValue = rand() % 12;
  cout << "The random month number is: "
    << RandomValue + 1 << endl;
  return 0;
}
```

REMEMBER

The Standard Library uses *pseudorandom* number generators: The numbers are distributed such that you appear to see a random sequence, but given enough time and patience, eventually the sequence repeats. In fact, if you don't set a seed value for your random number generator (or set it to a specific number), you can obtain predictable sequences of numbers every time. Most people use the time or some other automatically changing numeric source to set the seed value to make it more unpredictable. Here is typical output from this example:

```
The random month number is: 7
```

TIP

The first line of code in `main()` sets the seed by using the system time. Using the system time ensures a certain level of randomness in the starting value — and therefore a level of randomness for your application as a whole. If you comment out this line of code, you see the same output every time you run the application.

The example application uses `rand()` to create the random value. When you take the modulus of the random number, you obtain an output that is within a specific range — 12 in this case. The example ends by adding `1` to the random number because there isn't any month `0` in the calendar, and then outputs the month number for you.

Working with Temporary Buffers

Temporary buffers are useful for all kinds of tasks. Normally, you use them when you want to preserve the original data, yet you need to manipulate the data in some way. For example, creating a sorted version of your data is a perfect use of a temporary buffer. The `TemporaryBuffer` example shows how to use a temporary buffer to sort some strings:

```
#include <iostream>
#include <vector>
#include <memory>
#include <algorithm>

using namespace std;

int main() {
  vector<string> Words;
  Words.push_back("Blue");
  Words.push_back("Green");
  Words.push_back("Teal");
  Words.push_back("Brick");

  int Count = Words.size();
  cout << "Words contains: " << Count << " elements."
    << endl;

  // Create the buffer and copy the data to it.
  pair<string*, ptrdiff_t> Mem =
    get_temporary_buffer<string>(Count);
  uninitialized_copy(Words.begin(), Words.end(),
                     Mem.first);

  // Perform a sort and display the results.
  sort(Mem.first, Mem.first+Mem.second);
  for (int i = 0; i < Mem.second; i++)
      cout << Mem.first[i] << endl;
```

```
  // Show that the original list is unchanged.
  cout << "\nShowing Words Hasn't Changed" << endl;
  for (int i = 0; i < Count; i++)
    cout << Words[i] << endl;
  return 0;
}
```

The example starts with the now familiar list of color names. It then counts the number of entries in `Words` and displays the count onscreen.

At this point, the code creates the temporary buffer using `get_temporary_buffer()`. The output is `Mem` of type `pair`, with the first value containing a pointer to the string values and the second value containing the count of data elements. `Mem` doesn't contain anything — you have simply allocated memory for it.

The next task is to copy the data from `Words` to `Mem` using `uninitialized_copy()`. Now that `Mem` contains a copy of your data, you can organize it using the `sort()` function. The final step is to display the `Mem` content onscreen. Here is what you'll see:

```
Words contains: 4 elements.
Blue
Brick
Green
Teal

Showing Words Hasn't Changed
Blue
Green
Teal
Brick
```