Chapter 1: Exploring the Standard Library Further

In This Chapter

- Categorizing the Standard Library functions
- Working with container functions such as hash
- Performing random access with iterator functions
- Working with algorithms such as find
- Creating random numbers with functors
- Working with utilities such as min and max
- Creating temporary buffers with allocators

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, and renamed it the Standard Library. Consequently, when you see the STL online, don't get confused; it's merely an older version of the Standard Library.

For the purposes of this book, the differences between the Standard Library and the STL are so small that you can probably use the terms interchangeably. Just remember that the Standard Library is newer and does contain some changes to make the various versions of the STL work together. In addition, the STL won't provide support for features such as polymorphic allocators and optional inputs.

This chapter provides 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; we discuss more examples in later chapters and you can always refer to the Standard Library documentation for additional examples. Before the chapter moves on to any examples, however, it's important to know what the Standard Library contains, so the first section of this chapter provides you with a list of Standard Library function categories.

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 their document free or purchase a copy of it from `http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=50372`. 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) for a more reasonable sum. Check it out at `http://webstore.ansi.org/FindStandards.aspx` (simply type ISO/IEC 14882:2011 in the search field).

Because the STL 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.sgi.com/tech/stl/`. 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++.

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 1,338-page tome (in the current version) 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:

- Containers
- Iterators
- Algorithms
- Functors

SGI color-coding

The SGI website at `http://www.sgi.com/tech/stl/stl_index_cat.html` uses color-coding to tell you about the content. Here are the categories you'll see and their associated color:

- Concept is red
- Type is yellow
- Function is green
- Overview is purple

- Utilities
- Adaptors
- Allocators
- Polymorphic allocators

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.

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:

`basic_string`	`bit_vector`	`bitset`
`char_producer`	`deque`	`hash`
`list`	`map`	`multimap`
`multiset`	`priority_queue`	`queue`
`rope`	`set`	`slist`
`stack`	`vector`	

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 types. These types determine the kind of iterator you create in your code and the capabilities of that iterator. The following is a list of the iterator types:

back_insert_iterator	bidirectional_iterator	bidirectional_iterator_tag
forward_iterator	forward_iterator_tag	front_insert_iterator
input_iterator	input_iterator_tag	insert_iterator
istream_iterator	iterator_traits	ostream_iterator
output_iterator	output_iterator_tag	random_access_iterator
random_access_iterator_tag	raw_storage_iterator	reverse_bidirectional_iterator
reverse_iterator	sequence_buffer	

The Standard Library also includes a number of iterator-specific functions. These functions help you perform tasks such as advance (increment) the iterator by a certain number of positions. You can also measure the distance between the beginning and end of the iterator. The following is a list of iterator functions:

advance	distance	distance_type
iterator_category	value_type	

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 algorithm functions:

accumulate	adjacent_difference	adjacent_find
advance	binary_search	copy

copy_backward	copy_n	count
count_if	distance	equal
equal_range	fill	fill_n
find	find_end	find_first_of
find_if	for_each	generate
generate_n	includes	inner_product
inplace_merge	iota	is_heap
is_sorted	iter_swap	lexicographical_compare
lexicographical_compare_3way	lower_bound	make_heap
max	max_element	merge
min	min_element	mismatch
next_permutation	nth_element	partial_sort
partial_sort_copy	partial_sum	partition
pop_heap	power	prev_permutation
push_heap	random_sample	random_sample_n
random_shuffle	remove	remove_copy
remove_copy_if	remove_if	replace
replace_copy	replace_copy_if	replace_if
reverse	reverse_copy	rotate
rotate_copy	search	search_n
set_difference	set_intersection	set_symmetric_difference
set_union	sort	sort_heap
stable_partition	stable_sort	swap
swap_ranges	transform	uninitialized_copy
uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n
unique	unique_copy	upper_bound

Functors

Functors are a special class of object that acts as if it's a function. In most cases, you call a functor by using the same syntax you use for a function, but functors possess all the good elements of objects as well, such as the ability to instantiate them at runtime. (See Book IV Chapter 7 for an example of using a functor in this way.) Functors come in a number of forms.

For example, a binary function functor accepts two arguments as input and provides a result as output. Functors include a number of types that determine the kind of function the code creates, as shown in the following table:

binary_compose	binary_function	binary_negate
binder1st	binder2nd	divides
equal_to	greater	greater_equal
hash	identity	less
less_equal	logical_and	logical_not
logical_or	mem_fun1_ref_t	mem_fun1_t
mem_fun_ref_t	mem_fun_t	minus
modulus	multiplies	negate
not_equal_to	plus	pointer_to_binary_function
pointer_to_unary_function	project1st	project2nd
select1st	select2nd	subtractive_rng
unary_compose	unary_function	unary_negate

The Functors category contains only one function, `ptr_fun`. This function accepts a function pointer as input and outputs a function pointer adapter, which is a kind of function object. You use `ptr_fun` when you need to pass a function as input to another function such as `transform`. Here is an example of such code (as found in the `functor_ptr_fun` example):

```
#include <iostream>
#include <math.h>
#include <ext/functional>
#include <algorithm>

using namespace std;
using namespace __gnu_cxx;

int main()
{
    const int N = 10;
    double A[N];
    fill(A, A+N, 100);

    cout << A[0] << endl;
```

```
    transform(A, A+N, A, compose1(negate<double>(), ptr_fun(fabs)));

    cout << A[0] << endl;

    return 0;
}
```

This example begins by creating a constant that determines the number of elements in the array `A`. The code then fills every element in `A` with the value `100` and displays just one of those elements onscreen.

The tricky part comes next. The `transform()` algorithm accepts the beginning of an input iterator, the end of an input iterator, an output iterator, and the transformation you want to perform. The `transform()` algorithm takes each of the values in the input iterator, performs the transformation you requested, and places the result in the output iterator.

In this case, the code uses the nonstandard SGI functor `compose1()`, which takes two adaptable unary functions as input. Because `fabs()` is a standard function, you must use `ptr_fun()` to change it into a function pointer adapter before you can use it with `compose1()`. The result is that `A` contains the negation of the absolute value of the original value in `A` or `-100` when the transformation is complete. When you run this example, you see the following output:

```
100
-100
```

The GNU GCC compiler supports a number of STL features that don't appear as part of the Standard Library. In this case, `compose1()` appears in the `ext/functional` header, so you must provide the `#include <ext/functional>` line of code. In addition, because `compose1()` is nonstandard, it appears as part of a different namespace. Consequently, you must also provide the `using namespace __gnu_cxx;` line of code to access the functor without having to precede it with the namespace information.

Many C++ examples rely on the nonstandard parts of STL to perform tasks. If you want maximum compatibility and transportability for your code, you should avoid these nonstandard features.

Utilities

Utilities are functions and types that perform small service tasks within the Standard Library. The functions are `min()`, `max()`, and the relational operators. The types are `chart_traits` (the traits of characters used in other Standard Library features, such as `basic_string`) and `pair` (a pairing of two heterogeneous values).

Adaptors

Adaptors perform conversions of a sort. They make it possible to adapt one kind of data to another. In some cases, adaptors perform data conversion, such as negating numbers. The Adaptors category includes one function, `ptr_fun()`, which is explained in the "Functors" section of the chapter. In addition, the Adaptors category includes the types shown in the following table:

back_insert_ iterator	binary_compose	binary_negate
binder1st	binder2nd	front_insert_ iterator
insert_iterator	mem_fun1_ref_t	mem_fun1_t
mem_fun_ref_t	mem_fun_t	pointer_to_ binary_function
pointer_to_ unary_function	priority_queue	queue
raw_storage_ iterator	reverse_ bidirectional_ iterator	reverse_iterator
sequence_buffer	stack	unary_compose
unary_negate		

Allocators

Allocators manage resources, normally memory. In most cases, you won't ever need to use the members of the Allocators category. For example, you normally create new objects using the `new` operator. The `new` operator allocates memory for the object and then creates it by calling the object's constructor. In rare cases, such as when you want to implement a form of object pooling, you may want to separate the memory allocation process from the construction process. In this case, you call `construct()` to perform the actual task of constructing the object based on its class definition. The Allocators category has the following functions.

construct	destroy	get_temporary_ buffer
return_ temporary_buffer	uninitialized_ copy	uninitialized_ copy_n
uninitialized_ fill	uninitialized_ fill_n	

The Allocators category also includes a couple of types. These types help you manage memory, and you may find more use for them than you will the functions in this category. The types are

raw_storage_iterator	temporary_buffer

Polymorphic allocators

When working with older versions of the Standard Library, allocators used as arguments to templates create problems because they're bound by type. What this means is that a vector created using `std::vector<int>` is a completely different type from a vector created using `std::vector<int, myalloc>`, even though one is simply an extension of the other.

The `myalloc` part of the template simply defines the method used to allocate memory; it doesn't actually affect the type of data managed by the template. So, in both cases, you're created a `vector` to hold `int` data — the types are the same. The only difference is the method in which memory is allocated (the first uses standard memory allocation techniques, while the second uses a custom allocator). Using polymorphic allocators eliminates this problem by defining an abstract base memory class, `memory_resource`, to use for all memory allocators. This abstract class defines the following pure virtual methods:

allocate	deallocate	is_equal()

In order to use this new feature in Code::Blocks, you must enable support for C++ 11 extensions using the technique found in the "Configuring the IDE" section of Book IV, Chapter 6. In addition, you must use a version of Code::Blocks that supports C++ 14 because older versions won't include the required resources (such as header files). To add support for this feature, you must `#include <polymorphic_allocator>` and add `using namespace std::polyalloc`.

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 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.

The latest version of 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 "Configuring the IDE" section of Book IV, Chapter 6. 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:
4644931
while the hash of "Goodbye Cruel World" is:
4644988
```

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.* The STL uses an actual `hash_map<>` template. However, the Standard Library replaces `hash_map<>` with `unordered_map<>`, which means you must enable C++ extension support for this example. Except for the template name, you can actually use the two templates interchangeably, but using the `hash_map<>` template will display a warning message in newer versions of Code::Blocks.

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 the strings are equal, `eqstr` returns `true`.

Standard Library versus STL headers

You'll find a wealth of STL examples on the Internet because STL was around for a long time before the Standard Library appeared. In fact, some developers continue to prefer the STL simply because they're familiar with it. Here's a little secret: The STL headers use a `.h` extension and the Standard Library headers don't have an extension. For example, the now familiar `iostream` header used in every previous example in the book is actually the Standard Library form — the STL form is `iostream.h`.

Here's another secret. The Standard Library headers often call on the STL headers, so you've also been using STL throughout the book. It's amazing to see how these things work out.

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 `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`, while 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
```

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. Unlike your closet, you want your applications well organized and easy to manage! 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 `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 `vector`. However, the second attempt is thwarted because `Black` isn't one of the colors in `vector`. Here's the output from this example:

```
LightGray
Value not found!
```

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 the value wasn't found. However, in real-world code, you often must react to situations where 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

The `find()` algorithm is a personal favorite because it's so flexible. You can use it 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. Have some fun with this one — experiment with all the containers you come across.

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;
}
```

Actually, not one of the random number generators in the Standard Library works properly — imagine that! They are all *pseudorandom* number generators: The numbers are distributed such that it appears that you 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, you can obtain predictable sequences of numbers every time. How boring. Here is typical output from this example:

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

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. In our case, our system output 6 every time.

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.

The Standard Library provides access to two types of pseudorandom number generators. The first type requires that you set a seed value. The second type requires that you provide an input value with each call and doesn't require a seed value. Each generator outputs a different data type, so you can choose the kind of random number you obtain. Table 1-1 lists the random number generators and tells you what data type they output.

Table 1-1 Pseudorandom Number Generator Functions

Function	*Output Type*	*Seed Required?*
`rand`	`integer`	`yes`
`drand48`	`double`	`yes`
`erand48`	`double`	`no`
`lrand48`	`long`	`yes`
`nrand48`	`long`	`no`
`mrand48`	`signed long`	`yes`
`jrand48`	`signed long`	`no`

Now that you know about the pseudorandom number generators, look at the seed functions used to prime them. Table 1-2 lists the seed functions and their associated pseudorandom number generator functions.

Table 1-2	Seed Functions
Function	*Associated Pseudorandom Number Generator Function*
srand	rand
srand48	drand48
seed48	mrand48
lcong48	lrand48

Performing Comparisons Using min and max

Computer applications perform many comparisons. In most cases, you don't know what the values are in advance or you wouldn't be interested in performing the comparison in the first place. The `min()` and `max()` functions make it possible to look at two values and determine the minimum or maximum value. The `MinAndMax` example demonstrates how you use these two functions:

```
#include <iostream>

using namespace std;

int main()
{
    int Number1, Number2;

    cout << "Type the first number: ";
    cin >> Number1;

    cout << "Type the second number: ";
    cin >> Number2;

    cout << "The minimum number is: " << min(Number1, Number2) << endl;
    cout << "The maximum number is: " << max(Number1, Number2) << endl;

    return 0;
}
```

In this case, the code accepts two numbers as input and then compares them using `min()` and `max()`. The output you see depends on what you provide as input, but the first output line tells you which number is smaller and the second tells you which is larger. Assuming you provide values of 5 and 6, here is the application output you see:

```
Type the first number: 5
Type the second number: 6
The minimum number is: 5
The maximum number is: 6
```

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");
    Words.push_back("Purple");
    Words.push_back("Brown");
    Words.push_back("LightGray");

    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;

    return 0;
}
```

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

At this point, the code creates the temporary buffer using `get_temporary_buffer`. The output is `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 `vector` (`Words`) to `pair` (`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: 7 elements.
Blue
Brick
Brown
Green
LightGray
Purple
Teal
```