

Data Structures and Other Objects Using C++
Second Edition
Release date: July 30, 2000
ISBN 0-201-70297-5
Addison Wesley

Michael Main and Walter Savitch
main@colorado.edu
Supplements:
<http://www.cs.colorado.edu/~main/dsoc.html>

3 Container Classes

(I am large. I contain multitudes.)

WALT WHITMAN
Song of Myself

- 3.1 THE BAG CLASS
- 3.2 PROGRAMMING PROJECT: THE SEQUENCE CLASS
- 3.3 INTERACTIVE TEST PROGRAMS
- CHAPTER SUMMARY
- SOLUTIONS TO SELF-TEST EXERCISES
- PROGRAMMING PROJECTS

The throttle and point classes in Chapter 2 are good examples of abstract data types. But their applicability is limited to a few specialized programs. This chapter begins the presentation of several classes with broad applicability in programs large and small. The two particular classes in this chapter—bags and sequences—are examples of **container classes**. Intuitively, a container class is a class where each object contains a collection of items. For example, one program might keep track of a collection of integers, perhaps the ages of all the people in your family. Another program, perhaps a cryptography program, can use a collection of characters.

The bag and sequence classes are both simple versions of more complex classes from the C++ Standard Library. The goal is for you to understand and use the bag and sequence classes as a bridge to understanding and using the Standard container classes. Over the next few chapters, variations of the bag and sequence classes will teach you how to write your own container classes that are compliant

*a class in which
each object
contains a
collection of items*

with the C++ Standard Library, and therefore your own classes can take advantage of standard algorithms for such tasks as searching and sorting.

A key feature of a good container class is that it should be easy to change the type of item in the container so that a new application can use the container. With this kind of “easy reuse,” many different applications can use the same container class. The same container class can be used by one program for a collection of integers, and by another program for a collection of characters or some other data type. In this chapter we use **typedef statements** to provide the ability to easily change the type of item in a container class. In Chapter 6, which focuses explicitly on software reusability, we’ll use a different technique called **templates**, which is also used by the Standard Library container classes.

3.1 THE BAG CLASS

This section provides an example of a container class, called a *bag of integers*. To define the new bag data type, think about an actual bag—a grocery bag or a garbage bag—and imagine writing integers on slips of paper and putting them in the bag. A **bag of integers** is similar to this imaginary bag: a container that can hold a collection of integers that we place into it. A bag of integers can be used by any program that needs to store a collection of integers for later use. For example, later we will write a program that keeps track of the ages of your family’s members. If you have a large family with ten people, the program keeps track of ten ages—and these ages are kept in a bag of integers.

The Bag Class—Specification

We’ve given an intuitive description of a bag of integers, but for a more precise specification of the bag class, we must describe the collection of functions to manipulate a bag object. We’ll do this by providing a prototype for each of the functions, most of which are member functions. With each prototype we also specify the precise action that the function will perform. These specifications will later become our precondition/postcondition contracts. Let’s look at the functions one at a time.

The constructor. The bag class has a default constructor to initialize a bag so that it is empty. The name of the constructor must be the same as the name of the class itself, so the prototype for our constructor is the following:

```
bag( );
```

The value semantics. As part of our specification, we require that bag objects can be copied with an assignment statement. Also, a newly declared bag can be initialized as a copy of another bag, using the copy constructor such as:

```

bag b;
b.insert(42);
bag c(b);

```

b now contains a 42.

*c is initialized
with the copy constructor
to be a copy of b.*

At this point, because we are only specifying which operations can manipulate a bag, we don't need to say anything more about the value semantics.

A typedef for the value_type. So far we have considered only bags of integers. But to be more flexible, we won't actually use the name *int* when we refer to the types of the items in the bag. Instead, we will use the name `value_type` for the data type of the items in a bag. Some programs might need a bag of integers, and those programs will set the `value_type` to an *int*. Other programs might use a different `value_type`. In order for the bag to have this flexible `value_type`, we will place the following statement at the top of the public section of the bag's class definition:

```

class bag
{
public:
    typedef int value_type;
    ...

```

This statement is a **typedef** statement. It consists of the keyword *typedef* followed by a data type (such as *int*) and then a new identifier, such as `value_type`. We are not required to use the specific name `value_type`; we could have used any meaningful name. But the Standard Library container classes use the name `value_type`, so we have done so for consistency.

The effect of the typedef statement is that bag functions can use the name `value_type` as a synonym for the data type *int*. Wherever a bag member function uses the name `value_type`, the compiler will recognize it as simply another name for *int*. Other functions, which are not bag member functions, can use the name `bag::value_type` as the type of the items in a bag. Moreover, if we want a new kind of bag, we can simply change the word *int* to a new data type and recompile. No other changes will be needed anywhere in our program. For example, to declare a bag of double numbers we change the typedef statement to the following:

```

class bag
{
public:
    typedef double value_type;
    ...

```

In Chapter 6, we will use an alternative way to define `value_type`. The alternative, called a template class, is more cumbersome, but it overcomes some drawbacks of the typedef statement. Meanwhile, the top of the next page shows a summary of how we used the C++ typedef statement.



FEATURE

C++ Feature: Typedef Statements within a Class Definition

Within a class definition, we can place a typedef statement of the following form:

```
class <Name of the class>
{
public:
    typedef <A data type such as int or double> <A new name>
    ...
}
```

This statement is a **typedef** statement. It consists of the keyword *typedef* followed by a data type (such as *int*) and then a new identifier (such as *value_type*). The effect of this typedef statement is that member functions can use the new name *value_type* as a synonym for the data type. Functions that are not member functions can also use the name, but its use must be preceded by the class name and “::” (for example *bag::value_type*).

The *size_type*. In addition to the *value_type*, our *bag* defines another data type that can be used for variables that keep track of how many items are in a bag. This type will be called *size_type*, with its definition near the top of the *bag* class definition:

```
class bag
{
public:
    typedef int value_type;
    typedef <an integer type of some kind> size_type;
    ...
}
```

Once we have provided the *size_type* definition, we can use *size_type* for any variable that’s counting how many items are in a bag. This is another programming idea that we got from the Standard Library containers—they all have a built-in *size_type* as part of the class.

Of course, we still must decide which data type to use for “an integer type of some kind” in the typedef statement. We could use an ordinary *int*, but C++ provides a better alternative: the *size_t* data type, described here.



FEATURE

C++ Feature: The *std::size_t* Data Type

The data type *size_t* is an integer data type that can hold only non-negative numbers. Each C++ implementation guarantees that the values of the *size_t* type are sufficient to hold the size of any variable that can be declared on your machine. Therefore, when you want to describe the size of some array or other variable, the best choice is the *size_t* data type. The *size_t* type is part of the *std* namespace from the Standard Library facility, *cstdlib*. To use *size_t* in a header file, we must include *cstdlib* and use the full name *std::size_t*.

Our bag definition uses `size_t` as shown here:

```
class bag
{
public:
    typedef int value_type;
    typedef std::size_t size_type;
    ...
}
```

With the bag definition, or within an implementation of a bag member function we can use the type `size_type`. Other programmers can also use this data type, but they must write the full name `bag::size_type`.

The size member function. The bag has a constant member function called `size`. The prototype uses the bag's `size_type`:

```
size_type size( ) const;
```

As you might guess, the return value of the `size` function tells how many items are currently in the bag. To illustrate the use of the function, suppose `first_bag` contains one copy of the number 4 and two copies of the number 8. Then `first_bag.size()` returns 3.

The insert member function. This is a member function that places a new integer, called `entry`, into a bag. Here is the prototype:

```
void insert(const value_type& entry);
```

As an example, here is a sequence of function calls for a bag called `first_bag`:

```
bag first_bag;
first_bag.insert(8);
first_bag.insert(4);
first_bag.insert(8);
```

After these statements, first_bag contains two 8s and a 4.



After these statements are executed, `first_bag` contains three integers: the number 4 and two copies of the number 8. It is important to realize that a bag can contain many copies of the same integer, such as this example with two copies of 8.

Notice that the `entry` parameter is a `const` reference parameter. This may seem strange since the usual purpose of a `const` reference parameter is to improve efficiency when a parameter is a large object. Integers are not large, but we may later change the `value_type` to something that is large. With this in mind, we will use `const` reference parameters for `value_type` parameters, whenever this is possible (i.e., whenever the function's implementation does not change the value of the parameter).

The count member function. This is a constant member function that determines how many copies of a *particular* number are in a bag. The prototype uses `size_type`:

```
size_type count(const value_type& target) const;
```

The activation of `count(n)` returns the number of occurrences of `n` in a bag. For example, if `first_bag` contains the number 4 and two copies of the number 8, then we will have these values:

```
cout << first_bag.count(1) << endl; ← Prints 0
cout << first_bag.count(4) << endl; ← Prints 1
cout << first_bag.count(8) << endl; ← Prints 2
```

The erase_one and erase member functions. These two member functions have the following prototypes:

```
bool erase_one(const value_type& target);
size_type erase(const value_type& target);
```

Provided that the `target` is actually in the bag, the `erase_one` function removes one copy of `target` and returns `true`. If `target` is not in the bag, attempting to erase one copy has no effect on the bag, and the function returns `false`. The `erase` function removes all copies of the `target`; its return value tells how many copies were removed (which could be zero).

Union operator. The **union** of two bags is a new larger bag that contains all the numbers in the first bag plus all the numbers in the second bag, as shown here:



In the drawing we wrote “+” for “union.” To implement the union, we will overload the `+` operator as a nonmember function with this prototype:


```
bag operator +(const bag& b1, const bag& b2);
```

The function is not a member function because of our guidelines about overloading binary operators (see page 82).

Overloading the += operator. The + operator is defined for bags, so it is sensible to also overload +=. The overloaded += will allow us to add the contents of one bag to the existing contents of another bag in much the same way that += works for integers or real numbers. We intend to use += as shown here:

```
bag first_bag, second_bag;
first_bag.insert(8);
second_bag.insert(4);
second_bag.insert(8);
first_bag += second_bag;
```

*This adds the contents of
second_bag to what's
already in first_bag.*



After these statements `first_bag` contains one 4 and two 8s.

Our style preference is to overload += as a member function. The reason is that the first argument (to the left of the +=) has special significance: It is the argument that actually has its value changed. The second argument (to the right of the +=) never has its value changed. By making the operator += into a member function, we place special emphasis on the left argument in a statement such as:

*overload += as a
member function*

```
first_bag += second_bag;
```

This statement means “activate the += member function of `first_bag`, and use `second_bag` as the argument.” Here is the prototype of the member function:

```
void operator +=(const bag& addend);
```

There are several points to notice:

- This is a *void* function. It does not return a value. It only alters the contents of the bag that activates the function.
- The function has only one parameter, `addend`. This is the right-hand bag in an expression such as `first_bag += second_bag`. The left-hand bag is the bag that activates += and that has its contents altered.
- We use the name `addend` for the parameter, meaning “something to be added,” but you may use whatever name you like.

The bag’s CAPACITY. That’s the end of our list of functions, and we’re almost ready to write the header file. But first, we describe one more handy C++ feature that is related to how we will store the items in a bag.

Our plan is for bounded bags that can hold 30 items each. (Later we will remove this restriction, providing an unbounded bag class.) There is nothing magic about the number 30—we just picked it as a conveniently small size for our first bags. Later, we might want to change the size 30, allowing bags that hold 42 or 5000 or some other number of items. To make it easy to change the bag’s size, and also to make our programs more readable, we will use a name such as `CAPACITY` rather than simply using the number 30.

The best way to define `CAPACITY` is as a **static member constant**, as shown in the example here:.

```
class bag
{
public:
    typedef int value_type;
    typedef std::size_t size_type;
    static const size_type CAPACITY = 30;
    ...
}
```

The keyword `const` has the same meaning that we have seen with other constant declarations, so that the value of `CAPACITY` is defined once and cannot be changed while the program is running.

The keyword `static` modifies the definition in a useful way. Usually each object has its own copy of each member variable. But when the keyword `static` is used with a class member, it means that *all* of the class's objects use the *same* value. This is different! For example, with the bag's static member constant, every bag has the same `CAPACITY` of 30. In fact, the only reason that we can set the `CAPACITY` to 30 *within* the class definition is because every bag has the same value for `CAPACITY`. When a program declares a bag `b`, the program can refer to the capacity with the usual notation for selecting a member: `b.CAPACITY`. Because every bag has the same capacity, a program can also refer to a bag's capacity using the bag:: "scope resolution operator," as shown in this example:

```
bag b;
cout << "The capacity of b is " << b.CAPACITY << endl;
cout << "Every bag has capacity " << bag::CAPACITY << endl;
```

As shown in this example, we recommend all uppercase letters for the name of any constant. This makes it easy to recognize which values are constant.

In addition to declaring the static member constant within the class definition, the program must also repeat the declaration of the constant in the implementation file. In our example, the following single line must appear in the implementation file:

```
const bag::size_type bag::CAPACITY;
```

We have described the general format of a static member constant, but there are a few pitfalls to beware of:

- The keyword `static` is not repeated in the implementation file because `static` has a different meaning outside of the class definition.
- When the constant is declared in the implementation file, we must use the full type name (such as `bag::size_type`) rather than the short version (such as `size_type`) because the short version may be used only in the class definition or within an implementation of a member function.

- In the implementation file, we must also use the full name of the constant (such as `bag::CAPACITY`) rather than the short version (such as `CAPACITY`); otherwise the compiler won't know that this is a member of a class.

For future reference, here is a summary of static member constants, including a note about where the initial value must appear for different types of constants.

CLARIFYING THE CONST KEYWORD Part 4: Static Member Constants

A **static member constant** has the two keywords *static* and *const* before its declaration in a class. For example, in our bag class definition:

```
static const size_type CAPACITY = 30;
```

The keyword *static* indicates that the entire class has only one copy of this member, and the keyword *const* indicates that a program cannot change the value (which is just like ordinary declared constants).

In addition to declaring the static member constant within the class definition, the constant must be redeclared in the implementation file without the keyword *static*. For example:

```
const bag::size_type bag::CAPACITY;
```

Notice that the initial value (such as 30), is given only in the header file, not the implementation file. However, this technique of defining the value in the header file is allowed only for integer types such as *int* and *size_t*. Non-integer types must be done the other way around, leaving the value out of the header file and defining this value in the implementation file. The reason for this difference is that integral values are often used within the class definition to define something such as an array size.

1. DECLARED CONSTANTS: PAGE 12
2. CONSTANT MEMBER FUNCTIONS: PAGE 35
3. CONST REFERENCE PARAMETERS: PAGE 69
4. STATIC MEMBER CONSTANTS
5. CONST PARAMETERS THAT ARE POINTERS OR ARRAYS: PAGE 157
6. THE CONST KEYWORD WITH A POINTER TO A NODE, AND THE NEED FOR TWO VERSIONS OF SOME MEMBER FUNCTIONS: PAGE 212
7. CONST ITERATORS: PAGE 298

Older Compilers Do Not Support Initialization of Static Member Constants

The ability to initialize and use a static member constant within the class definition is a relatively new feature. If you have an older compiler that does not support static constant members, then Appendix E, “Dealing with Older Compilers,” provides an alternative for your programming.

The Bag Class—Documentation

We now know enough about the bag class to write the documentation of the header file, as shown in Figure 3.1. We've used the name `bag1.h` for this header file because it is the first of several different kinds of bags that we plan to implement.

The documentation includes information about the two typedef statements (`value_type` and `size_type`) and the static member constant (`CAPACITY`). In particular, notice that we have been very specific about what sort of data type is required for the `value_type`. The `value_type` may be any of the C++ built-in data types (such as `int` or `char`), or it may be a class with a default constructor, an assignment operator, and operators to test for equality (`x == y`) and non-equality (`x != y`).

Take a moment to read and understand all of the preconditions in Figure 3.1, such as this precondition for the `+=` operator:

Precondition: `size() + addend.size() <= CAPACITY`.

In this precondition, `size()` refers to the size of the bag that activates the function, and `CAPACITY` refers to the capacity of the bag that activates the function. On the other hand, `addend.size()` refers to the size of the addend, which is a parameter of the function.

Documenting the Value Semantics

One of the requirements for the `value_type` may seem peculiar—why do we require that `value_type` “must have an assignment operator”? Doesn't every data type permit assignments such as `x = y`? Won't there always be an automatic assignment operator? No! For example, `x = y` is forbidden when `x` and `y` are arrays. Later we will see other data types that forbid assignments, or at least require care in defining what the assignment operator actually means.

FIGURE 3.1 Documentation for the Bag Header File

Documentation for a Header File

```
// FILE: bag1.h
// CLASS PROVIDED: bag (part of the namespace main_savitch_3)
//
// TYPEDEFS and MEMBER CONSTANTS for the bag class:
//     typedef ____ value_type
//     bag::value_type is the data type of the items in the bag. It may be any of the C++
//     built-in types (int, char, etc.), or a class with a default constructor, an assignment
//     operator, and operators to test for equality (x == y) and non-equality (x != y).
//
// (continued)
```

(FIGURE 3.1 continued)

```

//  typedef ____ size_type
//      bag::size_type is the data type of any variable that keeps track of how many items
//      are in a bag.
//
//  static const size_type CAPACITY = ____
//      bag::CAPACITY is the maximum number of items that a bag can hold.
//
//  CONSTRUCTOR for the bag class:
//  bag( )
//      Postcondition: The bag has been initialized as an empty bag.
//
//  MODIFICATION MEMBER FUNCTIONS for the bag class:
//  size_type erase(const value_type& target)
//      Postcondition: All copies of target have been removed from the bag.
//      The return value is the number of copies removed (which could be zero).
//
//  bool erase_one(const value_type& target)
//      Postcondition: If target was in the bag, then one copy has been removed;
//      otherwise the bag is unchanged. A true return value indicates that one
//      copy was removed; false indicates that nothing was removed.
//
//  void insert(const value_type& entry)
//      Precondition: size() < CAPACITY.
//      Postcondition: A new copy of entry has been added to the bag.
//
//  void operator +=(const bag& addend)
//      Precondition: size() + addend.size() <= CAPACITY.
//      Postcondition: Each item in addend has been added to this bag.
//
//  CONSTANT MEMBER FUNCTIONS for the bag class:
//  size_type size( ) const
//      Postcondition: The return value is the total number of items in the bag.
//
//  size_type count(const value_type& target) const
//      Postcondition: The return value is number of times target is in the bag.
//
//  NONMEMBER FUNCTIONS for the bag class:
//  bag operator +(const bag& b1, const bag& b2)
//      Precondition: b1.size() + b2.size() <= bag::CAPACITY.
//      Postcondition: The bag returned is the union of b1 and b2.
//
//  VALUE SEMANTICS for the bag class:
//      Assignments and the copy constructor may be used with bag objects.

```

The Bag Class—Demonstration Program

With the documentation in hand, we can write a program that uses a bag. We don't need to know how the functions are implemented. As an example, a demonstration program appears in Figure 3.2. The program asks a user about the ages of family members. The user enters the ages followed by a negative number to indicate the end of the input, and these ages are put into a bag. The program then asks the user to type the ages again, as a simple test. A typical dialogue with the program looks like this:

```
Type the ages in your family.
Type a negative number when you are done:
5 19 47 -1
Type those ages again. Press return after each age:
19
Yes, I've found that age and removed it.
36
No, that age does not occur!
5
Yes, I've found that age and removed it.
47
Yes, I've found that age and removed it.
May your family live long and prosper.
```

FIGURE 3.2 Demonstration Program for the Bag Class

A Program

```
// FILE: bag_demo.cxx
// This is a small demonstration program showing how the bag class is used.
#include <iostream>    // Provides cout and cin
#include <cstdlib>     // Provides EXIT_SUCCESS
#include "bag1.h"      // With value_type defined as an int
using namespace std;
using namespace main_savitch_3;

// PROTOTYPES for functions used by this demonstration program:
void get_ages(bag& ages);
// Postcondition: The user has been prompted to type in the ages of family members. These
// ages have been read and placed in the ages bag, stopping when the bag is full or when the
// user types a negative number.

void check_ages(bag& ages);
// Postcondition: The user has been prompted to type in the ages of family members again.
// Each age is removed from the ages bag when it is typed, stopping when the bag is empty.
```

(continued)

(FIGURE 3.2 continued)

```

int main( )
{
    bag ages;

    get_ages(ages);
    check_ages(ages);
    cout << "May your family live long and prosper." << endl;
    return EXIT_SUCCESS;
}

void get_ages(bag& ages)
{
    int user_input;

    cout << "Type the ages in your family." << endl;
    cout << "Type a negative number when you are done:" << endl;
    cin >> user_input;
    while (user_input >= 0)
    {
        if (ages.size( ) < ages.CAPACITY)
            ages.insert(user_input);
        else
            cout << "I have run out of room and can't add that age." << endl;
        cin >> user_input;
    }
}

void check_ages(bag& ages)
{
    int user_input;

    cout << "Type those ages again. Press return after each age:" << endl;
    while (ages.size( ) > 0)
    {
        cin >> user_input;
        if (ages.erase_one(user_input))
            cout << "Yes, I've found that age and removed it." << endl;
        else
            cout << "No, that age does not occur!" << endl;
    }
}

```

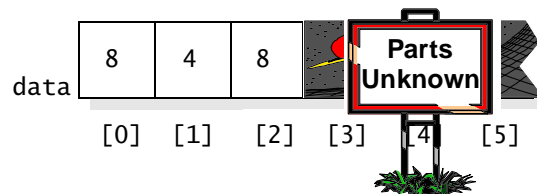
The Bag Class—Design

There are several ways to design the bag class. For now, we'll keep things simple and design a somewhat inefficient data structure using an array. The data structure will be redesigned several times to allow more efficient functions.

We start the design by thinking about the data structure—the actual configuration of private member variables used to implement the class. The primary structure for our design is an array that stores the items of a bag. Or, to be more precise, we use *the beginning part* of a large array. Such an array is called a **partially filled array**. For example, if the bag contains the integer 4 and two copies of 8, then the first part of the array could look this way:

*use the
beginning part
of an array*

Components of
the partially filled
array contain the
items of the bag.



This array will be one of the private member variables of the bag class. The length of the array will be determined by the constant `CAPACITY`, but as the picture indicates, when we are using the array to store a bag with just three items, we don't care what appears beyond the first three components. Starting at index 3, the array might contain all zeros, or it might contain garbage, or our favorite number—it really doesn't matter.

Because part of the array can contain garbage, the bag class must keep track of one other item: *How much of the array is currently being used?* For example, in the picture above, we are using only the first three components of the array because the bag contains three items. The amount of the array being used can be as small as zero (an empty bag) or as large as `CAPACITY` (a full bag). The amount increases as items are added to the bag, and it decreases as items are removed. In any case, we will keep track of the amount in a private member variable called `used`. With this approach, there are two private members for a bag. Notice that the total size of the array is determined by the `CAPACITY` constant.

*the bag's
member
variables*

```
class bag
{
public:
    // TYPEDEFS and MEMBER CONSTANTS
    typedef int value_type;
    typedef std::size_t size_type;
    static const size_type CAPACITY = 30;
    || The rest of the public members will be listed later.
private:
    value_type data[CAPACITY]; // An array to store items
    size_type used;           // How much of the array is used
};
```

*two private
member variables
for the bag*

Pitfall: The value_type Type Must Have a Default Constructor

The `value_type` is used as the component type of an array in the private member variable shown here:

```
class bag
{
...
private:
    value_type data[CAPACITY]; // An array to store items
...
}
```

If the `value_type` is a class with constructors (rather than one of the C++ built-in types), then the compiler must initialize each component of the data array using the item's default constructor. This is why our bag documentation includes the statement that the `value_type` type must be "a class with a default constructor..."

The point to remember is that when an array has a component type that is a class, the compiler uses the default constructor to initialize the array components.

The Invariant of a Class

We've defined the bag data structure, and we have a good intuitive idea of how the structure will be used to represent a bag of items. But as an aid in implementing the class we should also write down an explicit statement of how the data structure is used to represent a bag. In the case of the bag, we need to state how the member variables of the bag class are used to represent a bag of items. There are two rules for our bag implementation:

1. The number of items in the bag is stored in the member variable `used`.
2. For an empty bag, we do not care what is stored in any of `data`; for a non-empty bag, the items in the bag are stored in `data[0]` through `data[used-1]`, and we don't care what is stored in the rest of `data`.

rules that dictate how the member variables are used to represent a value

The rules that dictate how the member variables of a class represent a value (such as a bag of items) are called the **invariant of the class**. The knowledge of these rules is essential to the correct implementation of the class's functions. With the exception of the constructors, each function depends on the invariant being valid when the function is called. And each function, including the constructors, has a responsibility of ensuring that the invariant is valid when the function finishes. In some sense, the invariant of a class is a condition that is an *implicit* part of every function's postcondition. And (except for the constructors) it is also an *implicit* part of every function's precondition. The invariant is not usually written as an *explicit* part of the preconditions and postconditions because the programmer who uses the class does not need to know about these conditions. But to the implementor of the class, the invariant is indispensable. In other words, the invariant is a critical part of the implementation of a class, but it has no effect on the way the class is used.

Key Design Concept

The invariant is a critical part of a class's implementation.

PITFALL

The Invariant of a Class

Always make an explicit statement of the rules that dictate how the member variables of a class are used. These rules are called the **invariant of the class**. All of the functions (except the constructors) can count on the invariant being valid when the function is called. Each function also has the responsibility of ensuring that the invariant is valid when the function finishes.

The Bag Class—Implementation

Once the invariant of the bag is stated, the implementation of the functions is relatively simple because there is no interaction between the functions—except for their cooperation at keeping the invariant valid. Let’s discuss each function along with its implementation.

The constructor. The default constructor initializes a bag as an empty bag, and does no other work. The only task involved is to set the member used to zero, which can be accomplished with an inline member function:

implementing the constructor

```
bag( ) { used = 0; }
```

The value semantics. Our documentation indicates that assignments and the copy constructor may be used with a bag. Our plan is to use the automatic assignment operator and the automatic copy constructor, each of which simply copies the member variables from one bag to another. This is fine because the copying process will copy both the data array and the member variable used.

For example, if a programmer has two bags *x* and *y*, then the statement `y = x` will invoke the automatic assignment operator to copy all of *x.data* to *y.data*, and to copy *x.used* to *y.used*. This is exactly what we want the assignment operator to do, and the automatic copy constructor is also correct.

So, our only “work” for the value semantics is confirming that the automatic operations are correct. Don’t you wish all implementations were that easy?

The count member function. To count the number of occurrences of a particular item in a bag, we step through the used portion of the partially filled array. Remember that we are using locations `data[0]` through `data[used-1]`, so the correct loop is shown in this implementation:

implementing the count function

```
bag::size_type bag::count(const value_type& target) const
{
    size_type answer;
    size_type i;
    answer = 0;
    for (i = 0; i < used; ++i)
        if (target == data[i])
            ++answer;
    return answer;
}
```


Pitfall: When to Use the Full Type Name `bag::size_type`

When we implement the `count` function, we must take care to write the return type as shown here:

```
bag::size_type bag::count(const value_type& target)
```

We have used the completely specified type `bag::size_type` rather than just `size_type`. This is because many compilers do not recognize that you are implementing a bag member function until after seeing `bag::count`. In the implementation, after `bag::count`, we may use simpler names such as `size_type` and `value_type`, but before `bag::count`, we should use the full type name `bag::size_type`.

The insert member function. The `insert` function checks that there is room to insert a new item. If so, then the item is placed in the next available location of the array. What is the index of the next available location? For example, if `used` is 3, then `data[0]`, `data[1]`, and `data[2]` are already occupied, and the next location is `data[3]`. In general, the next available location is `data[used]`. We can place the new item in `data[used]`, as shown in this implementation:

```
void bag::insert(const value_type& entry)
// Library facilities used: cassert
{
    assert(size( ) < CAPACITY);
    data[used] = entry;
    ++used;
}
```

See Self-Test Exercise 7 for an alternative approach to these steps.

implementing
insert

Within a member function we can refer to the static member constant `CAPACITY` with no extra notation. This refers to the `CAPACITY` member constant of the bag that activates the `insert` function.

Programming Tip: Make Assertions Meaningful

At the start of the `insert` member function we wrote the assertion:

```
assert(size( ) < CAPACITY);
```

Of course, we could have written “`used < CAPACITY`” instead, but it is better to write assertions with public members (such as the `size` function). The public member is better because it has meaning to the programmer who uses our class. If the assertion fails, that programmer will understand the message “Assertion failed: `size() < CAPACITY`.”

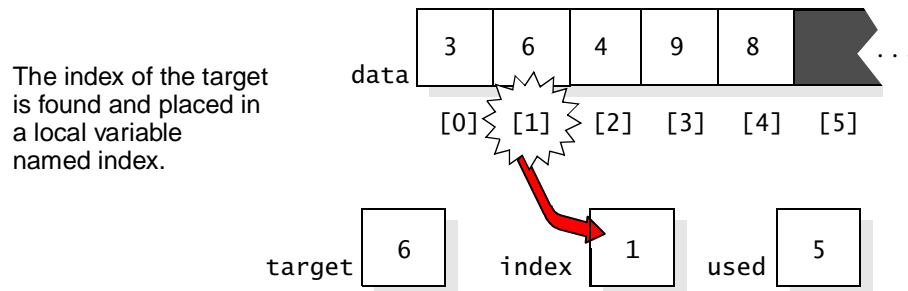
The erase_one member function. The `erase_one` function takes several steps to remove an item named `target` from a bag. In the first step, we find the index of `target` in the bag’s array, and store this index in a local variable named `index`. For example, suppose that `target` is the number 6 in the five-item bag drawn at the top of the next page.

PITFALL



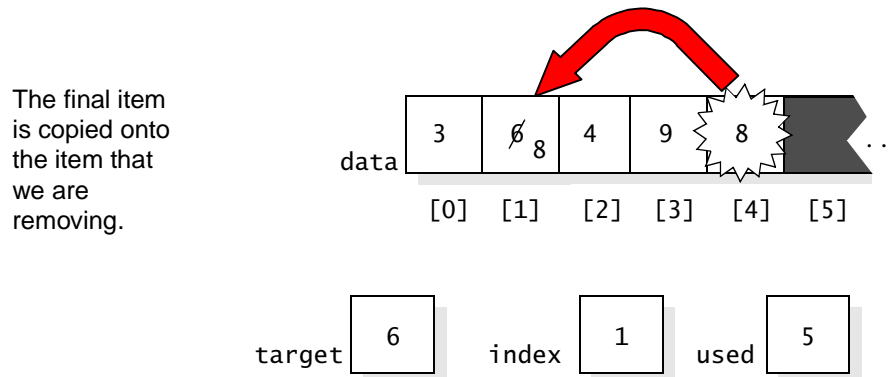
TIP



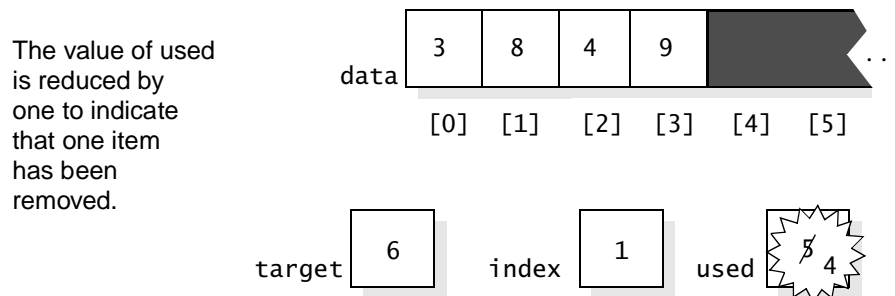


In this example, `target` is a parameter to the `erase_one` member function, `index` is a local variable in the `erase_one` member function, and `used` is the familiar bag member variable. As you can see in the drawing, the first step of `erase_one` was to locate the target (6) and place the index of the target in the local variable named `index`.

Once the index of the target is found, the second step is to take the *final* item in the bag and copy it to `data[index]`. The reason for this copying is so that all the bag's items stay together at the front of the partially filled array, with no holes. In our example, the number 8 is copied to `data[index]` as shown here:



The third step is to reduce the value of `used` by one—in effect reducing the used part of the array by one. In our example, `used` is reduced from 5 to 4:



The code for the `erase_one` function, shown in Figure 3.3, follows these three steps. The only item added is a check that the target is actually in the bag. If we discover that the target is not in the bag, then we do not need to remove anything (and the function returns false). Also note that our function works correctly for the boundary values of removing the first or last item in the array.

Before we continue, we want to point out some programming techniques. Look at the following while-loop from Figure 3.3:

*implementing
erase_one*

```
index = 0;
while ((index < used) && (data[index] != target))
    ++index;
```

To begin, the `index` is set to zero. The boolean expression indicates that the loop continues as long as `index` is still a location in the used part of the array (i.e., `index < used`) and we have not yet found the target (i.e., `data[index] != target`). Each time through the loop, the `index` is incremented by one

FIGURE 3.3 Implementation of the Member Function to Remove an Item

A Member Function Implementation

```
void bag::erase_one(const value_type& target)
// Postcondition: If target was in the bag, then one copy has been removed;
// otherwise the bag is unchanged. A true return value indicates that one
// copy was removed; false indicates that nothing was removed.
{
    size_type index; // The location of target in the data array

    // First, set index to the location of target in the data array, which could be as small as
    // 0 or as large as used-1. If target is not in the array, then index will be set equal to
    // used.
    index = 0;
    while ((index < used) && (data[index] != target))
        ++index;

    if (index == used)
        return false; // target isn't in the bag, so no work to do.

    // When execution reaches here, target is in the bag at data[index].
    // So, reduce used by 1 and copy the last item onto data[index].
    --used;
    data[index] = data[used]; ← See Self-Test Exercise 7 for an
    return true;                alternative approach to this step.
}
```

(++index). No other work is needed in the loop, so the body of the loop has no other statements.

An important programming technique concerns the boolean expression shown here:

```
index = 0;
while ((index < used) && (data[index] != target))
    ++index;
```

Look at the expression `data[index]` in the second part of the test. The valid indexes for `data` range from 0 to `used-1`. But, if the target is not in the array, then `index` will eventually reach `used`, which could be an invalid index. At that point, with `index` equal to `used`, we must not evaluate the expression `data[index]`. In some situations, trying to evaluate `data[index]` with an invalid index can even cause your program to crash. The general rule: *Never use an invalid index with an array.*

short-circuit
evaluation of
logical operations

Avoiding the invalid index is the reason for the first part of the logical test (i.e., `index < used`). Moreover, the test for `(index < used)` must appear *before* the other part of the test. Placing `(index < used)` first ensures that only valid indexes are used. The insurance comes from a technique called *short-circuit evaluation*, which C++ uses to evaluate boolean expressions. In **short-circuit evaluation** a boolean expression is evaluated from left to right, and the evaluation stops as soon as there is enough information to determine the value of the expression. In our example, if `index` equals `used`, then the first part of the logical expression `(index < used)` is false, so the entire `&&` expression *must* be false. It doesn't matter whether the second part of the `&&` expression is true or false. Therefore, C++ doesn't bother to evaluate the second part of the expression, and the potential error of an invalid index is avoided.

The operator +=. The operator `+=` is a member function. Most of the work of this function is accomplished by a loop that copies each of the items from `addend.data` to the `data` array of the object that activates `+=`. One possible implementation uses a loop, something like this:

```
void bag::operator +=(const bag& addend)
{
    ...
    for (i = 0; i < number of items to copy; ++i)
    {
        data[used] = addend.data[i];
        ++used;
    }
}
```

implementing
operator +=

The key assignment statement in the loop is highlighted. On the left of the assignment we have written `data[used]`, which is the next available location of the `data` array for the object that activated the function. On the right of the assignment we have written `addend.data[i]`, which is item number `i` from the `data` array that we are copying.

There's nothing wrong with the loop-based implementation, but an alternative that avoids an explicit loop is shown in Figure 3.4. The implementation uses the copy function from the `<algorithm>` Standard Library. This function can copy items from one array to another, as described in the following C++ Feature.

C++ Feature: The Copy Function from the C++ Standard Library

The Standard Library contains a copy function for easy copying of items from one location to another. The function is part of the `std` namespace in the `<algorithm>` facility, and is used as follows:

```
copy(<beginning location>, <ending location>, <destination>);
```

The function starts at the specified beginning location and copies an item to the destination. It continues beyond the beginning location, copying more and more items to the next spot of the destination, until we are about to copy the ending location. The ending location is **not** copied. All three parameters are often locations within arrays. For example, suppose that `b` and `c` are arrays. To copy the items `b[0]...b[9]` into locations `c[40]...c[49]`, we could write:

```
copy(b, b + 10, c + 40);
```

This call to copy starts copying items from `b[0]`, `b[1]`, `b[2]`, It stops when it reaches `b[10]` (and `b[10]` is not copied). The copied items go into array `c`, at locations `c[40]`, `c[41]`, `c[42]`, The destination must not overlap the source.

As shown in this example, to specify a location that is at the start of an array, just use the array name (such as `b`). To specify a location at index `i` of an array, write the array name followed by `+ i` (such as `b + 10` or `c + 40`).

The statement `copy(addend.data, addend.data + addend.used, data + used)` is used in Figure 3.4 to copy items from `addend.data` into the `data` array. The copied items come from the start of `addend.data`, continuing up to but not including `addend.data[addend.used]`. The copied items are placed in the `data` array starting at location `data[used]`.

FEATURE



FIGURE 3.4 Implementation of the Operator `+=` Member Function

A Member Function Implementation

```
void bag::operator +=(const bag& addend)
// Precondition: size() + addend.size() <= CAPACITY.
// Postcondition: Each item in addend has been added to this bag.
// Library facilities used: algorithm, cassert
{
    assert(size() + addend.size() <= CAPACITY);
    copy(addend.data, addend.data + addend.used, data + used);
    used += addend.used;
}
```

The copy function is from the `<algorithm>` part of the C++ Standard Library

The operator +. The operator + is different from our other functions. It is an ordinary function rather than a member function. The function must take two bags, add them together into a third bag, and return this third bag. The “third bag” is declared as a local variable called `answer` in this implementation:

```
bag operator +(const bag& b1, const bag& b2)
// Library facilities used: cassert
{
    bag answer;

    assert(b1.size( ) + b2.size( ) <= bag::CAPACITY);

    answer += b1;
    answer += b2;
    return answer;
}
```

Add in the items of b1.

Add in the items of b2.

Notice that this function does not need to be a friend function. Why not? (See the answer to Self-Test Exercise 5.) Also, the function implementation can access the static member constant with the notation `bag::CAPACITY`.

The Bag Class—Putting the Pieces Together

Only the `erase` and `size` functions remain to be implemented. We’ll leave `erase` as an exercise (it is similar to `erase_one`), and `size` will be an inline function of the class definition shown in the completed header file of Figure 3.5 on page 113. Notice that in the header file we also list the prototype of the bag’s `operator +` function. This is not a member function, so the prototype appears *after* the end of the bag class definition.

All the function implementations are collected in the implementation file of Figure 3.6 on page 114.



TIP

Programming Tip: Document the Class Invariant in the Implementation File

We wrote the invariant for the bag class at the top of the implementation file in Figure 3.6. This is the best place to document the class’s invariant. In particular, do not write the invariant in the header file, because a programmer who uses the class does not need to know about how the invariant dictates the use of private fields. But the programmer who implements the class does need to know about the invariant.

FIGURE 3.5 Header File for the Bag Class**A Header File**

```

// FILE: bag1.h
// CLASS PROVIDED: bag (part of the namespace main_savitch_3)

|| See Figure 3.1 on page 100 for the other documentation that goes here.

#ifndef MAIN_SAVITCH_BAG1_H
#define MAIN_SAVITCH_BAG1_H
#include <cstdlib> // Provides size_t

namespace main_savitch_3
{
    class bag
    {
    public:
        // TYPEDEFS and MEMBER CONSTANTS
        typedef int value_type;
        typedef std::size_t size_type;
        static const size_type CAPACITY = 30;
        // CONSTRUCTOR
        bag( ) { used = 0; }
        // MODIFICATION MEMBER FUNCTIONS
        size_type erase(const value_type& target);
        bool erase_one(const value_type& target);
        void insert(const value_type& entry);
        void operator +=(const bag& addend);
        // CONSTANT MEMBER FUNCTIONS
        size_type size( ) const { return used; }
        size_type count(const value_type& target) const;
    private:
        value_type data[CAPACITY]; // The array to store items
        size_type used;             // How much of array is used
    };

    // NONMEMBER FUNCTIONS for the bag class
    bag operator +(const bag& b1, const bag& b2);
}

#endif

```

If your compiler does not permit initialization of static constants, see Appendix E.

FIGURE 3.6 Implementation File for the Bag Class**An Implementation File**

```

// FILE: bag1.cxx
// CLASS IMPLEMENTED: bag (see bag1.h for documentation)
// INVARIANT for the bag class:
//   1. The number of items in the bag is in the member variable used.
//   2. For an empty bag, we do not care what is stored in any of data; for a non-empty bag,
//       the items in the bag are stored in data[0] through data[used-1], and we don't care
//       what's in the rest of data.

#include <algorithm> // Provides copy function
#include <cassert>   // Provides assert function
#include "bag1.h"
using namespace std;

namespace main_savitch_3
{
    const bag::size_type bag::CAPACITY;

    bag::size_type bag::erase(const value_type& target)
    {
        || See the solution to Self-Test Exercise 6 on page 134.
    }

    bool bag::erase_one(const value_type& target)
    {
        size_type index; // The location of target in the data array

        // First, set index to the location of target in the data array,
        // which could be as small as 0 or as large as used-1.
        // If target is not in the array, then index will be set equal to used.
        index = 0;
        while ((index < used) && (data[index] != target))
            ++index;

        if (index == used) // target isn't in the bag, so no work to do
            return false;

        // When execution reaches here, target is in the bag at data[index].
        // So, reduce used by 1 and copy the last item onto data[index].
        --used;
        data[index] = data[used];
        return true;
    }
}

```

See "Static Member Constants" on page 99 for an explanation of this line.

See Self-Test Exercise 7 for an alternative approach to this step.

(continued)

(FIGURE 3.6 continued)

```

void bag::insert(const value_type& entry)
// Library facilities used: cassert
{
    assert(size( ) < CAPACITY);
    data[used] = entry;
    ++used;
}

void bag::operator +=(const bag& addend)
// Library facilities used: algorithm, cassert
{
    assert(size( ) + addend.size( ) <= CAPACITY);
    copy(addend.data, addend.data + addend.used, data + used);
    used += addend.used;
}

bag::size_type bag::count(const value_type& target) const
{
    size_type answer;
    size_type i;

    answer = 0;
    for (i = 0; i < used; ++i)
        if (target == data[i])
            ++answer;
    return answer;
}

bag operator +(const bag& b1, const bag& b2)
// Library facilities used: cassert
{
    bag answer;

    assert(b1.size( ) + b2.size( ) <= bag::CAPACITY);

    answer += b1;
    answer += b2;
    return answer;
}

```

See Self-Test Exercise 7 for an alternative approach to these steps.

The copy function is from the <algorithm> part of the C++ Standard Library

The Bag Class—Testing

Thus far, we have focused on the design and implementation of new classes, including new member functions and operator overloading. But it's also important to continue practicing the other aspects of software development, particularly testing. Each of the bag's new functions must be tested, including the overloaded operators. As shown in Chapter 1, it is important to concentrate the testing on boundary values. At this point, we will alert you to only one potential pitfall, leaving the complete testing to Programming Project 1 on page 136.

PITFALL

Pitfall: An Object Can Be an Argument to Its Own Member Function

The same variable is sometimes used on both sides of an assignment or other operator. For example, the value of an integer `d` is doubled by the highlighted statement here:

```
int d = 5;
d += d;
```

Add the current value of `d` to `d`, giving it a value of 10.

A similar technique can be used with a bag, as shown here:

```
bag b;
b.insert(5);
b.insert(2);
b += b;
```

`b` now contains a 5 and a 2.

Now `b` contains two 5s and two 2s.

The highlighted statement takes all the items in `b` (the 5 and the 2) and adds them to what's already in `b`, so `b` ends up with two copies of each number.

In the `+=` statement, the bag `b` is activating the `+=` operator, but this same bag `b` is the actual argument to the operator. This is a situation that must be carefully tested. As an example of the danger, consider the incorrect implementation of `+=` in Figure 3.7. Do you see what goes wrong with `b += b`? (See the answer to Self-Test Exercise 8.)

The situation: A member function has a parameter type that is the same as the member function's class. For example, the bag's `+=` operator has a parameter that is itself a bag.

The danger: The member function might fail when an object activates the member function, and the same object is used as the actual argument. For example, a bag `b` could be used in the statement: `b += b`.

Always test this special situation.

FIGURE 3.7 Wrong Implementation of the Bag's += Operator**A Wrong Member Function Implementation**

```

void bag::operator +=(const bag& addend)
// Library facilities used: cassert
{
    size_type i; // An array index

    assert(size( ) + addend.size( ) <= CAPACITY);

    for (i = 0; i < addend.used; ++i)
    {
        data[used] = addend.data[i];
        ++used;
    }
}

```

WARNING!

There is a bug in this implementation. See Self-Test Exercise 8.

The Bag Class—Analysis

We finish this section with a time analysis of the bag's functions. We'll use the number of items in a bag as the input size. For example, if *b* is a bag containing *n* integers, then the number of operations required by *b.count* is a formula involving *n*. To count the operations, we'll count the number of statements executed by the function, although we won't need an exact count since our answer will use big-*O* notation. Except for the return statement, all of the work in *count* happens in this loop:

```

for (i = 0; i < used; ++i)
    if (target == data[i])
        ++answer;

```

We can see that the body of the loop will be executed exactly *n* times—once for each item in the bag. The body of the loop also has another important property: The body contains no other loops or calls to functions that contain loops. This is enough to conclude that the total number of statements executed by *count* is no more than:

$$n \times (\text{number of statements in the loop}) + 3$$

The “+3” at the end is for the initialization of *i*, the final test of (*i* < *used*), and the return statement. Regardless of how many statements are actually in the loop, the time expression is *always* *O*(*n*)—so the *count* function is linear.

constant time
 $O(1)$

A similar analysis shows that `erase_one` is also linear, although its loop sometimes executes fewer than n times. However, the fact that `erase_one` *sometimes* requires fewer than $n \times (\text{number of statements in the loop})$ does not change the fact that the function is $O(n)$. In the worst case, the loop does execute a full n iterations, therefore the correct time analysis is no better than $O(n)$.

Several of the other bag functions do not contain any loops at all, and do not call any functions with loops. This is a pleasant situation because the time required for any of these functions does not depend on the number of items in the bag. For example, when an item is added to a bag, the new item is always placed at the end of the array, and the `insert` function never looks at the items that were already in the bag. When the time required by a function does not depend on the size of the input, the procedure is called **constant time**, which is written $O(1)$. But be careful in analyzing the `+=` operator. Its call to the copy function requires time that is proportional to the size of the addend bag, so it is not constant time.

The time analyses of all the functions are summarized in Figure 3.8.

Self-Test Exercises

1. The bag's documentation in Figure 3.1 on page 100 says that the `value_type` may be a class, but only if it has a default constructor and several operators. Why?
2. Draw a picture of `mybag.data` after these statements:


```
bag mybag;
mybag.insert(1);
mybag.insert(2);
mybag.insert(3);
mybag.erase_one(1);
```
3. Write the invariant of the bag class.
4. Use the copy function to copy six elements from the start of an array `x` into an array `y` starting at `y[42]`.
5. Why isn't the bag's *operator +* function a friend function?
6. Implement the bag's `erase` member function.

FIGURE 3.8 Time Analysis for the Bag Functions (First Version)

Operation	Time Analysis		Operation	Time Analysis	
Default constructor	$O(1)$	Constant time	<code>+= another bag</code>	$O(n)$	n is the size of the other bag
<code>count</code>	$O(n)$	n is the size of the bag	<code>b1 + b2</code>	$O(n_1 + n_2)$	n_1 and n_2 are the sizes of the bags
<code>erase_one</code>	$O(n)$	Linear time	<code>insert</code>	$O(1)$	Constant time
<code>erase</code>	$O(n)$	Linear time	<code>size</code>	$O(1)$	Constant time

7. Rewrite the last two statements of `erase_one` (Figure 3.3 on page 109) as a single statement, using the expression `--used` as the index. (If you are unsure of the meaning of `--used` as an index, then go ahead and peek at our answer at the back of the chapter.) Use `used++` as the index to make a similar alteration to the `insert` function member.
8. Suppose we implement the `+=` operator as shown in Figure 3.7 on page 117. What goes wrong with `b += b`?

3.2 PROGRAMMING PROJECT: THE SEQUENCE CLASS

You are ready to tackle a container class implementation on your own. The class is a container class called a **sequence**. A sequence is similar to a bag—both contain a bunch of items. But unlike a bag, the items in a sequence are arranged in an order, one after another.

How does this differ from a bag? After all, aren't the bag items arranged one after another in the partially filled array that implements the bag? Yes, but that's a quirk of our particular bag implementation, and the order is just haphazard.

*how a sequence
differs from a bag*

In contrast, the items of a sequence are kept one after another, and member functions will allow a program to step through the sequence one item at a time. Member functions also permit a program to control precisely where items are inserted and removed within the sequence. The technique of using member functions to access items is called an **internal iterator**, which differs from **external iterators** of the Standard Library containers. Later, in Chapter 6, we will examine external iterators in detail and add them to both the bag and the sequence.

*internal iterators
versus
external iterators*

The Sequence Class—Specification

Our sequence is a class that depends on an underlying `value_type`, and the class also provides a `size_type`. It's a good habit to use these particular names for all our classes since you'll find the same names for the Standard Library container classes. At the moment, a sequence will be limited to no more than 30 items. As with our bag, the `value_type`, `size_type`, and sequence capacity will be defined in the public section of the class definition. Throughout the discussion, we will use examples in which the items are *double* numbers, and the sequence has no more than 30 items. So the header file has these definitions:

```
class sequence
{
public:
    // TYPEDEF and MEMBER CONSTANTS
    typedef double value_type;
    typedef std::size_t size_type;
    static const size_type CAPACITY = 30;
    ...
}
```

Keep in mind that the capacity and item type can easily be changed and recompiled if we need other kinds of sequences. Also, remember the alternatives if your compiler does not support this way of initializing a static constant in a class definition (see Appendix E).

The class that we implement will be called `sequence`. We'll now specify the member functions of this new class:

Default constructor. The sequence class has just one constructor—a default constructor that creates an empty sequence.

The size member function. The size member function returns the number of items in the sequence. The prototype is given here along with the postcondition:

```
size_type size( ) const;
// Postcondition: The return value is the number of items in the sequence.
```

For example, if `scores` is a sequence containing the values 10.1, 40.2, and 1.1, then `scores.size()` returns 3. Throughout our examples, we will draw sequences vertically, with the first item on top, as shown in the picture in the margin (where the first item is 10.1).

10.1
40.2
1.1

Member functions to examine a sequence. We will have member functions to build a sequence, but it will be easier to first explain the member functions that examine a sequence which has already been built. Now, with the bag class, all that we can do is inquire how many copies of a particular item are in the bag. A sequence is more flexible, allowing us to examine the items one after another. The items must be examined in order, from the front to the back of the sequence. Three member functions work together to enforce the in-order retrieval rule. The functions' prototypes are given here:

```
void start( );
value_type current( ) const;
void advance( );
```

When we want to retrieve the items in a sequence, we begin by activating `start`. After activating `start`, the `current` function returns the first item in the sequence. Each time we call `advance`, the `current` function changes so that it returns the next item in the sequence. For example, if a sequence named `numbers` contains the four numbers 37, 10, 83, and 42, then we can write the following code to print the first three numbers:

```
start,
current,
advance
numbers.start( );
cout << numbers.current( ) << endl; ← Prints 37
numbers.advance( );
cout << numbers.current( ) << endl; ← Prints 10
numbers.advance( );
cout << numbers.current( ) << endl; ← Prints 83
```

One other member function cooperates with `current`. The function, called `is_item`, returns a boolean value to indicate whether there actually is another item for `current` to provide, or whether `current` has advanced right off the end. The `is_item` prototype is given here with a postcondition:

```
bool is_item( ) const;
// Postcondition: A true return value indicates that there is a valid
// "current" item that can be obtained from the current member function.
// A false return value indicates that there is no valid current item.
```

Using all four of the member functions in a for-loop, we can print an entire sequence, as shown here for the numbers sequence:

```
for (numbers.start( ); numbers.is_item( ); numbers.advance( ))
    cout << numbers.current( ) << endl;
```

The insert and attach member functions. There are two member functions to add new items to a sequence. One of the functions, called `insert`, places a new item before the current item. For example, suppose that we have created the sequence shown to the right with three items, and that the current item is 8.8. In this example, we want to add 10.0, immediately before the current item. When 10.0 is inserted before the current item, other items—such as 8.8 and 99.0—will move down to make room for the new item. After the insertion, the sequence has the four items shown in the lower box.

If there is no current item, then `insert` places the new item at the front of the sequence. In any case, after the `insert` function returns, the newly inserted item will be the current item, as specified in this precondition/postcondition contract:

```
void insert(const value_type& entry);
// Precondition: size( ) < CAPACITY.
// Postcondition: A new copy of entry has been inserted in the sequence
// before the current item. If there was no current item, then the new entry
// has been inserted at the front. In either case, the new item is now the
// current item of the sequence.
```

A second member function, called `attach`, also adds a new item to a sequence, but the new item is added *after* the current item, as specified here:

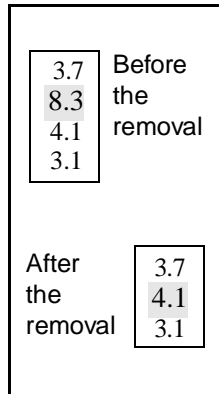
```
void attach(const value_type& entry);
// Precondition: size( ) < CAPACITY.
// Postcondition: A new copy of entry has been inserted in the sequence
// after the current item. If there was no current item, then the new entry
// has been attached to the end. In either case, the new item is now the
// current item of the sequence.
```

42.1
8.8
99.0

The sequence grows by inserting 10.0 before the current item.

42.1
10.0
8.8
99.0

If there is no current item, then the `attach` function places the new item at the end of the sequence (rather than the front). Either `insert` or `attach` can be used to place the first item on a sequence.



The `remove_current` member function. The current item can be removed from a sequence. The member function for a removal has no parameters:

```
void remove_current( );
// Precondition: is_item returns true.
// Postcondition: The current item has been removed from the sequence,
// and the item after this (if there is one) is now the new current item.
```

The function's precondition requires that there is a current item; it is this current item that is removed. For example, suppose `scores` is the four-item sequence shown at the top of the box in the margin, and the highlighted 8.3 is the current item. After activating `scores.remove_current()`, the 8.3 has been deleted, and the 4.1 is now the current item.

The Sequence Class—Documentation

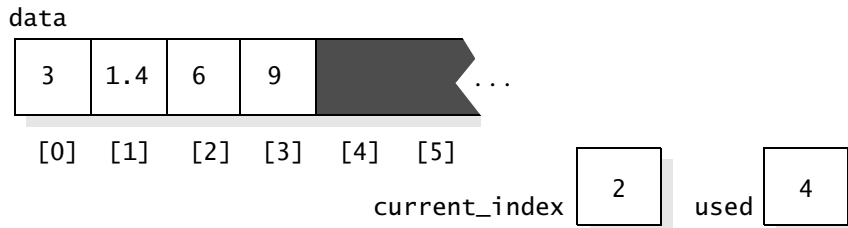
The header file for this first version of our sequence class is shown in Figure 3.9 on page 124. The header file includes the class definition with our suggestion for three member variables. We discuss these member variables next.

The Sequence Class—Design

Our suggested design for the sequence class has three private member variables. The first variable, `data`, is an array that stores the items of the sequence. Just like the bag, `data` is a partially filled array. A second member variable, called `used`, keeps track of how much of the `data` array is currently being used. Therefore, the used part of the array extends from `data[0]` to `data[used-1]`. The third member variable, `current_index`, gives the index of the “current” item in the array (if there is one). If there is no valid current item in the sequence, then `current_index` will be the same number as `used` (since this is larger than any valid index). Here is the complete invariant of our class, stated as three rules:

1. The number of items in the sequence is stored in the member variable `used`.
2. For an empty sequence, we do not care what is stored in any of `data`; for a non-empty sequence, the items are stored in their sequence order from `data[0]` to `data[used-1]`, and we don't care what is stored in the rest of `data`.
3. If there is a current item, then it lies in `data[current_index]`; if there is no current item, then `current_index` equals `used`.

As an example, suppose that a sequence contains four numbers, with the current item at `data[2]`. The member variables of the object might appear as shown here:



In this example, the current item is at `data[2]`, so the `current()` function would return the number 6. At this point, if we called `advance()`, then `current_index` would increase to 3, and `current()` would then return 9.

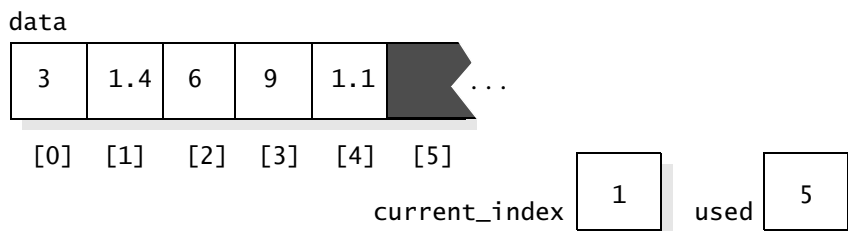
Normally, a sequence has a “current” item, and the member variable `current_index` contains the location of that current item. But if there is no current item, then `current_index` contains the same value as `used`. In our example, if `current_index` was 4, then that would indicate that there is no current item. Notice that this value (4) is beyond the used part of the array (which stretches from `data[0]` to `data[3]`).

The stated requirements for the member variables form the invariant of the sequence class. You should place this invariant at the top of your implementation file (`sequence1.cxx`). We will leave most of this implementation file up to you, but we will offer some hints and a bit of pseudocode.

invariant of the class

The Sequence Class—Pseudocode for the Implementation

The `remove_current` function. This function removes the current item from the sequence. First check that the precondition is valid (use `is_item()` in an assertion). Then remove the current item by shifting each of the subsequent items leftward one position. For example, suppose we are removing the current item from the sequence drawn here:



What is the current item in this picture? It is the 1.4 since `current_index` is 1, and `data[1]` contains the 1.4.

(text continues on page 126)

FIGURE 3.9 Header File for the Sequence Class**A Header File**

```
// FILE: sequence1.h
// CLASS PROVIDED: sequence (part of the namespace main_savitch_3)
//
// TYPEDEF and MEMBER CONSTANTS for the sequence class:
//     typedef ____ value_type
//         sequence::value_type is the data type of the items in the sequence. It may be any of the
//         C++ built-in types (int, char, etc.), or a class with a default constructor, an assignment
//         operator, and a copy constructor
//
//     typedef ____ size_type
//         sequence::size_type is the data type of any variable that keeps track of how many
//         items are in a sequence.
//
//     static const size_type CAPACITY = ____
//         sequence::CAPACITY is the maximum number of items that a sequence can hold.
//
// CONSTRUCTOR for the sequence class:
//     sequence( )
//         Postcondition: The sequence has been initialized as an empty sequence.
//
// MODIFICATION MEMBER FUNCTIONS for the sequence class:
//     void start( )
//         Postcondition: The first item in the sequence becomes the current item (but if the
//         sequence is empty, then there is no current item).
//
//     void advance( )
//         Precondition: is_item returns true.
//         Postcondition: If the current item was already the last item in the sequence, then there
//         is no longer any current item. Otherwise, the new item is the item immediately after
//         the original current item.
//
//     void insert(const value_type& entry)
//         Precondition: size( ) < CAPACITY.
//         Postcondition: A new copy of entry has been inserted in the sequence before the
//         current item. If there was no current item, then the new entry has been inserted at the
//         front. In either case, the new item is now the current item of the sequence.
//
//     void attach(const value_type& entry)
//         Precondition: size( ) < CAPACITY.
//         Postcondition: A new copy of entry has been inserted in the sequence after the current
//         item. If there was no current item, then the new entry has been attached to the end of
//         the sequence. In either case, the new item is now the current item of the sequence.
//
//     void remove_current( )
//         Precondition: is_item returns true.
//         Postcondition: The current item has been removed from the sequence, and the
//         item after this (if there is one) is now the new current item. (continued)
```

(FIGURE 3.9 continued)

```
// CONSTANT MEMBER FUNCTIONS for the sequence class:
//   size_type size( ) const
//       Postcondition: The return value is the number of items in the sequence.
//
//   bool is_item( ) const
//       Postcondition: A true return value indicates that there is a valid "current" item that
//       may be retrieved by the current member function (listed below). A false return value
//       indicates that there is no valid current item.
//
//   value_type current( ) const
//       Precondition: is_item( ) returns true.
//       Postcondition: The item returned is the current item in the sequence.
//
// VALUE SEMANTICS for the sequence class:
//   Assignments and the copy constructor may be used with sequence objects.

#ifndef MAIN_SAVITCH_SEQUENCE_H
#define MAIN_SAVITCH_SEQUENCE_H
#include <cstdlib> // Provides size_t

namespace main_savitch_3
{
    class sequence
    {
    public:
        // TYPEDEFS and MEMBER CONSTANTS
        typedef double value_type;
        typedef std::size_t size_type;
        static const size_type CAPACITY = 30;
        // CONSTRUCTOR
        sequence( );
        // MODIFICATION MEMBER FUNCTIONS
        void start( );
        void advance( );
        void insert(const value_type& entry);
        void attach(const value_type& entry);
        void remove_current( );
        // CONSTANT MEMBER FUNCTIONS
        size_type size( ) const;
        bool is_item( ) const;
        value_type current( ) const;
    private:
        value_type data[CAPACITY];
        size_type used;
        size_type current_index;
    };
}

#endif
```

If your compiler does not permit initialization of static constants, see Appendix E.

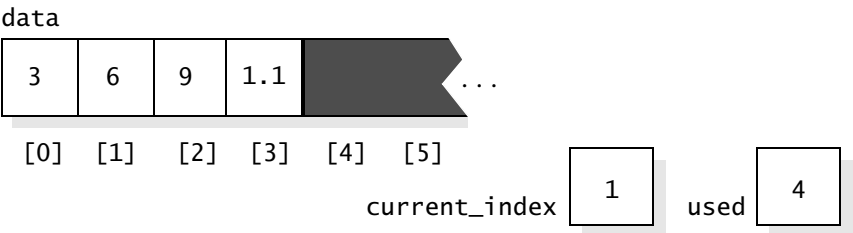
The three private member variables are discussed in the section "The Sequence Class—Design" on page 122.

In the case of the bag, we could remove an element such as 1.4 by copying the final item (1.1) onto the 1.4. But this approach won't work for the *sequence* because the items would lose their sequence order. Instead, each item after the 1.4 must be moved leftward one position. The 6 moves from data[2] to data[1]; the 9 moves from data[3] to data[2]; the 1.1 moves from data[4] to data[3]. This is a lot of movement, but a simple for-loop suffices to carry out all the work. This is the pseudocode:

```
for (i = the index after the current item; i < used; ++i)
    Move an item from data[i] back to data[i-1];
```

do not use the
copy function

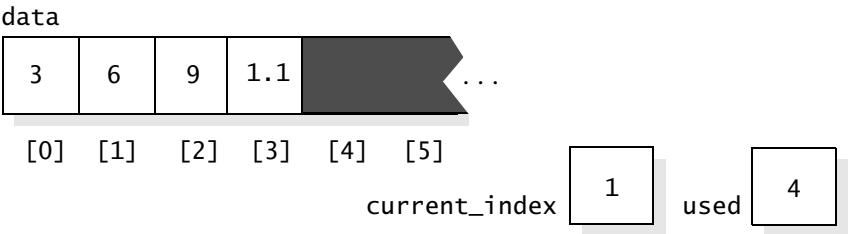
You should not use the copy function from <algorithm> since that function forbids the overlap of the source with the destination.
When the loop completes, you should reduce used by one. The final result for our example is shown here:



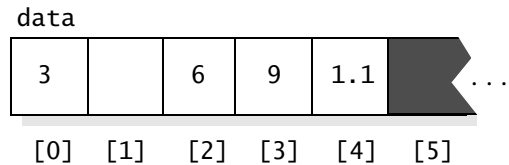
After the removal, the `current_index` is unchanged. In effect, this means that the item that was just after the removed item is now the current item. You should check that the function works correctly for boundary values—removing the first item and removing the final item. In fact, both these cases do work fine. When the final item is removed, `current_index` will end up with the same value as `used`, indicating that there is no longer a current item.

The insert function. If there is a current item, then the insert function must take care to insert the new item just before the current position. Items that are already at or after the current position must be shifted rightward to make room for the new item. We suggest that you start by checking the precondition. Then shift items at the end of the array rightward one position each until you reach the position for the new item.

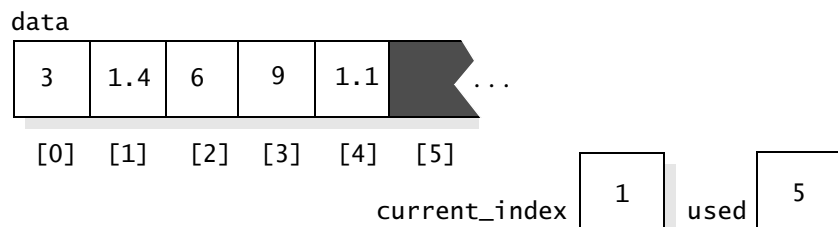
For example, suppose you are inserting 1.4 at the location data[1] in this sequence:



You would begin by shifting the 1.1 rightward from `data[3]` to `data[4]`; then move the 9 from `data[2]` to `data[3]`; then the 6 moves from `data[1]` rightward to `data[2]`. At this point, the array looks like this:



Of course, `data[1]` actually still contains a 6 since we just copied the 6 from `data[1]` to `data[2]`. But we have drawn `data[1]` as an empty box to indicate that `data[1]` is now available to hold the new item (that is, the 1.4 that we are inserting). At this point we can place the 1.4 in `data[1]` and add one to `used`, as shown here:



The pseudocode for shifting the items rightward uses a for-loop. Each iteration of the loop shifts one item, as shown here:

```
for (i = used; data[i] is the wrong spot for entry ; --i)
    data[i] = data[i-1];
```

The key to the loop is the test `data[i] is the wrong spot for entry`. How do we test whether a position is the wrong spot for the new item? A position is wrong if $(i > \text{current_index})$. Can you now write the entire member function in C++? (See the solution to Self-Test Exercise 9, and don't forget to handle the special case when there is no current item.)

Other Member Functions. The other member functions are straightforward; for example, the `attach` function is similar to `insert`. You'll need to watch out for the pitfall about using full names (see page 107). Some additional useful member functions are described in Programming Projects 3 and 4 on page 136.

Self-Test Exercises

9. Write the `insert` function for the sequence. Why should this implementation avoid using the copy function from `<algorithm>`?
10. Suppose that a sequence has 24 items, and there is no current item. According to the invariant of the class, what is `current_index`?
11. Suppose `g` is a sequence with 10 items. You activate `g.start()`, then activate `g.advance()` three times. What value is then in `g.current_index`?
12. What are good boundary values to test the `remove_current` function?
13. Write a demonstration program that asks the user for a list of family member ages, then prints the list in the same order that it was given.
14. Write a new member function to remove a specified item from a sequence. The function has one parameter (the item to remove).
15. For a sequence of numbers, suppose that you attach 1, then 2, then 3, and so on up to n . What is the big- O time analysis for the combined time of attaching all n numbers? How does the analysis change if you insert n first, then $n-1$, and so on down to 1—always using `insert` instead of `attach`?

3.3 INTERACTIVE TEST PROGRAMS

Your sequence class is a good candidate for an interactive test program that follows a standard format. The format, illustrated by the program of Figure 3.10, can be used with any class. The start of the main program declares an object—in this case, a sequence object. The rest of the main program is an interactive loop that continues as long as the user wants. Three things occur inside the loop:

1. A small menu of choices is written for the user. Each choice is printed along with a letter or other meaningful character to allow the user to select the choice.
2. The user's selection from the menu is read.
3. Based on the user's selection, some action is taken on the sequence object.

Our example interactive test program for the sequence is shown in Figure 3.10, with part of a sample dialogue in Figure 3.11 on page 132. Some of the techniques used in the test program are familiar. For example, subtasks, such as printing the menu, are accomplished with functions. Two techniques in the test program may be new to you: converting input to uppercase letters, and acting on the input via a *switch* statement. We'll discuss these two techniques after you've looked through the program.

C++ Feature: Converting Input to Uppercase Letters

Even small test programs should have some flexibility regarding user input. For example, the program should accept either upper- or lowercase letters for each menu choice. We accomplish this by reading the user's input and then, if necessary, converting a lowercase letter to the corresponding uppercase letter. The conversion is carried out by a function `toupper` with this specification:

```
char toupper(char c);
// Postcondition: If c is a lowercase letter, then the return value is the
// uppercase equivalent of c. Otherwise the return value is just c itself.
```

The `toupper` function is part of the `<cctype>` facility. In our main program, we use `toupper` to convert the result of the `get_user_command` function, as shown here:

```
choice = toupper(get_user_command( ));
```

FEATURE ++**FIGURE 3.10** Interactive Test Program for the Sequence Class**A Program**

```
// FILE: sequence_test.cxx
// An interactive test program for the new sequence class
#include <cctype>           // Provides toupper
#include <iostream>        // Provides cout and cin
#include <cstdlib>          // Provides EXIT_SUCCESS
#include "sequence1.h"     // With value_type defined as double
using namespace std;
using namespace main_savitch_3;

// PROTOTYPES for functions used by this test program:
void print_menu( );
// Postcondition: A menu of choices for this program has been written to cout.

char get_user_command( );
// Postcondition: The user has been prompted to enter a one-character command.
// The next character has been read (skipping blanks and newline characters),
// and this character has been returned.

void show_sequence(sequence display);
// Postcondition: The items on display have been printed to cout (one per line).

double get_number( );
// Postcondition: The user has been prompted to enter a real number. The
// number has been read, echoed to the screen, and returned by the function. (continued)
```

(FIGURE 3.10 continued)

```

int main( )
{
    sequence test; // A sequence that we'll perform tests on
    char choice;   // A command character entered by the user

    cout << "I have initialized an empty sequence of real numbers." << endl;

    do
    {
        print_menu( );
        choice = toupper(get_user_command( ));
        switch (choice)
        {
            case '!': test.start( );
                       break;
            case '+': test.advance( );
                       break;
            case '?': if (test.is_item( ))
                       cout << "There is an item." << endl;
                       else
                           cout << "There is no current item." << endl;
                       break;
            case 'C': if (test.is_item( ))
                       cout << "Current item is: " << test.current( ) << endl;
                       else
                           cout << "There is no current item." << endl;
                       break;
            case 'P': show_sequence(test);
                       break;
            case 'S': cout << "Size is " << test.size( ) << "." << endl;
                       break;
            case 'I': test.insert(get_number( ));
                       break;
            case 'A': test.attach(get_number( ));
                       break;
            case 'R': test.remove_current( );
                       cout << "The current item has been removed." << endl;
                       break;
            case 'Q': cout << "Ridicule is the best test of truth." << endl;
                       break;
            default:  cout << choice << " is invalid." << endl;
        }
    }
    while ((choice != 'Q'));

    return EXIT_SUCCESS;
}

```

(continued)

(FIGURE 3.10 continued)

```

void print_menu( )
// Library facilities used: iostream
{
    cout << endl; // Print blank line before the menu
    cout << "The following choices are available: " << endl;
    cout << " !   Activate the start( ) function" << endl;
    cout << " +   Activate the advance( ) function" << endl;
    cout << " ?   Print the result from the is_item( ) function" << endl;
    cout << " C   Print the result from the current( ) function" << endl;
    cout << " P   Print a copy of the entire sequence" << endl;
    cout << " S   Print the result from the size( ) function" << endl;
    cout << " I   Insert a new number with the insert(...) function" << endl;
    cout << " A   Attach a new number with the attach(...) function" << endl;
    cout << " R   Activate the remove_current( ) function" << endl;
    cout << " Q   Quit this test program" << endl;
}

char get_user_command( )
// Library facilities used: iostream
{
    char command;

    cout << "Enter choice: ";
    cin >> command; // Input of characters skips blanks and newline character

    return command;
}

void show_sequence(sequence display)
// Library facilities used: iostream
{
    for (display.start( ); display.is_item( ); display.advance( ))
        cout << display.current( ) << endl;
}

double get_number( )
// Library facilities used: iostream
{
    double result;

    cout << "Please enter a real number for the sequence: ";
    cin >> result;
    cout << result << " has been read." << endl;
    return result;
}

```

FIGURE 3.11 Part of a Sample Dialogue from the Program of Figure 3.10**A Sample Dialogue**

I have initialized an empty sequence of real numbers.

The following choices are available:

- ! Activate the start() function
- + Activate the advance() function
- ? Print the result from the is_item() function
- C Print the result from the current() function
- P Print a copy of the entire sequence
- S Print the result from the size() function
- I Insert a new number with the insert(...) function
- A Attach a new number with the attach(...) function
- R Activate the remove_current() function
- Q Quit this test program

Enter choice: **A**

Please enter a real number for the sequence: **3.14**

3.14 has been read.

The following choices are available:

- ! Activate the start() function
- + Activate the advance() function
- ? Print the result from the is_item() function
- C Print the result from the current() function
- P Print a copy of the entire sequence
- S Print the result from the size() function
- I Insert a new number with the insert(...) function
- A Attach a new number with the attach(...) function
- R Activate the remove_current() function
- Q Quit this test program

Enter choice: **S**

Size is 1.

|| The dialogue continues until the user types **Q** to stop the program.

C++ Feature: The Switch Statement

After the user's choice is read, the main program takes an action. The action depends on the single character that the user typed from the menu. An effective statement to select among many possible actions is the *switch* statement, with the general form:

```
switch (<Control value>)
{
    <Body of the switch statement>
}
```

When the switch statement is reached, the control value is evaluated. The program then looks through the body of the switch statement for a matching case label. For example, if the control value is the character 'A', then the program looks for a case label of the form `case 'A':`. If a matching case label is found, then the program goes to that label and begins executing statements. Statements are executed one after another—but if a **break** statement (of the form `break;`) occurs, then the program skips to the end of the body of the switch statement.

If the control value has no matching case label, then the program will look for a **default label** of the form `default:`. This label handles any control values that don't have their own case label.

If there is no matching case label and no default label, then the whole body of the switch statement is skipped.

For an interactive test program, the switch statement has one case label for each of the menu choices. For example, one of the menu choices is the character 'A', which allows the user to attach a new number to the sequence. In the switch statement, the 'A' command is handled as shown here:

```
switch (choice)
{
    ...
    case 'A': test.attach(get_number( ));
              break;
    ...
}
```

FEATURE ++**Self-Test Exercises**

16. What are the values of `toupper('a')`, `toupper('A')`, and `toupper('+')`?
17. What situation calls for a switch statement?
18. The `show_sequence` function on page 131 uses a *value* parameter rather than a *reference* parameter. Why?

CHAPTER SUMMARY

- A *container class* is a class where each object contains a collection of items. Bags and sequences are two examples of container classes; the C++ Standard Library also provides a variety of flexible container classes.
- A container class should be implemented in a way that makes it easy to alter the data type of the underlying items. In C++, the simple approach to this problem uses a typedef statement to define the type of the container's item.
- The simplest implementations of container classes use a *partially filled array*. Using a partially filled array requires each object to have at least two member variables: the array itself and another variable to keep track of how much of the array is being used.
- When you design a class, always make an explicit statement of the rules that dictate how the member variables are used. These rules are called the *invariant of the class*, and should be written at the top of the implementation file for easy reference.
- Small classes can be tested effectively with an *interactive test program* that follows the standard format of our sequence test program.

?

Solutions to Self-Test Exercises

1. The default constructor is required because `value_type` is used as the component type of an array. Each of the required operators (`=`, `==`, and `!=`) is used with the `value_type` in at least one of the bag's member functions.
2.

3	2
---	---

[0] [1]

data[1].

We don't care what appears beyond
3. See the two rules on page 105.
4. `copy(x, x+6, y+42);`
5. It does not need to be a friend function because it does not directly access any private members of the bag.
6.

```

bag::size_type
bag::erase(const value_type& target)
{
    size_type index = 0;
    size_type many_removed = 0;

    while (index < used)
    {
        if (data[index] == target)
        {
            --used;
            data[index] = data[used];
            ++many_removed;
        }
        else
            ++index;
    }

    return many_removed;
}

```

7. The two statements can be replaced by one statement: `data[index] = data[--used];` When `--used` appears as an expression, the variable `used` is decremented by one, and the resulting value is the value of the expression. (On the other hand, if `used--` appears as an expression, the value of the expression is the value of `used` prior to subtracting one.) Similarly, the last two statements of `insert` can be combined to `data[used++] = entry;`. In this case, we have the expression `used++` as the index because we want to use the old value of `used` (before adding one) as the index.
8. If we activate `b += b`, then the private member variable `used` is the same variable as `addend.used`. Each iteration of the loop adds 1 to `used`, and hence `addend.used` is also increasing, and the loop never ends. To correct the problem, you could store the initial value of `addend.used` in a local variable, and use this local variable to determine when the loop ends.
9.

```
void sequence::insert
(const value_type& entry)
{
    size_type i;

    assert(size() < CAPACITY);

    if (!is_item())
        current_index = 0;
    for (i = used; i > current_index; --i)
        data[i] = data[i-1];
    data[current_index] = entry;
    ++used;
}
```
10. 24
11. `g.current_index` will be 3 (since the 4th item occurs at `data[3]`).
12. The `remove_current` function should be tested when the sequence size is just 1, and when the sequence is at its full capacity. At full capacity you should try removing the first item, and the last item in the sequence.
13. Your program can be similar to Figure 3.2 on page 102.
14. Here is our function's prototype, with a post-condition:
- ```
void
remove(const value_type& target);
// Postcondition: If target was in the
// sequence then the first copy of target has
// been removed, and the item after
// the removed item (if there is one)
// becomes the new current item; otherwise
// the sequence remains unchanged.
```
- The easiest implementation searches for the index of the target. If this index is found, then set `current_index` to this index, and activate the ordinary `remove_current` function.
15. The total time to attach 1, 2, ...,  $n$  is  $O(n)$ . The total time to insert  $n, n-1, \dots, 1$  is  $O(n^2)$ . The larger time for the insert is because an insertion at the front of the sequence requires all of the existing items to be shifted right to make room for the new item. Hence, on the second insertion, one item is shifted. On the third insertion, two items are shifted. And so on to the  $n^{\text{th}}$  item, which needs  $n-1$  shifts. The total number of shifts is  $1+2+\dots+(n-1)$ , which is  $O(n^2)$ . (To show that this sum is  $O(n^2)$ , use a technique similar to that used in Figure 1.2 on page 17.)
16. The first two calls return 'A'. The function call `toupper('+')` returns '+'.
17. Use a switch statement when a single control value determines which of several possible actions is to be taken.
18. With a reference parameter, the advancing of the current element through the sequence would alter the actual argument.



## PROGRAMMING PROJECTS

For more in-depth projects, please see [www.cs.colorado.edu/~main/projects/](http://www.cs.colorado.edu/~main/projects/)

**1** A **black box** test of a class is a program that tests the correctness of the class's member functions without directly examining the private members of the class. You can imagine that the private members are inside an opaque black box where they cannot be seen, so all testing must occur only through activating the public member functions.

Write a black box test program for the `bag` class. Make sure that you test the boundary values, such as an empty bag, a bag with one item, and a full bag.

**2** Implement operators for `-` and `-=` for the `bag` class from Section 3.1. For two bags `x` and `y`, the bag `x-y` contains all the items of `x`, with any items from `y` removed. For example, suppose that `x` has seven copies of the number 3, and `y` has two copies of the number 3. Then `x-y` will have five copies of the number 3 (i.e.,  $7 - 2$  copies of the number 3). In the case where `y` has more copies of an item than `x` does, the bag `x-y` will have no copies of that item. For example, suppose that `x` has nine copies of the number 8, and `y` has ten copies of the number 8. Then `x-y` will have no 8s. The statement `x -= y` should have the same effect as the assignment `x = x-y`;

**3** Implement the `sequence` class from Section 3.2. You may wish to provide some additional useful member functions, such as: (1) a function to add a new item at the front of the sequence; (2) a function to remove the item from the front of the sequence; (3) a function to add a new item at the end of the sequence; (4) a function that makes the last item of the sequence become the current item; (5) operators for `+` and `+=`. For the `+` operator, `x + y` contains all the items of `x`, followed by all the items of `y`. The statement `x += y` appends all of the items of `y` to the end of what's already in `x`.

**4** For a sequence `x`, we would like to be able to refer to the individual items using the usual C++ notation for arrays. For example, if

`x` has three items, then we want to be able to write `x[0]`, `x[1]`, and `x[2]` to access these three items. This use of the square brackets is called the **subscript operator**. The subscript operator may be overloaded as a member function, with the prototype shown here as part of the `sequence` class:

```
class sequence
{
public:
 ...
 value_type operator [] (size_type index)
 const;
 ...
}
```

As you can see, the `operator [ ]` is a member function with one parameter. The parameter is the index of the item that we want to retrieve. The implementation of this member function should check that the index is a valid index (i.e., `index` is less than the sequence size), and then return the specified item.

For this project, specify, design, and implement this new subscript operator for the `sequence`.

**5** A bag can contain more than one copy of an item. For example, the chapter describes a bag that contains the number 4 and two copies of the number 8. This bag behavior is different from a **set**, which can contain only a single copy of any given item. Write a new container class called `set`, which is similar to a bag, except that a set can contain only one copy of any given item. You'll need to change the interface a bit. For example, instead of the bag's `count` function, you'll want a constant member function such as this:

```
bool set::contains
(const value_type& target) const;
// Postcondition: The return value is true if
// target is in the set; otherwise the return
// value is false.
```

Make an explicit statement of the invariant of the `set` class. Do a time analysis for each operation. At this

point, an efficient implementation is not needed. For example, just adding a new item to a set will take linear time because you'll need to check that the new item isn't already present. Later we'll explore more efficient implementations (including the implementation of `set` in the C++ Standard Library).

You may also want to add additional operations to your set class, such as an operator for subtraction.

**6** Suppose that you implement a sequence where the `value_type` has a comparison operator `<` to determine when one item is "less than" another item. For example, integers, double numbers, and characters all have such a comparison operator (and classes that you implement yourself may also be given such a comparison). Rewrite the sequence class using a new class name, `sorted_sequence`. In a sorted sequence, the insert function always inserts a new item so that all the items stay in order from smallest to largest. There is no attach function. All the other functions are the same as the original sequence class.

**7** In this project, you will implement a new class called a **bag with receipts**. This new class is similar to an ordinary bag, but the way that items are added and removed is different. Each time an item is added to a bag with receipts, the `insert` function returns a unique integer called the **receipt**. Later, when you want to remove an item, you must provide a copy of the receipt as a parameter to the `remove` function. The `remove` function removes the item whose receipt has been presented, and also returns a copy of that item through a reference parameter.

Here's an implementation idea: A bag with receipts can have *two* private arrays, like this:

```
class bag_with_receipts
{
...
private:
 value_type data[CAPACITY];
 bool in_use[CAPACITY];
};
```

Arrays such as these, which have the same size, are

called **parallel arrays**. The idea is to keep track of which parts of the data array are being used by placing boolean values in the second array. When `in_use[i]` is true, then `data[i]` is currently being used; when `in_use[i]` is false, then `data[i]` is currently unused. When a new item is added, we will find the first spot that is currently unused and store the new item there. The receipt for the item is the index of the location where the new item is stored.

**8** Another way to store a collection of items is called a **keyed bag**. In this type of bag, whenever an item is added, the programmer using the bag also provides an integer called the **key**. Each item added to the keyed bag must have a unique key; two items cannot have the same key. So, the insertion function has the specification shown here:

```
void keyed_bag::insert
(const value_type& entry, int key);
// Precondition: size() < CAPACITY, and the
// bag does not yet contain any item with
// the given key.
// Postcondition: A new copy of entry has
// been added to the bag, with the given key.
```

When the programmer wants to remove an item from a keyed bag, the key of the item must be specified, rather than the item itself. The keyed bag should also have a boolean member function that can be used to determine whether the bag has an item with a specified key.

A keyed bag differs from the bag with receipts (in the previous project). In a keyed bag, the programmer using the class specifies a particular key when an item is inserted. In contrast, for a bag with receipts, the `insert` function returns a receipt, and the programmer using the class has no control over what that receipt might be.

For this project, do a complete specification, design, and implementation of a keyed bag.

**9** This is a simple version of a longer project that will be developed in Chapter 4. The project starts with the definition of a one-variable **polynomial**, which is an arithmetic

expression of the form:

$$a_0 + a_1x + a_2x^2 + \dots + a_kx^k$$

The highest exponent,  $k$ , is called the **degree** of the polynomial, and the constants  $a_0, a_1, \dots$  are the **coefficients**. For example, here are two polynomials with degree three:

$$2.1 + 4.8x + 0.1x^2 + (-7.1)x^3$$

$$2.9 + 0.8x + 10.1x^2 + 1.7x^3$$

Specify, design, and implement a class for polynomials. The class may contain a static member constant, `MAXDEGREE`, which indicates the maximum degree of any polynomial. (This allows you to store the coefficients in an array with a fixed size.) Spend some time thinking about operations that make sense on polynomials. For example, you can write an operation that adds two polynomials. Another operation should evaluate the polynomial for a given value of  $x$ .

- 10** Specify, design, and implement a class that can be one player in a game of tic-tac-toe. The constructor should specify whether the object is to be the first player (X's) or the second player (O's). There should be a member function to ask the object to make its next move, and a member function that tells the object what the opponent's next move is. Also include other useful member functions, such as a function to ask whether a given spot of the tic-tac-toe board is occupied, and if so, whether the occupation is with an X or an O. Also, include a member function to determine when the game is over, and whether it was a draw, an X win, or an O win.

Use the class in two programs: a program that plays tic-tac-toe against the program's user, and a program that has two tic-tac-toe objects that play against each other.

- 11** Specify, design, and implement a container class that can hold up to five playing cards. Call the class `pokerhand`, and overload the boolean comparison operators to allow you to compare two poker hands. For two hands  $x$  and  $y$ , the relation  $x > y$  means that  $x$  is a better hand than  $y$ . If you do not play in a weekly poker game yourself,

then you may need to consult a card rule book for the rules on the ranking of poker hands.

- 12** Specify, design, and implement a class that keeps track of rings stacked on a peg, rather like phonograph records on a spindle. An example with five rings is shown here:

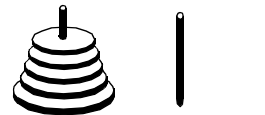
Rings stacked  
on a peg



The peg may hold up to 64 rings, with each ring having its own diameter. Also, there is a rule that requires each ring to be smaller than any ring underneath it, as shown in our example. The class's member functions should include: (a) a constructor that places  $n$  rings on the peg (where  $n$  may be as large as 64); use 64 for a default argument. These  $n$  rings have diameters from  $n$  inches (on the bottom) to one-inch (on the top). (b) a constant member function that returns the number of rings on the peg. (c) a constant member function that returns the diameter of the topmost ring. (d) a member function that adds a new ring to the top (with the diameter of the ring as a parameter to the function). (e) a member function that removes the topmost ring. (f) an overloaded output function that prints some clever representation of the peg and its rings. Make sure that all functions have appropriate preconditions to guarantee that the rule about ring sizes is enforced. Also spend time designing appropriate private data fields.

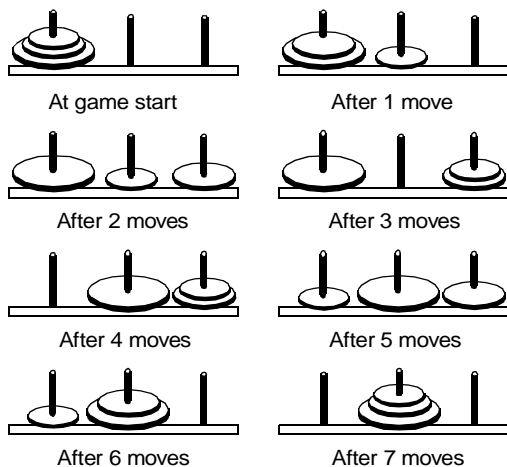
- 13** In this project, you will design and implement a class called `towers`, which is part of a program that lets a child play a game called Towers of Hanoi. The game consists of three pegs and a collection of rings that stack on the pegs. The rings are different sizes. The initial configuration for a five-ring game is shown here, with the first tower having rings from one-inch (on the top) to five-inches (on the bottom).

Initial configuration for  
a five-ring game of  
Towers of Hanoi





The rings are stacked in decreasing order of their size, and the second and third towers are initially empty. During the game, the child may transfer rings one-at-a-time from the top of one peg to the top of another. The object of the game is to move all the rings from the first peg to the second peg. The difficulty is that the child may not place a ring on top of one with a smaller diameter. There is the one extra peg to hold rings temporarily, but the prohibition against a larger ring on a smaller ring applies to it as well as to the other two pegs. A solution for a three-ring game is shown here:



The towers class must keep track of the status of all three pegs. You might use an array of three pegs, where each peg is an object from the previous project. The towers functions are specified here:

```
towers::towers(size_t n = 64);
// Precondition: 1 <= n <= 64.
// Postcondition: The towers have been initialized
// with n rings on the first peg and no rings on
// the other two pegs. The diameters of the first
// peg's rings are from one-inch (on the top) to n
// inches (on the bottom).

size_t towers::many_rings
(int peg_number) const;
// Precondition: peg_number is 1, 2, or 3.
// Postcondition: The return value is the number
// of rings on the specified peg.
```

```
size_t towers::top_diameter
(int peg_number) const;
// Precondition: peg_number is 1, 2, or 3.
// Postcondition: If many_rings(peg_number) > 0,
// then the return value is the diameter of the top
// ring on the specified peg; otherwise the return
// value is zero.

void towers::move
(int start_peg; int end_peg);
// Precondition: start_peg is a peg number
// (1, 2, or 3), and many_rings(start_peg) > 0;
// end_peg is a different peg number (not equal
// to start_peg), and top_diameter(end_peg) is
// either 0 or more than top_diameter(start_peg).
// Postcondition: The top ring has been moved
// from start_peg to end_peg.
```

Also overload the output operator so that a towers object may be displayed easily.

Use the towers object in a program that allows a child to play Towers of Hanoi. Make sure that you don't allow the child to make any illegal moves.

**14** Specify, design, and implement a class where each object keeps track of a large integer with up to 100 digits in base 10. The digits can be stored in an array of 100 elements and the sign of the number can be stored in a separate member variable, which is +1 for a positive number and -1 for a negative number.

The class should include several convenient constructors, such as a constructor to initialize an object from an ordinary `int`. Also overload the usual arithmetic operators and comparison operators (to carry out arithmetic and comparisons on these big numbers) and overload the input and output operators.

For more in-depth projects, please see [www.cs.colorado.edu/~main/projects/](http://www.cs.colorado.edu/~main/projects/)