Chapter 2

Arrays and Pointers

C++ is a large and complex language, in part because of some of its design goals. One goal that contributes to its complexity is the requirement that C++ be backward compatible with C. That is, any program written in C should compile on a C++ compiler. The C language was originally designed to write the Unix operating system. That goal required the language to have low-level access to hardware features and to be efficient.

The advantage of backward compatibility is that programmers who know the C language can use the C++ language to gradually learn the object-oriented features of the newer language. This advantage is a primary reason for the widespread adoption of the language. The disadvantage is that the low-level features of C must be compatible with C++, which makes some of the capabilities of C++ difficult to use. Dynamic memory allocation with pointers is a case in point. C++ introduced high-level operators that facilitate dynamic memory allocation. However, in C++, it is still easy to make errors with memory allocation and deallocation that are difficult to detect with casual testing.

This chapter describes dynamic memory allocation with pointers. It shows how arrays can be dynamically allocated and constructs some simple classes using them.

2.1 Pointer Types

The unary * operator is used in C++ to declare a pointer variable. When placed after the type, it declares a pointer to that type. For example the statement

```cpp
double* pd;
```

declares the variable pd to be a pointer to a double precision value. However, before you can store a double precision value in the variable you must allocate storage for it from the heap.

**Memory allocation**

`new` is a unary operator that allocates storage from the heap. It expects a type `T` on its right, allocates enough storage for `T`, and returns a pointer to the newly allocated storage. The statement

```cpp
pd = new double;
```

allocates storage from the heap for a double precision value and returns a pointer to it. Variable pd gets the value. Figure 2.1(a) shows the effect of the memory allocation. You can combine the declaration and allocation as follows.

```cpp
double* pd = new double;
```

The * operator is also used to dereference a pointer. When placed before a pointer variable, the notation indicates the cell to which the pointer points. For example, the statement

```cpp
*pd = 5.7;
```

assigns the value 5.7 to the memory cell to which pd points, as in Figure 2.1(b). From the hardware point of view, pd is the memory address of the location where the value is stored, and *pd is the memory location itself.

The type of pd is a pointer to double; the type of *pd is double. Because *pd is a double, C++ allows you to associate the * operator with the variable in the declaration. An equivalent declaration of pd is

```cpp
double *pd;
```

which is consistent with the use of * in the above assignment statement. You must be careful when defining two pointers with the same declaration. The declaration

```cpp
double* a, b;
```

does not define two pointers. Rather, the C++ compiler interprets the declaration as

```cpp
double *a, b;
```

so that b is a double, not a pointer. You can assign one pointer to another, but you must be careful to consider the effect of such an assignment. Because a pointer “points to” an item, if you give the pointer’s value to a second pointer, the second pointer will point to the same item to which the first pointer points. Consider the following code fragment, illustrated in Figure 2.2.

```cpp
int *pI = new int;
int *pJ = new int;
int *pK;
pI = 5;
pJ = 3;
pK = pI;
pI = pJ;
*pI = 2 + *pK;
```
2.1 Pointer Types

Figure 2.2 Allocating storage for pointers to integers and performing integer and pointer assignments.

The assignment of \( pI \) to \( pK \) is a pointer assignment, not an integer assignment. It makes \( pK \) point to the same cell to which \( pI \) points as shown in Figure 2.2(b). There is no change of any cell content. Similarly, the assignment of \( pJ \) to \( pI \) makes \( pI \) point to the same cell to which \( pJ \) points as in Figure 2.2(c). Execution of

\[
\text{cout} \ll *pI \ll *pJ \ll *pK \ll \text{endl;}
\]

following the code fragment would output

7 7 5

Because \( pI \) and \( pJ \) now point to the same memory cell, the cell containing 7, its value is printed twice.

Pointers are memory addresses, and at a low level of abstraction a memory address is an unsigned integer. In the C++ language, the pointer value 0 is reserved as a special sentinel value because a cell can never be allocated from the heap at address 0. It is now considered good practice to use an alternative representation for 0 called \textit{nullptr}, which is a new keyword introduced in C++11. The use of \textit{nullptr} in place of 0 makes the code easier to read because it shows clearly that the value is to be considered a pointer instead of an integer.

Figure 2.2(a) shows the state of \( pK \) after its declaration

\[
\text{int } *pK;
\]
as being a pointer with no value. Allocated variables always have values, even if they are unknown. After the above declaration of \( pK \), its value is not known. It is a serious mistake to assume that pointers with unassigned values have the value \textit{nullptr}. You cannot assume that \( pK \) has the value \textit{nullptr} in Figure 2.2(a). The reason this analysis error is so insidious is that a program that makes that assumption might work sometimes, but crash other times with the exact same input.

The address operand is \&. If \( x \) is a double then \&\( x \) is the address of \( x \). That is, \&\( x \) is a pointer to the cell that contains the double precision value of \( x \). Here is an example.

\[
\text{double } *py;
\text{double } dx;
\text{dx} = 6.2;
\text{py} = &dx;
\]
Chapter 2  Arrays and Pointers

(a) first = new Node;
(b) first->Value = 7;
    first->next = nullptr;
(c) p = first;
    first = new Node;
(d) first->value = 4;
    first->next = p;

Figure 2.3  Trace of a code fragment that uses the -> operator. The dashed triangle represents the nullptr value.

py and &dx are both addresses, that is, pointers. The last statement in the above fragment is a pointer assignment that makes py point to the same cell to which &dx points. Execution of

```
cout << *py << endl;
```

following the above code fragment would output 6.2.

In practice, pointers rarely point to primitive types like double and int. Instead, they point to a struct or a class. An example is a node structure for a linked list declared as

```
struct Node {
    int value;
    Node *next;
}
```

Node *first, *p;

Variable p has type pointer to Node. Therefore, *p has type Node, which is struct. You access a field of a struct or a class with the period operator . which is placed between the reference to the struct and its field. Thus, (*p).value is the value field of the struct to which p points, and (*p).next is the next field of the struct. The parentheses are necessary because the period operator has higher precedence than the * operator as Figure A.1 of the Appendix shows. Fortunately, C++ provides the -> operator, which allows the programmer to combine the * operator and the period operator without using parentheses. For example,

p->value
is equivalent to

(*p).value
Figure 2.3 is a trace of the execution of the following code fragment, which uses the -> operator.

```cpp
first = new Node;
first->value = 7;
first->next = nullptr;
p = first;
first = new Node;
first->value = 4;
first->next = p;
```

In C++, the programmer is responsible for deallocating memory that is allocated from the heap with the `new` operator. This responsibility is in marked contrast to those more modern object-oriented languages that provide automatic garbage collection. The operator that returns unneeded storage to the heap is `delete`, which expects a pointer on its right hand side. For example, to delete the two nodes of the previous code fragment requires execution of

```cpp
delete p;
delete first;
```

The effect of the `delete` operation is that heap memory is recycled and can be allocated again with later executions of `new`.

If a program fails to correctly delete all the unused storage from the heap, unused memory cells will accumulate in the heap and will not be available with later executions of `new`. Such a bug is called a *memory leak*. It is possible that after some period of time the entire heap will be occupied with unused cells, but allocation from the heap will nevertheless be impossible. When that happens, at best the program will cease operation with an error message warning the user that the program is out of memory, and at worse will crash. Memory leaks are notoriously difficult to detect, because the symptoms of the error are usually not manifest with short test runs of the program.

### Reference types and parameters

The Appendix describes the three parameter passing mechanisms of C++:

- Pass by value
- Pass by reference
- Pass by constant reference

The general rule is to use pass by reference when you want the function to change the value of the actual parameter, and to use the others when you do not. Here are three example programs that show the ramifications of using the different passing mechanisms in a function.

Figure 2.4 shows the effect of passing parameters by value. When the function call executes, the processor copies the value of the actual parameter onto the run-time stack. During execution of the called function, any changes that it makes to the formal parameters it makes to the copies. The actual parameters in the calling function are not affected by the changes, because the changes are made to the copies, not to the original
```cpp
#include <iostream>
using namespace std;

void swapVal(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
    cout << "a == " << a << " b == " << b << endl;
}

int main() {
    int i = 4;
    int j = 5;
    swapVal(i, j);
    cout << "i == " << i << " j == " << j << endl;
    return EXIT_SUCCESS;
}
```

Output

```
a == 5 b == 4
i == 4 j == 5
```

Figure 2.4 SwapValMain.cpp. A program to illustrate call by value. The actual parameters are i and j, which correspond to formal parameters a and b. Although the formal parameters change in the function swapValue, the actual parameters do not change in main.

actual parameters. The output in the figure shows that swapVal() changes the values of formal parameters a and b but not the actual parameters i and j.

Figure 2.5 shows call by reference. The formal parameters of swapRef() are a and b, each one of which is prefixed by & indicating call by reference. The actual parameters, i and j, declared to be integers. When the function call executes, the processor passes a reference to the formal parameters. You can think of a as being a reference to i and b as being a reference to j. So, in the function when

```
b = temp;
```

executes, it is as if

```
j = temp;
```

executes.

In C++, int& is known as a reference type. Furthermore, reference types are not limited to function parameters. You can declare a reference type outside a parameter list, but if you do so you must initialize it when you declare it. For example, this code fragment

```
int k;
```
#include <iostream>
using namespace std;

void swapRef(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
    cout << "a == " << a << " b == " << b << endl;
}

int main() {
    int i = 4;
    int j = 5;
    swapRef(i, j);
    cout << "i == " << i << " j == " << j << endl;
    return EXIT_SUCCESS;
}

Output
a == 5  b == 4
i == 5  j == 4

Figure 2.5 SwapRefMain.cpp. A program to illustrate call by reference. The call to swapRef passes an integer as the actual parameter with a reference type as the formal parameter. Unlike Figure 2.4 the value of actual parameter i is not passed. Rather, the address of i is passed.

int &c = k; // Must be initialized here.
c = 8;
cout << "k == " << k << " c == " << c << endl;

is legal and produces the output
k == 8  c == 8

If you replace the statement c = 8; with k = 8; in the above code fragment the result is unchanged. Both c and k refer to the same cell in memory. Reference types appear to have some characteristics of pointers, but they are not the same. Reference types have more restrictions. Once you set the reference of a reference type when you create it, you can never change what it refers to. If you have another integer variable s declared as
int s = 9;

and execute

\[ c = s; \]
#include <iostream>
using namespace std;

void swapPtr(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
    cout << "*a == " << *a << " *b == " << *b << endl;
}

int main() {
    int i = 4;
    int j = 5;
    swapPtr(&i, &j);
    cout << "i == " << i << " j == " << j << endl;
    return EXIT_SUCCESS;
}

Output
*a == 5  *b == 4
i == 5  j == 4

Figure 2.6 SwapPtrMain.cpp. A program to illustrate passing addresses by value. This technique is how to achieve the effect of call by reference in the C language, for which the technique of Figure 2.5 is not legal.

c does not refer to s. Instead, both c and k, which refer to the same cell in memory, get the value of s, which is 9.

This behavior is consistent with reference types in parameter lists. Think of a function call as creating a reference type on the run-time stack and initializing it the same way that c is initialized in the above code fragment. You use the formal parameter in a function the same way you use the variable c in the above code fragment. Once you set which cell a call-by-reference formal parameter refers to when the function is called, you can never change it to point to a different cell in memory.

The program in Figure 2.6 achieves the effect of pass by reference by explicitly passing addresses. The formal parameters a and b are pointers to integers called by value. Because they are called by value, the pointers themselves cannot change. However, the values in the cells to which they point can change.

The call to swapPtr() illustrates the fact that a pointer is an address and a reference is also an address. Because formal parameter a is a pointer, it expects an address. Actual parameter &i obliges by explicitly giving it the address of i with the & operator. The C programming language does not have the call by reference mechanism of Figure 2.5. So, when you program in C you must use the technique of Figure 2.6 if you want the function to change the values of the actual parameters.

Behind the scenes, the C++ compiler generates code like that of Figure 2.6 to imple-
2.1 **Pointer Types**

[Diagram: Figure 2.7 Trace of a code fragment that uses the fact that an array name without the subscript brackets is synonymous with a pointer to the first cell in the array.]

...
cout << *p;
the value 7 will be output.

C++ allows you to do arithmetic on pointers as if they were integers. However, when
you add a value to a pointer the addition always takes place in units of the number of
bytes occupied by the type to which the pointer points. In this example, the pointer p
points to an integer. If integer values for this computer each occupy four bytes, then
a C++ statement that adds 1 to p actually adds 4 to the address, so that p will point to
arr[1]. The statement
p++;
results in Figure 2.7(b). Now if you execute the statement
cout << *p;
the value 9 will be output.

Adding values to pointers works this way regardless of the number of bytes occupied
by an individual cell in the array. If arr were an array of double, each cell occupying
eight bytes instead of four, then the statement p++ would increment p by eight, but p
would still point to arr[1].

Because arr is itself an address, which is a pointer, you can do pointer arithmetic
on it. In general, any reference to
arr[i]
can be written
*(arr + i)
So you can dispense with the bracket subscript notation altogether and write the two
assignments
arr[0] = 7;
arr[1] = 9;
equivalently as
*arr = 7; *(arr + 1) = 9;

C++ programs sometimes use the equivalence of pointers and array addresses in their
specification of formal parameters. Instead of specifying a parameter to be an array of
type T, it is specified to be a pointer to type T. For example, the procedure heading
void addRational(Rational r[], int cap)
from Figure A.12 in the Appendix specifies r to be an array of Rational. It can be
written equivalently as
void addRational(Rational *r, int cap)

With this viewpoint, the function is expecting a pointer to the first element of the
array. The pointer is called by value. Any changes to the pointer that are made in the
function are not reflected in the calling function. But any changes in the array cells
pointed to by pointer r are reflected in the calling function. The equivalence of an array
name to a pointer to its first cell is the reason that arrays are in effect called by reference
in C++.
2.1 Pointer Types

#include <iostream> // istream, ostream.
using namespace std;

void readStream(istream &is, double d[], int cap, int &len);
// Pre: d is allocated with capacity cap.
// Post: num values are input from is to d[0..num - 1], where
// num == min(number of elements in is, cap).
// len == num.

void writeStream(ostream &os, double const d[], int cap, int len);
// Pre: d is allocated with capacity cap.
// Post: num values are output from d[0..num - 1] to os, where
// num == min(len, cap).

Figure 2.8 ArrayClassicMain.hpp. The specification of a main program that uses a dynamically allocated array of doubles.

Dynamically allocated arrays

In Chapter 1, Figure 1.19 shows the allocation of an array of pointers as

const int NUM_SHAPES = 5;
AShape *shapes[NUM_SHAPES];

This allocation is static. That is, the array of pointers is allocated on the runtime stack of the main program. It is true that the objects to which each pointer points in Figure 1.18(b) are allocated from the heap. But the array itself is allocated statically on the runtime stack.

A disadvantage of allocating an array statically is that the number of elements must be known at compile time. That is, NUM_SHAPES in the above declaration must be a constant. It cannot be a variable. If you want to change the number of shapes to process in the program, you must change the constant with your text editor and recompile the program. With static arrays, you cannot prompt the user for the size of the array as that would require a variable in the declaration of the array.

With dynamic allocation, the program allocates the array itself from the heap. Figure 2.8 shows the header file for a main program that uses dynamic allocation of an array of doubles, and Figure 2.9 shows its implementation. The main program prompts the user for the capacity of the array, which it stores in the integer variable cap. The statement
double *array = new double[cap];

allocates the array dynamically from the heap. Allocation from the heap allows cap to be a variable. The name of the array is arr. It appears from the declaration that it is a pointer to a double, but that is a consequence of the C++ equivalence of an array name without subscript brackets to a pointer to the first cell.

If an array is allocated statically on a function’s runtime stack, it will be deallocated automatically when the function returns. To avoid memory leaks, however, anything allocated from the heap with the new operator must be deleted with the delete operator. The main program in Figure 2.9 deletes arr with
```cpp
#include <cstdlib> // EXIT_SUCCESS.
#include <fstream> // ifstream.
#include "ArrayClassicMain.hpp"
#include "Utilities.hpp" // promptIntGE.
using namespace std;

int main() {
    int cap = promptIntGE("Enter array capacity", 1);
    double *arr = new double[cap]; // Dynamic allocation.
    ifstream ifs;
    promptFileOpen(ifs);
    if (ifs) {
        int length = 0;
        readStream(ifs, arr, cap, length);
        ifs.close();
        cout << "Read count == " << length << endl;
        cout << "Array data:" << endl;
        writeStream(cout, arr, cap, length);
        // arr[2 * cap] = 123.4;
        // cout << arr[2 * cap] << endl;
    }
    delete [] arr;
    return EXIT_SUCCESS;
}

void readStream(istream &is, double d[], int cap, int &len) {
    len = 0;
    for (int i = 0; i < cap && is >> d[i]; i++) {
        len++;
    }
}

void writeStream(ostream &os, double const d[], int cap, int len) {
    for (int i = 0; i < len && i < cap; i++) {
        os.width(12);
        os << d[i];
        if (i % 6 == 5) {
            os << endl;
        }
    }
    os << endl;
}
```

Figure 2.9 ArrayClassicMain.cpp. The implementation of the main program specified in Figure 2.8.
2.2 Array Classes

The objective of this section is to construct a class that behaves like an array but that does not permit the corruption that occurs if a value is stored outside the range of the index. An array is a collection of values, all of which have the same type. So, the question

delete [ ] arr;

Deallocation of a dynamically allocated array presents a typical C++ opportunity to produce a memory leak. If you forget the brackets [ ] in the deallocation statement as in

delete arr; // MEMORY LEAK

the statement will compile, because arr is a pointer to a double. The operation will delete only the first cell of the array, and you will probably be none the wiser when your program apparently passes its tests successfully.

Function readStream inputs double precision values from the input stream. There are three possibilities—the number of values in the stream is less than, equal to, or greater than the capacity of the array. If the number of values in the stream is less than or equal to the capacity of the array the function reads all the values into the array. If the number is greater than the capacity the function fills the array to its capacity and leaves the remaining values in the stream unprocessed. The statement that is responsible for deciding when to stop inputting is

for (int i = 0; i < cap && is >> d[i]; i++)

It uses a common C++ pointer idiom. The input stream is the class reference is, which is a pointer. The expression

is >> d[i]

attempts to input a value into d[i]. If there is a value in is to input, d[i] gets the value and the expression returns a non nullptr value. Because pointers are equivalent to integers, and nonzero integers are interpreted as true, the for loop continues. If there is no value to input, the expression returns nullptr, which is equivalent to 0 and interpreted as false. The loop terminates.

The two statements that are commented out in the main program

arr[2 * cap] = 123.4;
cout << arr[2 * cap] << endl;

are to illustrate what happens when you program with C++ primitive arrays. The first statement stores a value outside the boundary of the array, and the second statement outputs the value from the same location. Because C++ does not check its array bounds at execution time, the above statements will execute and even occasionally work. The first statement clobbers some cell in main memory in an unpredictable way. If the damage is benign the program will work. But if the corruption is fatal the program will crash. The following section constructs a safe array class that automatically checks for out-of-range indexing when you access an element of the array.
arises, What type should this safe array be? If you design the array to hold integers, your client is sure to want an array of doubles, and if you design an array of doubles, the client will want strings.

One alternative is to use the `typedef` facility described in Section A.4 of the Appendix. At the beginning of the class implementation you could place the definition

```cpp
typedef int T;
```

which makes the name `T` a synonym for `int`. Everywhere the implementation needs to refer to the type you write `T` instead of `int`. If the client needs a safe array of doubles, you replace the one line above with

```cpp
typedef double T;
```

and recompile the implementation. The problem with this approach is that someone still needs to change a line of code with a text editor. If you want to store this class as a service in a software library it would not be feasible to require clients to modify the source code they want to use. You could make available many different compiled servers for many different types, but that would be wasteful and difficult to maintain. Furthermore, how could you anticipate all the possible types that a client might need?

**Templates**

The template facility of C++ is designed to solve the problem of programming a service when the type to be used by the client is not known. The advantage of using a template instead of a `typedef` is that you can write the implementation of a class for a generic type and provide just one copy of the implementation as a service. Two different clients can use the service with two different types, yet only one generic class need be provided in the library. Writing a service with a generic type using templates is referred to as **generic programming**. The beauty of generic programming in C++ is that there is no performance penalty compared to the direct approach. That is, client programs that use the services of a template class execute just as fast as if the class were provided with the desired type built in. The concept of a template is similar to the concept of a function that provides a parameter list for its clients. For example, the greatest common divisor function

```cpp
int gcd(int m, int n)
```

has formal parameters `m` and `n`. A client calls `gcd` with specific integer values as actual parameters for `m` and `n`. With a template class, the generic type is like a formal parameter. The client supplies a specific type as the actual parameter. Because passing a type to a template is similar to passing an actual parameter to a formal parameter, the template facility is also known as parametric polymorphism.

**A safe array of doubles**

Figure 2.10 is a main program that accesses the services of a template class that implements a safe array. The class is implemented in `ArrayT.hpp`, which is included. Because `main()` calls no functions other than those in various libraries, there is no corresponding `ArrayMain.hpp` file. After reading the virtues of templates compared to `typedef`, you may wonder about the purpose of the statement
2.2 Array Classes

```cpp
#include <cstdlib> // EXIT_SUCCESS.
#include <iostream> // cout.
#include <fstream> // ifstream.
#include "Utilities.hpp"
#include "ArrayT.hpp"
using namespace std;

typedef ArrayT<double> ArrayDouble;

int main() {
    ArrayDouble arr(promptIntGE("Enter array capacity", 1));
    ifstream ifs;
    promptFileOpen(ifs);
    if (ifs) {
        int length = 0;
        readStream(ifs, arr, length);
        ifs.close();
        cout << "Read count == " << length << endl;
        cout << "Array data:" << endl;
        writeStream(cout, arr, length);
        // arr[2 * arr.cap()] = 123.4;
        // cout << arr[2 * arr.cap()] << endl;
        return EXIT_SUCCESS;
    }
}
```

**Figure 2.10** ArrayTMain.cpp. A main program that uses a dynamically allocated safe array of doubles from a template class.

typedef ArrayT<double> ArrayDouble;

Its purpose is strictly a convenience. You could eliminate the above typedef, and everywhere in the program that ArrayDouble appears, simply replace it with ArrayT<double>. The safe array is provided as a class, which is named ArrayT. A template class requires the client to supply a type as an actual parameter. Rather than enclose the parameter list in parentheses as is the case with a function’s parameter list, the list is enclosed in angle brackets < >. In the above statement, double is the actual parameter for the template class. The client is defining ArrayDouble to be a type that corresponds to the template class safe array for storing double precision real values.

The first statement in the main program

ArrayDouble arr(promptIntGE("Enter array capacity", 1))

makes object arr an instantiation of the class ArrayDouble. The class provides a constructor with a parameter list having a single integer parameter, which is executed when arr is declared. Whatever integer value the user enters is used internally in the class to set the capacity of the dynamically allocated array. Because ArrayDouble is
Chapter 2  Arrays and Pointers

Figure 2.11  The UML diagram for the template classes that provide safe arrays of nonpointer values ArrayT and pointer values ArrayP.

a class, there is no need for main() to explicitly deallocate arr before it terminates. The destructor for the class will be called automatically to deallocate the array when main() terminates.

The main program is short because functions readStream and writeStream are provided along with the template class in the library. They perform input and output like the corresponding functions in Figures 2.8 and 2.9 for the primitive array. There is no need to supply the capacity of the array as a parameter to these versions of the functions because the array class stores the capacity of the array as an attribute.

The statements that are commented out

arr[2 * arr.cap()] = 123.4;
cout << arr[2 * arr.cap()] << endl;

show that you can access the elements of arr with square brackets [ ] as if it were a primitive array. Unlike a primitive array, however, execution of the above assignment statement is guaranteed to not corrupt your computer in some unknown way. Instead, a precondition will be violated, an error message will direct you to the offending code, and the program will terminate.

The program of Figure 2.10 requires the services of a safe array of double precision real values. The client programs in the distribution software, however, need the services of a safe array of pointers as well as a safe array of nonpointers like double. A template class for an array of pointers in C++ differs slightly from the corresponding template class for an array of nonpointers. Therefore, the distribution software provides two safe array template classes, one for pointers named ArrayP, and one for nonpointers named ArrayT, which is used by the client in Figure 2.10. Figure 2.11 is the UML diagram of these two classes. Their common characteristics are factored out and specified by their abstract superclass ASeq. The Seq in the class name stands for sequence.
2.2 Array Classes

template <class S>
class ASeq {
    public:
        ASeq(int cap = 0);
        virtual ~ASeq() = 0;
        virtual S &operator[](int i) = 0; // For read/write.
        virtual S const &operator[](int i) const = 0; // For read-only.
        virtual int cap() const = 0;
    private:
        ASeq(ASeq const &rhs); // Disabled.
        ASeq &operator=(ASeq const &rhs); // Disabled.
};

template <class S>
ASeq<S>::ASeq(int cap) { }

template <class S>
ASeq<S>::~ASeq() { }

Figure 2.12 ASeq.hpp. The template class for an abstract safe array.

The UML diagram shows no attributes for ASeq and four operations—ASeq, which is the constructor, two versions of operator[], which are function overloading of the subscript operator [], and cap, which is a function that returns the capacity of the sequence. The dashed box in the upper right side of the UML diagram is the symbol for a template class. The symbol S is the formal parameter whose actual parameter is supplied by the client. The symbol T for the subclass ArrayT is the formal parameter for a nonpointer type, and the symbol P for the subclass ArrayP is the formal parameter for a pointer type. Each of the subclasses contain two private attributes, _data, a primitive C++ array that stores the values, and _cap, an integer that stores the capacity of the array. Figure 2.12 shows the listing for ASeq.

The declaration of the ASeq class contains a prefix
template <class S>

positioned before the usual class ASeq. The prefix has the formal parameter list <class S>, which states that S is the generic type for this class. There is no difference between the declaration of the template class array and what it would be with the generic type built in. Everywhere the formal parameter S appears, imagine that it is replaced by some primitive type like double. For example, in the declaration for the operator []

virtual S &operator[](int i) = 0;

you can imagine that it reads like

virtual double &operator[](int i) = 0;
// ========= Destructor =========
template<class P>
ArrayP<P>::~ArrayP() {
    for (int i = 0; i < _cap; i++) { // Differs for ArrayT.
        delete _data[i];
        _data[i] = nullptr;
    }
    delete [] _data;
    _data = nullptr;
}

Figure 2.13  ArrayP.hpp. The template class that provides a safe array of pointer values. Only the destructor is shown here, which differs from the destructor for ArrayT.

which is the effect it would have if the actual parameter from the client were double.

Figure 2.12 is the listing of the template class for the safe array of nonpointer values. It turns out to be difficult with C++ to separate the specification from the implementation of a template service. Consequently, the template class ArrayT is both specified and implemented in the .hpp file. There is no corresponding .cpp file. The attributes of the class are _data and _cap. It would appear that _data is a pointer to a cell of type T, but the appearance is deceiving. _data is an array name without square brackets, and is equivalent to a pointer to the first cell of the array. It is used in the class as an array of elements of type T. _cap stores the capacity of the array.

Allocation in the constructor is accomplished with

    _data = new T[cap];

Again, because T is the formal parameter of the template, if the actual parameter in the client were double, the allocation in the constructor would have the same effect as

    _data = new double[cap];

The corresponding deallocation in the destructor is accomplished with

default (1) _data; // Differs for ArrayP.

The destructor differs between the two classes derived from ASeq. To deallocate an array of pointers requires that you delete not only the array, but also the object to which each pointer points. Figure 2.13 shows the destructor for the ArrayP template class.

Overloading operator [ ]

Figures 2.14, 2.15, and 2.16 show the implementation of the methods that allow main() to access elements of the class by indexing as if the class were a primitive array. The idea is to treat the subscript brackets [ ] as an operator, and overload the operator name. When a client writes an object name followed by the subscript brackets, C++ will detect the class of the object and determine if the subscript operator has been defined for that class. If it has, the compiler will invoke the method.
// ========= ArrayT =========
// Pre: The parameter T is not a pointer type.
template<class T>
class ArrayT : public ASeq<T> {
private:
    T* _data;
    int _cap;

public:
    ArrayT(int cap = 1);
    virtual ~ArrayT();
    int cap() const override;
    T& operator[](int i) override; // For read/write.
    T const &operator[](int i) const override; // For read-only.

private:
    ArrayT(ArrayT const &rhs); // Disabled.
    ArrayT &operator=(ArrayT const &rhs); // Disabled.
};

// ========= Constructor =========
template<class T>
ArrayT<T>::ArrayT(int cap) {
    if (cap < 1) {
        cerr << "ArrayT constructor precondition 0 < cap violated."
             << endl;
        cerr << "cap == " << cap << endl;
        throw -1;
    }
    _data = new T[cap];
    _cap = cap;
}

// ========= Destructor =========
template<class T>
ArrayT<T>::~ArrayT() {
    delete [] _data; // Differs for ArrayP.
    _data = nullptr;
}

// ========= cap =========
template<class T>
int ArrayT<T>::cap() const {
    return _cap;
}

Figure 2.14  ArrayT.hpp. The template class that provides a safe array of non-pointer values. ArrayT is the constructor and ~ArrayT is the destructor. The listing continues in the next figure.
// ========= operator[] =========
template<class T>
T &ArrayT<T>::operator[](int i) {
    if (i < 0 || _cap <= i) {
        cerr << "ArrayT index out of bounds: index == " << i << endl;
        throw -1;
    }
    return _data[i];
}

template<class T>
T const &ArrayT<T>::operator[](int i) const {
    if (i < 0 || _cap <= i) {
        cerr << "ArrayT index out of bounds: index == " << i << endl;
        throw -1;
    }
    return _data[i];
}

Figure 2.15 ArrayT.hpp (continued). Overloading the subscript operator []. The listing continues in the next figure.

There are two ways to declare an array—with const and without. In the same way that the value of a constant integer cannot be changed, the values of a constant array cannot be changed either. It is possible to initialize the values of a constant array when the array is first declared. Once initialized, the values will never change. Another use of a constant array is as a formal parameter in a function’s parameter list. Even if the actual parameter is not a constant array, the compiler will not permit any changes to be made to the elements of the formal parameter within the scope of the function.

Corresponding to these two ways to declare an array, class ArrayT must implement two versions of the overloaded index operator. For an object that does not have const, it implements the method

T &ArrayT<T>::operator[](int i)

You can think of operator[] as the name of the function and (int i) as its formal parameter list. With the array declared as in Figure 2.10, you might write a statement like

num = arr[13];

C++ treats the index 13 as the actual parameter, which corresponds to formal parameter i. The function returns T & , which is a reference to T. Because the actual type is double in Figure 2.10, you can think of the function as returning double & , that is, a reference to a double precision real.

The first statement in the method
// readStream

```cpp
template<class T>
void readStream(istream &is, ArrayT<T> &a, int &len) {
  // Pre: >> is defined for T.
  // Post: num values are input from is to a, where
  // num == min(number of elements in is, a.cap()).
  // len == num.
  len = 0;
  int cap = a.cap();
  for (int i = 0; i < cap && is >> a[i]; i++) {
    len++;
  }
}
```

// writeStream

```cpp
template<class T>
void writeStream(ostream &os, ArrayT<T> const &a, int len) {
  // Pre: << is defined for T, which is a number.
  // Post: num values are output from a to os, six items per line,
  // where num == min(len, a.cap()).
  int cap = a.cap();
  for (int i = 0; i < len && i < cap; i++) {
    os.width(12);
    os.precision(6);
    os.setf(ios::fixed | ios::showpoint);
    os << a[i];
    if (i % 6 == 5) {
      os << endl;
    }
  }
  os << endl;
}
```

Figure 2.16  ArrayT.hpp (continued). The readStream and writeStream operators, which are called from the program in Figure 2.10. This completes the listing.

```cpp
if (i < 0 || _cap <= i) {
    cerr << "ArrayT index out of bounds: "
        << "index == " << i << endl;
    throw -1;
}
```

is what makes the array safe. The if test will be true if the value supplied by the actual parameter is outside the allowable range. In the above example, if the capacity of the array is 8, and the array is subscripted with 13, the precondition is not met, and the if
statement executes. An appropriate error message is generated and your application will not be corrupted in some unknown way.

The second statement in the method

```
return _data[i];
```

uses the index supplied by the client to access the primitive array stored as an attribute in the class. Because the type returned is \( T \& \), the `return` statement returns a reference to \( _\text{data}[i] \). For example, suppose the main program of Figure 2.10 executes the statement

```
arr[13] = 7.1;
```

The assignment operator `=` expects a reference to a memory cell on its left hand side to which it will assign a value. The return statement obliges by returning a reference to \( _\text{data}[13] \), which then gets the value 7.1.

The second version of the overloaded operator is for arrays that are declared with `const`. It is identical to the first version except for its heading

```
T \text{const\&} ArrayT<T>::\text{operator[]}(\text{int } i) \text{ const}
```

And in the heading the only difference is the existence of the keyword `const` in two places. The first `const` applies to the return type \( T \text{ const\&} \). This `const` informs the compiler that the values of the constant array cannot change. The second `const` makes the method a constant member function. A constant member function is prevented from modifying any attributes of its class.

For example, suppose `arr` is a constant array in the formal parameter list of a function as follows.

```
void Alpha(ArrayDouble const &arr)
```

If the compiler encounters the following statement within the function

```
num = arr[13];
```

it notes that `arr` is a constant class that uses the index operator. So, it looks for a constant member function of that class that implements the overloaded index operator and finds this second version. It notes that the return type is \( T \text{ const\&} \), which cannot change. No problem here, because the returned value is used on the right hand side of the assignment.

On the other hand, if the compiler encounters

```
arr[13] = 7.1;
```

it notes again that `arr` is a constant class, looks for a constant member function, and finds the second version. But now it notes that the return type is \( T \text{ const\&} \), which does not permit the referenced cell, \( a[13] \), to change. Therefore, the compiler issues an error and does not permit the assignment.
2.3 A Vector Class

One annoying feature about arrays is that you must specify how many cells will be in the array when you allocate it. This is true whether it is allocated on the stack and you must commit to its capacity at compile time, or whether you allocate it dynamically during program execution. In both cases, you are committed to the capacity of the array. After you make that commitment and begin populating the array with values, you can no longer increase the capacity of the array.

Properties of vectors

A vector is a data structure that is similar to an array because you access its values with the usual square bracket operator. For example, if \( v \) is a vector of \( \text{int} \), and you want to set its third element to 50, you execute

\[
\text{v}[2] = 50;
\]

In addition to this usual random access feature, a vector provide two advantages over an array:

- Its capacity increases automatically even after you begin populating it with values.
- It provides an insert operation that shifts current values to the right to accommodate the inserted value, and a remove operation that shifts current values to the left to fill the cell whose value is removed.

The capacity of a vector begins at one and automatically doubles when necessary to accommodate a new value. Figure 2.17 shows both these features for vector \( v \). At a given point in time a vector has a size and a capacity, with the invariant that the size is always less than or equal to the capacity. In Figure 2.17(d), the size and the capacity are both four. In Figure 2.17(e), to insert the value 50 at index four the size increases to five and the capacity doubles to eight.
// ======= VectorT =======
// Pre: The parameter T is not a pointer type.
template<class T>
class VectorT : public ASeq<T> {
private:
    T *data; // Invariant: 0 < _cap, and _cap is a power of 2.
    int _cap; // Invariant: 0 <= _size <= _cap.
    int _size;

    void doubleCapacity();

    VectorT(VectorT const &rhs); // Disabled.
    VectorT &operator=(VectorT const &rhs); // Disabled.

(a) Specification.

// ======= doubleCapacity =======
template<class T>
void VectorT<T>::doubleCapacity() {
    _cap *= 2;
    T *newDat = new T[_cap];
    for (int k = 0; k < _size; k++) {
        newDat[k] = _data[k];
    }
    delete[] _data;
    _data = newDat;
}

(b) Implementation.

Figure 2.18 Attributes and the doubleCapacity() method of the vector class VectorT.

A vector implementation

Vector class VectorT inherits from the abstract sequence ASeq in the same way that ArrayT does. The UML diagram for VectorT is similar to the one in Figure 2.11 for ArrayT and is not shown. There is a corresponding implementation for VectorP, a vector of pointers, that corresponds to ArrayP and is also not shown here.

Figures 2.18(a) shows the attributes and the declaration of the doubleCapacity() method of the vector class VectorT. A vector has three attributes: _data is the array of values, _cap is the capacity of the vector, and _size is its size.

Figure 2.18(b) shows the implementation of doubleCapacity(), which doubles the capacity of the vector. It allocates a new array named newDat with twice the capacity of the current array attribute _data. Then it copies the values from _data to
2.3 A Vector Class

```cpp
public:
    VectorT();
    // Post: This vector is initialized with capacity of 1 and size of 0.

    virtual ~VectorT();
    // Post: This vector is initialized with capacity of 1 and size of 0.

    int cap() const override {return _cap;}
    // Post: The capacity of this vector is returned.

    int size() const {return _size;}
    // Post: The size of this vector is returned.

(a) Specification, and implementation of cap() and size().

    // ========= Constructor =========
    template<class T>
    VectorT<T>::VectorT() {
        _data = new T[1];
        _cap = 1;
        _size = 0;
    }

    // ========= Destructor =========
    template<class T>
    VectorT<T>::~VectorT() {
        delete [] _data; // Differs for VectorP.
        _data = nullptr;
    }

(b) Implementation.

Figure 2.19 The constructor, destructor, and the cap() and size() methods of the vector class vectorT.
```

newDat, deletes _data from the heap, and sets _data to the enlarged array. An alternative strategy would be to increase the capacity by one, which would conserve memory. However, such a strategy would be more time consuming because you would have to copy over the entire array each time an element is appended. With the doubling strategy, you anticipate that additional append operations will execute in the future and preallocate storage for them.

Figure 2.19 shows the constructor and destructor, and the cap() and size() methods. The constructor allocates a new array of type T with one cell and sets _cap to one and _size to zero. The destructor deletes the array, similar to the destructor for ArrayT. For VectorP, not shown here, the destructor must deallocate the objects pointed to by the pointers in the vector, as does the destructor for ArrayP.
void append(T const &e);
// Post: Element e is appended to this vector, possibly increasing cap().

void insert(int i, T const &e);
// Pre: 0 <= i && i <= size().
// Post: Items [i..size()-1] are shifted right and element e
// is inserted at position i.
// size() is increased by 1, possibly increasing cap().

T remove(int i);
// Pre: 0 <= i && i < size(). T has a copy constructor.
// Post: Element e is removed from position i and returned.
// Items [i+1..size()-1] are shifted left.
// size() is decreased by 1 (and cap() is unchanged).

(a) Specification.

// ======== append ========
template<class T>
void VectorT<T>::append(T const &e) {
    if (_size == _cap) {
        doubleCapacity();
    }
    _data[_size++] = e;
}

(b) Implementation.

Figure 2.20 Methods to append to, insert into, and remove from VectorT. Implementation of insert() and remove() are exercises for the student.

The declarations and implementations of cap() and size() are combined because they are simple one-liners.

Figure 2.20 shows that method append() has no precondition. You can append a value to an empty vector. And even if you append a value to a vector that is full, it will automatically double its capacity to accommodate the appended value.

Compare the preconditions of insert() and remove(). The precondition for insert() is
// Pre: 0 <= i && i <= size().
and the precondition for remove() is
// Pre: 0 <= i && i < size().

The preconditions differ because you can insert an element after the last one in the vector, but you cannot remove an element after the last one. That is, if i has the value size()
// ========= operator[] =========
// For read/write.
template<class T>
T &VectorT<T>::operator[](int i) {
    if (i < 0 || _size <= i) {
        cerr << "VectorT index out of bounds: index == " << i << endl;
        throw -1;
    }
    return _data[i];
}

// For read-only.
template<class T>
T const &VectorT<T>::operator[](int i) const {
    if (i < 0 || _size <= i) {
        cerr << "VectorT index out of bounds: index == " << i << endl;
        throw -1;
    }
    return _data[i];
}

Figure 2.21  Overloading the [ ] operator of the vector class VectorT.

then the precondition for insert() is satisfied, but the precondition for remove() is not. Furthermore, you can insert an element in an empty vector, but you cannot remove an element from one. That is, if the value of size() is zero and i is also zero the precondition for insert() is satisfied, but the precondition for remove is not because 0 < 0 is false.

Figure 2.21 shows that overloading the [ ] operator is accomplished as it is with the safe array classes.

The safe array ArrayT provides readStream() to input values from an input stream and writeStream() to output values to an output stream. Figure 2.16 shows that these functions are not methods, i.e. member functions. They take an input or output stream, an ArrayT, and an integer length as parameters. VectorT handles input/output differently. Figure 2.22 shows that function toStream() is a method and has only one parameter, an output stream. Because VectorT has attribute _size there is no need for a length parameter.

Figure 2.22 shows how to overload operator<< so vectors can use the binary << output operator. The overloaded operator<< cannot be a method of VectorT, because of the requirements that C++ places on its signature. It must return a reference to an input stream, its first parameter must be an input stream, and its second parameter must correspond to the right hand side (rhs) of the << operator. In this case, rhs is a VectorT.

You might be tempted to dispense with method toStream() altogether and incorporate its logic into operator<<. The problem with that approach is that toStream() needs access to the private attributes of VectorT to be able to output a representation
// ========= operator<< =========
template<class T>
ostream &operator<<(ostream &os, VectorT<T> const &rhs) {
    rhs.toStream(os);
    return os;
}

// ========= toStream =========
template<class T>
void VectorT<T>::toStream(ostream &os) const {
    os << "(
    for (int i = 0; i < _size - 1; i++) {
        os << _data[i] << ", ";
    }
    if (_size > 0) {
        os << _data[_size-1];
    }
    os << ")";
}

// ========= operator>> =========
template<class T>
istream &operator>>(istream &is, VectorT<T> &rhs) {
    rhs.fromStream(is);
    return is;
}

// ========= fromStream =========
template<class T>
void VectorT<T>::fromStream(istream &is) {
    T temp;
    while (is >> temp) {
        append(temp);
    }
}

Figure 2.22 The input/output streaming operators of the vector class VectorT.hpp.

of the vector. But operator<< is not a method, and therefore does not have access to
the attributes of the vector. Because toStream() is a method it does have access, and
so can use the attributes to generate the stream of characters to the output stream.

Figures 2.23 and 2.24 show the listing of a main program to test the implementa-
tion of the VectorT data structure. It prompts the user for a one-letter response, then
depending on the response, calls one of the VectorT methods.
typedef VectorT<int> VectorInt;

int main() {
    VectorInt v;
    int value, index;
    ifstream ifs;
    char response;
    do {
        cout << "(c)ap (s)ize (a)ppend (f)ileAppend (i)nsert "
        << "(r)emove se(t) (w)rite (q)uit: ";
        cin >> response;
        switch (toupper(response)) {
        case 'C':
            cout << "\nThe capacity is \n" << v.cap() << endl;
            break;
        case 'S':
            cout << "\nThe size is \n" << v.size() << endl;
            break;
        case 'A':
            cout << "Append what integer value? ";
            cin >> value;
            v.append(value);
            break;
        case 'F':
            promptFileOpen(ifs);
            if (ifs) {
                ifs >> v;
                ifs.close();
            }
            break;
        case 'I':
            cout << "Insert what integer value? ";
            cin >> value;
            cout << "Insert at what location? ";
            cin >> index;
            v.insert(index, value);
            break;
        case 'R':
            cout << "Remove from what location? ";
            cin >> index;
            value = v.remove(index);
            cout << "\n" << value << " removed." << endl;
            break;
        }
    } while (response != 'q');
}

Figure 2.23  VectorTMain.cpp. A main program to test VectorT. The listing continues in the next figure.
case 'T':
    cout << "Set what integer value? ";
    cin >> value;
    cout << "Set at what location? ";
    cin >> index;
    v[index] = value;
    cout << "\nValue at index " << index << " is now "
    << v[index] << endl;
    break;
  case 'W':
    cout << "\n" << v << endl;
    break;
  case 'Q':
    break;
  default:
    cout << "\nIllegal command." << endl;
    break;
}
} while (toupper(response) != 'Q');
return EXIT_SUCCESS;

Figure 2.24 VectorTMain.cpp (continued). A main program to test VectorT.
This completes the listing.

Unit tests

After you implement a data structure you must test it to make sure it satisfies its specification. Most data structures have many methods, which raises the question of how to test the data structure. One approach would be to write all the methods of the data structure, then write some application that uses the data structure, then test the application to see how well it works. The problem with this approach is that if you discover a bug in the application you do not know if the error is in the application or in the data structure.

To alleviate this problem, common software engineering practice is to provide what is known as a unit test to test an individual method of a data structure. The idea is to thoroughly test each individual method of a data structure in isolation from the other methods and from any application that will use the data structure. Then, when you encounter a bug in an application you can rule out any errors in the data structure implementation, which in turn makes it easier to track down the bug.

Many systems for unit testing exist, each one depending on the programming language and the integrated development environment (IDE) that the programmer uses. Instead of using a commercial unit test system, all the data structures projects in the dp4ds software distribution for this book use a unit test system based on the main program for the project as a driver.

Figure 2.25 shows the unit test for the insert method of VectorT. The first group of lines in the file represent the user input for the main program in Figures 2.23 and 2.24.
2.3 A Vector Class

VectorT unit-insert

(10)
The capacity is 1
The size is 1

(20, 10)
The capacity is 2
The size is 2

(30, 20, 10)
The capacity is 4
The size is 3

(30, 40, 20, 10)
The capacity is 4
The size is 4

(30, 40, 20, 10, 50)
The capacity is 8
The size is 5

(30, 40, 60, 20, 10, 50)
The capacity is 8
The size is 6

Figure 2.25 unit-insert.txt. The unit test for the insert method of VectorT.
This unit test is contained in the dp4ds distribution software for this book. The sequence of insertions corresponds to those of Figure 2.17.

For example, the first line in Figure 2.25 is

i 10 0 w c s

You can see from Figure 2.23 that if you run the main program it will prompt you for a one-letter response. If you enter i as in the first line above, the program will branch to case 'I' then prompt you for the value to insert. If you enter 10 for the value, it will prompt you for the location to insert. If you enter 0, it will prompt you for another one-letter response. If you enter w, it will write the data structure to the output stream and prompt you for another one-letter response. Entering c then s will cause the capacity
Chapter 2  Arrays and Pointers

and size to output. If the input method is implemented correctly, the result of inputting the above line will produce the output

(10)
The capacity is 1
The size is 1

as shown in Figure 2.25.

Similarly, entering the other lines at the top of Figure 2.25 will produce the output shown in the rest of the figure if method insert() is implemented correctly. So, to test your implementation you would need to enter the sequence of prompts shown at the top of the figure and compare them with the expected output shown at the bottom of the figure. Note that the last one-letter response is q which terminates the main program.

Fortunately, you do not need to manually enter the responses to run the unit test. Instead, you can redirect the standard input for the program to come from the unit test file instead of from the keyboard. Because the sequence of responses is at the top of the file and the last q will terminate the main program, the program will not encounter the remainder of the file in its input stream.

The above technique is convenient if you are developing in a command line environment. If you are running in an IDE it should have a way to redirect the input to come from a file. However, a more convenient way to run a unit test in an IDE is to simply run the main program and wait for its first prompt in the console pane. Then you can simply copy the responses from the top of the unit test file and paste them into the console pane. Most IDEs will take the paste as if the stream of characters are entered from the keyboard. This technique usually works in a command line environment as well.

Every project in the dp4ds software distribution comes with a set of unit tests and a main program to drive them. For brevity, none of the later chapters show the main program driver or the unit tests. However, you should avail yourself of the unit tests when asked to implement a method of a data structure.

Exercises

2–1  Execute the main program ArrayClassicMain from Figure 2.9. Verify that it works correctly when the number of values in the input stream is less than the capacity of the array and when it is greater. Then remove the comment characters // to execute the statements that access memory beyond the range of the array. Experiment to find a value for the capacity of the array that will allow the out-of-bounds reference to apparently work correctly. Experiment to find a value that will crash your program.

2–2  Execute the main program ArrayTMain from Figure 2.10. Verify that it works correctly when the number of values in the input stream is less than the capacity of the array and when it is greater. Then remove the comment characters // to execute the statements that access memory beyond the range of the array. What error message do you get?

2–3  Implement the methods insert() and remove() in VectorT.hpp. Be sure to implement the preconditions. Test your implementation with the unit tests in the dp4ds distribution software for those cases that satisfy the preconditions. Test with interactive input to verify that your preconditions are implemented correctly. For example, if you enter -1 for the value of the index in remove the error message should be
2.3 A Vector Class

VectorT remove precondition 0 <= i & i < size() violated.
i == -1

2–4 Work Exercise 2–3 for VectorP.hpp.

2–5 The specification of VectorT never calls for decreasing the capacity. Modify the implementation of Exercise 2–3 so that when the size decreases to one fourth the capacity, the capacity decreases by one half. Maintain the invariant on _cap specified in Figure 2.18. Devise a new unit test named unit-collapse to test your feature. Use the format of Figure 2.25 including the input stream of responses to the main program and the listing of the expected output for a successful test.

2–6 Work Exercise 2–5 for VectorP.hpp.