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.

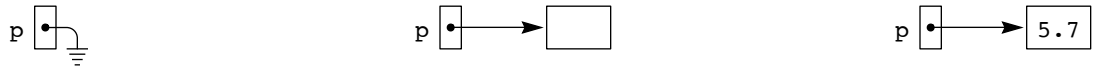
Early versions of C++ provide raw pointers that behave like pointers in C. With C pointers and C++ raw pointers, it is easy to make errors with memory deallocation that are difficult to detect with casual testing. Specifically, if a program fails to correctly delete all the unused storage from the heap, unused memory cells will accumulate in the heap and will not be available with later attempts to allocate from the heap. 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.

2.1 Pointer Types

A shared pointer is a class that contains, as one of its attributes, a raw pointer. This section shows how to allocate storage from the heap with shared pointers and how to manipulate them.

Shared pointers

Shared pointers alleviate the problem of memory leaks. The programmer is no longer responsible for deallocating storage from the heap with the `delete` operation. Instead,



(a) `shared_ptr<double> p;` **(b)** `p = make_shared<double>();` **(c)** `*p = 5.7;`

Figure 2.1 Allocating storage for a pointer to a double precision real, and assigning a real value to the allocated cell. The dashed triangle represents the `nullptr` value.

the execution system uses a reference counting algorithm that keeps track of how many pointers point to a shared object. When the number of references to the object decreases to zero so that the shared object can no longer be accessed, the system automatically deletes the object from the heap.

A shared pointer is an object of the standard class `shared_ptr`, which contains a raw pointer as an attribute. The non-member function `make_shared<>()` creates a shared pointer and takes the place of the `new` operation for raw pointers. The dereferencing operators `*` and `->` are provided as member functions of the class `shared_ptr` so you can access dynamically allocated storage with shared pointers just like you can with raw pointers. Here is an example of how to declare a shared pointer with the equivalent operations for a raw pointer.

```
shared_ptr<double> p;
p = make_shared<double>();
*p = 5.7;
```

```
double *p;
p = new double;
*p = 5.7;
```

The `shared_ptr` type is used in C++ to declare a shared pointer variable. The statement

```
shared_ptr<double> p;
```

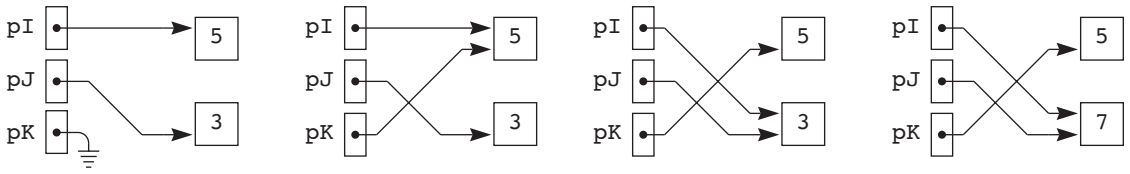
declares the variable `p` to be a pointer to a double precision value. Figure 2.1(a) shows the effect of the declaration.

Pointers are memory addresses, and at a low level of abstraction a memory address is an unsigned integer. In the C and C++ languages, 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 `nullptr`, which is a keyword introduced in C++11. The use of `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 2.1(a) shows the state of `p` after its declaration as being a pointer attached to a dashed triangle, which represents the special `nullptr` value. Smart pointers are automatically initialized to `nullptr`.

Before you can store a double precision value in the variable you must allocate storage for it from the heap. `make_shared<>()` is a function that allocates storage from the heap. It expects a type `T` in the angle brackets `<>`, allocates enough storage for `T`, and returns a pointer to the newly allocated storage. The statement

```
p = make_shared<double>();
```

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



(a) `*pI = 5;`
`*pJ = 3;`

(b) `pK = pI;`

(c) `pI = pJ;`

(d) `*pI = 2 + *pK;`

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

```
shared_ptr<double> p = make_shared<double>();
```

The `*` operator is 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

```
*p = 5.7;
```

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

You can combine the declaration, allocation, and initialization all in one statement as follows.

```
shared_ptr<double> p = make_shared<double>(5.7);
```

The function `make_shared<>()` has an optional parameter for initializing the value to which the pointer points.

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.

```
shared_ptr<int> pI = make_shared<int>();
shared_ptr<int> pJ = make_shared<int>();
shared_ptr<int> pK;
*pI = 5;
*pJ = 3;
pK = pI;
pI = pJ;
*pI = 2 + *pK;
cout << *pI << " " << *pJ << " " << *pK << endl;
```

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 the output statement streams

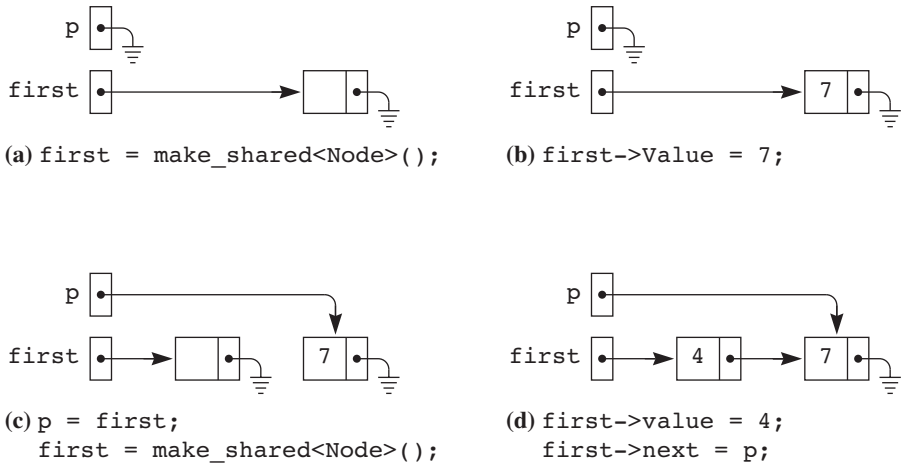


Figure 2.3 Trace of a code fragment that uses the `->` operator.

```
7 7 5
```

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

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;
    shared_ptr<Node> next;
};
shared_ptr<Node> first, p;
```

Variable `p` has type pointer to `Node`. Therefore, *p has type `Node`, which is a 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.


```
first = make_shared<Node>();
first->value = 7;
p = first;
first = make_shared<Node>();
first->value = 4;
first->next = p;
for (shared_ptr<Node> q = first; q != nullptr; q = q-> next) {
    cout << q -> value << " ";
}
```

The `for` loop outputs

```
4 7
```

The loop can be simplified to

```
for (shared_ptr<Node> q = first; q; q = q-> next) {
```

which is also a common coding pattern in C. The value `nullptr` is interpreted as `false`, and so `q` is interpreted as `true` when it has any value other than `nullptr`.

Placeholder type specifiers

C++11 introduced placeholder type specifiers, also known as automatic type deduction. When you declare a variable with an initializer you can omit the type of the variable and replace it with the keyword `auto`. The compiler will then automatically deduce the type of the variable from the initializer.

For example, instead of declaring `i` explicitly to be an integer as

```
int i = 7;
```

replace `int` with `auto` as

```
auto i = 7;
```

The compiler will deduce that the type of `i` is `int` from the constant `7` because it does not contain a decimal point. Here are some type deductions provided by `auto`.

```
auto i = 7;                                 int
auto x = 6.2;                               double
auto p = make_shared<double>(5.7);          shared_ptr<double>
auto pI = make_shared<int>();               shared_ptr<int>
auto first = make_shared<Node>();           shared_ptr<Node>
for (auto q = first; q; q = q-> next)       shared_ptr<Node>
```

The benefit of using `auto` may not seem like much in these simple examples. However, when dealing with complex data structures with involved types using `auto` as an abbreviation can shorten and simplify your code.


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

void swapVal(int g, int h) {
    auto temp = g;
    g = h;
    h = temp;
    cout << "g == " << g << ", h == " << h << endl;
}

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

Output

```
g == 5, h == 4
i == 4, j == 5
```

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

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. Following are 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 actual parameters. The output in the figure shows that `swapVal()` changes the values of formal parameters `g` and `h` but not the actual parameters `i` and `j`.


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

void swapRef(int &g, int &h) {
    auto temp = g;
    g = h;
    h = temp;
    cout << "g == " << g << ", h == " << h << endl;
}

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

Output

```
g == 5, h == 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.

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

```
h = temp;
```

executes, it is as if

```
j = temp;
```

executes.

Pointer parameters

The program in Figure 2.6 achieves the effect of pass by reference by explicitly passing pointers by value. The formal parameters `g` and `h` 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.


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

void swapPtrVal(shared_ptr<int> g, shared_ptr<int> h) {
    auto temp = *g; // before
    *g = *h;
    *h = temp;
    cout << "*g == " << *g << ", *h == " << *h << endl; // after
}

int main() {
    auto i = make_shared<int>(4);
    auto j = make_shared<int>(5);
    swapPtrVal(i, j);
    cout << "*i == " << *i << ", *j == " << *j << endl;
    return EXIT_SUCCESS;
}
```

Output

```
*g == 5, *h == 4
*i == 5, *j == 4
```

(a) Passing pointers by value, before. **(b)** Passing pointers by value, after.

Figure 2.6 `SwapPtrValMain.cpp`. A program to illustrate passing pointers by value. This technique achieves the effect of call by reference by changing the values to which `i` and `j` point.

Corresponding to the formal parameters, the actual parameters in the call to the function are `i` and `j`, also pointers to integers, and initialized to 4 and 5 respectively. Figure 2.6(a) shows the memory allocation immediately after the function is called. Actual parameter `i` and formal parameter `g` are pointers with the same value. That is, they point to the same memory cell in the heap.

Figure 2.6(b) shows the memory allocation immediately before the return from the function. The code in the function has changed the values in the cells to which the pointers point.

Behind the scenes, the C++ compiler generates code like that of Figure 2.6 to implement call by reference when programmers use the technique of Figure 2.5. In Figure 2.5 the actual parameters are not pointers, but integers. So, the compiler passes the addresses of the integers. Any assignment to a formal parameter in the function, like

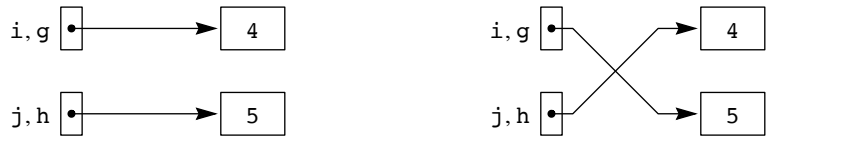

```
#include <iostream>
using namespace std;

void swapPtrRef(shared_ptr<int> &g, shared_ptr<int> &h) {
    auto temp = g; // before
    g = h;
    h = temp;
    cout << "*g == " << *g << ", *h == " << *h << endl; // after
}

int main() {
    auto i = make_shared<int>(4);
    auto j = make_shared<int>(5);
    swapPtrRef(i, j);
    cout << "*i == " << *i << ", *j == " << *j << endl;
    return EXIT_SUCCESS;
}
```

Output

```
*g == 5, *h == 4
*i == 5, *j == 4
```



(a) Passing pointers by reference, before. **(b)** Passing pointers by reference, after.

Figure 2.7 `SwapPtrRefMain.cpp`. A program to illustrate passing pointers by reference. This technique achieves the effect of call by reference by changing what `i` and `j` point to.

```
g = h;
```

generates code to assign the value of `h` to the value of `g`. At the lowest level of abstraction every parameter is passed by value. The only question is, What kind of value is passed, a data value or an address value? The processor passes a data value to implement pass by value and an address value to implement pass by reference.

In Figure 2.7, the pointer itself changes what it points to, instead of changing the value in the cell to which it points. The pointer parameters are called by reference. Figure 2.7(a) shows the memory allocation immediately after the function is called. Actual parameter `i` and formal parameter `g` are, in effect, the same pointer. When `i` is changed in the function, `g` is changed. Figure 2.7(b) shows the memory allocation immediately before the return from the function. The code in the function has changed the pointers instead of changing the content of the cells to which they point. If you delete the `&`

symbols in the parameter list, the output will be

```
*g == 5, *h == 4
*i == 4, *j == 5
```

because the pointers will not be changed in the main program.

Reference types

In the function of Figure 2.5,

```
void swapRef(int &g, int &h) {
```

which is called as follows

```
auto i = 4;
auto j = 5;
swapRef(i, j);
```

`int &` is known as a *reference type*.

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

```
auto i = 4;
int &g = i; // Must be initialized here.
cout << "g == " << g << ", i == " << i << endl;
g = 5;
cout << "g == " << g << ", i == " << i << endl;
```

is legal and produces the output

```
g == 4, i == 4
g == 5, i == 5
```

Because `g` refers to `i`, when you change the value of `g` you also change the value of `i`. This behavior is identical to that in function `swapRef()`. When you change the value of formal parameter `g` in the function, you change the value of actual parameter `i`.

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` as in the following code fragment

```
auto i = 4;
int &g = i; // Must be initialized here.
auto s = 6;
g = s;
s = 7;
cout << "g == " << g << ", i == " << i << ", s == " << s << endl;
```

the assignment of `s` to `g` does not make `g` refer to `s`. Instead, it gives the value of `s` to `g`. The output is

```
g == 6, i == 6, s == 7
```



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

void readStream(istream &is, unique_ptr<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, unique_ptr<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.

Changing the value of `s` to 7 does not change the value of `g`, which still refers to `i`.

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 `g` is initialized to refer to `i` in the above code fragments. You use the formal parameter in a function the same way you use the reference variable `g` in the above code. Once you set which cell a call-by-reference formal parameter refers to when the function is called, you can never change it to refer to a different cell in memory.

Dynamically allocated arrays

In Chapter 1, Figure 1.19 (page 26) shows the allocation of an array of pointers as

```
const int NUM_SHAPES = 5;
shared_ptr<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) (page 25) 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

```
auto arr = make_unique<double[]>(cap);
```



```
#include <cstdlib> // EXIT_SUCCESS.
#include <fstream> // ifstream.
#include <memory>
#include "ArrayClassicMain.hpp"
#include "Utilities.hpp" // promptIntGE.
using namespace std;

int main() {
    int cap = promptIntGE("Enter array capacity", 1);
    auto arr = make_unique<double[]>(cap);
    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;
    }
    return EXIT_SUCCESS;
}

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

void writeStream(ostream &os, unique_ptr<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.

allocates the array dynamically from the heap. Allocation from the heap allows `cap` to be a variable.

The C++20 standard provides the ability to have a shared pointer to an array. Unfortunately, at the time of this writing C++20 compilers are not widely available. So, the dp4ds Distribution is designed for standard C++17, which does not allow shared pointers to arrays. The `unique_ptr` type is for pointers to objects that are not shared and provides the same automatic garbage collection the `shared_ptr` type provides. The `make_unique<>()` funcation is for dynamic allocation for unique pointers corresponding to the `make_shared<>()` function for shared pointers.

This restriction to C++17 requires the array parameters in functions `readStream()` and `writeStream()` to be more complicated than would otherwise be the case. The formal parameter `unique_ptr<double[]> *d` is a raw pointer to a unique pointer to an array. The actual parameter `&arr` is the address of the unique pointer to the array. In the function, `(*d)` is the array and `(*d)[i]` is the *i* th element of the array.

Function `readStream` inputs double precision values from the input stream. There are two possibilities—the number of values in the stream is less than or equal to the capcity of the array, 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.

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 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 an array of 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

```
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

```
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 and recompile the app. 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

```
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.


```
// ========= ASeq =========
template<class T>
class ASeq {
public:
    explicit ASeq(int cap = 0) {}; // Avoid implicit conversion.
    virtual ~ASeq() = default;
    virtual T &operator[](int i) = 0; // For read/write.
    virtual T const &operator[](int i) const = 0; // For read-only.
    virtual int cap() const = 0;

    virtual void toStream(ostream &os) const = 0;
    // Pre: operator<< is defined for T.
    // Post: A string representation of this sequence is streamed to os.

    virtual int fromStream(istream &is) = 0;
    // Pre: operator>> is defined for T.
    // Post: The items of input stream is are appended to this sequence.

    ASeq(ASeq const &rhs) = delete; // Disabled.
    ASeq &operator=(ASeq const &rhs) = delete; // Disabled.
};
```

Figure 2.10 The template class for an abstract sequence. Partial contents of `ASeq.hpp`.

An abstract sequence class

Figure 2.10 shows the template class for the `ASeq` abstract class. An array is a specific example of a more general sequence. The mathematical notion of a sequence is a function f from the integers to a set of data of type T indexed by integers. For example, the sequence

40 20 70 50 10

is a set of data of type `int` indexed by integers 0..4 with the function f defined as

$f(0) = 40, f(1) = 20, f(2) = 70, f(3) = 50, f(4) = 10$

A specific implementation of an abstract sequence encloses the index in square brackets. For example, if `myArr` inherits from `ASeq` and its elements are the integers in the above sequence, then `myArr[2]` has the value 70.

The code

```
template <class T> class ASeq
```

declares `ASeq` to be a template class. `T` is like a formal parameter for the type. The actual parameter could be `int`, `double`, or some other type or class. Parameter `cap` in


```
// ========= operator<< =========
template<class T>
ostream &operator<<(ostream &os, ASeq<T> const &rhs) {
    rhs.toStream(os);
    return os;
}

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

Figure 2.11 The non-member functions for streaming output from and input to `ASeq`. Partial contents of `ASeq.hpp`.

the constructor is the capacity of the sequence with a default value of 0. Any subclass of `ASeq` is required to implement the square bracket operator specified by

```
virtual T &operator[](int i) = 0;
virtual T const &operator[](int i) const = 0;
```

It must also implement function `cap()`, which returns the capacity of the sequence. The methods `toStream` and `fromStream` are for output from and input to an object that inherits from `ASeq`. The functions in Figure 2.11 use them to overload the streaming operators. For example, if `myArr` inherits from `ASeq` and it implements `toStream` you could output the entire array as follows.

```
cout << myArr;
```

Figure 2.12 is the UML diagram for `ASeq` and two of its subclasses, `ArrayT` and `VectorT`, in the dp4ds Distribution. `ArrayT` is a safe array that guards against accessing memory outside the range of the array. `VectorT` is a sequence that automatically expands during program execution to accommodate an increase in the amount of data being processed. The dashed box around the T in Figure 2.12 is the UML notation for the template parameter.

A safe array of doubles

Figure 2.13 is a main program that uses a template class that implements a safe array. `ArrayT.hpp` implements the class. 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

```
typedef ArrayT<double> ArrayDouble;
```

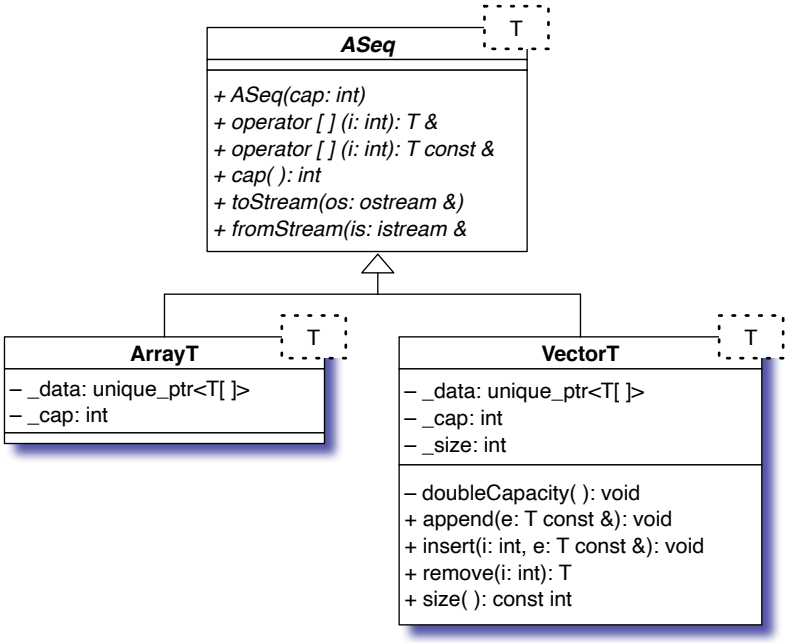



Figure 2.12 The UML diagram for the template classes that provide safe arrays and vectors.

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 the `ArrayT` class. 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.

The main program is short because functions `readStream()` and `writeStream()` are provided along with the template class in the library. They perform general input and output like the corresponding functions in Figures 2.8 and 2.9 for the primitive array. The `ArrayT` project also supplies the `writeFormatted()` function that gives more control over the output format. 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


```
#include <cstdlib> // EXIT_SUCCESS.
#include <iostream> // cout.
#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 = arr.fromStream(ifs);
        ifs.close();
        cout << "Read count == " << length << endl;
        cout << "Array data:" << endl;
        writeFormatted(cout, arr, length, 16, 1, 6);
        // arr[2 * arr.cap()] = 123.4;
        // cout << arr[2 * arr.cap()] << endl;
        return EXIT_SUCCESS;
    }
}
```

Figure 2.13 `ArrayTMain.cpp`. A main program that uses a dynamically allocated safe array of doubles from a template class.

```
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.

Figure 2.14 is a partial listing of the template class for the safe array. 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.

Attribute `_data` is declared as

```
unique_ptr<T[]> _data;
```

It is a pointer to an array of elements of type `T`. Allocation in the constructor is accomplished with

```
_data = make_unique<T[]>(cap);
```



```
// ========= ArrayT =========
template<class T>
class ArrayT : public ASeq<T> {
private:
    unique_ptr<T[]> _data;
    int _cap;
public:
    explicit ArrayT(int cap = 1);
    int cap() const override;
    T &operator[](int i) override; // For read/write.
    T const &operator[](int i) const override; // For read-only.

    void toStream(ostream &os) const override;
    // Pre: operator<< is defined for T.
    // Post: A string representation of this array is returned to output stream os.

    int fromStream(istream &is) override;
    // Pre: operator>> is defined for T.
    // Post: The items of input stream is are appended to this array.
};

// ========= 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 = make_unique<T[]>(cap);
    _cap = cap;
}

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

Figure 2.14 The template class that provides a safe array. Partial contents of `ArrayT.hpp`.

which allocates the array from the heap. 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


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

// ========= fromStream =========
template<class T>
int ArrayT<T>::fromStream(istream &is) {
    int len = 0;
    if constexpr(!is_shared_ptr<T>::value)
        for (int i = 0; i < _cap && is >> _data[i]; i++) {
            len++;
        }
    return len;
}
```

Figure 2.15 The `toStream()` and `fromStream()` functions. Partial contents of `ArrayT.hpp`.

```
_data = make_unique<double[]>(cap);
```

The constructor sets attribute `_cap` to the value passed to it as a parameter.

Figure 2.15 shows the `toStream()` and `fromStream()` functions. If `myArr` has values 10, 70, and 40 for `myArr[0]`, `myArr[1]`, and `myArr[2]` respectively then `toStream()` streams

```
(10, 70, 40)
```

The `writeFormatted()` function is not shown.

Overloading operator [ ]

Figure 2.16 shows 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.

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


```
// ========= 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 2.16 Overloading the subscript operator `[]`. Partial contents of `ArrayT.hpp`.

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 2.13, 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 2.13, you can think of the function as returning `double &`, that is, a reference to a double precision real.

The first statement in the method

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

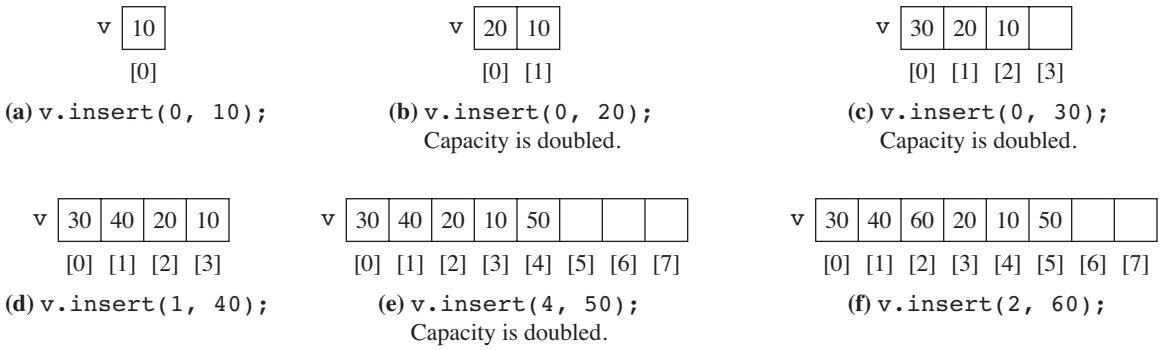



Figure 2.17 The effect of the insert operation on vector `v`. The first parameter of the insert method is the index of where to insert the value. The second parameter is the value to be inserted.

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 `_data[i]`. For example, suppose the main program of Figure 2.13 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 `_data[13]`, which has the value 7.1 given to `arr[13]`.

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



```
template<class T>
// ========= VectorT =========
template<class T>
class VectorT : public ASeq<T> {
private:
    unique_ptr<T[]> _data;
    int _cap;  // Invariant: 0 < _cap, and _cap is a power of 2.
    int _size; // Invariant: 0 <= _size <= _cap.

    void doubleCapacity();
```

Figure 2.18 Specification of the private part of the vector class `VectorT`.

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 `const T&`, 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 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 `int`, and you want to set its third element to 50, you execute

```
v[2] = 50;
```



```
// ========= 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];
    }
    _data.reset(newDat);
}
```

Figure 2.19 Implementaation of the `doubleCapacity()` method of the vector class `VectorT`.

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.

A vector implementation

Vector class `VectorT` inherits from the abstract sequence `ASeq` in the same way that `ArrayT` does. Figure 2.12 shows the UML diagram for `VectorT` and Figure 2.18 is the specification if its private part. A vector has three attributes. `_data` is the raw array of values, `_cap` is the capacity of the vector, and `_size` is its size.

Figure 2.19 shows the implementation of `doubleCapacity()`, which doubles the capacity of the vector. First, it allocates a new array named `newDat` with twice the capacity of the current array attribute `_data`.

```
_cap *= 2;
T *newDat = new T[_cap];
```

`newDat` is a pointer to a raw array, not a `unique_pointer` to a raw array. It is allocated with the `new` operator, and, by itself, there would be no automatic garbage collection. Second, it copies the values from `_data` to `newDat` in a `for` loop. Third, the statement

```
_data.reset(newDat);
```



```
public:
    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 of the constructor. Specification and implementation of `cap()` and `size()`.

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

(b) Implementation of the constructor.

Figure 2.20 The constructor, `cap()`, and `size()` methods of the vector class `vectorT`.

makes `newDat` an attribute of the `_data` object, which is a `unique_pointer` to a raw array. Automatic garbage collection does deallocate the old `_data` 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 2.20(a) shows the specification of the constructor and part (b) shows its implementation. The constructor allocates a new array of type `T` with one cell and sets `_cap` to one and `_size` to zero. The declarations and implementations of `cap()` and `size()` in Figure 2.20(a) are combined because they are simple one-liners.

Figure 2.21 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


```
public:
    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 of `append()`, `insert()`, and `remove()`.

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

(b) Implementation of `append()`.

Figure 2.21 Methods to append to, insert into, and remove from `VectorT`. Implementation of `insert()` and `remove()` are exercises for the student.

```
// 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()` 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 2.22 shows that overloading the `[]` operator is accomplished as it is with the safe array classes.

Figure 2.23 shows how `VectorT` handles input and output. Function `toStream()` is a method and has only one parameter, an output stream. Because `VectorT` has at-


```
T &operator[](int i) override; // For read/write.
T const &operator[](int i) const override; // For read-only.
```

(a) Specification of `operator[]`.

```
// ========= operator[] =========
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];
}

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];
}
```

(b) Implementation of `operator[]`.

Figure 2.22 Overloading the `[]` operator of the vector class `VectorT`.

tribute `_size` there is no need for a length parameter as there is in `writeFormatted()` for `ArrayT` in Figure 2.13.

Figure 2.23 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 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 2.24 and 2.25 show the listing of a main program to test the implementation 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.


```
// ========= 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 2.23 The input/output streaming operators of the vector class `VectorT.hpp`.

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


```
typedef VectorT<int> VectorInt;

int main() {
    VectorInt v;
    int value, index;
    char response;
    do {
        cout << "\n(c)ap  (s)ize  (a)ppend  (i)nsert  (r)emove  "
             << "se(t)  (w)rite  (q)uit: ";
        cin >> response;
        switch (toupper(response)) {
        case 'C':
            cout << "\nThe capacity is " << v.cap() << endl;
            break;
        case 'S':
            cout << "\nThe size is " << v.size() << endl;
            break;
        case 'A':
            cout << "\nAppend what integer value? ";
            cin >> value;
            v.append(value);
            break;
        case 'I':
            cout << "\nInsert what integer value? ";
            cin >> value;
            cout << "\nInsert at what location? ";
            cin >> index;
            v.insert(index, value);
            break;
```

Figure 2.24 `VectorTMain.cpp`. A main program to test `VectorT`. The listing continues in the next figure.

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 Distribution for this book use a unit test system based on the main program for the project as a driver.

Figure 2.26 shows the unit test for the insert method of `VectorT`. The first group of


```
        case 'R':
            cout << "\nRemove from what location? ";
            cin >> index;
            value = v.remove(index);
            cout << "\n" << value << " removed" << endl;
            break;
        case 'T':
            cout << "\nSet what integer value? ";
            cin >> value;
            cout << "\nSet 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');
    cout << endl;
    return EXIT_SUCCESS;
}
```

Figure 2.25 `VectorTMain.cpp` (continued). A main program to test `VectorT`. This completes the listing.

lines in the file represent the user input for the main program in Figures 2.24 and 2.25. For example, the first line in Figure 2.26 is

```
i 10 0 w c s
```

You can see from Figure 2.24 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 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
```



```
i 10 0 w c s
i 20 0 w c s
i 30 0 w c s
i 40 1 w c s
i 50 4 w c s
i 60 2 w c s q

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 2.26 `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.

as shown in Figure 2.26.

Similarly, entering the other lines at the top of Figure 2.26 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 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 2.13. 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

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

2–4 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 2.26 including the input stream of responses to the main program and the listing of the expected output for a successful test.

2–5 The text (page 52) gives the output of Figure 2.7 if you delete the `&` symbols in the parameter list. Draw the memory allocation corresponding to Figures 2.7(a) and (b) for this case immediately before the return from the function.

