18 Vectors and Arrays

"Caveat emptor!"

—Good advice

This chapter describes how vectors are copied and accessed through subscripting. To do that, we discuss copying in general and consider **vector**'s relation to the lower-level notion of arrays. We present arrays' relation to pointers and consider the problems arising from their use. We also present the five essential operations that must be considered for every type: construction, default construction, copy construction, copy assignment, and destruction. In addition, a container needs a move constructor and a move assignment.

18.1 Introduction

To get into the air, a plane has to accelerate along the runway until it moves fast enough to "jump" into the air. While the plane is lumbering along the runway, it is little more than a particularly heavy and awkward truck. Once in the air, it soars to become an altogether different, elegant, and efficient vehicle. It is in its true element.

In this chapter, we are in the middle of a "run" to gather enough programming language features and techniques to get away from the constraints and difficulties of plain computer memory. We want to get to the point where we can program using types that provide exactly the properties we want based on logical needs. To "get there" we have to overcome a number of fundamental constraints related to access to the bare machine, such as the following:

- An object in memory is of fixed size.
- An object in memory is in one specific place.
- The computer provides only a few fundamental operations on such objects (such as copying a word, adding the values from two words, etc.).

Basically, those are the constraints on the built-in types and operations of C++ (as inherited through C from hardware; see §22.2.5 and Chapter 27). In Chapter 17, we saw the beginnings of a **vector** type that controls all access to its elements and provides us with operations that seem "natural" from the point of view of a user, rather than from the point of view of hardware.

This chapter focuses on the notion of copying. This is an important but rather technical point: What do we mean by copying a nontrivial object? To what extent are the copies independent after a copy operation? What copy operations are there? How do we specify them? And how do they relate to other fundamental operations, such as initialization and cleanup?

Inevitably, we get to discuss how memory is manipulated when we don't have higher-level types such as **vector** and **string**. We examine arrays and pointers, their relationship, their use, and the traps and pitfalls of their use. This is essential information to anyone who gets to work with low-level uses of C++ or C code.

Please note that the details of **vector** are peculiar to **vector**s and the C++ ways of building new higher-level types from lower-level ones. However, every "higher-level" type (**string**, **vector**, **list**, **map**, etc.) in every language is somehow built from the same machine primitives and reflects a variety of resolutions to the fundamental problems described here.

18.2 Initialization

Consider our **vector** as it was at the end of Chapter 17:

```
class vector {
        int sz;                     // the size
        double* elem;               // a pointer to the elements
public:
        vector(int s)                                   // constructor
                :sz{s}, elem{new double[s]} { /* . . . */ }   // allocates memory
        ~vector()                                       // destructor
                { delete[] elem; }                      // deallocates memory
        // . . .
};
```

That's fine, but what if we want to initialize a vector to a set of values that are not defaults? For example:

```
vector v1 = {1.2, 7.89, 12.34 };
```

We can do that, and it is much better than initializing to default values and then assigning the values we really want:

```
vector v2(2);          // tedious and error-prone
v2[0] = 1.2;
v2[1] = 7.89;
v2[2] = 12.34;
```

Compared to **v1**, the "initialization" of **v2** is tedious and error-prone (we deliberately got the number of elements wrong in that code fragment). Using **push_back()** can save us from mentioning the size:

```
vector v3;                  // tedious and repetitive
v2.push_back(1.2);
v2.push_back(7.89);
v2.push_back(12.34);
```

But this is still repetitive, so how do we write a constructor that accepts an initializer list as its argument? A **{ }**-delimited list of elements of type **T** is presented to the programmer as an object of the standard library type **initializer_list<T>**, a list of **T**s, so we can write

```
class vector {
        int sz;                     // the size
        double* elem;               // a pointer to the elements
public:
        vector(int s)               // constructor (s is the element count)
                :sz{s}, elem{new double[sz]}    // uninitialized memory for elements
        {
                for (int i = 0; i<sz; ++i) elem[i] = 0.0;     // initialize
        }

        vector(initializer_list<double> lst)             // initializer-list constructor
                :sz{lst.size()}, elem{new double[sz]}    // uninitialized memory
                                                         // for elements
        {
                copy( lst.begin(),lst.end(),elem);  // initialize (using std::copy(); §B.5.2)
        }
        // . . .
};
```

We used the standard library **copy** algorithm (§B.5.2). It copies a sequence of elements specified by its first two arguments (here, the beginning and the end of the **initializer_list**) to a sequence of elements starting with its third argument (here, the **vector**'s elements starting at **elem**).

Now we can write

```
vector v1 = {1,2,3};      // three elements 1.0, 2.0, 3.0
vector v2(3);             // three elements each with the (default) value 0.0
```

Note how we use **( )** for an element count and **{ }** for element lists. We need a notation to distinguish them. For example:

```
vector v1 {3};          // one element with the value 3.0
vector v2(3);           // three elements each with the (default) value 0.0
```

This is not very elegant, but it is effective. If there is a choice, the compiler will interpret a value in a **{ }** list as an element value and pass it to the initializer-list constructor as an element of an **initializer_list**.

In most cases – including all cases we will encounter in this book – the **=** before an **{ }** initializer list is optional, so we can write

```
vector v11 = {1,2,3};   // three elements 1.0, 2.0, 3.0
vector v12 {1,2,3};     // three elements 1.0, 2.0, 3.0
```

The difference is purely one of style.

Note that we pass **initializer_list<double>** by value. That was deliberate and required by the language rules: an **initializer_list** is simply a handle to elements allocated "elsewhere" (see §B.6.4).

18.3 Copying

Consider again our incomplete **vector**:

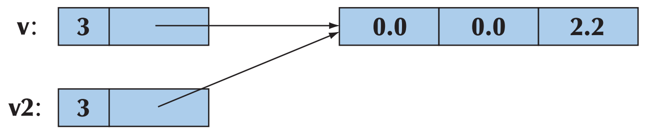
```
class vector {
        int sz;                 // the size
        double* elem;           // a pointer to the elements
public:
        vector(int s)                                       // constructor
                :sz{s}, elem{new double[s]} { /* . . . */ }  // allocates memory
        ~vector()                                           // destructor
                { delete[] elem; }                          // deallocates memory
        // . . .
};
```

Let's try to copy one of these vectors:

```
void f(int n)
{
        vector v(3);            // define a vector of 3 elements
        v.set(2,2.2);           // set v[2] to 2.2
        vector v2 = v;        // what happens here?
        // . . .
}
```

Ideally, v2 becomes a copy of v (that is, = makes copies); that is, v2.size()==v.size() and v2[i]==v[i] for all is in the range [0:v.size()). Furthermore, all memory is returned to the free store upon exit from f(). That's what the standard library vector does (of course), but it's not what happens for our still-far-too-simple vector. Our task is to improve our vector to get it to handle such examples correctly, but first let's figure out what our current version actually does. Exactly what does it do wrong? How? And why? Once we know that, we can probably fix the problems. More importantly, we have a chance to recognize and avoid similar problems when we see them in other contexts.

The default meaning of copying for a class is "Copy all the data members." That often makes perfect sense. For example, we copy a Point by copying its coordinates. But for a pointer member, just copying the members causes problems. In particular, for the vectors in our example, it means that after the copy, we have v.sz==v2.sz and v.elem==v2.elem so that our vectors look like this:



That is, v2 doesn't have a copy of v's elements; it shares v's elements. We could write

```
v.set(1,99);          // set v[1] to 99
v2.set(0,88);         // set v2[0] to 88
cout << v.get(0) << ' ' << v2.get(1);
```

The result would be the output 88 99. That wasn't what we wanted. Had there been no "hidden" connection between v and v2, we would have gotten the output 0 0, because we never wrote to v[0] or to v2[1]. You could argue that the behavior we got is "interesting," "neat!" or "sometimes useful," but that is not what we intended or what the standard library vector provides. Also, what happens when we return from f() is an unmitigated disaster. Then, the destructors for v and v2 are implicitly called; v's destructor frees the storage used for the elements using

```
delete[] elem;
```

and so does v2's destructor. Since elem points to the same memory location in both v and v2, that memory will be freed twice with likely disastrous results (§17.4.6).

18.3.1 Copy constructors

So, what do we do? We'll do the obvious: provide a copy operation that copies the elements and make sure that this copy operation gets called when we initialize one **vector** with another.

Initialization of objects of a class is done by a constructor. So, we need a constructor that copies. Unsurprisingly, such a constructor is called a *copy constructor*. It is defined to take as its argument a reference to the object from which to copy. So, for class **vector** we need

```
vector(const vector&);
```

This constructor will be called when we try to initialize one **vector** with another. We pass by reference because we (obviously) don't want to copy the argument of the constructor that defines copying. We pass by **const** reference because we don't want to modify our argument (§8.5.6). So we refine **vector** like this:

```
class vector {
        int sz;
        double* elem;
public:
        vector(const vector&) ;           // copy constructor: define copy
        // . . .
};
```

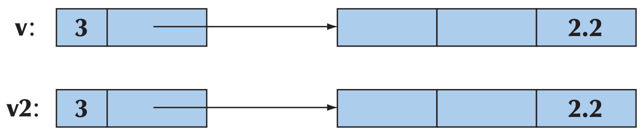
The copy constructor sets the number of elements (**sz**) and allocates memory for the elements (initializing **elem**) before copying element values from the argument **vector**:

```
vector:: vector(const vector& arg)
// allocate elements, then initialize them by copying
        :sz{arg.sz}, elem{new double[arg.sz]}
{
        copy(arg,arg+sz,elem);        // std::copy(); see §B.5.2
}
```

Given this copy constructor, consider again our example:

```
vector v2 = v;
```

This definition will initialize **v2** by a call of **vector**'s copy constructor with **v** as its argument. Again given a **vector** with three elements, we now get



Given that, the destructor can do the right thing. Each set of elements is correctly freed. Obviously, the two **vector**s are now independent so that we can change the value of elements in **v** without affecting **v2** and vice versa. For example:

```
v.set(1,99);                // set v[1] to 99
v2.set(0,88);               // set v2[0] to 88
cout << v.get(0) << ' ' << v2.get(1);
```

This will output **0 0**.

Instead of saying

```
vector v2 = v;
```

we could equally well have said

```
vector v2 {v};
```

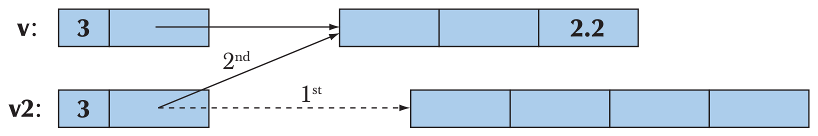
When **v** (the initializer) and **v2** (the variable being initialized) are of the same type and that type has copying conventionally defined, those two notations mean exactly the same thing and you can use whichever notation you like better.

18.3.2 Copy assignments

We handle copy construction (initialization), but we can also copy **vector**s by assignment. As with copy initialization, the default meaning of copy assignment is memberwise copy, so with **vector** as defined so far, assignment will cause a double deletion (exactly as shown for copy constructors in §18.3.1) plus a memory leak. For example:

```
void f2(int n)
{
        vector v(3);        // define a vector
        v.set(2,2.2);
        vector v2(4);
        v2 = v;             // assignment: what happens here?
        // . . .
}
```

We would like **v2** to be a copy of **v** (and that's what the standard library **vector** does), but since we have said nothing about the meaning of assignment of our **vector**, the default assignment is used; that is, the assignment is a memberwise copy so that **v2**'s **sz** and **elem** become identical to **v**'s **sz** and **elem**, respectively. We can illustrate that like this:



When we leave **f2()**, we have the same disaster as we had when leaving **f()** in §18.3 before we added the copy constructor: the elements pointed to by both **v** and **v2** are freed twice (using **delete[]**). In addition, we have leaked the memory initially allocated for **v2**'s four elements. We "forgot" to free those. The remedy for this copy assignment is fundamentally the same as for the copy initialization (§18.3.1). We define an assignment that copies properly:

```
class vector {
        int sz;
        double* elem;
public:
        vector& operator=(const vector&) ;      // copy assignment
        // . . .
};

vector& vector::operator=(const vector& a)
        // make this vector a copy of a
{
        double* p = new double[a.sz];          // allocate new space
        copy(a.elem,a.elem+a.sz,elem);         // copy elements
        delete[] elem;                         // deallocate old space
        elem = p;                              // now we can reset elem
        sz = a.sz;
        return *this;                  // return a self-reference (see §17.10)
}
```

Assignment is a bit more complicated than construction because we must deal with the old elements. Our basic strategy is to make a copy of the elements from the source **vector**:

```
        double* p = new double[a.sz];          // allocate new space
        copy(a.elem,a.elem+a.sz,elem);         // copy elements
```

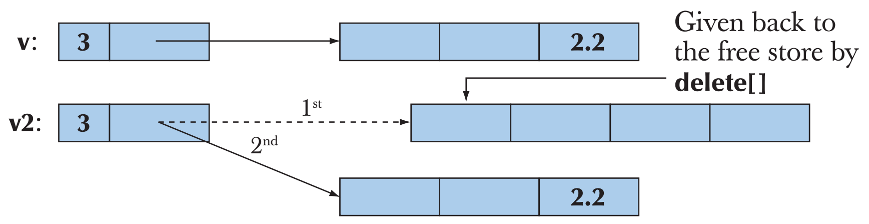
Then we free the old elements from the target **vector**:

```
delete[] elem;                                 // deallocate old space
```

Finally, we let **elem** point to the new elements:

```
elem = p;                    // now we can reset elem
sz = a.sz;
```

We can represent the result graphically like this:



We now have a **vector** that doesn't leak memory and doesn't free (**delete[]**) any memory twice.

When implementing the assignment, you could consider simplifying the code by freeing the memory for the old elements before creating the copy, but it is usually a very good idea not to throw away information before you know that you can replace it. Also, if you did that, strange things would happen if you assigned a **vector** to itself:

```
vector v(10);
         v = v;    // self-assignment
```

Please check that our implementation handles that case correctly (if not with optimal efficiency).

18.3.3 Copy terminology

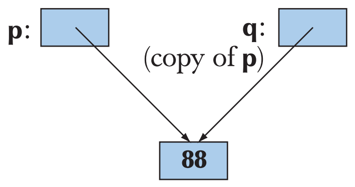
Copying is an issue in most programs and in most programming languages. The basic issue is whether you copy a pointer (or reference) or copy the information pointed to (referred to):

- *Shallow copy* copies only a pointer so that the two pointers now refer to the same object. That's what pointers and references do.
- *Deep copy* copies what a pointer points to so that the two pointers now refer to distinct objects. That's what **vector**s, **string**s, etc. do. We define copy constructors and copy assignments when we want deep copy for objects of our classes.

Here is an example of shallow copy:

```
int* p = new int{77};
int* q = p;              // copy the pointer p
*p = 88;                 // change the value of the int pointed to by p and q
```

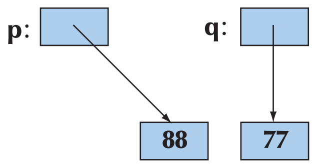
We can illustrate that like this:



In contrast, we can do a deep copy:

```
int* p = new int{77};
int* q = new int{*p};   // allocate a new int, then copy the value pointed to by p
*p = 88;                // change the value of the int pointed to by p
```

We can illustrate that like this:



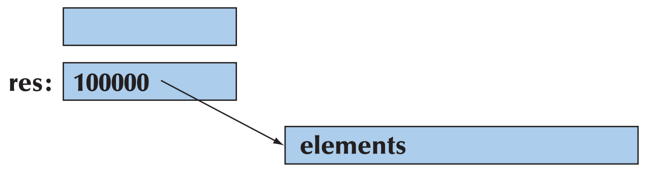
Using this terminology, we can say that the problem with our original **vector** was that it did a shallow copy, rather than copying the elements pointed to by its **elem** pointer. Our improved **vector**, like the standard library **vector**, does a deep copy by allocating new space for the elements and copying their values. Types that provide shallow copy (like pointers and references) are said to have *pointer semantics* or *reference semantics* (they copy addresses). Types that provide deep copy (like **string** and **vector**) are said to have *value semantics* (they copy the values pointed to). From a user perspective, types with value semantics behave as if no pointers were involved – just values that can be copied. One way of thinking of types with value semantics is that they "work just like integers" as far as copying is concerned.

18.3.4 Moving

If a **vector** has a lot of elements, it can be expensive to copy. So, we should copy **vector**s only when we need to. Consider an example:

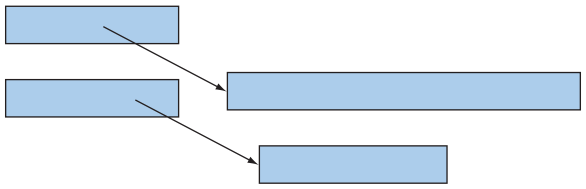
```
vector fill(istream& is)
{
        vector res;
        for (double x; is>>x; ) res.push_back(x);
        return res;
}

void use()
{
        vector vec = fill(cin);
        // ... use vec ...
}
```

Here, we fill the local vector **res** from the input stream and return it to **use()**. Copying **res** out of **fill()** and into **vec** could be expensive. But why copy? We don't want a copy! We can never use the original (**res**) after the return. In fact, **res** is destroyed as part of the return from **fill()**. So how can we avoid the copy? Consider again how a vector is represented in memory:



We would like to "steal" the representation of **res** to use for **vec**. In other words, we would like **vec** to refer to the elements of **res** without any copy.

After moving **res**'s element pointer and element count to **vec**, **res** holds no elements. We have successfully moved the value from **res** out of **fill()** to **vec**. Now, **res** can be destroyed (simply and efficiently) without any undesirable side effects:



We have successfully moved 100,000 **double**s out of **fill()** and into its caller at the cost of four single-word assignments.

How do we express such a move in C++ code? We define move operations to complement the copy operations:

```
class vector {
        int sz;
        double* elem;
public:
        vector(vector&& a);                  // move constructor
        vector& operator=(vector&&);         // move assignment
        // . . .
        };
```

The funny **&&** notation is called an "rvalue reference." We use it for defining move operations. Note that move operations do not take **const** arguments; that is, we write **(vector&&)** and not **(const vector&&)**. Part of the purpose of a move operation is to modify the source, to make it "empty." The definitions of move operations tend to be simple. They tend to be simpler and more efficient than their copy equivalents. For **vector**, we get

```
vector::vector(vector&& a)
        :sz{a.sz}, elem{a.elem}              // copy a's elem and sz
{
        a.sz = 0;                            // make a the empty vector
        a.elem = nullptr;
}

vector& vector::operator=(vector&& a) // move a to this vector
{
        delete[] elem;                       // deallocate old space
        elem = a.elem;                       // copy a's elem and sz
        sz = a.sz;
        a.elem = nullptr;                    // make a the empty vector
        a.sz = 0;
        return *this;                        // return a self-reference (see §17.10)
}
```

By defining a move constructor, we make it easy and cheap to move around large amounts of information, such as a vector with many elements. Consider again:

```
vector fill(istream& is)
{
        vector res;
        for (double x; is>>x; ) res.push_back(x);
        return res;
}
```

The move constructor is implicitly used to implement the return. The compiler knows that the local value returned (**res**) is about to go out of scope, so it can move from it, rather than copying.

The importance of move constructors is that we do not have to deal with pointers or references to get large amounts of information out of a function. Consider this flawed (but conventional) alternative:

```
vector* fill2(istream& is)
{
        vector* res = new vector;
        for (double x; is>>x; ) res->push_back(x);
        return res;
}

void use2()
{
        vector* vec = fill(cin);
        // ... use vec ...
        delete vec;
}
```

Now we have to remember to delete the **vector**. As described in §17.4.6, deleting objects placed on the free store is not as easy to do consistently and correctly as it might seem.

18.4 Essential operations

We have now reached the point where we can discuss how to decide which constructors a class should have, whether it should have a destructor, and whether you need to provide copy and move operations. There are seven essential operations to consider:

- Constructors from one or more arguments
- Default constructor
- Copy constructor (copy object of same type)
- Copy assignment (copy object of same type)
- Move constructor (move object of same type)
- Move assignment (move object of same type)
- Destructor

Usually we need one or more constructors that take arguments needed to initialize an object. For example:

```
string s {"cat.jpg"};                 // initialize s to the character string "cat.jpg"
Image ii {Point{200,300},"cat.jpg"};  // initialize a Point with the
                                      // coordinates{200,300},
                                      // then display the contents of file
                                      // cat.jpg at that Point
```

The meaning/use of an initializer is completely up to the constructor. The standard **string**'s constructor uses a character string as an initial value, whereas **Image**'s constructor uses the string as the name of a file to open. Usually we use a constructor to establish an invariant (§9.4.3). If we can't define a good invariant for a class that its constructors can establish, we probably have a poorly designed class or a plain data structure.

Constructors that take arguments are as varied as the classes they serve. The remaining operations have more regular patterns.

How do we know if a class needs a default constructor? We need a default constructor if we want to be able to make objects of the class without specifying an initializer. The most common example is when we want to put objects of a class into a standard library **vector**. The following works only because we have default values for **int**, **string**, and **vector<int>**:

```
vector<double> vi(10);          // vector of 10 doubles, each initialized to 0.0
vector<string> vs(10);          // vector of 10 strings, each initialized to ""
vector<vector<int>> vvi(10);    // vector of 10 vectors, each initialized to vector{}
```

So, having a default constructor is often useful. The question then becomes: "When does it make sense to have a default constructor?" An answer is: "When we can establish the invariant for the class with a meaningful and obvious default value." For value types, such as **int** and **double**, the obvious value is **0** (for **double**, that becomes **0.0**). For **string**, the empty **string**, **""**, is the obvious choice. For **vector**, the empty **vector** serves well. For every type **T**, **T{}** is the default value, if a default exists. For example, **double{}** is **0.0**, **string{}** is **""**, and **vector<int>{}** is the empty **vector** of **int**s.

A class needs a destructor if it acquires resources. A resource is something you "get from somewhere" and that you must give back once you have finished using it. The obvious example is memory that you get from the free store (using **new**) and have to give back to the free store (using **delete** or **delete[ ]**). Our **vector** acquires memory to hold its elements, so it has to give that memory back; therefore, it needs a destructor. Other resources that you might encounter as your programs increase in ambition and sophistication are files (if you open one, you also have to close it), locks, thread handles, and sockets (for communication with processes and remote computers).

Another sign that a class needs a destructor is simply that it has members that are pointers or references. If a class has a pointer or a reference member, it often needs a destructor and copy operations.

A class that needs a destructor almost always also needs a copy constructor and a copy assignment. The reason is simply that if an object has acquired a resource (and has a pointer member pointing to it), the default meaning of copy (shallow, memberwise copy) is almost certainly wrong. Again, **vector** is the classic example.

Similarly, a class that needs a destructor almost always also needs a move constructor and a move assignment. The reason is simply that if an object has acquired a resource (and has a pointer member pointing to it), the default meaning of copy (shallow, memberwise copy) is almost certainly wrong and the usual remedy (copy operations that duplicate the complete object state) can be expensive. Again, **vector** is the classic example.

In addition, a base class for which a derived class may have a destructor needs a **virtual** destructor (§17.5.2).

18.4.1 Explicit constructors

A constructor that takes a single argument defines a conversion from its argument type to its class. This can be most useful. For example:

```
class complex {
public:
        complex(double);           // defines double-to-complex conversion
        complex(double,double);
        // . . .
};

complex z1 = 3.14;                 // OK: convert 3.14 to (3.14,0)
complex z2 = complex{1.2, 3.4};
```

However, implicit conversions should be used sparingly and with caution, because they can cause unexpected and undesirable effects. For example, our **vector**, as defined so far, has a constructor that takes an **int**. This implies that it defines a conversion from **int** to **vector**. For example:

```
class vector {
        // . . .
        vector(int);
        // . . .
};
```

```
vector v = 10;            // odd: makes a vector of 10 doubles
v = 20;                   // eh? Assigns a new vector of 20 doubles to v

void f(const vector&);
f(10);                    // eh? Calls f with a new vector of 10 doubles
```

It seems we are getting more than we have bargained for. Fortunately, it is simple to suppress this use of a constructor as an implicit conversion. A constructor-defined **explicit** provides only the usual construction semantics and not the implicit conversions. For example:

```
class vector {
        // . . .
        explicit vector(int);
        // . . .
};

vector v = 10;            // error: no int-to-vector conversion
v = 20;                   // error: no int-to-vector conversion
vector v0(10);            // OK

void f(const vector&);
f(10);                    // error: no int-to-vector<double> conversion
f(vector(10));            // OK
```

To avoid surprising conversions, we – and the standard – define **vector**'s single-argument constructors to be **explicit**. It's a pity that constructors are not **explicit** by default; if in doubt, make any constructor that can be invoked with a single argument **explicit**.

18.4.2 Debugging constructors and destructors

Constructors and destructors are invoked at well-defined and predictable points of a program's execution. However, we don't always write **explicit** calls, such as **vector(2)**; rather we do something, such as declaring a **vector**, passing a **vector** as a by-value argument, or creating a **vector** on the free store using **new**. This can cause confusion for people who think in terms of syntax. There is not just a single syntax that triggers a constructor. It is simpler to think of constructors and destructors this way:

- Whenever an object of type **X** is created, one of **X**'s constructors is invoked.
- Whenever an object of type **X** is destroyed, **X**'s destructor is invoked.

A destructor is called whenever an object of its class is destroyed; that happens when names go out of scope, the program terminates, or **delete** is used on a pointer to an object. A constructor (some appropriate constructor) is invoked whenever an object of its class is created; that happens when a variable is initialized, when an object is created using **new** (except for built-in types), and whenever an object is copied.

But when does that happen? A good way to get a feel for that is to add print statements to constructors, assignment operations, and destructors and then just try. For example:

```
struct X {              // simple test class
    int val;

    void out(const string& s, int nv)
        { cerr << this << "–>" << s << ": " << val << " (" << nv << ")\n"; }

    X(){ out("X()",0); val=0; }                              // default constructor
    X(int v) { val=v; out( "X(int)",v); }
    X(const X& x){ val=x.val; out("X(X&) ",x.val); }         // copy constructor
    X& operator=(const X& a)                                 // copy assignment
        { out("X::operator=()",a.val); val=a.val; return *this; }
    ~X() { out("~X()",0); }                                  // destructor
};
```

Anything we do with this **X** will leave a trace that we can study. For example:

```
X glob(2);              // a global variable

X copy(X a) { return a; }

X copy2(X a) { X aa = a; return aa; }

X& ref_to(X& a) { return a; }

X* make(int i) { X a(i); return new X(a); }

struct XX { X a; X b; };

int main()
{
    X loc {4};          // local variable
    X loc2 {loc};       // copy construction
```

```
        loc = X{5};               // copy assignment
        loc2 = copy(loc);         // call by value and return
        loc2 = copy2(loc);
        X loc3 {6};
        X& r = ref_to(loc);       // call by reference and return
        delete make(7);
        delete make(8);
        vector<X> v(4);           // default values
        XX loc4;
        X* p = new X{9};          // an X on the free store
        delete p;
        X* pp = new X[5];         // an array of Xs on the free store
        delete[] pp;
}
```

Try executing that.

TRY THIS

We really mean it: do run this example and make sure you understand the result. If you do, you'll understand most of what there is to know about construction and destruction of objects.

Depending on the quality of your compiler, you may note some "missing copies" relating to our calls of **copy()** and **copy2()**. We (humans) can see that those functions do nothing: they just copy a value unmodified from input to output. If a compiler is smart enough to notice that, it is allowed to eliminate the calls to the copy constructor. In other words, a compiler is allowed to assume that a copy constructor copies and does nothing but copy. Some compilers are smart enough to eliminate many spurious copies. However, compilers are not guaranteed to be that smart, so if you want portable performance, consider move operations (§18.3.4).

Now consider: Why should we bother with this "silly class **X**"? It's a bit like the finger exercises that musicians have to do. After doing them, other things – things that matter – become easier. Also, if you have problems with constructors and destructors, you can insert such print statements in constructors for your real classes to see that they work as intended. For larger programs, this exact kind of tracing becomes tedious, but similar techniques apply. For example, you can determine whether you have a memory leak by seeing if the number of constructions minus the number of destructions equals zero. Forgetting to define copy constructors and copy assignments for classes that allocate memory or hold pointers to objects is a common – and easily avoidable – source of problems.

If your problems get too big to handle by such simple means, you will have learned enough to be able to start using the professional tools for finding such problems; they are often referred to as "leak detectors." The ideal, of course, is not to leak memory by using techniques that avoid such leaks.

18.5 Access to vector elements

So far (§17.6), we have used **set()** and **get()** member functions to access elements. Such uses are verbose and ugly. We want our usual subscript notation: **v[i]**. The way to get that is to define a member function called **operator[]**. Here is our first (naive) try:

```
class vector {
        int sz;                        // the size
        double* elem;                  // a pointer to the elements
public:
        // . . .
        double operator[](int n) { return elem[n]; }      // return element
};
```

That looks good and especially it looks simple, but unfortunately it is too simple. Letting the subscript operator (**operator[]()**) return a value enables reading but not writing of elements:

```
vector v(10);
double x = v[2];                       // fine
v[3] = x;                              // error: v[3] is not an lvalue
```

Here, **v[i]** is interpreted as a call **v.operator[](i)**, and that call returns the value of **v**'s element number **i**. For this overly naive **vector**, **v[3]** is a floating-point value, not a floating-point variable.

TRY THIS

Make a version of this **vector** that is complete enough to compile and see what error message your compiler produces for **v[3]=x;**.

Our next try is to let **operator[]** return a pointer to the appropriate element:

```
class vector {
        int sz;                        // the size
        double* elem;                  // a pointer to the elements
public:
        // . . .
        double* operator[](int n) { return &elem[n]; }       // return pointer
};
```

Given that definition, we can write

```
vector v(10);
for (int i=0; i<v.size(); ++i) {          // works, but still too ugly
        *v[i] = i;
        cout << *v[i];
}
```

Here, **v[i]** is interpreted as a call **v.operator[](i)**, and that call returns a pointer to **v**'s element number **i**. The problem is that we have to write ***** to dereference that pointer to get to the element. That's almost as bad as having to write **set()** and **get()**. Returning a reference from the subscript operator solves this problem:

```
class vector {
        // . . .
        double& operator[ ](int n) { return elem[n]; }    // return reference
};
```

Now we can write

```
vector v(10);
for (int i=0; i<v.size(); ++i) {          // works!
        v[i] = i;                           // v[i] returns a reference element i
        cout << v[i];
}
```

We have achieved the conventional notation: **v[i]** is interpreted as a call **v.operator[](i)**, and that returns a reference to **v**'s element number **i**.

18.5.1 Overloading on const

The **operator[]()** defined so far has a problem: it cannot be invoked for a **const vector**. For example:

```
void f(const vector& cv)
{
        double d = cv[1];                   // error, but should be fine
        cv[1] = 2.0;                        // error (as it should be)
}
```

The reason is that our **vector::operator[]()** could potentially change a **vector**. It doesn't, but the compiler doesn't know that because we "forgot" to tell it. The solution is to provide a version that is a **const** member function (see §9.7.4). That's easily done:

```
class vector {
        // . . .
        double& operator[](int n);          // for non-const vectors
        double operator[](int n) const;     // for const vectors
};
```

We obviously couldn't return a **double&** from the **const** version, so we returned a **double** value. We could equally well have returned a **const double&**, but since a **double** is a small object there would be no point in returning a reference (§8.5.6), so we decided to pass it back by value. We can now write

```
void ff(const vector& cv, vector& v)
{
        double d = cv[1];          // fine (uses the const [])
        cv[1] = 2.0;               // error (uses the const [])
        double d = v[1];           // fine (uses the non-const [])
        v[1] = 2.0;                // fine (uses the non-const [])
}
```

Since **vector**s are often passed by **const** reference, this **const** version of **operator[]()** is an essential addition.

18.6 Arrays

For a while, we have used *array* to refer to a sequence of objects allocated on the free store. We can also allocate arrays elsewhere as named variables. In fact, they are common

- As global variables (but global variables are most often a bad idea)
- As local variables (but arrays have serious limitations there)
- As function arguments (but an array doesn't know its own size)
- As class members (but member arrays can be hard to initialize)

Now, you might have detected that we have a not-so-subtle bias in favor of **vector**s over arrays. Use **std::vector** where you have a choice – and you have a choice in most contexts. However, arrays existed long before **vector**s and are roughly equivalent to what is offered in other languages (notably C), so you must know arrays, and know them well, to be able to cope with older code and with code written by people who don't appreciate the advantages of **vector**.

So, what is an array? How do we define an array? How do we use an array? An *array* is a homogeneous sequence of objects allocated in contiguous memory; that is, all elements of an array have the same type and there are no gaps between the objects of the sequence. The elements of an array are numbered from 0 upward. In a declaration, an array is indicated by "square brackets":

```
const int max = 100;
int gai[max];              // a global array (of 100 ints); "lives forever"

void f(int n)
{
        char lac[20];      // local array; "lives" until the end of scope
        int lai[60];
        double lad[n];     // error: array size not a constant
        // . . .
}
```

Note the limitation: the number of elements of a named array must be known at compile time. If you want the number of elements to be a variable, you must put it on the free store and access it through a pointer. That's what **vector** does with its array of elements.

Just like the arrays on the free store, we access named arrays using the subscript and dereference operators (**[ ]** and *****). For example:

```
void f2()
{
        char lac[20];      // local array; "lives" until the end of scope

        lac[7] = 'a';
        *lac = 'b';        // equivalent to lac[0]='b'

        lac[–2] = 'b';     // huh?
        lac[200] = 'c';    // huh?
}
```

This function compiles, but we know that "compiles" doesn't mean "works correctly." The use of **[ ]** is obvious, but there is no range checking, so **f2()** compiles, and the result of writing to **lac[–2]** and **lac[200]** is (as for all out-of-range access) usually disastrous. Don't do it. Arrays do not range check. Again, we are dealing directly with physical memory here; don't expect "system support."

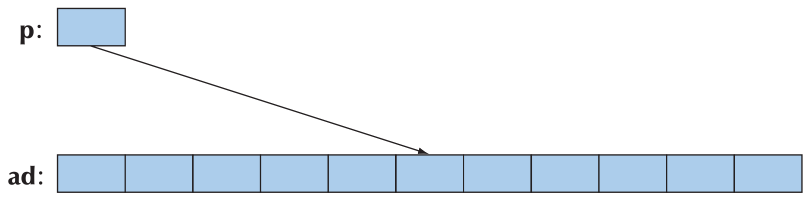
But couldn't the compiler see that **lac** has just 20 elements so that **lac[200]** is an error? A compiler could, but as far as we know no production compiler does. The problem is that keeping track of array bounds at compile time is impossible in general, and catching errors in the simplest cases (like the one above) only is not very helpful.

18.6.1 Pointers to array elements

A pointer can point to an element of an array. Consider:

```
double ad[10];
double* p = &ad[5];          // point to ad[5]
```

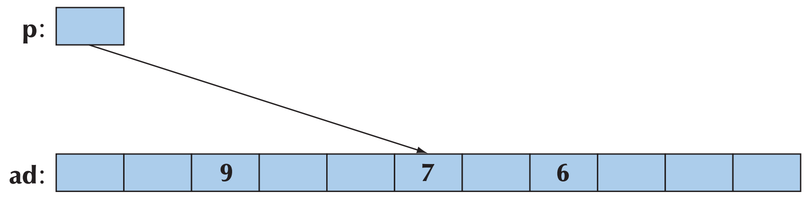
We now have a pointer **p** to the **double** known as **ad[5]**:



We can subscript and dereference that pointer:

```
*p =7;
p[2] = 6;
p[–3] = 9;
```

We get

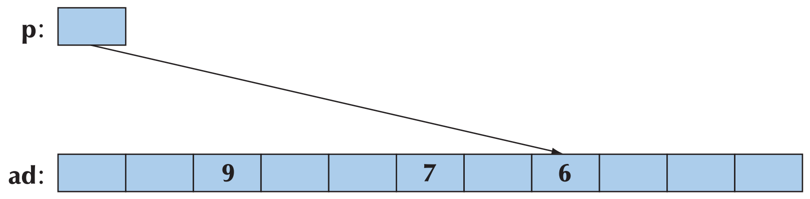


That is, we can subscript the pointer with both positive and negative numbers. As long as the resulting element is in range, all is well. However, access outside the range of the array pointed into is illegal (as with free-store-allocated arrays; see §17.4.3). Typically, access outside an array is not detected by the compiler and (sooner or later) is disastrous.

Once a pointer points into an array, addition and subscripting can be used to make it point to another element of the array. For example:

```
p += 2;              // move p 2 elements to the right
```

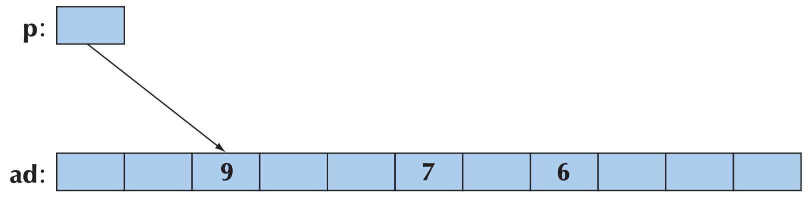
We get



And

```
p –= 5;              // move p 5 elements to the left
```

We get



Using **+**, **–**, **+=**, and **–=** to move pointers around is called *pointer arithmetic*. Obviously, if we do that, we have to take great care to ensure that the result is not a pointer to memory outside the array:

```
p += 1000;           // insane: p points into an array with just 10 elements
double d = *p;       // illegal: probably a bad value
                     // (definitely an unpredictable value)
*p = 12.34;          // illegal: probably scrambles some unknown data
```

Unfortunately, not all bad bugs involving pointer arithmetic are that easy to spot. The best policy is usually simply to avoid pointer arithmetic.

The most common use of pointer arithmetic is incrementing a pointer (using **++**) to point to the next element and decrementing a pointer (using **––**) to point to the previous element. For example, we could print the value of **ad**'s elements like this:

```
for (double* p = &ad[0]; p<&ad[10]; ++p) cout << *p << '\n';
```

Or backward:

```
for (double* p = &ad[9]; p>=&ad[0]; −−p) cout << *p << '\n';
```

This use of pointer arithmetic is not uncommon. However, we find the last ("backward") example quite easy to get wrong. Why **&ad[9]** and not **&ad[10]**? Why **>=** and not **>**? These examples could equally well (and equally efficiently) be done using subscripting. Such examples could be done equally well using subscripting into a **vector**, which is more easily range checked.

Note that most real-world uses of pointer arithmetic involve a pointer passed as a function argument. In that case, the compiler doesn't have a clue how many elements are in the array pointed into: you are on your own. That is a situation we prefer to stay away from whenever we can.

Why does C++ have (allow) pointer arithmetic at all? It can be such a bother and doesn't provide anything new once we have subscripting. For example:

```
double* p1 = &ad[0];
double* p2 = p1+7;
double* p3 = &p1[7];
if (p2 != p3) cout << "impossible!\n";
```

Mainly, the reason is historical. These rules were crafted for C decades ago and can't be removed without breaking a lot of code. Partly, there can be some convenience gained by using pointer arithmetic in some important low-level applications, such as memory managers.

18.6.2 Pointers and arrays

The name of an array refers to all the elements of the array. Consider:

```
char ch[100];
```

The size of **ch**, **sizeof(ch)**, is 100. However, the name of an array turns into ("decays to") a pointer with the slightest excuse. For example:

```
char* p = ch;
```

Here **p** is initialized to **&ch[0]** and **sizeof(p)** is something like 4 (not 100).

This can be useful. For example, consider a function **strlen()** that counts the number of characters in a zero-terminated array of characters:

```
int strlen(const char* p)      // similar to the standard library strlen()
{
        int count = 0;
        while (*p) { ++count; ++p; }
        return count;
}
```

We can now call this with **strlen(ch)** as well as **strlen(&ch[0])**. You might point out that this is a very minor notational advantage, and we'd have to agree.

One reason for having array names convert to pointers is to avoid accidentally passing large amounts of data by value. Consider:

```
int strlen(const char a[ ])      // similar to the standard library strlen()
{
        int count = 0;
        while (a[count]) { ++count; }
        return count;
}

char lots [100000];

void f()
{
        int nchar = strlen(lots);
        // . . .
}
```

Naively (and quite reasonably), you might expect this call to copy the 100,000 characters specified as the argument to **strlen()**, but that's not what happens. Instead, the argument declaration **char p[ ]** is considered equivalent to **char* p**, and the call **strlen(lots)** is considered equivalent to **strlen(&lots[0])**. This saves you from an expensive copy operation, but it should surprise you. Why should it surprise you? Because in every other case, when you pass an object and don't explicitly declare an argument to be passed by reference (§8.5.3–6), that object is copied.

Note that the pointer you get from treating the name of an array as a pointer to its first element is a value and not a variable, so you cannot assign to it:

```
char ac[10];
ac = new char [20];          // error: no assignment to array name
&ac[0] = new char [20];      // error: no assignment to pointer value
```

Finally! A problem that the compiler will catch!

As a consequence of this implicit array-name-to-pointer conversion, you can't even copy arrays using assignment:

```
int x[100];
int y[100];
// . . .
x = y;                          // error
int z[100] = y;                 // error
```

This is consistent, but often a bother. If you need to copy an array, you must write some more elaborate code to do so. For example:

```
for (int i=0; i<100; ++i) x[i]=y[i];     // copy 100 ints
memcpy(x,y,100*sizeof(int));             // copy 100*sizeof(int) bytes
copy(y,y+100, x);                        // copy 100 ints
```

Note that the C language doesn't support anything like **vector**, so in C, you must use arrays extensively. This implies that a lot of C++ code uses arrays (§27.1.2). In particular, C-style strings (zero-terminated arrays of characters; see §27.5) are very common.

If we want assignment, we have to use something like the standard library **vector**. The **vector** equivalent to the copying code above is

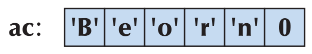
```
vector<int> x(100);
vector<int> y(100);
// . . .
x = y;                          // copy 100 ints
```

18.6.3 Array initialization

An array of **char**s can be initialized with a string literal. For example:

```
char ac[] = "Beorn";            // array of 6 chars
```

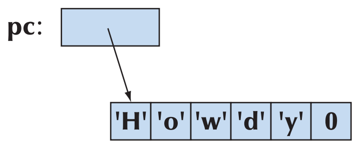
Count those characters. There are five, but **ac** becomes an array of six characters because the compiler adds a terminating zero character at the end of a string literal:



A zero-terminated string is the norm in C and many systems. We call such a zero-terminated array of characters a *C-style string*. All string literals are C-style strings. For example:

```
char* pc = "Howdy";                    // pc points to an array of 6 chars
```

Graphically:



Note that the **char** with the numeric value **0** is not the character **'0'** or any other letter or digit. The purpose of that terminating zero is to allow functions to find the end of the string. Remember: An array does not know its size. Relying on the terminating zero convention, we can write

```
int strlen(const char* p)              // similar to the standard library strlen()
{
        int n = 0;
        while (p[n]) ++n;
        return n;
}
```

Actually, we don't have to define **strlen()** because it is a standard library function defined in the **<string.h>** header (§27.5, §B.11.3). Note that **strlen()** counts the characters, but not the terminating **0**; that is, you need *n*+1 **char**s to store *n* characters in a C-style string.

Only character arrays can be initialized by literal strings, but all arrays can be initialized by a list of values of their element type. For example:

```
int ai[] = { 1, 2, 3, 4, 5, 6 };       // array of 6 ints
int ai2[100] = {0,1,2,3,4,5,6,7,8,9};  // the last 90 elements are initialized to 0
double ad[100] = { };                  // all elements initialized to 0.0
char chars[] = {'a', 'b', 'c'};        // no terminating 0!
```

Note that the number of elements of **ai** is six (not seven) and the number of elements for **chars** is three (not four) – the "add a **0** at the end" rule is for literal character strings only. If an array isn't given a size, that size is deduced from the initializer list. That's a rather useful feature. If there are fewer initializer values than array elements (as in the definitions of **ai2** and **ad**), the remaining elements are initialized by the element type's default value.

18.6.4 Pointer problems

Like arrays, pointers are often overused and misused. Often, the problems people get themselves into involve both pointers and arrays, so we'll summarize the problems here. In particular, all serious problems with pointers involve trying to access something that isn't an object of the expected type, and many of those problems involve access outside the bounds of an array. Here we will consider

- Access through the null pointer
- Access through an uninitialized pointer
- Access off the end of an array
- Access to a deallocated object
- Access to an object that has gone out of scope

In all cases, the practical problem for the programmer is that the actual access looks perfectly innocent; it is "just" that the pointer hasn't been given a value that makes the use valid. Worse (in the case of a write through the pointer), the problem may manifest itself only a long time later when some apparently unrelated object has been corrupted. Let's consider examples:

Don't access through the null pointer:

```
int* p = nullptr;
*p = 7;          // ouch!
```

Obviously, in real-world programs, this typically occurs when there is some code in between the initialization and the use. In particular, passing **p** to a function and receiving it as the result from a function are common examples. We prefer not to pass null pointers around, but if you have to, test for the null pointer before use:

```
int* p = fct_that_can_return_a_nullptr();

if (p == nullptr) {
        // do something
}
else {
        // use p
        *p = 7;
}
```

and

```
void fct_that_can_receive_a_nullptr(int* p)
{
        if (p == nullptr) {
                // do something
        }
        else {
                // use p
                *p = 7;
        }
}
```

Using references (§17.9.1) and using exceptions to signal errors (§5.6 and §19.5) are the main tools for avoiding null pointers.

Do initialize your pointers:

```
int* p;
*p = 9;          // ouch!
```

In particular, don't forget to initialize pointers that are class members.

Don't access nonexistent array elements:

```
int a[10];
int* p = &a[10];
*p = 11;         // ouch!
a[10] = 12;      // ouch!
```

Be careful with the first and last elements of a loop, and try not to pass arrays around as pointers to their first elements. Instead use **vector**s. If you really must use an array in more than one function (passing it as an argument), then be extra careful and pass its size along.

Don't access through a deleted pointer:

```
int* p = new int{7};
// . . .
delete p;
// . . .
*p = 13;         // ouch!
```

The **delete p** or the code after it may have scribbled all over ***p** or used it for something else. Of all of these problems, we consider this one the hardest to systematically avoid. The most effective defense against this problem is not to have "naked" **news** that require "naked" **deletes**: use **new** and **delete** in constructors and destructors or use a container, such as **Vector_ref** (§E.4), to handle **deletes**.

Don't return a pointer to a local variable:

```
int* f()
{
        int x = 7;
        // . . .
        return &x;
}

// . . .

int* p = f();
// . . .
*p = 15;                // ouch!
```

The return from **f()** or the code after it may have scribbled all over ***p** or used it for something else. The reason for that is that the local variables of a function are allocated (on the stack) upon entry to the function and deallocated again at the exit from the function. In particular, destructors are called for local variables of classes with destructors (§17.5.1). Compilers could catch most problems related to returning pointers to local variables, but few do.

Consider a logically equivalent example:

```
vector& ff()
{
        vector x(7);    // 7 elements
        // . . .
        return x;
}     // the vector x is destroyed here

// . . .

vector& p = ff();
// . . .
p[4] = 15;              // ouch!
```

Quite a few compilers catch this variant of the return problem.

It is common for programmers to underestimate these problems. However, many experienced programmers have been defeated by the innumerable variations and combinations of these simple array and pointer problems. The solution is not to litter your code with pointers, arrays, **new**s, and **delete**s. If you do, "being careful" simply isn't enough in realistically sized programs. Instead, rely on vectors, RAII ("Resource Acquisition Is Initialization"; see §19.5), and other systematic approaches to the management of memory and other resources.

18.7 Examples: palindrome

Enough technical examples! Let's try a little puzzle. A *palindrome* is a word that is spelled the same from both ends. For example, *anna*, *petep*, and *malayalam* are palindromes, whereas *ida* and *homesick* are not. There are two basic ways of determining whether a word is a palindrome:

- Make a copy of the letters in reverse order and compare that copy to the original.
- See if the first letter is the same as the last, then see if the second letter is the same as the second to last, and keep going until you reach the middle.

Here, we'll take the second approach. There are many ways of expressing this idea in code depending on how we represent the word and how we keep track of how far we have come with the comparison of characters. We'll write a little program that tests whether words are palindromes in a few different ways just to see how different language features affect the way the code looks and works.

18.7.1 Palindromes using string

First, we try a version using the standard library **string** with **int** indices to keep track of how far we have come with our comparison:

```
bool is_palindrome(const string& s)
{
        int first = 0;                  // index of first letter
        int last = s.length()–1;        // index of last letter
        while (first < last) {          // we haven't reached the middle
                if (s[first]!=s[last]) return false;
                ++first;                // move forward
                ––last;                 // move backward
        }
        return true;
}
```

We return **true** if we reach the middle without finding a difference. We suggest that you look at this code to convince yourself that it is correct when there are no letters in the string, just one letter in the string, an even number of letters in the string, and an odd number of letters in the string. Of course, we should not just rely on logic to see that our code is correct. We should also test. We can exercise **is_palindrome()** like this:

```
int main()
{
        for (string s; cin>>s; ) {
                cout << s << " is";
                if (!is_palindrome(s)) cout << " not";
                cout << " a palindrome\n";
        }
}
```

Basically, the reason we are using a **string** is that "**string**s are good for dealing with words." It is simple to read a whitespace-separated word into a string, and a **string** knows its size. Had we wanted to test **is_palindrome()** with strings containing whitespace, we could have read using **getline()** (§11.5). That would have shown *ah ha* and *as df fd sa* to be palindromes.

18.7.2 Palindromes using arrays

What if we didn't have **string**s (or **vector**s), so that we had to use an array to store the characters? Let's see:

```
bool is_palindrome(const char s[], int n)
        // s points to the first character of an array of n characters
{
        int first = 0;                  // index of first letter
        int last = n–1;                 // index of last letter
        while (first < last) {          // we haven't reached the middle
                if (s[first]!=s[last]) return false;
                ++first;                // move forward
                ––last;                 // move backward
        }
        return true;
}
```

To exercise **is_palindrome()**, we first have to get characters read into the array. One way to do that safely (i.e., without risk of overflowing the array) is like this:

```
istream& read_word(istream& is, char* buffer, int max)
        // read at most max–1 characters from is into buffer
```

```
{
        is.width(max);          // read at most max–1 characters in the next >>
        is >> buffer;           // read whitespace-terminated word,
                                // add zero after the last character read into buffer
        return is;
}
```

Setting the **istream**'s width appropriately prevents buffer overflow for the next **>>** operation. Unfortunately, it also means that we don't know if the read terminated by whitespace or by the buffer being full (so that we need to read more characters). Also, who remembers the details of the behavior of **width()** for input? The standard library **string** and **vector** are really better as input buffers because they expand to fit the amount of input. The terminating **0** character is needed because most popular operations on arrays of characters (C-style strings) assume 0 termination. Using **read_word()** we can write

```
int main()
{
        constexpr int max = 128;
        for (char s[max]; read_word(cin,s,max); ) {
                cout << s << " is";
                if (!is_palindrome(s,strlen(s))) cout << " not";
                cout << " a palindrome\n";
        }
}
```

The **strlen(s)** call returns the number of characters in the array after the call of **read_word()**, and **cout<<s** outputs the characters in the array up to the terminating **0**.

We consider this "array solution" significantly messier than the "**string** solution," and it gets much worse if we try to seriously deal with the possibility of long strings. See exercise 10.

18.7.3 Palindromes using pointers

Instead of using indices to identify characters, we could use pointers:

```
bool is_palindrome(const char* first, const char* last)
        // first points to the first letter, last to the last letter
{
        while (first < last) {          // we haven't reached the middle
                if (*first!=*last) return false;
                ++first;                // move forward
                ––last;                 // move backward
        }
        return true;
}
```

Note that we can actually increment and decrement pointers. Increment makes a pointer point to the next element of an array and decrement makes a pointer point to the previous element. If the array doesn't have such a next element or previous element, you have a serious uncaught out-of-range error. That's another problem with pointers.

We call this **is_palindrome()** like this:

```
int main()
{
        const int max = 128;
        for (char s[max]; read_word(cin,s,max); ) {
                cout << s << " is";
                if (!is_palindrome(&s[0],&s[strlen(s)–1])) cout << " not";
                cout << " a palindrome\n";
        }
}
```

Just for fun, we rewrite **is_palindrome()** like this:

```
bool is_palindrome(const char* first, const char* last)
        // first points to the first letter, last to the last letter
{
        if (first<last) {
                if (*first!=*last) return false;
                return is_palindrome(first+1,last–1);
        }
        return true;
}
```

This code becomes obvious when we rephrase the definition of *palindrome*: a word is a palindrome if the first and the last characters are the same and if the substring you get by removing the first and the last characters is a palindrome.

Drill

In this chapter, we have two drills: one to exercise arrays and one to exercise **vector**s in roughly the same manner. Do both and compare the effort involved in each.

Array drill:

1. Define a global **int** array **ga** of ten **int**s initialized to 1, 2, 4, 8, 16, etc.
2. Define a function **f()** taking an **int** array argument and an **int** argument indicating the number of elements in the array.
3. In **f()**:
   a. Define a local **int** array **la** of ten **int**s.
   b. Copy the values from **ga** into **la**.
   c. Print out the elements of **la**.
   d. Define a pointer **p** to **int** and initialize it with an array allocated on the free store with the same number of elements as the argument array.
   e. Copy the values from the argument array into the free-store array.
   f. Print out the elements of the free-store array.
   g. Deallocate the free-store array.
4. In **main()**:
   a. Call **f()** with **ga** as its argument.
   b. Define an array **aa** with ten elements, and initialize it with the first ten factorial values (1, 2*1, 3*2*1, 4*3*2*1, etc.).
   c. Call **f()** with **aa** as its argument.

Standard library vector drill:

1. Define a global **vector<int> gv**; initialize it with ten **int**s, 1, 2, 4, 8, 16, etc.
2. Define a function **f()** taking a **vector<int>** argument.
3. In **f()**:
   a. Define a local **vector<int> lv** with the same number of elements as the argument **vector**.
   b. Copy the values from **gv** into **lv**.
   c. Print out the elements of **lv**.
   d. Define a local **vector<int> lv2**; initialize it to be a copy of the argument **vector**.
   e. Print out the elements of **lv2**.
4. In **main()**:
   a. Call **f()** with **gv** as its argument.
   b. Define a **vector<int> vv**, and initialize it with the first ten factorial values (1, 2*1, 3*2*1, 4*3*2*1, etc.).
   c. Call **f()** with **vv** as its argument.

Review

1. What does "Caveat emptor!" mean?
2. What is the default meaning of copying for class objects?
3. When is the default meaning of copying of class objects appropriate? When is it inappropriate?
4. What is a copy constructor?
5. What is a copy assignment?
6. What is the difference between copy assignment and copy initialization?
7. What is shallow copy? What is deep copy?
8. How does the copy of a **vector** compare to its source?
9. What are the five "essential operations" for a class?
10. What is an **explicit** constructor? Where would you prefer one over the (default) alternative?
11. What operations may be invoked implicitly for a class object?
12. What is an array?
13. How do you copy an array?
14. How do you initialize an array?
15. When should you prefer a pointer argument over a reference argument? Why?
16. What is a C-style string?
17. What is a palindrome?

Terms

array	deep copy	move assignment
array initialization	default constructor	move construction
copy assignment	essential operations	palindrome
copy constructor	**explicit** constructor	shallow copy

Exercises

1. Write a function, **char* strdup(const char*)**, that copies a C-style string into memory it allocates on the free store. Do not use any standard library functions. Do not use subscripting; use the dereference operator ***** instead.
2. Write a function, **char* findx(const char* s, const char* x)**, that finds the first occurrence of the C-style string **x** in **s**. Do not use any standard library functions. Do not use subscripting; use the dereference operator ***** instead.
3. Write a function, **int strcmp(const char* s1, const char* s2)**, that compares C-style strings. Let it return a negative number if **s1** is lexicographically

before **s2**, zero if **s1** equals **s2**, and a positive number if **s1** is lexicographically after **s2**. Do not use any standard library functions. Do not use subscripting; use the dereference operator ***** instead.

4. Consider what happens if you give **strdup()**, **findx()**, and **strcmp()** an argument that is not a C-style string. Try it! First figure out how to get a **char*** that doesn't point to a zero-terminated array of characters and then use it (never do this in real – non-experimental – code; it can create havoc). Try it with free-store-allocated and stack-allocated "fake C-style strings." If the results still look reasonable, turn off debug mode. Redesign and re-implement those three functions so that they take another argument giving the maximum number of elements allowed in argument strings. Then, test that with correct C-style strings and "bad" strings.
5. Write a function, **string cat_dot(const string& s1, const string& s2)**, that concatenates two strings with a dot in between. For example, **cat_dot("Niels", "Bohr")** will return a string containing **Niels.Bohr**.
6. Modify **cat_dot()** from the previous exercise to take a string to be used as the separator (rather than dot) as its third argument.
7. Write versions of the **cat_dot()**s from the previous exercises to take C-style strings as arguments and return a free-store-allocated C-style string as the result. Do not use standard library functions or types in the implementation. Test these functions with several strings. Be sure to free (using **delete**) all the memory you allocated from free store (using **new**). Compare the effort involved in this exercise with the effort involved for exercises 5 and 6.
8. Rewrite all the functions in §18.7 to use the approach of making a backward copy of the string and then comparing; for example, take **"home"**, generate **"emoh"**, and compare those two strings to see that they are different, so *home* isn't a palindrome.
9. Consider the memory layout in §17.4. Write a program that tells the order in which static storage, the stack, and the free store are laid out in memory. In which direction does the stack grow: upward toward higher addresses or downward toward lower addresses? In an array on the free store, are elements with higher indices allocated at higher or lower addresses?
10. Look at the "array solution" to the palindrome problem in §18.7.2. Fix it to deal with long strings by (a) reporting if an input string was too long and (b) allowing an arbitrarily long string. Comment on the complexity of the two versions.
11. Look up (e.g., on the web) *skip list* and implement that kind of list. This is not an easy exercise.
12. Implement a version of the game "Hunt the Wumpus." "Hunt the Wumpus" (or just "Wump") is a simple (non-graphical) computer game originally invented by Gregory Yob. The basic premise is that a rather smelly

monster lives in a dark cave consisting of connected rooms. Your job is to slay the wumpus using bow and arrow. In addition to the wumpus, the cave has two hazards: bottomless pits and giant bats. If you enter a room with a bottomless pit, it's the end of the game for you. If you enter a room with a bat, the bat picks you up and drops you into another room. If you enter the room with the wumpus or he enters yours, he eats you. When you enter a room you will be told if a hazard is nearby:

"I smell the wumpus": It's in an adjoining room.

"I feel a breeze": One of the adjoining rooms is a bottomless pit.

"I hear a bat": A giant bat is in an adjoining room.

For your convenience, rooms are numbered. Every room is connected by tunnels to three other rooms. When entering a room, you are told something like "You are in room 12; there are tunnels to rooms 1, 13, and 4; move or shoot?" Possible answers are **m13** ("Move to room 13") and **s13–4–3** ("Shoot an arrow through rooms 13, 4, and 3"). The range of an arrow is three rooms. At the start of the game, you have five arrows. The snag about shooting is that it wakes up the wumpus and he moves to a room adjoining the one he was in – that could be your room.

Probably the trickiest part of the exercise is to make the cave by selecting which rooms are connected with which other rooms. You'll probably want to use a random number generator (e.g., **randint()** from **std_lib_facilities.h**) to make different runs of the program use different caves and to move around the bats and the wumpus. Hint: Be sure to have a way to produce a debug output of the state of the cave.

Postscript

The standard library **vector** is built from lower-level memory management facilities, such as pointers and arrays, and its primary role is to help us avoid the complexities of those facilities. Whenever we design a class, we must consider initialization, copying, and destruction.