15

Vector and Free Store

Use **vector** *as the default!*
– Alex Stepanov

This chapter and the next five describe the containers and algorithms part of the C++ standard library, traditionally called the STL. We describe the key facilities from the STL and some of their uses. In addition, we present the key design and programming techniques used to implement the STL and some low-level language features used for that. Among those are pointers, arrays, and free store. The focus of this chapter and the next three is the design and implementation of the most common and most useful STL container: **vector**.

§15.1 Introduction
§15.2 **vector** basics
§15.3 Memory, addresses, and pointers
The **sizeof** operator
§15.4 Free store and pointers
Free-store allocation; Access through pointers; Initialization; The null pointer; Free-store deallocation
§15.5 Destructors
Generated destructors; Virtual destructors
§15.6 Access to elements
§15.7 An example: lists
List operations; List use
§15.8 The **this** pointer
More link use

15.1 Introduction

CC The most useful container in the C++ standard library is **vector**. A **vector** provides a sequence of elements of a given type. You can refer to an element by its index (subscript), extend the **vector** by using **push_back()** or **resize()**, ask a **vector** for the number of its elements using **size()**, and have access to the **vector** checked against attempts to access out-of-range elements. The standard library **vector** is a convenient, flexible, efficient (in time and space), statically type-safe container of elements. The standard **string** has similar properties, as have other useful standard container types, such as **list** and **map**, which we describe in Chapter 20. However, a computer's memory doesn't directly support such useful types. All that the hardware *directly* supports is sequences of bytes. For example, for a **vector<double>v**, the operation **v.push_back(2.3)** adds **2.3** to a sequence of **double**s and increases the element count of **v** (**v.size()**) by 1. At the lowest level, the computer knows nothing about anything as sophisticated as **push_back()**; all it knows is how to read and write a few bytes at a time.

In this and the following three chapters, we show how to build a **Vector** from the basic language facilities available to every programmer. Gradually, our **Vector** approximates the standard-library **vector**. This approach allows us to illustrate useful concepts and programming techniques, and to see how they are expressed using C++ language features. The language facilities and programming techniques we encounter in our **Vector** implementation are generally useful and very widely used.

Once we have seen how **vector** is designed, implemented, and used, we can proceed to look at other standard library containers, such as **map**, and examine the elegant and efficient facilities for their use provided by the C++ standard library (Chapter 20 and Chapter 21). These facilities save us from programming common tasks involving data ourselves. Instead, we can use what is available as part of every C++ implementation to ease the writing and testing of our code.

AA We approach the standard library **vector** through a series of increasingly sophisticated **Vector** implementations. First, we build an extremely simple **Vector**. Then, we see what's undesirable about that **Vector** and fix it. Please don't confuse the versions of **Vector** with the real **vector**. These versions are simplified to address a single issue at a time and therefore flawed. Be happy when you discover a flaw before we get around to mention and fix it in a later version. Spotting a flaw means that you are likely to recognize it when you see it in your own code or in that of others. When we have improved the seriously flawed early versions a few times, we reach a **Vector** that approximates the standard-library **vector** – shipped with your C++ compiler, the one that you have been using in the previous chapters. This process of gradual refinement mirrors the way we typically approach a new programming task. Along the way, we encounter and explore many classical problems related to the use of memory and data structures. The basic plan is this:

- *Chapter 15 (this chapter)*: How can we deal with varying amounts of memory? In particular, how can different **vector**s have different numbers of elements? This leads us to examine free store (also called *heap* and *dynamic memory*) and pointers. We introduce the crucial notion of a destructor.
- *Chapter 16*: We introduce arrays and explore their relation to pointers. We discuss the relationship between pointers and references. We present C-style strings. We offer **span**, **array**, and **string** as alternatives to the use of low-level constructs.
- *Chapter 17*: How can we copy **vector**s? How can we provide a subscript operation for them? How can we change the size of a **vector**? We introduce the notion of a set of

essential operations that a class needs for its lifetime and resources to be managed.

- *Chapter 18*: How can we have **vector**s with different element types? And how can we deal with out-of-range errors? To answer those questions, we explore the C++ template and exception facilities. We present a **Vector** that approximates the standard-library **vector** and also the resource-management pointers **unique_ptr** and **shared_ptr**.

In addition to the new language facilities and techniques that we introduce to handle the implementation of a flexible, efficient, and type-safe vector, we also use many of the language facilities and programming techniques we have already seen. Occasionally, we take the opportunity to give those a slightly more formal and technical definition.

So, this is the point at which we finally get to deal directly with memory. Why do we have to? Our **vector** and **string** are extremely useful and convenient; we can just use those. After all, containers, such as **vector** and **string**, are designed to insulate us from some of the unpleasant aspects of real memory. However, unless we are content to believe in magic, we must examine the lowest level of memory management. Why shouldn't you "just believe in magic"? Or – to put a more positive spin on it – why shouldn't you "just trust that the implementers of **vector** knew what they were doing"? After all, we don't suggest that you examine the device physics that allow our computer's memory to function. If so, we could skip right to Chapter 19.

CC

Well, we are programmers (computer scientists, software developers, or whatever) rather than physicists. Had we been studying device physics, we would have had to look into the details of computer memory design. However, since we are studying programming, we must look into the detailed design of programs. In theory, we could consider the low-level memory access and management facilities "implementation details" just as we do the device physics. However, if we did that, we would not just have to "believe in magic"; we would be unable to implement a new container (should we need one, and that's not uncommon). Also, we would be unable to read huge amounts of C and older C++ code that directly uses memory. As we will see over the next few chapters, pointers (a low-level and direct way of referring to an object) are also useful for a variety of reasons not related to memory management. It is not easy to use C++ well without sometimes using pointers.

XX

More philosophically, I am among the large group of computer professionals who are of the opinion that if you lack a basic and practical understanding of how a program maps onto a computer's memory and operations, you will have problems getting a solid grasp of higher-level topics, such as data structures, algorithms, and operating systems.

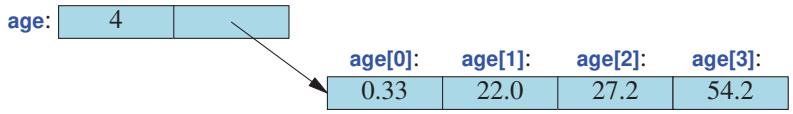
15.2 vector basics

We start our incremental design of our **Vector** by considering a very simple use of **vector**:

```
vector<double> age(4);          // a vector with 4 elements of type double
age[0]=0.33;
age[1]=22.0;
age[2]=27.2;
age[3]=54.2;
```

This creates a **vector** with four elements of type **double** and gives those four elements the values **0.33**, **22.0**, **27.2**, and **54.2**. The four elements are numbered 0, 1, 2, 3. The numbering of elements in

C++ standard library containers always starts from 0 (zero). Numbering from 0 is very common, and it is a universal convention among C++ programmers. The number of elements of a **vector** is called its size. So, the size of **age** is 4. The elements of a **vector** are numbered (indexed) from 0 to size-1. For example, the elements of **age** are numbered **0** to **age.size()–1**. We can represent **age** graphically like this:

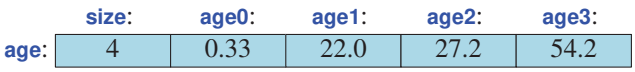


How do we make this "graphical design" real in a computer's memory? How do we get the values stored and accessed like that? Obviously, we have to define a class and we want to call this class **Vector**. Furthermore, it needs a data member to hold its size and one to hold its elements. But how do we represent a set of elements where the number of elements can vary? We could use a standard library **vector**, but that would – in this context – be cheating: we are building our own **Vector** here.

So, how do we represent that arrow in the drawing above? Consider doing without it. We could define a fixed-size data structure:

```
class Vector {
    int size;
    double age0, age1, age2, age3;
};
```

Ignoring some notational details, we'll have something like this:



CC That's simple and nice, but the first time we try to add an element with **push_back()** we are sunk: we have no way of adding an element; the number of elements is fixed to four in the program text. We need something more than a data structure holding a fixed number of elements. Operations that change the number of elements of a **vector**, such as **push_back()**, can't be implemented if we defined our **Vector** to have a fixed number of elements. Basically, we need a data member that points to the set of elements so that we can make it point to a different set of elements when we need more space. We need something like the memory address of the first element. In C++, a data type that can hold an address is called a *pointer* and is syntactically distinguished by the suffix ∗, so that **double∗** means "pointer to **double**." Given that, we can define our first version of a **Vector** class:

```
class Vector {          // a very simplified vector of doubles (like vector<double>)
    int sz;                     // the size
    double* elem;               // pointer to the first element (of type double)
public:
    Vector(int s);              // constructor: allocate s doubles,
                                // let elem point to them and sz hold the size (s)
    int size() const { return sz; }  // the current size
};
```

Before proceeding with the Vector design, let us study the notion of "pointer" in some detail. The notion of "pointer" – together with its closely related notion of "array" – is key to C++'s notion of "memory" (§16.1).

15.3 Memory, addresses, and pointers

CC

A computer's memory is a sequence of bytes. We can number the bytes from 0 to the last one. We call such "a number that indicates a location in memory" an *address*. You can think of an address as a kind of integer value. The first byte of memory has the address 0, the next the address 1, and so on. We can visualize a megabyte of memory like this:

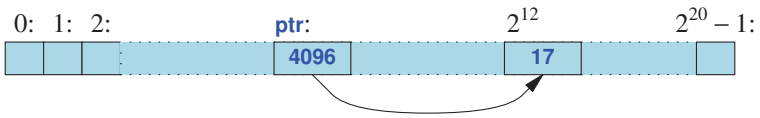
Everything we put in memory has an address. For example:

```
int var = 17;
```

This will set aside an "int-sized" piece of memory for var somewhere and put the value 17 into that memory. We can also store and manipulate addresses. An object that holds an address value is called a *pointer*. For example, the type needed to hold the address of an int is called a "pointer to int" or an "int pointer" and the notation is int*:

```
int* ptr = &var;      // ptr holds the address of var
```

The "address of" operator, unary &, is used to get the address of an object. So, if var happens to start at address 4096 (also known as 2^{12}), ptr will hold the value 4096:



Basically, we view our computer's memory as a sequence of bytes numbered from 0 to the memory size minus 1. On some machines that's a simplification, but as an initial programming model of the memory, it will suffice.

Each type has a corresponding pointer type. For example:

```
int x = 17;
int* pi = &x;          // pointer to int

double e = 2.71828;
double* pd = &e;       // pointer to double
```

If we want to see the value of the object pointed to, we can do that using the "contents of" operator, unary *. For example:

```
cout << "pi == " << pi << "; contents of pi == " << *pi << "\n";
cout << "pd == " << pd << "; contents of pd == " << *pd << "\n";
```

The output for *pi will be the integer 17 and the output for *pd will be the double 2.71828. The output for pi and pd will vary depending on where the compiler allocated our variables x and e in memory. The notation used for the pointer value (address) may also vary depending on which conventions your system uses.

The *contents of* operator (often called the *dereference* operator) can also be used on the left-hand side of an assignment:

```
*pi = 27;          // OK: you can assign 27 to the int pointed to by pi
*pd = 3.14159;     // OK: you can assign 3.14159 to the double pointed to by pd
*pd = *pi;         // OK: you can assign an int (*pi) to a double (*pd)
```

XX Note that even though a pointer value can be printed as an integer, a pointer is not an integer. "What does an int point to?" is not a well-formed question; ints do not point, pointers do. A pointer type provides the operations suitable for addresses, whereas int provides the arithmetic operations suitable for integers. So pointers and integers do not implicitly mix:

```
int i = pi;        // error: can't assign an int* to an int
pi = 7;            // error: can't assign an int to an int*
```

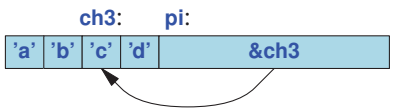
Similarly, a pointer to char (a char*) is not a pointer to int (an int*). For example:

```
char* pc = pi;     // error: can't assign an int* to a char*
pi = pc;           // error: can't assign a char* to an int*
```

Why is it an error to assign pc to pi? Consider one answer: a char is usually much smaller than an int, so consider this:

```
char ch1 = 'a';
char ch2 = 'b';
char ch3 = 'c';
char ch4 = 'd';
int* pi = &ch3;    // error: we cannot assign a char* to an int*, but let's pretend we could
*pi = 12345;       // write to an int-sized piece of memory
*pi = 67890;
```

Exactly how the compiler allocates variables in memory is implementation defined, but we might very well get something like this:



Now, had the compiler allowed the code, we would have been writing 12345 to the memory starting at &ch3. That would definitely have changed the value of some nearby memory, such as ch2 or ch4. If we were really unlucky (which is likely), we would have overwritten part of pi itself! In that case, the next assignment (*pi=67890) would place 67890 in some completely different part of memory. Be glad that such assignment is disallowed. It is called *memory corruption*. However, this is

one of the very few protections offered by the compiler at this low level of programming.

We are really close to the hardware here. This is not a particularly comfortable place to be for a programmer. We have only a few primitive operations available and hardly any support from the language or the standard library. However, we had to get here to know how higher-level facilities, such as **vector**, are implemented. We need to understand how to write code at this level because not all code can be "high-level" (PPP2.Ch25). Also, we might better appreciate the convenience and relative safety of the higher levels of software once we have experienced their absence. Our aim is always to work at the highest level of abstraction that is possible given a problem and the constraints on its solution. In this chapter and in Chapter 16 to Chapter 18, we show how to get back to a more comfortable level of abstraction by implementing a **Vector**.

15.3.1 The sizeof operator

So how much memory does an **int** really take up? A pointer? The operator **sizeof** answers such questions: **CC**

```
void sizes(char ch, int i, int* p)
{
    cout << "the size of char is " << sizeof(char) << ' ' << sizeof(ch) << '\n';
    cout << "the size of int is " << sizeof(int) << ' ' << sizeof(i) << '\n';
    cout << "the size of int* is " << sizeof(int*) << ' ' << sizeof(p) << '\n';
}
```

As you can see, we can apply **sizeof** either to a type name or to an expression; for a type, **sizeof** gives the size of an object of that type, and for an expression, it gives the size of the type of the result. The result of **sizeof** is a positive integer and the unit is **sizeof(char)**, which is defined to be **1**. Typically, a **char** is stored in a byte, so **sizeof** reports the number of bytes.

> TRY THIS
>
> Execute the example above and see what you get. Then extend the example to determine the size of **bool**, **double**, and some other type.

The size of a type is *not* guaranteed to be the same on every implementation of C++. These days, **sizeof(int)** is typically 4 on a laptop or desktop machine. With an 8-bit byte, that means that an **int** is 32 bits. However, embedded systems processors with 16-bit **int**s and high-performance architectures with 64-bit **int**s are common.

How much memory is used by a **vector**? We can try

```
vector<int> v(1000);          // vector with 1000 elements of type int
cout << "the size of vector<int>(1000) is " << sizeof (v) << '\n';
```

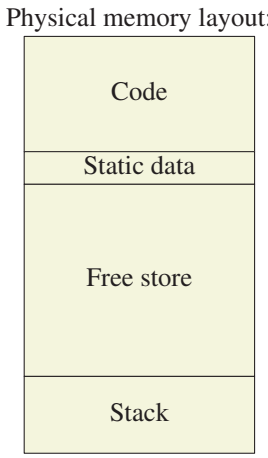
The output will be something like

```
the size of vector<int>(1000) is 20
```

The explanation will become obvious over this chapter and the next (see also §16.1.1), but clearly, **sizeof** is not counting the **vector** elements.

15.4 Free store and pointers

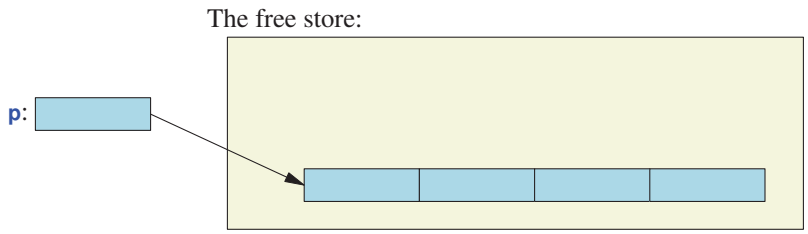
CC Consider the implementation of **Vector** from the end of §15.2. From where does the **Vector** get the space for the elements? How do we get the pointer **elem** to point to them? When you start a C++ program, the compiler sets aside memory for your code (sometimes called *code storage* or *text storage*) and for the global variables you define (called *static storage*). It also sets aside some memory to be used when you call functions, and they need space for their arguments and local variables (§7.4.8) called *stack storage* or *automatic storage*. The rest of the computer's memory is potentially available for other uses; it is "free." We can illustrate that graphically:



The C++ language makes this *free store* (also called the *heap* and *dynamic memory*) available through an operator called **new**. For example:

```
double* p = new double[4];          // allocate 4 doubles on the free store
```

This asks the C++ run-time system to allocate four **double**s on the free store and return a pointer to the first **double** to us. We use that pointer to initialize our pointer variable **p**. We can represent this graphically:



The **new** operator returns a pointer to the object it creates. If it created several objects (an array of objects), it returns a pointer to the first of those objects. If that object is of type **X**, the pointer returned by **new** is of type **X***. For example:

```
char* q = new double[4];        // error: a double* assigned to a char*
```

That **new** returns a pointer to a **double** and a **double** isn't a **char**, so we should not (and cannot) assign it to the pointer to **char** variable **q**.

We say that the pointer **q** points to an array of four elements of type **double**. An *array* is a contiguous sequence of elements of a given type.

15.4.1 Free-store allocation

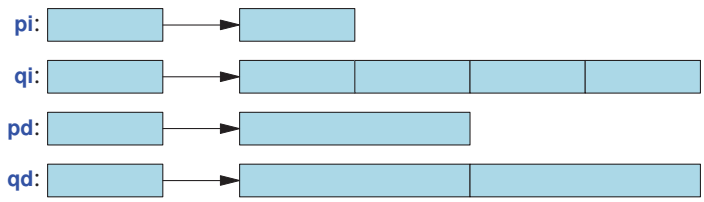
We request memory to be *allocated* on the *free store* by the **new** operator: **CC**

- The **new** operator returns a pointer to the allocated memory.
- A pointer value is the address of the first byte of the memory.
- A pointer points to an object of a specified type.
- A pointer does *not* know how many elements it points to (this is the root cause of many problems).

The **new** operator can allocate individual elements or sequences (arrays) of elements. For example:

```
int* pi = new int;              // allocate one int
int* qi = new int[4];           // allocate 4 ints (an array of 4 ints)

double* pd = new double;        // allocate one double
double* qd = new double[n];     // allocate n doubles (an array of n doubles)
```

Note that the number of objects allocated can be a variable. That's important because that allows us to select how many objects we allocate at run time. If **n** is **2**, we get



Pointers to objects of different types are different types. For example:

```
pi = pd;        // error: can't assign a double* to an int*
pd = pi;        // error: can't assign an int* to a double*
```

Why? After all, we can assign an **int** to a **double** and a **double** to an **int**. However, an **int** and a **double** may have different sizes. If so:

- Writes through a pointer to a larger element than was allocated can overwrite unrelated variables (§15.3).
- The **[ ]** operator relies on the size of the element type to figure out where to find an element. So if **sizeof(int)!=sizeof(double)** we could get some rather strange results if we allowed **qi** to point to the memory allocated for **qd**.

That's the "practical explanation." The theoretical explanation is simply "Allowing assignment of pointers to different types would allow type errors."

15.4.2 Access through pointers

In addition to using the dereference operator * on a pointer, we can use the subscript operator [ ]. For example:

```
double* p = new double[4];      // allocate 4 doubles on the free store
double x = *p;                  // read the (first) object pointed to by p
double y = p[0];                // read the 1st object pointed to by p
double z = p[2];                // read the 3rd object pointed to by p
```

Unsurprisingly, the subscript operator for a pointer counts from 0 just like **vector**'s subscript operator, so **p[2]** refers to the third element; **p[0]** is the first element so **p[0]** means exactly the same as ***p**. The [ ] and * operators can also be used for writing:

```
*p = 7.7;                       // write to the (first) object pointed to by p
p[2] = 9.9;                     // write to the 3rd object pointed to by p
```

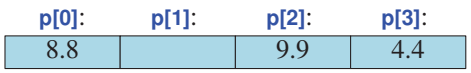
A pointer points to an object in memory. The "contents of" operator (also called the *dereference* operator) allows us to read and write the object pointed to by a pointer **p**:

```
double x = *p;                  // read the object pointed to by p
*p = 8.8;                       // write to the object pointed to by p
```

When applied to a pointer, the [ ] operator treats memory as a sequence of objects (of the type specified by the pointer declaration) with the first one pointed to by a pointer **p**:

```
double x = p[3];                // read the 4th object pointed to by p
p[3] = 4.4;                     // write to the 4th object pointed to by p
double y = p[0];                // p[0] is the same as *p
```

That's all. There is no checking, no implementation cleverness, just simple access to our computer's memory:



This is exactly the simple and optimally efficient mechanism for accessing memory that we need to implement a **vector**.

We can have pointers to any object in memory. That includes object of class types, such as **vector**s. For example:

```
vector<int>* p = new vector<int>{7,8,9};
cout << p->size();              // access using ->
cout << (*p).size();            // access using .
```

We access a class member through a pointer using the **->** operator. Alternatively, and equivalently, we can dereference the pointer (using operator *) and then use operator **.** (dot).

15.4.3 Initialization

As ever, we would like to ensure that an object has been given a value before we use it; that is, we want to be sure that our pointers are initialized and also that the objects they point to have been initialized. Consider:

```
double* p0;                 // uninitialized: likely trouble
double* p1 = new double;    // get (allocate) an uninitialized double
double* p2 = new double{5.5}; // get a double initialized to 5.5
double* p3 = new double[5]; // get (allocate) 5 uninitialized doubles
```

Obviously, declaring **p0** without initializing it is asking for trouble. Consider:

```
*p0 = 7.0;
```

This will assign **7.0** to some location in memory. We have no idea which part of memory that will be. It could be harmless, but never, never ever, rely on that. Sooner or later, we get the same result as for an out-of-range access: "My program crashed mysteriously" or "My program gives wrong output." A scary percentage of serious problems with old-style C++ programs ("C-style programs") is caused by access through uninitialized pointers and out-of-range access. We must do all we can to avoid such access, partly because we aim at professionalism, partly because we don't care to waste our time searching for that kind of error. There are few activities as frustrating and tedious as tracking down this kind of bug. It is much more pleasant and productive to prevent bugs than to hunt for them. Always initialize your variables. XX

Memory allocated by **new** is not initialized for built-in types. If you don't like that, you can specify values, as we did for **p2**: ***p2** is **5.5**. Note the use of **{ }** for initialization. This contrasts to the use of **[ ]** to indicate "array." XX

We can specify an initializer list for an array of objects allocated by **new**. For example:

```
double* p4 = new double[5] {0,1,2,3,4};
double* p5 = new double[] {0,1,2,3,4};
```

Now **p4** points to objects of type **double** containing the values **0.0**, **1.0**, **2.0**, **3.0**, and **4.0**. So does **p5**; the number of elements can be left out when a set of elements is provided.

As usual, we should worry about uninitialized objects and make sure we give them a value before we read them. Beware that compilers often have a "debug mode" where they by default initialize every variable to a predictable value (often 0). That implies that when turning off the debug features to ship a program, when running an optimizer, or simply when compiling on a different machine, a program with uninitialized variables may suddenly run differently. Don't get caught with an uninitialized variable. The standard-library **vector** helps avoid that by initializing its elements. XX

When we define our own types, we have better control of initialization. If a type **X** has a default constructor (§8.4.2), we get

```
X* px1 = new X;     // one default-initialized X
X* px2 = new X[17]; // 17 default-initialized Xs
```

If a type **Y** has a constructor, but not a default constructor, we have to explicitly initialize:

```
Y* py1 = new Y;          // error: no default constructor
Y* py2 = new Y{13};      // OK: initialized to Y{13}
Y* py3 = new Y[17];      // error: no default constructor
Y* py4 = new Y[17] {0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
```

Long initializer lists for **new** can be impractical, but they can come in very handy when we want just a few elements, and that is a common case.

15.4.4 The null pointer

If you have no other pointer to use for initializing a pointer, use the null pointer, **nullptr**:

```
double* p0 = nullptr;          // the null pointer
```

Often, we test whether a pointer is valid (i.e., whether it points to something) by checking whether it is **nullptr**. For example:

```
if (p0 != nullptr)          // consider p0 valid
```

This is not a perfect test, because **p0** may contain a "random" value that happens to be nonzero (e.g., if we forgot to initialize) or the address of an object that has been **delete**d (see §15.4.5). However, that's often the best we can do. We don't actually have to mention **nullptr** explicitly because an **if**-statement really checks whether its condition is **nullptr**:

```
if (p0)                     // consider p0 valid; equivalent to p0!=nullptr
```

AA We prefer this shorter form, considering it a more direct expression of the idea "**p0** is valid," but opinions vary.

We need to use the null pointer when we have a pointer that sometimes points to an object and sometimes not. That's rarer than many people think; consider: If you don't have an object for a pointer to point to, why did you define that pointer? Couldn't you wait until you have an object?

In older code, people often use **0** (zero) or **NULL** instead of **nullptr**. Both older alternatives can lead to confusion and/or errors, so prefer the more specific **nullptr**.

15.4.5 Free-store deallocation

The **new** operator allocates ("gets") memory from the free store. Since a computer's memory is limited, it is usually a good idea to return memory to the free store once we are finished using it. That way, the free store can reuse that memory for a new allocation. For large programs and for long-running programs such freeing of memory for reuse is essential. For example:

```
double* calc(int res_size, int max)
{
    double* p = new double[max];
    double* res = new double[res_size];
    // ... use p to calculate results to be put in res ...
    delete[] p;          // return the array pointed to by p to the free store
    return res;
}
```

```
double* r = calc(100,1000);
// ... use the result ...
delete[] r;          // return the array pointed to by r to the free store
```

The operator for returning memory to the free store is called **delete**. We apply **delete** to a pointer returned by **new** to make the memory available to the free store for future allocation. If the memory allocated by **new** is an array, we add **[ ]** to **delete**.

Forgetting to **delete** an object that was created by **new** is called a *memory leak* and is usually a bad mistake.

Deleting an object twice is also a bad mistake. For example:

```
int* p = new int{5};
delete p;            // fine: p points to an object created by new
// ...
delete p;            // error: p points to memory owned by the free-store manager
```

After the first **delete**, you don't own the object pointed to anymore so the free-store manager may have modified the memory pointed to or given that memory to some other part of the program that used **new**.

15.5 Destructors

Having to remember to call **delete** is an error-prone nuisance, so leave that to classes, such as **vector** that do so implicitly. For example:

```
vector<double> calc2(int res_size, int max)
{
    vector<double> p(max);
    vector<double> res(res_size);
    // ... use p to calculate results to be put in res ...
    return res;
}

void use()
{
    // ...
    vector<double> r = calc2(100,1000);
    // ... use the result ...
}
```

This is shorter and much cleaner code – no **new**s and **delete**s to keep track of – and (surprisingly?) just as efficient as the original version.

The **new**s and **delete**s disappeared into the **vector**. Let's see how that's done:

```
class Vector {       // a very simplified vector of doubles
public:
    Vector(int s);                  // constructor: allocate elements on the free store and initialize them
    ~Vector() { delete[] elem; }    // destructor: return elements to free store
    // ...
```

```
private:
        int sz;                         // the size
        double* elem;                   // a pointer to the elements

};

Vector::Vector(int s)                   // constructor
        :sz{s},                         // initialize sz
        elem{new double[s]}             // initialize elem to elements on the free store
{
        for (int i=0; i<s; ++i)         // initialize elements
                elem[i]=0;
}


Vector::~Vector()               // destructor
{
        delete[] elem;          // return elements to free store
}
```

So, **sz** is the number of elements. We initialize it in the constructor, and a user of **vector** can get the number of elements by calling **size()**. Space for the elements is allocated using **new** in the constructor, and the pointer returned from the free store is stored in the member pointer **elem**.

This **Vector** does not leak memory. The basic idea is to have the compiler know about a function that does the opposite of a constructor, just as it knows about the constructor. Inevitably, such a function is called a *destructor*, and its name is the name of the class preceded by the complement operator ~, here **~Vector**. In the same way that a constructor is implicitly called when an object of a class is created, a destructor is implicitly called when an object goes out of scope. A constructor makes sure that an object is properly created and initialized. Conversely, a destructor makes sure that an object is properly cleaned up before it is destroyed.

AA We are not going to go into great detail about the uses of destructors here, but they are great for handling resources that we need to first acquire (from somewhere) and later give back: files, threads, locks, etc. Remember how **iostream**s clean up after themselves? They flush buffers, close files, free buffer space, etc. That's done by their destructors. Every class that "owns" a resource needs a destructor.

The use of constructor/destructor pairs is the key to some of the most effective C++ programming techniques. Destructors make returning resources implicit. That's essential because we know from many fields of life and many programming problems that remembering to give something back when we are done using it is surprisingly hard.

15.5.1 Generated destructors

If a member of a class has a destructor, then that destructor will be called when the object containing the member is destroyed. For example:

```
struct Customer {
        string name;
        vector<string> addresses;
        // ...
};

void some_fct()
{
        Customer fred { "Fred", {"17 Oak St.", "232 Rock Ave."}};
        // ... use fred ...
}
```

When we exit **some_fct()**, so that **fred** goes out of scope, **fred** is destroyed; that is, the destructors for **name** and **addresses** are called. This is obviously necessary for destructors to be useful and is sometimes expressed as "The compiler generated a destructor for **Customer**, which calls the members' destructors." That is indeed often how the obvious and necessary guarantee that destructors are called is implemented. A generated destructor is often called the *default destructor*.

The destructors for members – and for bases (§12.3) – are implicitly called from a derived class destructor (whether user-defined or generated). Basically, all the rules add up to: "Destructors are called when the object is destroyed" (by going out of scope, by **delete**, etc.).

15.5.2 Virtual destructors

Destructors are conceptually simple but are the foundation for many of the most effective C++ programming techniques. The basic idea is simple: **CC**

- Whatever resources a class object needs to function, it acquires it in a constructor.
- During the object's lifetime it may release resources and acquire new ones.
- At the end of the object's lifetime, the destructor releases every resource still owned by the object.

The matched constructor/destructor pair handling free-store memory for **vector** is the archetypical example. We'll get back to that idea with more examples in §18.4. Here, we will examine an important application that comes from the use of free-store and class hierarchies in combination. Consider a use of **Shape** and **Text** from §12.2 and §11.8:

```
Shape* fct()
{
        Text tt {Point{200,200},"Anya"};                    // local Text variable
        // ...
        return new Text{Point{100,100},"Courtney"};         // Text object on the free store
}

void user()
{
        Shape* q = fct();
        // ... use the Shape without caring exactly which kind of shape it is ...
        delete q;
}
```

This looks fairly plausible – and it is. It all works, but let's see how, because that exposes an elegant, important, simple technique. The **Text** (§11.8) object **tt** is properly destroyed at the exit from **fct()**. **Text** has a **string** member, which obviously needs to have its destructor called – **string** handles its memory acquisition and release exactly like **vector**. For **tt**, that's easy; the compiler just calls **Text**'s generated destructor as described in §15.5.1. But what about the **Text** object returned from **fct()**? The calling function **user()** has no idea that **q** points to a **Text**; all it knows is that it points to a **Shape**. Then how does **delete q** get to call **Text**'s destructor?

In §12.2, we breezed past the fact that **Shape** has a destructor. In fact, **Shape** has a **virtual** destructor. That's the key. When we say **delete q**, **delete** looks at **q**'s type to see if it needs to call a destructor, and if so it calls it. So, **delete q** calls **Shape**'s destructor **~Shape()**. But **~Shape()** is **virtual**, so – using the **virtual** call mechanism (§12.3.2) – that call invokes the destructor of **Shape**'s derived class, in this case **~Text()**. Had **Shape::~Shape()** not been **virtual**, **Text::~Text()** would not have been called and **Text**'s **string** member wouldn't have been properly destroyed.

AA As a rule of thumb: if you have a class with a **virtual** function, it needs a **virtual** destructor. The reason is that if a class has a **virtual** function, it is likely to be used as a base class and objects of its derived class are likely to be allocated using **new** and manipulated through pointers to their base. Thus, such derived class objects are likely to be **delete**d through pointers to their base.

CC Destructors are invoked implicitly for scoped objects or indirectly through **delete**. They are not called directly. That saves a lot of tricky work.

> TRY THIS
>
> Write a little program using base classes and members where you define the constructors and destructors to output a line of information when they are called. Then, create a few objects and see how their constructors and destructors are called.

But what about that **delete q;**? We just deemed having to remember to **delete** tedious and error-prone (§15.4.5). The standard library has a *smart pointer* (also called a *resource-management pointer)* to deal with that:

```
unique_ptr<Shape> fct()
{
    Text tt {Point{200,200},"Annemarie"};                    // local Text variable
    // ...
    return make_unique<Text>(Point{100,100},"Nicholas");     // Text object on the free store
}

void user()
{
    unique_ptr<Shape> q = fct();
    // ... use the Shape without caring exactly which kind of shape it is ...
}
```

A **unique_ptr** object holds a pointer (§18.5.2). When the **unique_ptr** goes out of scope, its destructor calls **delete** on that pointer. Again, we see the code become shorter and simpler by relying on a class with a destructor.

XX Also, please remember that a "naked" **new** outside a constructor is an opportunity to forget to **delete** the object that **new** created, thus creating a potentially serious resource leak. "Naked" **new**s

and "naked" **delete**s are sources of serious errors. Instead, use resource-management classes, such as **vector**, **unique_ptr** (§18.5.2), or **Vector_ref** (§11.7.3). Keep **new**s in constructors and **delete**s in destructors (and in related resource-management functions (§17.4)).

15.6 Access to elements

For **Vector** to be usable, we need a way to read and write elements. For starters, we can provide simple **get()** and **set()** member functions:

```
class Vector {          // a very simplified vector of doubles
public:
    Vector(int s) :sz{s}, elem{new double[s]} { /* ... */ }     // constructor
    ~Vector() { delete[] elem; }                                // destructor

    int size() const { return sz; }                             // the current size

    double get(int n) const { return elem[n]; }                 // access: read
    void set(int n, double v) { elem[n]=v; }                    // access: write
private:
    int sz;                 // the size
    double* elem;           // a pointer to the elements
};
```

Both **get()** and **set()** access the elements using the **[ ]** operator on the **elem** pointer: **elem[n]**.

Now we can make a **Vector** of **double**s and use it:

```
Vector v(5);
for (int i=0; i<v.size(); ++i) {
    v.set(i,1.1*i);
    cout << "v[" << i << "]==" << v.get(i) << '\n';
}
```

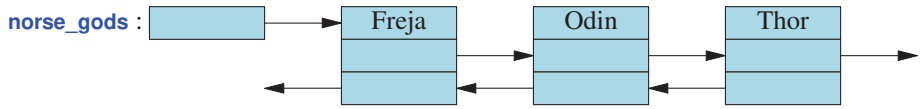
This will output

```
v[0]==0
v[1]==1.1
v[2]==2.2
v[3]==3.3
v[4]==4.4
```

This is still an overly simple **vector**, and the code using **get()** and **set()** is rather ugly compared to the usual subscript notation. However, we start small and simple and then grow our programs step by step, testing along the way. This strategy of growth and repeated testing minimizes errors and debugging.

15.7 An example: lists

Lists are among the most common and useful data structures. Usually, a list is made out of "links" where each link holds some information and pointers to other links. This is one of the classical uses of pointers. For example, we could represent a short list of Norse gods like this:



CC A list like this is called a *doubly-linked list* because given a link, we can find both the predecessor and the successor. A list where we can find only the successor is called a *singly-linked list*. We use doubly-linked lists when we want to make it easy to remove an element. We can define these links like this:

```
struct Link {
        Link(const string& v, Link* p = nullptr, Link* s = nullptr)
                : value{v}, prev{p}, succ{s} { }
        string value;
        Link* prev;
        Link* succ;
};
```

That is, given a **Link**, we can get to its successor using the **succ** pointer and to its predecessor using the **prev** pointer. We use **nullptr** to indicate that a **Link** doesn't have a successor or a predecessor. We can build our list of Norse gods like this:

```
Link* norse_gods = new Link{"Thor",nullptr,nullptr};
norse_gods = new Link{"Odin",nullptr,norse_gods};
norse_gods->succ->prev = norse_gods;
norse_gods = new Link{"Freja",nullptr,norse_gods};
norse_gods->succ->prev = norse_gods;
```

We built that list by creating the **Link**s and tying them together as in the picture: first Thor, then Odin as the predecessor of Thor, and finally Freja as the predecessor of Odin. You can follow the pointer to see that we got it right, so that each **succ** and **prev** points to the right god. However, the code is obscure because we didn't explicitly define and name an insert operation:

```
Link* insert(Link* p, Link* n)        // insert n before p (incomplete)
{
        n->succ = p;                  // p comes after n
        p->prev->succ = n;            // n comes after what used to be p's predecessor
        n->prev = p->prev;            // p's predecessor becomes n's predecessor
        p->prev = n;                  // n becomes p's predecessor
        return n;
}
```

This works provided that **p** really points to a **Link** and that the **Link** pointed to by **p** really has a predecessor. Please convince yourself that this really is so. When thinking about pointers and linked structures, such as a list made out of **Link**s, we invariably draw little box-and-arrow diagrams on

paper to verify that our code works for small examples. Please don't be too proud to rely on this effective low-tech design technique.

That version of **insert()** is incomplete because it doesn't handle the cases where **n**, **p**, or **p->prev** is **nullptr**. We add the appropriate tests for the null pointer and get the messier, but correct, version:

```
Link* insert(Link* p, Link* n)        // insert n before p; return n
{
    if (n==nullptr)
        return p;
    if (p==nullptr)
        return n;
    n->succ = p;                      // p comes after n
    if (p->prev)
        p->prev->succ = n;
    n->prev = p->prev;                // p's predecessor becomes n's predecessor
    p->prev = n;                      // n becomes p's predecessor
    return n;
}
```

Given that, we could write

```
Link* norse_gods = new Link{"Thor"};
norse_gods = insert(norse_gods,new Link{"Odin"});
norse_gods = insert(norse_gods,new Link{"Freja"});
```

Now all the error-prone fiddling with the **prev** and **succ** pointers has disappeared from sight. Pointer fiddling is tedious and error-prone and *should* be hidden in well-written and well-tested functions. In particular, many errors in conventional code come from people forgetting to test pointers against **nullptr** – just as we (deliberately) did in the first version of **insert()**. AA

Note that we used default arguments (§13.3.1) to save users from mentioning predecessors and successors in every constructor use.

15.7.1 List operations

The standard library provides a **list** class, which we will describe in §19.3. It hides all link manipulation, but here we will elaborate on our notion of **list** based on the **Link** class to get a feel for what goes on "under the covers" of **list** classes and see more examples of pointer use.

What operations does our **Link** class need to allow its users to avoid "pointer fiddling"? That's to some extent a matter of taste, but here is a useful set:

- The constructor
- **insert**: insert before an element
- **add**: insert after an element
- **erase**: remove an element
- **find**: find a **Link** with a given value
- **advance**: get the *n* th successor

We could write these operations like this:

```
Link* add(Link* p, Link* n)             // insert n after p; return n
{
    // ... much like insert() ...
}

Link* erase(Link* p)        // remove *p from list; return p's successor
{
    if (p==nullptr)
        return nullptr;
    if (p->succ)
        p->succ->prev = p->prev;
    if (p->prev)
        p->prev->succ = p->succ;
    return p->succ;
}

Link* find(Link* p, const string& s)        // find s in list; return nullptr for "not found"
{
    while (p) {
        if (p->value == s)
            return p;
        p = p->succ;
    }
    return nullptr;
}

Link* advance(Link* p, int n)           // move n positions in list; return nullptr for "not found"
    // positive n moves forward, negative backward
{
    if (p==nullptr)
        return nullptr;
    while (0<n) {
        --n;
        if (p->succ)
            p = p->succ;
        return nullptr;
    }
    while (n<0) {
        ++n;
        if (p->prev)
            p = p->prev;
        return nullptr;
    }
    return p;
}
```

We could have considered an attempt to **advance()** outside the list an error and thrown an exception, but we chose to return the **nullptr** thereby forcing the user to deal with that possibility. Why? Primarily because at this level of pointer use, the user has to be constantly aware of the possibility of

nullptr anyway. Yes, that's error prone, so we use higher-level constructs (such as the standard-library list; §20.6) whenever we can.

15.7.2 List use

As a little exercise, let's build two lists:

```
Link* norse_gods = new Link{"Thor"};
norse_gods = insert(norse_gods,new Link{"Odin"});
norse_gods = insert(norse_gods,new Link{"Zeus"});
norse_gods = insert(norse_gods,new Link{"Freja"});

Link* greek_gods = new Link{"Hera"};
greek_gods = insert(greek_gods,new Link{"Athena"});
greek_gods = insert(greek_gods,new Link{"Mars"});
greek_gods = insert(greek_gods,new Link{"Poseidon"});
```

"Unfortunately," we made a couple of mistakes: Zeus is a Greek god, rather than a Norse god, and the Greek god of war is Ares, not Mars (Mars is his Latin/Roman name). We can fix that:

```
Link* p = find(greek_gods, "Mars");
if (p)
    p->value = "Ares";
```

Note how we were cautious about find() returning a nullptr. We think that we know that it can't happen in this case (after all, we just inserted Mars into greek_gods), but in a real example someone might change that code.

Similarly, we can move Zeus into his correct Pantheon:

```
Link* p = find(norse_gods,"Zeus");
if (p) {
    erase(p);
    insert(greek_gods,p);
}
```

Did you notice the bug? It's quite subtle (unless you are used to working directly with links). What if the Link we erase() is the one pointed to by norse_gods? Again, that doesn't actually happen here, but to write good, maintainable code, we have to take that possibility into account:

```
Link* p = find(norse_gods, "Zeus");
if (p) {
    if (p==norse_gods)
        norse_gods = p->succ;
    erase(p);
    greek_gods = insert(greek_gods,p);
}
```

While we were at it, we also corrected the second bug: when we insert Zeus *before* the first Greek god, we need to make greek_gods point to Zeus's Link. Pointers are extremely useful and flexible, but subtle. They have to be handled with care and avoided in favor of less error-prone alternatives wherever possible.

Finally, let's print out those lists:

```
void print_all(Link* p)
{
        cout << "{ ";
        while (p) {
                cout << p->value;
                if (p=p->succ)
                        cout << ", ";
        }
        cout << " }";
}

print_all(norse_gods);
cout << '\n';

print_all(greek_gods);
cout << '\n';
```

This should give

```
{ Freja, Odin, Thor }
{ Zeus, Poseidon, Ares, Athena, Hera }
```

15.8 The this pointer

Note that each of our list functions takes a `Link*` as its first argument and accesses data in that object. That's the kind of function that we often make a member function. Could we simplify `Link` (or link use) by making the operations members? Could we maybe make the pointers private so that only the member functions have access to them? We could:

```
class Link {
public:
        string value;

        Link(const string& v, Link* p = nullptr, Link* s = nullptr)
                : value{v}, prev{p}, succ{s} { }

        Link* insert(Link* n);                  // insert n before this object
        Link* add(Link* n);                     // insert n after this object
        Link* erase();                          // remove this object from list

        Link* find(const string& s);            // find s in list
        const Link* find(const string& s) const;    // find s in const list (see _oper.const_)

        Link* advance(int n) const;             // move n positions in list
        Link* next() const { return succ; }
        Link* previous() const { return prev; }
```

```
private:
        Link* prev;
        Link* succ;
};
```

This looks promising. We defined the operations that don't change the state of a **Link** into **const** member functions. We added (nonmodifying) **next()** and **previous()** functions so that users could iterate over lists (of **Link**s) – those are needed now that direct access to **succ** and **prev** is prohibited. We left **value** as a public member because (so far) we have no reason not to; it is "just data."

Now, let's try to implement **Link::insert()** by copying our global **insert()** and modifying it:

```
Link* Link::insert(Link* n)          // insert n before p; return n
{
        Link* p = this;              // pointer to this object
        if (n==nullptr)              // nothing to insert
                return p;
        if (p==nullptr)              // nothing to insert into
                return n;
        n->succ = p;                 // p comes after n
        if (p->prev)
                p->prev->succ = n;
        n->prev = p->prev;           // p's predecessor becomes n's predecessor
        p->prev = n;                 // n becomes p's predecessor
        return n;
}
```

But how do we get a pointer to the object for which **Link::insert()** was called? Without help from the language we can't. However, in every member function, the identifier **this** is a pointer that points to the object for which the member function was called. Alternatively, we could simply use **this** instead of **p**: **CC**

```
Link* Link::insert(Link* n)          // insert n before this object; return n
{
        if (n==nullptr)
                return this;
        if (this==nullptr)
                return n;
        n->succ = this;              // this object comes after n
        if (this->prev)
                this->prev->succ = n;
        n->prev = this->prev;        // this object's predecessor becomes n's predecessor
        this->prev = n;              // n becomes this object's predecessor
        return n;
}
```

This is a bit verbose, but we don't need to mention **this** to access a member, so we can abbreviate:

```
Link* Link::insert(Link* n)      // insert n before this object; return n
{
      if (n==nullptr)
            return this;
      if (this==nullptr)
            return n;
      n->succ = this;                  // this object comes after n
      if (prev)
            prev->succ = n;
      n->prev = prev;                  // this object's predecessor becomes n's predecessor
      prev = n;                        // n becomes this object's predecessor
      return n;
}
```

In other words, we have been using the **this** pointer – the pointer to the current object – implicitly every time we accessed a member. It is only when we need to refer to the whole object that we need to mention it explicitly.

Note that **this** has a specific meaning: it points to the object for which a member function is called. It does not point to any old object. The compiler ensures that we do not change the value of **this** in a member function. For example:

```
struct S {
      // . . .
      void mutate(S* p)
      {
            this = p;          // error: this is immutable
            // ...
      }
};
```

15.8.1 More link use

Having dealt with the implementation issues, we can see how the use now looks:

```
Link* norse_gods = new Link{"Thor"};
norse_gods = norse_gods->insert(new Link{"Odin"});
norse_gods = norse_gods->insert(new Link{"Zeus"});
norse_gods = norse_gods->insert(new Link{"Freja"});

Link* greek_gods = new Link{"Hera"};
greek_gods = greek_gods->insert(new Link{"Athena"});
greek_gods = greek_gods->insert(new Link{"Mars"});
greek_gods = greek_gods->insert(new Link{"Poseidon"});
```

That's very much like before. As before, we correct our "mistakes." In this case, the name of the god of war is wrong, so we change it:

```
Link* p = greek_gods->find("Mars");
if (p)
        p->value = "Ares";
```

Move Zeus into his correct Pantheon:

```
Link* p2 = norse_gods->find("Zeus");
if (p2) {
        if (p2==norse_gods)
                norse_gods = p2->next();
        p2->erase();
        greek_gods = greek_god->insert(p2);
}
```

Finally, let's print out those lists:

```
void print_all(Link* p)
{
        cout << "{ ";
        while (p) {
                cout << p->value;
                if (p=p->next())
                        cout << ", ";
        }
        cout << " }";
}

print_all(norse_gods);
cout << '\n';

print_all(greek_gods);
cout << '\n';
```

This should again give

```
{ Freja, Odin, Thor }
{ Zeus, Poseidon, Ares, Athena, Hera }
```

So, which version do you like better: the one where **insert()**, etc. are member functions or the one where they are freestanding functions? In this case the differences don't matter much, but see §8.7.5.

One thing to observe here is that we still don't have a list class, only a link class. That forces us to keep worrying about which pointer is the pointer to the first element. This **Link** code is littered with naked **new**s and there isn't a **delete** in sight. We hope you noticed and that it made you a bit queasy. We can do better – by defining a class **List** – but designs along the lines presented here are very common and the **Link** examples are meant to illustrate pointer manipulation, rather than resource management. The standard-library **list** is presented in §19.3. In general, subtle pointer manipulation is best encapsulated in a class (§12.3). **AA**

Drill

This drill has two parts. The first exercises/builds your understanding of free-store-allocated arrays and contrasts arrays with vectors:

[1] Allocate an array of ten ints on the free store using new.
[2] Print the values of the ten ints to cout.
[3] Deallocate the array (using delete[]).
[4] Write a function print_array(ostream& os, int* a, int n) that prints out the values of a (assumed to have n elements) to os.
[5] Allocate an array of ten ints on the free store; initialize it with the values 100, 101, 102, etc.; and print out its values.
[6] Allocate an array of 11 ints on the free store; initialize it with the values 100, 101, 102, etc.; and print out its values.
[7] Allocate an array of 20 ints on the free store; initialize it with the values 100, 101, 102, etc.; and print out its values.
[8] Did you remember to delete the arrays? (If not, do it.)
[9] Do 5, 6, and 7 using a vector instead of an array and a print_vector() instead of print_array().

The second part focuses on pointers and their relation to arrays. Use print_array():

[1] Allocate an int, initialize it to 7, and assign its address to a variable p1.
[2] Print out the value of p1 and of the int it points to.
[3] Allocate an array of seven ints; initialize it to 1, 2, 4, 8, etc.; and assign its address to a variable p2.
[4] Print out the value of p2 and of the array it points to.
[5] Declare an int* called p3 and initialize it with p2.
[6] Assign p1 to p2.
[7] Assign p3 to p2.
[8] Print out the values of p1 and p2 and of what they point to.
[9] Deallocate all the memory you allocated from the free store.
[10] Allocate an array of ten ints; initialize it to 1, 2, 4, 8, etc.; and assign its address to p1.
[11] Allocate an array of ten ints, and assign its address to a variable p2.
[12] Copy the values from the array pointed to by p1 into the array pointed to by p2.
[13] Repeat 10–12 using a vector rather than an array.

Review

[1] Why do we need data structures with varying numbers of elements?
[2] What four kinds of storage do we have for a typical program?
[3] What is the free store? What other name is commonly used for it? What operators support it?
[4] What is a pointer?
[5] What is a dereference operator and why do we need one?
[6] What is an address? How are memory addresses manipulated in C++?
[7] What information about a pointed-to object does a pointer have? What useful information does it lack?

[8] What can a pointer point to?
[9] What is a leak?
[10] What is a resource?
[11] What is another term for "free store"?
[12] How can we initialize a pointer?
[13] What is a null pointer? When do we need to use one?
[14] When do we need a pointer (instead of a reference or a named object)?
[15] What is a destructor? When do we want one?
[16] When do we want a **virtual** destructor?
[17] How are destructors for members called?
[18] How do we access a member of a class through a pointer?
[19] What is a doubly-linked list?
[20] What is **this** and when do we need to use it?

Terms

address	destructor	**nullptr**	address of: **&**
free store	pointer	allocation	**Link**
array	list	resource leak	dereference: *
vector	container	member access: **->**	subscript: **[ ]**
deallocation	memory	**this**	memory corruption
delete	memory leak	null pointer	**delete[]**
new	**virtual**		

Exercises

[1] What is the output format of pointer values on your implementation? Hint: Don't read the documentation.
[2] How many bytes are there in an **int**? In a **double**? In a **bool**? Do not use **sizeof** except to verify your answer.
[3] List two ways that a pointer can be misused in potentially disastrous ways. Give examples.
[4] Consider the memory layout in §15.4. 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?
[5] We have not said what happens when you run out of memory using **new**. That's called *memory exhaustion*. Find out what happens. You have two obvious alternatives: look for documentation, or write a program with an infinite loop that allocates but never deallocates. Try both. Approximately how much memory did you manage to allocate before failing?
[6] Write a program that reads characters from **cin** into an array that you allocate on the free store. Read individual characters until an exclamation mark (**!**) is entered. Do not use a **std::string**. Do not worry about memory exhaustion.

[7] Do the previous exercise again, but this time read into a **std::string** rather than to memory you put on the free store (**string** knows how to use the free store for you).

[8] Which way does the stack grow: up (toward higher addresses) or down (toward lower addresses)? Which way does the free store initially grow (that is, before you use **delete**)? Write a program to determine the answers.

[9] Look at your solution of exercise 6. Is there any way that input could get the array to overflow; that is, is there any way you could enter more characters than you allocated space for (a serious error)? Does anything reasonable happen if you try to enter more characters than you allocated?

[10] Complete the "list of gods" example from §15.7 and run it.

[11] Why did we define two versions of **find()** in §15.8?

[12] Modify the **Link** class from §15.7 to hold a value of a **struct God**. **struct God** should have members of type **string**: name, mythology, vehicle, and weapon. For example, **God{"Zeus", "Greek", "", "lightning"}** and **God{"Odin", "Norse", "Eight–legged flying horse called Sleipner", "Spear called Gungnir"}**. Write a **print_all()** function that lists gods with their attributes one per line. Add a member function **add_ordered()** that places its new element in its correct lexicographical position. Using the **Link**s with the values of type **God**, make a list of gods from three mythologies; then move the elements (gods) from that list to three lexicographically ordered lists – one for each mythology.

[13] Modify the "list of gods" example from §15.7 not to leak memory.

[14] Could the "list of gods" example from §15.7 have been written using a singly-linked list; that is, could we have left the **prev** member out of **Link**? Why might we want to do that? For what kind of examples would it make sense to use a singly-linked list? Re-implement that example using only a singly-linked list.

[15] Look up (e.g., on the Web) *skip list* and implement that kind of list. This is not an easy exercise.

Postscript

Why bother with messy low-level stuff like pointers and free store when we can simply use **vector**? Well, one answer is that someone has to design and implement **vector** and similar abstractions, and it's generally useful to know how that's done. There are programming languages that dodge the problems with low-level programming. Basically, programmers of such languages delegate the tasks that involve direct access to hardware to C++ programmers (and programmers of other languages suitable for low-level programming). Our favorite reason, however, is simply that you can't really claim to understand computers and programming until you have seen how software meets hardware. People who don't know about pointers, memory addresses, and other basic low-level facilities often have the strangest ideas of how their programming language facilities work; such wrong ideas can lead to code that's "interestingly poor" (i.e., slow and/or unmaintainable).

16

Arrays, Pointers, and References

Caveat emptor!
– Good advice

This chapter describes the lower-level notions of arrays and pointers. We consider the uses of pointers, such as array traversal and address arithmetic, and the problems arising from such use. We also present the widely used C-style string; that is, a zero-terminated array of **char**s. Pointers and arrays are key to the implementation of types that save us from error-prone uses of pointers, such as **vector**, **string**, **span**, **not_null**, **unique_ptr**, and **shared_ptr**. As an example, we show a few ways we can implement a function that determines whether a sequence of characters represents a palindrome.

§16.1 Arrays
Pointer arithmetic
§16.2 Pointers and references
Pointer and reference parameters; Pointers as parameters
§16.3 C-style strings
§16.4 Alternatives to pointer use
span; **array**; **not_null**
§16.5 An example: palindromes
Palindromes using **string**; Palindromes using arrays; Palindromes using pointers; Palindromes using **span**

16.1 Arrays

So far, we have allocated all of our arrays on the free store; that's what we need to implement **vector**. However, we can also have arrays on the stack and in static memory (§15.4). For example:

```
int ai[4];              // static array of 4 ints

void fct()
{
    char ac[8];         // on-stack array of 8 chars
}
```

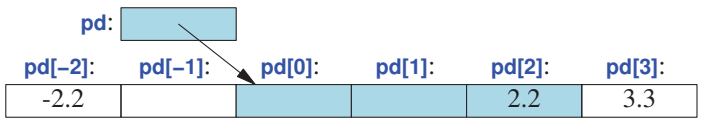
C++ uses the conventional subscript notation **[ ]** to indicate "array." For most uses, **vector<T>** is a better choice than **T[]** for representing a contiguous sequence of elements of a type **T**. "Contiguous" means that there are no gaps between the elements.

AA 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 we 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**.

XX The major problem with pointers to arrays is that a pointer doesn't "know" how many elements it points to. A pointer to an array is a pointer to the first element of the array, rather than to some "array object." Consider:

```
void use(double* pd)
{
    pd[2] = 2.2;
    pd[3] = 3.3;
    pd[–2] = –2.2;
}

void test()
{
    double a[3];
    use(a);         // a converts to a pointer to a[0] when used as an argument
}
```

What **use()** "sees" can be graphically represented:



XX Does **pd** have a third element **pd[2]**? Does it have a fourth element **pd[3]**? If we look at **use(a)**, we find that the answers are yes and no, respectively. However, the compiler doesn't know that; it does not keep track of pointer values. Our code will simply access memory as if we had allocated enough memory. It will even access **pd[–2]** as if the location two **double**s before what **pd** points to was part of our allocation.

We have no idea what the memory locations marked **pd[–2]** and **pd[3]** are used for. However, we do know that they weren't meant to be used as part of our array of three **double**s pointed to by **pd**. Most likely, they are parts of other objects and we just scribbled all over those. That's not a good idea. In fact, it is typically a disastrously poor idea: "disastrous" as in "My program crashes mysteriously" or "My program gives wrong output." Try saying that aloud; it doesn't sound nice at all. We'll go a long way to avoid that.

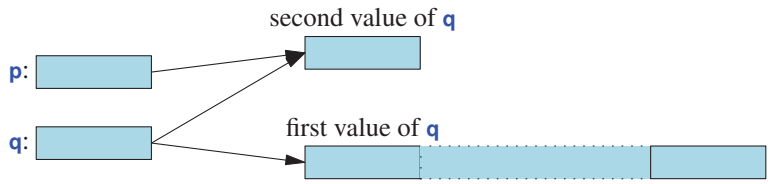
Out-of-range access is often called *range error* or *buffer overflow*. Such errors are particularly nasty because apparently unrelated parts of a program are affected. An out-of-range read gives us a "random" value that may depend on some completely unrelated computation. An out-of-range write can put some object into an "impossible" state or simply give it a totally unexpected and wrong value. Such writes typically aren't noticed until long after they occurred, so they are particularly hard to find. Worse still: run a program with an out-of-range error twice with slightly different input and it may give different results. Bugs of this kind ("transient bugs") are some of the most difficult bugs to find.

We have to ensure that such out-of-range access doesn't happen. One of the reasons we use **vector** rather than directly using memory allocated by **new** is that a **vector** knows its size so that it (or we) can easily prevent out-of-range access.

One thing that can make it hard to prevent out-of-range access is that we can assign one **double*** to another **double*** independently of how many objects each points to. A pointer really doesn't know how many objects it points to. For example:

```
double* p = new double;          // allocate a double
double* q = new double[1000];    // allocate 1000 doubles

q[700] = 7.7;                    // fine: q points to 1000 doubles
q = p;                           // let q point to the same object as p does
double d = q[700];               // bad: q points to a single double: out-of-range access!
```

Here, in just three lines of code, **q[700]** refers to two different memory locations, and the last use is an out-of-range access and a likely disaster.



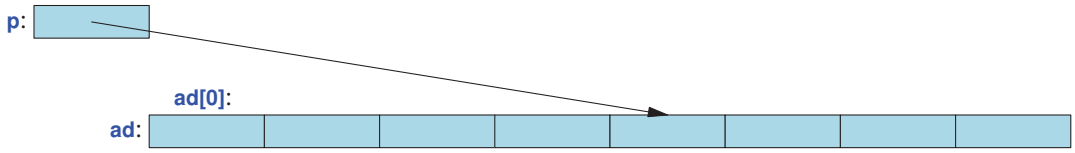
By now, we hope that you are asking, "But why can't pointers remember the size?" Obviously, we could design a "pointer" that did exactly that – a **vector** is almost that, and if you look through the C++ literature and libraries, you'll find many "smart pointers" that compensate for weaknesses of the low-level built-in pointers (e.g., see **span** in §16.4.1). However, somewhere we need to reach the hardware level and understand how objects are addressed – and a machine address does not "know" what it addresses. Also, understanding pointers is essential for understanding lots of real-world code.

16.1.1 Pointer arithmetic

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

```
double ad[8];
double* p = &ad[4];        // point to ad[4]; the 5th element of ad
```

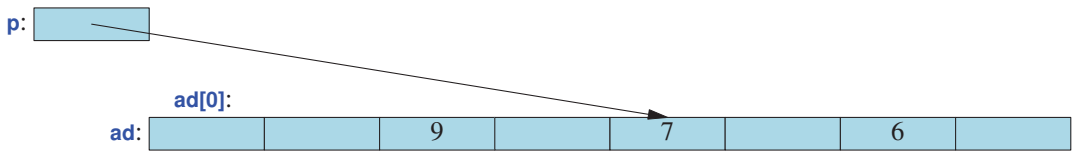
We now have a pointer **p** to the **double** known as **ad[4]**:



We can subscript and dereference that pointer:

```
*p =7;
p[2] = 6;
p[–2] = 9;
```

We get

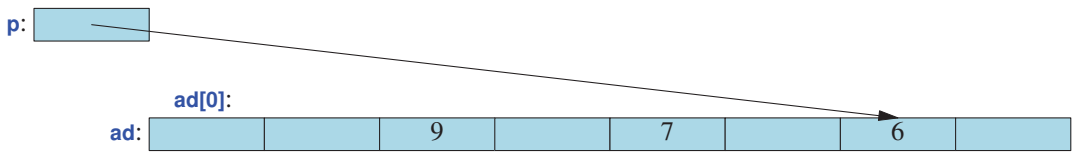


That is, we can subscript the pointer with both positive and negative numbers. As long as the resulting element is in range, all is well. However, access outside the range of the array pointed into is illegal (as with free-store-allocated arrays; see §16.1). Typically, access outside an array is not 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
```

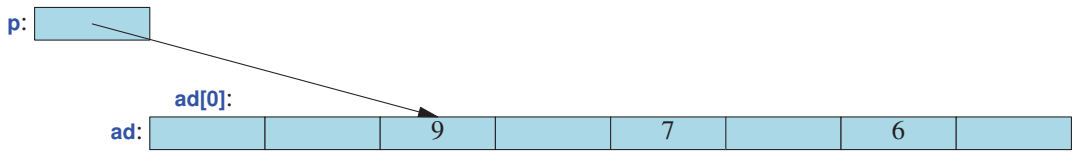
We get



We can also move backwards:

```
p -= 4;          // move p 4 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 doesn't point to memory outside the array: CC

```
p += 1000;          // insane: p points into an array with just 8 elements
double d = *p;      // illegal: probably a bad 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 simply to avoid pointer arithmetic except when implementing facilities for which there is no reasonable alternative.

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:

```
const int max = sizeof(ad)/sizeof(*ad);          // one way to determine the number of elements of ad

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

Or backward:

```
for (double* p = &ad[max-1]; 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[max-1]` and not `&ad[max]`? 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` or a `span` (§16.4.1), which is more easily range checked. Also, many examples can be done using a range-`for` (§3.6.1) rather than by subscripting.

The way of finding the number of elements in an array (here, `sizeof(ad)/sizeof(*ad)`) may seem odd, but consider: `sizeof` reports sizes in the number of bytes used to store an object, so `sizeof(ad)` will be `8*sizeof(double)` because we gave `ad` eight elements of type `double`. To get the number of elements (here `8`) back, we must divide with the size of an element (here, `*ad`). When we lower the abstraction level, our code gets messier and more error-prone, and `sizeof` is about as low as we get.

Note that most real-world uses of pointer arithmetic involve a pointer passed as a function argument (like the `use()` example in §16.1). 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* pd = &ad[0];
double d1 = *(pd+7);
double d2 = &pd[7];
if (d2 != d3)
    cout << "impossible!\n";
```

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

16.2 Pointers and references

CC You can think of a reference as an automatically dereferenced immutable pointer or as an alternative name for an object. Pointers and references differ in these ways:

- Assignment to a pointer changes the pointer's value (not the pointed-to value).
  - Assignment to a reference changes the value of the object referred to (not the reference).
- To initialize a pointer you use a pointer, a `new`, a `&`, or the name of an array.
  - To initialize a reference you use an object (maybe a pointer dereferenced by `*` or `[]`).
- To access an object pointed to by a pointer, we use `*` or `[]` (§16.1).
  - To access an object referred to by a reference, we just use the name of the reference.
- Beware of null pointers (§15.4.4).
  - A reference must be initialized to refer to an object, and cannot be made to refer to another object.

For example:

pointer	reference	comment
`int x = 10;`	`int x = 10;`	
`int* p = &x;`	`int& r = x;`	`&x` to get a pointer
`p = x;`	`r = &x;`	type errors
`*p = 7;`	`r = 7;`	write to the object pointed/referred to
`p = 7;`	`*r = 7;`	type errors
`int x2 = *p;`	`int x2 = r;`	read the value of the object pointed/referred to
`int* p2 = p;`	`int& r2 = r;`	make `p2` point to the object pointed to by `p` make `r2` refer to the object referred to by `r`
`p = nullptr;`	`r = "nullref";`	there is no nullref
`p = &x2;`		make `p` point to `x2`
	`r = x2;`	assign `x2` to the object referred to by `r`

AA Note that `r=x2` will not make the reference refer to `x2`. There is no way to get a reference to refer to a different object after initialization, and that is a valuable guarantee. Instead, `r=x2` assigns the value of `x2` to the object referred to by `r`; that is, to `x`. If you need to point to something different at different times, use a pointer.

For ideas about when and how to use pointers, see §15.7 and §16.5.

A reference and a pointer are both implemented by using a memory address. They just use that address differently to provide you – the programmer – slightly different facilities.

16.2.1 Pointer and reference parameters

When you want to change the value of a variable to a value computed by a function, you have three choices. For example:

```
int incr_v(int x) { return x+1; }      // compute a new value and return it
void incr_p(int* p) { ++*p; }          // pass a pointer (dereference it and increment the result)
void incr_r(int& r) { ++r; }           // pass a reference
```

How do you choose? We think returning the value often leads to the most obvious (and therefore least error-prone) code; that is:

```
int x = 2;
x = incr_v(x);      // copy x to incr_v(); then copy the result out and assign it
```

We prefer that style for small objects, such as an **int**. In addition, if a "large object," such as **vector**, has a move constructor (§17.4.4) we can efficiently pass it back and forth.

How do we choose between using a reference argument and using a pointer argument? Unfortunately, either way has both attractions and problems, so again the answer is less than clear-cut. You have to make a decision based on the individual function and its likely uses.

If you use a pointer as a function argument, the function has to beware that someone might call it with a null pointer, that is, with a **nullptr**. For example:

XX

```
incr_p(nullptr);    // crash: incr_p() will try to dereference the null pointer
int* p = nullptr;
incr_p(p);          // crash: incr_p() will try to dereference the null pointer
```

This is obviously nasty. The person who writes **incr_p()** can protect against this:

```
void incr_p(int* p)
{
    if (p==nullptr)
        error("null pointer argument to incr_p()");
    ++*p;           // dereference the pointer and increment the object pointed to
}
```

But now **incr_p()** suddenly doesn't look as simple and attractive as before. Chapter 4 discusses how to cope with bad arguments. In contrast, users of a reference (such as **incr_r()**) are entitled to assume that a reference refers to an object.

If "passing nothing" (passing no object) is acceptable from the point of view of the semantics of the function, we must use a pointer argument. However, "passing nothing" is not acceptable for our increment operation – hence the need for throwing an exception for **p==nullptr**.

So, the real answer is: "The choice depends on the nature of the function": AA

- For tiny objects prefer pass-by-value. By "tiny," we mean one or two values of built-in types (e.g., two **double**s), or one or two class objects of equivalent size.

- For functions where "no object" (represented by **nullptr**) is a valid argument use a pointer parameter (and remember to test for **nullptr**).
- Otherwise, use a reference parameter.

See also §7.4.6, and don't forget pass-by-**const**-reference.

16.2.2 Pointers as parameters

XX Pointers are popular as parameters. For example:

```
void print_n(int* p, int n)
{
    if (p==nullptr)        // protect against nullptr
        return;

    for (int i = 0; i<n; ++i)
        cout << v[i] << ' ';
}

void user()
{
    int a[12];
    int* p = new int [10];
    // ... fill a and *p ...
    print_n(a,12);
    print_n(p,12);
}
```

There is much wrong with this code: verbosity, magic constants (§3.3.1), a memory leak (§15.4.5), and a range error (§4.6.2).

AA The root of the problem is that a pointer (by definition) doesn't "know" how many elements it points to is used as a parameter. We recommend that you don't pass a sequence of elements as a (pointer,count) pair. A pointer argument should be assumed to point to a single object or be the **nullptr**. Instead, pass an object that represents a range, e.g., a **span** (§16.4.1). For example:

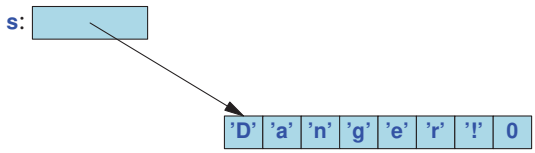
```
void print_n(span<int> s)      // a span points to an array and "remembers" its number of elements (§16.4.1)
{
    for (int x : s)
        cout << x << ' ';
}

void user()
{
    int a[12];
    vector<int> v(10);
    // ... fill a and v ...
    print_n(a);            // prints 12 ints
    print_n(v);            // prints 10 ints
}
```

16.3 C-style strings

Since the earliest days of the C language, a character string has been represented in memory as a zero-terminated array of characters and accessed through pointers (PPP2.§27.5). For example:

```
const char* s = "Danger!";
```

We can represent **s** like this:



Our literal strings are C-style strings. Note that the characters of a literal string are **const**. If we want to change the characters, we need to use an array:

```
char s[] = "Modifiable";
cout << "modifiable: " << s;          // writes "Modifiable"

s[6] = 'e';
s[7] = 'd';
s[8] = 0;          // terminating zero

cout << "modifiable: " << s;          // writes "Modified"
```

To get the number of characters in a C-style string, we use **strlen()**:

```
cout << strlen(s);    //      writes 8
```

Note that **strlen()** does not count the terminating **0**; it gives the number of characters up to, and not including, that **0**. That also means that if you place a **0** in the middle of a string, as we did for **s**, we can no longer determine how much memory was allocated for that string.

Our C-style "strings" are pointers and have pointer semantics. That implies that assignment, copy, and **sizeof** applies to the pointer, rather than what it points to. Consider:

```
string cat(const string& name, const string& addr)
{
      return id + '@' + addr;
}
```

Using C-style strings instead of **std::string**, this becomes:

```
char* cat2(const char* name, const char* addr)
{
      int nsz = strlen(name);
      int sz = nsz+strlen(addr)+2;          // +2 for '@' and the terminating zero
      char* res = (char*) malloc(sz);       // using the old allocation function
      strcpy(res,name);                     // copy from *name to *res until a zero is seen
      res[nsz+1] = '@';
```

```
        strcpy(res+2,addr);
        return res;
    }
```

The C and C++ standard-library function **malloc()** allocates memory on the free store; the memory allocated by **malloc()** must be given back to the free-store manager using **free()**. You often see this instead of **new**/**delete** in C-style code – **string**, **new**, and **delete** are not part of C. The C and C++ standard-library function **strcpy()** copies the characters pointed to by its second argument into the location pointed to by its first argument until it encounters a terminating zero. The terminating zero is also copied.

XX Such code is verbose and error-prone. It uses the messy pointer arithmetic, and who will **free()** the memory returned by **cat2()**? We strongly recommend the use of **std::string** or similar high-level strings over C-style strings. As a reader or maintainer of code, would you rather see **cat()** or **cat2()**?

Why would anyone design something like C-string? When C-style strings were invented, a computer's memory was measured in dozens of kilobytes or – if you were lucky – hundreds of kilobytes. I remember being ecstatic when I got hold of a computer with a whole megabyte! C-style strings were close to optimal in time and space for the kinds of programs written then. The reason is that they don't compute or store any information that you might not need. Also, the initial users of C-style strings were far better programmers than today's average. They simply didn't make most of the obvious programming mistakes.

AA Why would anyone write such code today? Well, there are a lot of C programmers who don't have any alternatives and often bring their habits to C++. Also, many programmers believe that they can write optimal code without making errors. They are almost always wrong.

Really, a C-style string has many features that might come as a surprise to someone unacquainted with it. C-style strings really are pointers. For example, **=** assigns pointers, rather than the values they point to.

You have been warned! Prefer **std::string** when you have the option. Look up additional tutorial information and documentation (e.g., [cppref]) if you don't.

16.4 Alternatives to pointer use

AA Pointers can be used for essentially anything. That's why we try to avoid their use: looking at pointer-heavy code we cannot reliably determine the intent of the programmer. That makes many uses of pointers error prone. Consider alternatives:

- To hold a collection of values, use a standard-library container, such as **vector**, **set** (§20.5), **map** (§20.2), **unordered_map** (§20.3), or **array** (§16.4.2).
- To hold a string of characters, use the standard-library **string** (§2.4, §9.10.3 PPP2.§23.2).
- To point to an object, you own (i.e., must **delete**), use the standard-library **unique_ptr** (§18.5.2) or **shared_ptr** (§18.5.3).
- To point to a contiguous sequence of elements that you don't own, use the standard-library **span** (§16.4.1).
- To systematically avoid dereferencing a null pointer, use **not_null** (§16.4.3}

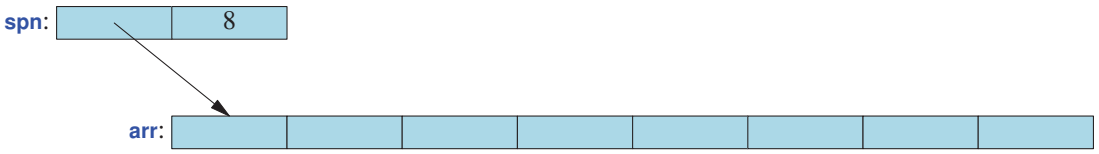
Often, there are alternatives to the standard-library components, but make the standard library as your default choice.

16.4.1 Span

Most of the uses of pointer involve keeping track of the number of elements that a pointer points to. Most of the problems that don't relate to ownership come from getting the number of elements wrong. So, we apply the obvious remedy: a "pointer" that keeps track of the number of its element. That type is called **span**. For example:

```
int arr[8];
span spn {arr};        // a span<int> that points to 8 ints
```

or graphically:



Here, **spn** was defined without mentioning the element type or the number of elements, those were deduced from the definition of **arr**. We don't always have that information available and we don't always want all of an array. Then, we must be explicit:

```
const int max = 1024;
int buf[max];
span<int>sp {buf,max/2};        // first half of buf
```

With **span**, we can get range checking and range-**for**:

```
void test(span<int> s)
{
        cout << "size: " << s.size() << '\n';
        for (int x : s)
                cout << x << '\n';
        try {
                int y = s[size()];
        }
        catch (...) {
                cout << "we have range checking\n";
                return;
        }
        cout << "no range checking! Boo Hoo!\n";
        terminate();            // exit the program
}
```

Unfortunately, **std::span** is not guaranteed to range check, though the version in **PPP_support** does.

16.4.2 array

The built-in array converts to a pointer at the slightest provocation. The resulting pointer lacks size information and can be confused with pointers that need to be **deleted**. However, the standard library provides an **array** type that doesn't implicitly produce a pointer. For example:

```
std::array<int,8> arr { 0,1,2,3,4,5,6,7 };
int* p = arr;                          // error (and that's good)
```

Unlike vector and string, array is not a handle to elements allocated elsewhere:

For good and bad, like the built-in array, array is a simpler and less flexible type than vector. Its size is not stored in memory, but remembered by the type system. An array doesn't use the free store unless you create it using new. Instead, an array uses only the kind of memory (e.g., stack or static) in which you create it, and that can be important.

Unfortunately, the number of elements of an array is never deduced, so it must be explicitly stated. On the other hand, if there are more elements than initializers, the remainder is default initialized. For example:

```
array<string,4> as { "Hello", " ", "World!" };          // as[3] is the empty string
```

16.4.3 not_null

Consider a possible implementation of strlen():

```
int strlen(const char* p)
{
    if (p==nullptr)
        return 0;
    int n = 0;
    while (*p++)
        ++n;
    return n;
}
```

Is the test for nullptr necessary? If we don't have it, strlen(p), where p is a null pointer, will most likely cause a crash; if not, it will give a wrong result. If we do have it, should it give a result? That is, should we pretend that the (imaginary) string pointed to by nullptr has zero elements? Or would it be better to give an error (e.g., throw an exception)?

> TRY THIS
>
> Look up the definition of std::strlen() to see what the standard requires. Then, try char* p = nullptr; size_t x = strlen(p); to see what your implementation does.

Whenever we use pointers, the question of what to do with the nullptr immediately surfaces. Does nullptr have a defined meaning? Is it a bug? If it is a bug, who is responsible for catching it? The caller or the callee? Documentation or comments may give the answers, but we don't always read the manuals. PPP_support provides a simple solution: a type that checks if its argument is the nullptr and throws not_null_error if it is. After that check, a not_null behaves just like a pointer.
If strlen() does not allow nullptr, we could have defined it like this:

```
int strlen(not_null<const char*> p)
{
        int n = 0;
        while (*p++)
                ++n;
        return n;
}
```

If `strlen()` isn't defined like this – and being a C function from the 1970s it isn't – the implementer has to decide whether to trust the caller or defend against the possibility of a `nullptr` argument. In addition, every user would have to decide whether to make sure that no argument to `strlen` is `nullptr`. That's life at the lower levels of abstraction. Ideally, we stay out of such dilemmas by using higher-level types, such as `vector` and `string`. Where that's not possible, use `not_null` for pointer arguments that come from code you do not control.

16.5 An example: palindromes

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 comparing 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.

16.5.1 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 "strings 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.

16.5.2 Palindromes using arrays

What if we didn't have strings (or vectors), 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 and add zero after the last character read
        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. That terminating 0 character is needed because many popular operations on arrays of characters (C-style strings) assume a terminating zero. 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. AA

16.5.3 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;
}
```

This is arguably the cleanest is_palindrome() so far, but its simplicity has been achieved by pushing the task of getting the range right onto its users:

```
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)
        return (*first==*last) ? is_palindrome(first+1,last–1) : false;
    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 last characters is a palindrome.

16.5.4 Palindromes using span

Basically, a `span` is simply a different, and usually better, alternative to using arrays or pointers. For example:

```
bool is_palindrome(span<char> s)
{
    return (s.size()) ? is_palindrome(s.data(),s.data()+s.size()) : true;   // implemented using pointers
}
```

Like all standard-library types, `span` has more useful member functions than fit into this book. Here, we used `data()` that returns a pointer to the first element of the `span`. An empty `span` doesn't have a first element, we first checked `s`'s size.

This `is_palindrome()` is an example of how to map from one style to another, here from a modern style using `span` to an older one using pointers. In large, multi-year projects, it is not uncommon to have several styles present as facilities and fashions evolve. We can also map the other way:

```
bool is_palindrome(const char* first, const char* last)
{
    return is_palindrome(span<char>{first,last–first});                    // implemented using span
}
```

Unfortunately, there is no simple and reliable run-time test that the pair of pointers `first` and `last` is plausible, so it is generally better to use `span` throughout:

```
bool is_palindrome(span<char> s)
{
    if (s.size()<2)
        return true;
    return (s.front()==s.back()) ? is_palindrome(span<int>{s.data()+1,s.size()–2}) : false;
}
```

Drill

In this chapter, we have two drills: one to exercise arrays and one to exercise vectors 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 ints 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():
- Define a local int array la of ten ints.
- Copy the values from ga into la.
- Print out the elements of la.
- 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.
- Copy the values from the argument array into the free-store array.
- Print out the elements of the free-store array.
- Deallocate the free-store array.

[4] In main():
- Call f() with ga as its argument.
- 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.).
- Call f() with aa as its argument.

Standard-library vector drill:

[1] Define a global vector<int> gv; initialize it with ten ints, 1, 2, 4, 8, 16, etc.

[2] Define a function f() taking a vector<int> argument.

[3] In f():
- Define a local vector<int> lv with the same number of elements as the argument vector.
- Copy the values from gv into lv.
- Print out the elements of lv.
- Define a local vector<int> lv2; initialize it to be a copy of the argument vector.
- Print out the elements of lv2.

[4] In main():
- Call f() with gv as its argument.
- 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.).
- Call f() with vv as its argument.

Review

[1] What does "Caveat emptor!" mean?

[2] What is an array?

[3] How do you copy an array?

[4] How do you initialize an array?

[5] When should you prefer a pointer argument over a reference argument? Why?

[6] When should you prefer a **span** over a pointer? Why?
[7] How does **std::array** differ from a built-in array?
[8] What good is range checking?
[9] What information do you need to do range checking?
[10] What good can a **not_null** do?
[11] What is a C-style string?
[12] What is a palindrome?

Terms

array	pointer	palindrome	pointer arithmetic
span	**array**	**not_null**	C-style string
*	**&**	**->**	subscripting
[ ]	**strlen()**	range error	**nullptr** dereference

Exercises

[1] Write a function, **void to_lower(char* s)**, that replaces all uppercase characters in the C-style string **s** with their lowercase equivalents. For example, **Hello, World!** becomes **hello, world!**. Do not use any standard-library function. A C-style string is a zero-terminated array of characters, so if you find a **char** with the value **0** you are at the end.
[2] Write a function, **char* str_dup(const char*)**, that copies a C-style string into memory it allocates on the free store. Do not use any standard-library function.
[3] Write a function, **char* find_x(const char* s, const char* x)**, that finds the first occurrence of the C-style string **x** in **s**.
[4] Write a function, **int str_cmp(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.
[5] Consider what happens if you give your **str_dup()**, **find_x()**, and **str_cmp()** a pointer 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.
[6] See what happens if you give the standard-library function **strcmp()** a pointer argument that is not a C-style string.
[7] Write a function, **string cat_dot(const char* s1, const char* s2)**, that concatenates two strings with a dot in between. For example, **cat_dot("Niels", "Bohr")** will return a string containing **Niels.Bohr**.

[8] Write a version of **cat_dot()** that takes **const string&** arguments.

[9] Modify **cat_dot()** from the previous two exercises to take a string to be used as the separator (rather than dot) as its third argument.

[10] 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.

[11] Rewrite all the functions in §16.5 (palindromes) 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.

[12] Look at the "array solution" to the palindrome problem in §16.5.2. 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.

[13] 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 **PPP_support**) 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

Pointers and arrays are ubiquitous in C code and older C++ code. For example, string literals are C-style strings. Thus, we have to understand their use and learn how to avoid their misuses. The standard-library **span**, **array**, and **not_null** address many problems related to range errors and can be used where high-level types, such as **vector**, cannot be used consistently.

This page intentionally left blank

17

Essential Operations

When someone says
I want a programming language in which
I need only say what I wish done,
give him a lollipop.
– Alan Perlis

This chapter describes how vectors are copied and accessed through subscripting. To do that, we discuss copying in general and present the essential operations that must be considered for every type: construction, default construction, copy, move, and destruction. Like many types, **vector** offers comparisons, so we show how to provide operations such as **==** and **<**. Finally, we grapple with the problems of changing the size of a **vector**: why and how?

§17.1 Introduction
§17.2 Access to elements
§17.3 List initialization
§17.4 Copying and moving
Copy constructors; Copy assignments; Copy terminology; Moving
§17.5 Essential operations
Explicit constructors; Debugging constructors and destructors
§17.6 Other useful operations
Comparison operators; Related operators
§17.7 Remaining **Vector** problems
§17.8 Changing size
Representation; **reserve()** and **capacity()**; **resize()**; **push_back()**; Assignment
§17.9 Our **Vector** so far

17.1 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.

CC 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 PPP2.§22.2.5 and PPP2.Ch27). In Chapter 15, 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?

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.

17.2 Access to elements

The **Vector** as we left it at the end of Chapter 15 is still woefully incomplete compared to **std::vector**. In particular, it lacks an elegant way to access its elements. All it had was a pair of **get()** and **set()** functions. Code using **get()** and **set()** to access elements is rather ugly compared to the usual subscript notation:

```
Vector v(3);
v.set(0,1);
v.set(1,2);
v.set(2,3);
int x = v.get(2);
```

We can do better. The way to get the usual **v[i]** notation 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 (§3.3)
```

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 {
    // ...
    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) {
    *v[i] = i;          // works, but still too ugly
    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 a reference
    const double& operator[](int n) const { return elem[n]; }     // return a const& for a const (§8.7.4)
};
```

References are not just for function arguments.

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**.

Since **vector**s are often passed by **const** reference, the **const** version of **operator[]()** is an essential addition. That version could return a plain **double**, rather than a **double&**.

17.3 List initialization

Consider yet again our **Vector**:

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

That's fine, but we would like to initialize a **Vector** with a list of values. For example:

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

Just setting the size and then assigning the values we want is tedious and error-prone:

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

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. Its first element is identified by **begin()** and the end of the list by **end()**, 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[s]}         // uninitialized memory for elements
    {
        for (int i = 0; i<sz; ++i)
            elem[i] = 0.0;                  // initialize to a default value
    }
```

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

        // ...
};
```

We used the standard-library **copy** algorithm. It copies a sequence of elements identified by its first two arguments (here, the **initializer_list**'s **begin()** and **end()**) into the memory identified by its third argument (here, **elem**). This style of code is pervasive in the standard library, and is described briefly in §17.6 and in detail in Chapter 19 and Chapter 21.

Member initializers must appear in the order of the members themselves. This means that we can use a member as part of subsequent member initializers (here, as we did with **sz**).

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**. **CC**

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."

Naturally, when we can initialize a **Vector** with a list, we expect to be able to assign a list to a **Vector**. For example:

```
v1 = {7,8,9,0};
```

For that, we define **Vector::operator=(initializer_list<double>)**.

17.4 Copying and moving

Consider again our incomplete **Vector**:

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

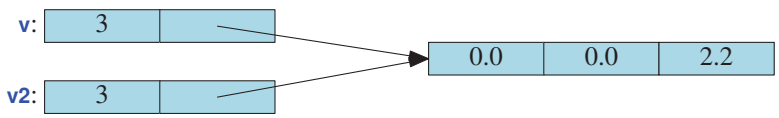
Let's try to copy one of these vectors:

```
void f(int n)
{
        Vector v(3);            // define a vector of 3 elements
        v[2] = 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 **i**s 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**. If you are lucky, your compiler will warn you.

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.

CC 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 **Vector**s in our example, it means that after the copy, we have **v.sz==v2.sz** and **v.elem==v2.elem** so that our **vector**s look like this:



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

```
v[1] = 99;                      // modify v
v2[0] = 88;                     // modify v2
cout << v[0] << ' ' << v2[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.

17.4.1 Copy constructors

So, what do we do? We 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&);          // copy a 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 (§7.4.4). So we refine **Vector** like this:

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

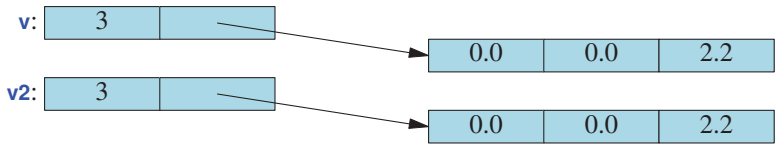
The copy constructor sets the number of elements (**sz**) and allocates memory for the elements (initializing **elem**) before copying element values from the argument **Vector**:

```
Vector::Vector(const Vector& arg)       // allocate elements, then initialize them by copying
        :sz{arg.sz}, elem{new double[arg.sz]}
{
        copy(arg.elem,arg.elem+sz,elem);  // copy elements [0:sz) from elem.arg into elem
}
```

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[1] = 99;                  // modify v
v2[0] = 88;                 // modify v2
cout << v[0] << ' ' << v2[1];
```

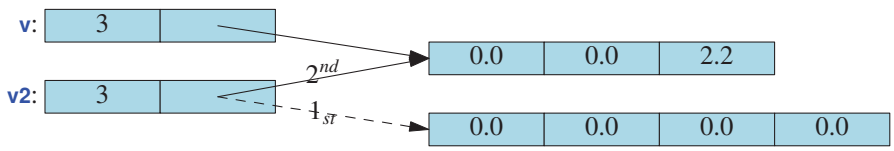
This will output **0 0**.

17.4.2 Copy assignments

CC 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 §17.4) plus a memory leak. For example:

```
void f2(int n)
{
    Vector v(3);            // define a vector
    v[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 §17.4 before we added the copy constructor: the elements pointed to by both **v** and **v2** are freed twice (implicitly using **delete[]**). In addition, we have leaked the memory initially allocated for **v2**'s four elements. We "forgot" to free those. If you ever make that mistake, hope for a compiler warning.

The remedy for this flawed copy assignment is fundamentally the same as for the flawed copy initialization: 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,p);         // copy elements [0:sz) from a.elem into p
        delete[] elem;                      // deallocate old space
        elem = p;                           // now we can reset elem
        sz = a.sz;
        return *this;                       // return a self-reference (§15.8)
}
```

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,p);         // copy elements [0:sz) from a.elem into p
```

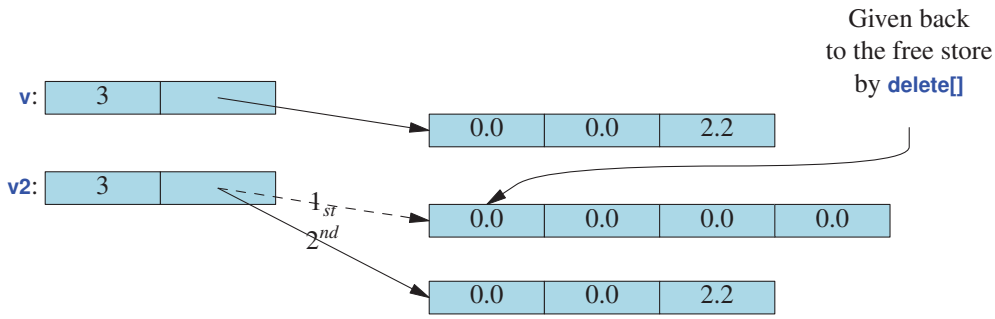
Then we free the old elements from the target **Vector**:

```
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

AA

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 optimally efficient).

17.4.3 Copy terminology

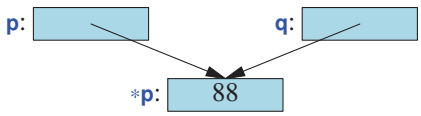
CC Copying is an issue in most programs and in most programming languages. The basic issue is whether you copy a pointer (or reference) or copy the information pointed to (referred to):

- *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
```

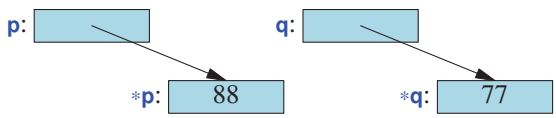
We can illustrate that like this:



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:



AA 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 **std::vector**, does a deep copy by allocating new space for the elements and copying their values. Types that provide shallow copy (like pointers) 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

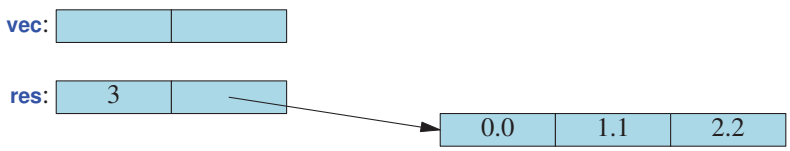
semantics behave as if no pointers were involved – just values that can be copied. One way of thinking of types with value semantics is that they "work just like integers" as far as copying is concerned.

17.4.4 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:

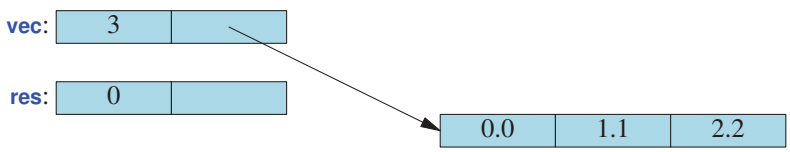
```
Vector fill(istream& is)
{
    Vector res;                     // assuming we have a default constructor (§17.5)
    for (double x; is>>x; )         // assuming support for range-for (§17.6)
        res.push_back(x);           // assuming we have a Vector::push_back (§17.8.4)
    return res;
}

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 how **res** would be represented in memory if we had entered **0.0 1.1 2.2**:



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. AA

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 3 **double**s out of **fill()** and into its caller at the cost of 4 single-word assignments. The cost would be the same had we moved 100,000 **double**s out of **fill()**.

In a simple case like this, a compiler can figure out how to do even better, and some have done so for decades. It simply constructs **res** in **vec**'s location. That way, there never is a separate **res** to copy. This is called *copy elision*. However, to avoid copying in every case, we need to express such a move in C++ code. How? We define move operations to complement the copy operations:

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

AA 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&& arg)
    :sz{arg.sz}, elem{arg.elem}          // copy arg's elem and sz
{
    arg.sz = 0;                          // make arg the empty Vector
    arg.elem = nullptr;
}

Vector& Vector::operator=(Vector&& arg)          // move arg to this Vector
{
    if (this != &arg) {                  // protect against self-assignment (e.g., v=v)
        delete[] elem;                   // deallocate old space
        elem = arg.elem;                 // copy arg's elem and sz
        sz = arg.sz;
        arg.elem = nullptr;              // make arg the empty Vector
        arg.sz = 0;
    }
    return *this;                        // return a self-reference (§15.8)
}
```

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;                          // assuming we have a default constructor (§17.5)
```

```
        for (double x; is>>x; )          // assuming support for range-for (§17.6)
            res.push_back(x);            // assuming we have a Vector::push_back (§17.8.4)
        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 copy.

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: XX

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

void use2()
{
    Vector* vec = fill2(cin);
    // ... use vec ...
    delete vec;
}
```

Now we have to remember to delete the **Vector**. As described in §15.4.5, deleting objects placed on the free store is not as easy to do consistently and correctly as it might seem.

17.5 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: CC

- Constructors from one or more arguments
- Default constructor (§17.5)
- Copy constructor (copy object of same type; §17.4.1)
- Copy assignment (copy object of same type; §17.4.2)
- Move constructor (move object of same type; §17.4.4)
- Move assignment (move object of same type; §17.4.4)
- Destructor (§15.5)

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 (§8.4.3). 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 every type **T**, **T{}** is the default value, if a default exists. For example, **double{}** is **0.0**, **string{}** is **""**, **vector<int>{}** is the empty **vector** of **int**s, and in §8.4.2 we made our **Date{}** January 1, 2001. For our **Vector** that would be:

```
class Vector {
public:
    Vector() : sz{0}, elem{nullptr} {}
    // ...
};
```

AA 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).

AA 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.

AA A class that needs a destructor almost always also needs a copy and/or move operations (constructor and assignment). The reason is 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.

AA In addition, a base class for which a derived class may have a destructor needs a **virtual** destructor (§15.5.2).

AA If a class needs any one of the essential operations, it probably needs all. This adds up to two popular rules of thumb:

- *Rule of zero*: If you don't need to, don't define any essential operation.
- *Rule of all*: if you need to define any essential operation, define them all.

That second rule is often called "the rule of three" or "the rule of five" because people don't agree on how to count operations (e.g., do you count assignment and construction as one or two? **const** and non-**const** versions?).

For example, a class like this doesn't need explicitly defined essential operations because the compiler correctly generates them from the ones provided by **string** and **vector**:

```
struct Club {
        string name;
        vector<Member> members;
};
```

17.5.1 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: AA

```
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: CC

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

Vector v = 10;          // error: no implicit int-to-Vector conversion
v = 20;                 // error: no implicit int-to-Vector conversion
Vector v(10);           // OK: considered explicit

void f(const Vector&);
f(10);                  // error: no implicit int-to-Vector conversion
f(Vector(10));          // OK: considered explicit
```

To avoid surprising conversions, we – like 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**.

17.5.2 Debugging constructors and destructors

AA 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<double>(2)**; rather we do something, such as returning a **vector** from a function, 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) { cout << this << "->" << s << ": " << val << " (" << nv << ")\n"; }

    X(){ out("X()",0); val=0; }                                 // default constructor
    X(int x) { out( "X(int)",x); val=x; }
    X(const X& x){out("X(X&) ",x.val); val=x.val;  }            // copy constructor
    X(X&& x){  out("X(X&&) ",x.val); val=x.val; x.val=0; }      // move constructor
```

```
        X& operator=(const X& x) { out("X copy assignment",x.val); val=x.val; return *this; }
        X& operator=(X&& x) { out("X move assignment",x.val); val=x.val; x.val=0; 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) { cout << "copy()\n"; return a; }
X copy2(X a) { cout << "copy2()\n"; X aa = a; return aa; }
X& ref_to(X & a) { cout << "ref_to()\n"; return a; }
X* make(int i) { cout << "make()\n";  X a(i); return new X(a); }

struct XX { X a; X b; };        // members

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 THIS
>
> Try executing that. Then remove the move operations and run it again. The compilers can be very clever at avoiding unnecessary copies. We really mean it: do run this example and try to explain 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. A compiler is allowed to assume that a copy operation copies and does nothing else but copy. Based on that, it can eliminate redundant copies (copy elision; §17.4.4). Similarly, a compiler assumes that move operations do nothing but move. AA

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.

AA 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 prevent such leaks.

17.6 Other useful operations

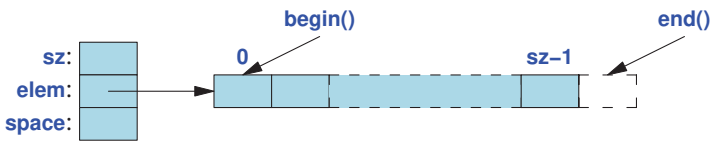
In addition to the essential operations, there are quite a few that are often important for common uses:

- Comparison operators, such as `==` and `<` (§17.6.1)
- `initializer_list` construction and assignment (§17.3)
- Iteration support functions, such as `begin()` and `end()`, as required for range-`for`
- `swap()` (§7.4.5, §18.4.3))

For `vector`s, `begin()` gives the position of the first element of a sequence and `end()` the one past the end location. For `Vector`, that becomes:

```
double* Vector::begin() const { return elem; }
double* Vector::end() const { return elem+sz; }
```

Or graphically:



The `begin()`/`end()` pair is used to implement traversal of elements by many algorithms (Chapter 21) and by range-`for`.

17.6.1 Comparison operators

We can define operators for our types, not just `[ ]` and `=`, but essentially all operators. When we define assignment, `=`, we typically also need to define equality, `==`, so that we can say what it means for the target of an assignment to have the same value as its source. Usually, `a=b` implies that after the assignment we have `a==b`. For our `Vector`, that's easily done:

```
bool operator==(const Vector& v1, const Vector& v2)
{
    if (v1.size()!=v2.size())
        return false;
    for (int i = 0; i<v1.size(); ++i)
        if (v1[i]!=v2[i])
            return false;
    return true;
}
```

Note the initial comparison of the number of elements. Without that, we could be wasting time comparing many elements before finding two that compared not equal. Also, having determined that the sizes are equal, we don't have to check both sizes as we loop through the elements. When dealing with containers, such simple optimizations can be most effective.

17.6.2 Related operators

Operators rarely stand by themselves, they come in "clusters" of related operators that jointly deliver a desired semantics:

- `+`, `-`, `*`, `/`, and sometimes `%` tend to go together.
- We saw that we needed `*`, `->`, and `[]` for pointers.
- When we have `==` we usually also want `!=`.
- When we have `=` we usually also want `==` and `!=`.
- `<`, `<=`, `>`, and `>=` are comparisons. When we have those, we usually also want `==` and `!=`.

Operators have conventional meanings, and we should be careful to conform to those. A `+` that subtracted would seriously confuse readers.

Often the easiest and most efficient way to build one operator from a cluster of related operators is in terms of one of the others. Here is a `!=` for our `Vector`:

```
bool operator!=(const Vector& v1, const Vector& v2)
{
    return !(v1==v2);
}
```

For historical reasons defining an operator is called *operator overloading* because doing so adds a meaning to that operator.

Naturally, the standard-library `std::vector` has comparison operators: `==`, `!=`, `<`, `<=`, `>`, and `>=`. All we have been doing here is to continue our efforts to build our `Vector` in the image of the standard-library one.

Other operators that you can define for your own types include

- `()` application/call
- `,` comma
- `<<` and `>>`
- `&` bitwise and, `|` bitwise or, `^` bitwise exclusive or, and `~` bitwise complement
- `&&` logical and, and `||` logical or
- but unfortunately not `.` (dot)

No, you can't define your own operators (e.g., ** or =~=) or redefine the meaning of operators on the built-in types (e.g., + on integers always adds). We decided that the added confusion from allowing such flexibility outweighed the benefits gained in relatively few cases.

17.7 Remaining Vector problems

Our Vector class has reached the point where we can

- Create Vectors of double-precision floating-point elements (objects of class Vector) with whatever number of elements we want.
- Copy our Vectors using assignment and initialization.
- Move our Vectors cheaply from one scope to another using assignment and initialization.
- Use initializer_lists for assignment and initialization.
- Rely on Vectors to correctly release their memory when they go out of scope.
- Access Vector elements using the conventional subscript notation (on both the right-hand side and the left-hand side of an assignment).
- Compare Vectors using operators == and !=.
- Support range-for with begin() and end().

That's all good and useful, but to reach the level of sophistication we expect (based on experience with std::vector), we need to address three more concerns:

- How do we change the size of a Vector (change the number of elements)?
- How do we catch and report out-of-range Vector element access?
- How do we specify the element type of a Vector as an argument?

For example, how do we define Vector so that this is legal, as it is for std::vector:

```
Vector<double> vd;          // elements of type double
for (double d; cin>>d; )
    vd.push_back(d);        // grow vd to hold all the elements

Vector<char> vc(100);       // elements of type char
int n = 0;
if (cin>>n && 0<n)
    vc.resize(n);           // make vc have n elements
```

CC Obviously, it is nice and useful to have Vectors that allow this, but why is it important from a programming point of view? What makes it interesting to someone collecting useful programming techniques for future use? We are using two kinds of flexibility. We have a single entity, the Vector, for which we can vary two things:

- The number of elements
- The type of elements

Those kinds of variability are useful in rather fundamental ways. We always collect data. Looking around my desk, I see piles of bank statements, credit card bills, and phone bills. Each of those is basically a list of lines of information of various types: strings of letters and numeric values. In front of me lies a phone; it keeps lists of phone numbers and names. In the bookcases across the room, there are shelf after shelf of books. Our programs tend to be similar: we have containers of elements of various types. We have many different kinds of containers (vector is just the most

widely useful), and they contain information such as phone numbers, names, transaction amounts, and documents. Essentially all the examples from my desk and my room originated in some computer program or another. The obvious exception is the phone: it *is* a computer, and when I look at the numbers on it, I'm looking at the output of a program just like the ones we're writing. In fact, those numbers may very well be stored in a **std::vector<Number>**.

Not all **vector**s have the same type of elements. We need **vector**s of **double**s, temperature readings, records (of various kinds), **string**s, operations, GUI buttons, **Shape**s, dates, pointers to **Window**s, etc. The possibilities are endless. We leave that problem to Chapter 18.

Not all containers have the same number of elements. Could we live with a **vector** that had its size fixed by its initial definition; that is, could we write our code without **push_back()**, **resize()**, and equivalent operations? Sure we could, and such vector-like containers can be most useful. However, that would put an unnecessary burden on the programmer: the basic trick for living with fixed-size containers is to move the elements to a bigger container when the number of elements grows too large for the initial size. For example, we could read into a **vector** without ever changing the size of a **vector** like this:

```
void grow(Vector& v)            // read elements into a vector without using push_back:
{
        int n = 0;                      // number of elements
        for (double d; cin>>d; ) {
                if (n==v.size()) {
                        Vector v2(v.size()+1);
                        for (int i; i<v.size(); ++i)
                                v2[i] = v[i];
                        v = v2;
                }
        }
        v[n] = d;                       // add the new element
}
```

That's not pretty. Are you convinced that we got it right? It's horrendously inefficient if we ever exceed the original size! One of the reasons to use containers, such as **vector**, is to do better; that is, we want our **Vector** to handle such size changes internally to save us – its users – the bother and the chance to make mistakes. In other words, we often prefer containers that can grow to hold the exact number of elements we happen to need. The standard-library equivalent to the code above is far simpler and also far more efficient:

```
void grow(vector<double>& v)    // use the standard library
{
        for (double d; cin>>d; )
                vd.push_back(d);
}
```

Are such changes of size common? If they are not, facilities for changing size are simply minor conveniences. However, such size changes are very common. The most obvious example is reading an unknown number of values from input. Other examples are collecting a set of results from a search (we don't in advance know how many results there will be) and removing elements from a collection one by one. Thus, the question is not whether we should handle size changes for

CC

containers, but how.

XX Why do we bother with changing sizes at all? Why not "just allocate enough space and be done with it"? That appears to be the simplest and most efficient strategy. However, it is that only if we can reliably allocate enough space without allocating grossly too much space – and we can't. When we can, we still have to keep track of how much of our pre-allocated space has been used so far. Experience shows that doing so is not trivial and a source of errors. Unless we carefully and systematically check for out-of-range access we suffer disasters. Thus, we prefer to let a properly designed and carefully implemented library take care of that.

There are many kinds of containers. This is an important point, and because it has important implications it should not be accepted without thought. Why can't all containers be **vector**s? If we could make do with a single kind of container (e.g., **vector**), we could dispense with all the concerns about how to program it and just make it part of the language. If we could make do with a single kind of container, we needn't bother learning about different kinds of containers; we'd just use **vector** all the time.

Well, data structures are the key to most significant applications. There are many thick and useful books about how to organize data, and much of that information could be described as answers to the question "How do I best store my data?" So, the answer is that we need many different kinds of containers, but it is too large a subject to adequately address here. However, we have already used **vector**s and **string**s (a **string** is a container of characters) extensively. In the next chapters, we will see **list**s, **map**s (a **map** is a tree of pairs of values), and matrices (PPP2.§24.5). Because we need many different containers, the language features and programming techniques needed to build and use containers are widely useful. In fact, the techniques we use to store and access data are among the most fundamental and most useful for all nontrivial forms of computing.

CC At the most basic memory level, all objects are of a fixed size and no types exist. What we do here is to introduce language facilities and programming techniques that allow us to provide containers of objects of various types for which we can vary the number of elements. This gives us a fundamentally useful degree of flexibility and convenience.

17.8 Changing size

What facilities for changing size does **std::vector** offer? It provides three simple operations. Given

```
vector<double> v;        // v.size()==0
```

we can change its size in three ways:

```
v.resize(10);            // v now has 10 elements

v.push_back(7);          // add an element with the value 7 to the end of v; v.size() increases by 1

v = v2;                  // assign another vector; v is now a copy of v2; v.size() now equals v2.size()
```

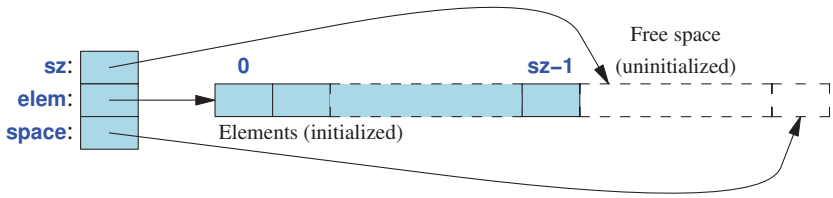
The standard-library **vector** offers more operations that can change a **vector**'s size, such as **erase()** and **insert()**, but here we will just see how we can implement **resize()**, **push_back()**, and **operator=()** for our **Vector**.

17.8.1 Representation

In §17.7, we show a simple, naive strategy for changing size: just allocate space for the new number of elements and copy the old elements into the new space. However, if you resize often, that's inefficient. In practice, if we change the size once, we usually do so many times. In particular, we rarely see just one **push_back()**. So, we can optimize our programs by anticipating such changes in size. In fact, all **vector** implementations keep track of both the number of elements and an amount of "free space" reserved for "future expansion." For example:

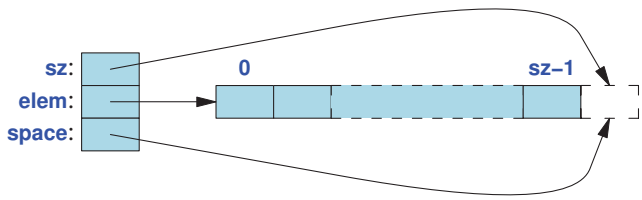
```
class Vector {
      // ...
private:
      int sz;                 // number of elements
      double* elem;           // address of first element
      int space;              // number of elements plus "free space"/"slots" for new elements
};
```

We can represent this graphically like this:



We number elements starting from **0**, so **sz** (the number of elements) refers to one beyond the last element and **space** refers to one beyond the last allocated slot. The pointers shown are really **elem+sz** and **elem+space**.

When a **Vector** is first constructed, **space==sz**; that is, there is no "free space":

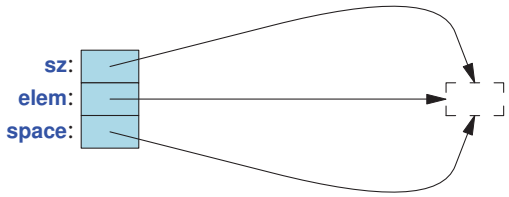


We don't start allocating extra slots until we begin changing the number of elements. Typically, **space==sz**, so there is no memory overhead unless we use **push_back()**.

The default constructor for **Vector** sets its integer members to **0** and its pointer member to **nullptr**:

```
Vector() :sz{0}, elem{nullptr}, space{0} {}
```

That gives



That one-beyond-the-end element is completely imaginary. The default constructor does no free-store allocation and occupies minimal storage (but see exercise 16).

Please note that our **Vector** illustrates techniques that can be used to implement a standard **vector** (and other data structures), but a fair amount of freedom is given to standard-library implementations so that **std::vector** on your system may use different techniques.

17.8.2 reserve() and capacity()

The most fundamental operation when we change sizes (that is, when we change the number of elements) is **Vector::reserve()**. That's the operation we use to add space for new elements:

```
void Vector::reserve(int newalloc)
{
        if (newalloc<=space)                    // never decrease allocation
                return;
        double* p = new double[newalloc];       // allocate new space
        for (int i=0; i<sz; ++i)                // copy old elements
                p[i] = elem[i];
        delete[] elem;                          // deallocate old space
        elem = p;
        space = newalloc;
}
```

Note that we don't initialize the elements of the reserved space. After all, we are just reserving space; using that space for elements is the job of **push_back()** and **resize()**.

Obviously, the amount of free space available in a **Vector** can be of interest to a user, so we (like the standard) provide a member function for obtaining that information:

```
int Vector::capacity() const { return space; }
```

That is, for a **Vector** called **v**, **v.capacity()–v.size()** is the number of elements we could **push_back()** to **v** without causing reallocation.

The standard-library **vector** never implicitly decreases its allocation. It assumes that if we ever needed an amount of memory, we are likely to do so again. In case we don't, **v.shrink_to_fit()** will reduce **capacity** to **size()**.

17.8.3 resize()

Given **reserve()**, implementing **resize()** for our **Vector** is fairly simple. We have to handle several cases:

- The new size is larger than the old allocation.
- The new size is larger than the old size, but smaller than or equal to the old allocation.
- The new size is equal to the old size.
- The new size is smaller than the old size.

Let's see what we get:

```
void Vector::resize(int newsize)
    // make the vector have newsize elements
    // initialize each new element with the default value 0.0
{
    reserve(newsize);
    for (int i=sz; i<newsize; ++i)          // initialize new elements
        elem[i] = 0;
    sz = newsize;
}
```

We let **reserve()** do the hard work of dealing with memory. The loop initializes new elements (if there are any).

We didn't explicitly deal with any cases here, but you can verify that all are handled correctly nevertheless.

> TRY THIS
>
> What cases do we need to consider (and test) if we want to convince ourselves that this **resize()** is correct? How about **newsize == 0**? How about **newsize == –77**?

17.8.4 push_back()

When we first think of it, **push_back()** may appear complicated to implement, but given **reserve()** it is quite simple:

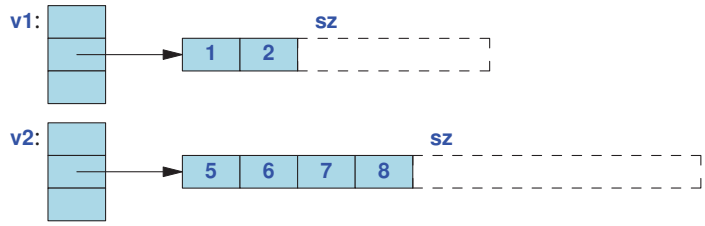
```
void Vector::push_back(double d)
    // increase vector size by one; initialize the new element with d
{
    if (space==0)                   // start with space for 8 elements
        reserve(8);
    else if (sz==space)
        reserve(2*space);           // get more space
    elem[sz] = d;                   // add d at end
    ++sz;                           // increase the size (sz is the number of elements)
}
```

In other words, if we have no spare space, we double the size of the allocation. In practice that turns out to be a very good choice for the vast majority of uses of **vector**, and that's the strategy used by most implementations of **std::vector**.

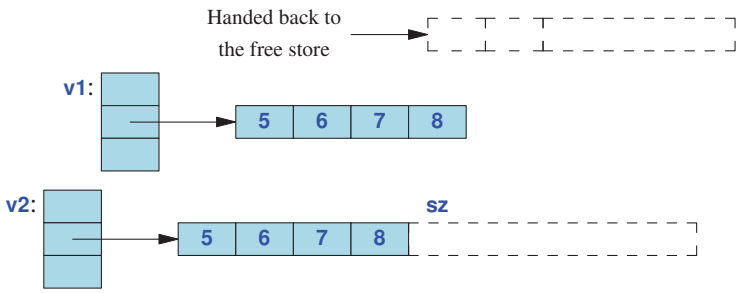
17.8.5 Assignment

We can define vector assignment in several different ways. For example, we could have decided that assignment was legal only if the two vectors involved had the same number of elements.

However, in §17.4 we decided that vector assignment should have the general and arguably the most obvious meaning: after assignment **v1=v2**, the vector **v1** is a copy of **v2**. Consider:



Obviously, we need to copy the elements as we did in §17.4.2, but what about the spare space? Do we "copy" the "free space" at the end? We don't: the new **Vector** will get a copy of the elements, but since we have no idea how that new **Vector** is going to be used, we don't bother with extra space at the end:



The simplest implementation of that is:

- Allocate memory for a copy.
- Copy the elements.
- Delete the old allocation.
- Set the **sz**, **elem**, and **space** to the new values.

Like this:

```
Vector& Vector::operator=(const Vector& a)
    // like copy constructor, but we must deal with old elements
{
    double* p = new double[a.sz];        // allocate new space
    for (int i = 0; i<a.sz; ++i)         // copy elements
        p[i] = a.elem[i];
    delete[] elem;                       // deallocate old space

    space = sz = a.sz;                   // set new size
    elem = p;                            // set new elements
    return *this;                        // return self-reference
}
```

By convention, an assignment operator returns a reference to the object assigned to. The notation for that is *this, which is explained in §15.8.

This implementation is correct, but when we look at it a bit, we realize that we do a lot of redundant allocation and deallocation. What if the Vector we assign to has more elements than the one we assign? What if the Vector we assign to has the same number of elements as the Vector we assign? In many applications, that last case is very common. In either case, we can just copy the elements into space already available in the target Vector:

```
Vector& Vector::operator=(const vector& a)
{
    if (this==&a)                    // self-assignment, no work needed
        return *this;

    if (a.sz<=space) {               // enough space, no need for new allocation
        for (int i = 0; i<a.sz; ++i) // copy elements
            elem[i] = a.elem[i];
        sz = a.sz;
        return *this;
    }

    double* p = new double[a.sz];    // allocate new space
    for (int i = 0; i<a.sz; ++i)     // copy elements
        p[i] = a.elem[i];
    delete[] elem;                   // deallocate old space

    space = sz = a.sz;               // set new size
    elem = p;                        // set new elements
    return *this;                    // return a self-reference
}
```

Here, we first test for self-assignment (e.g., v=v); in that case, we just do nothing. That test is logically redundant but sometimes a significant optimization. It does, however, show a common use of the this pointer: checking if the argument a is the same object as the object for which a member function (here, operator=()) was called.

Please convince yourself that this code actually works if we remove the this==&a case. The a.sz<=space is also just an optimization. Please convince yourself that this code actually works if we remove the a.sz<=space case.

17.9 Our Vector so far

Now we have an almost real Vctor of doubles:

```
class Vector {
/*
    invariant:
    if 0<=n<sz, elem[n] is element n
    sz<=space;
    if sz<space there is space for (space-sz) doubles after elem[sz-1]
*/
    int sz;                 // the size
    double* elem;           // pointer to the elements (or 0)
    int space;              // number of elements plus number of free slots
public:
    Vector() : sz{0}, elem{nullptr}, space{0} { }
    explicit Vector(int s) :sz{s}, elem{new double[s]}, space{s}
    {
        for (int i=0; i<sz; ++i)
            elem[i]=0;      // elements are initialized
    }

    Vector(initializer_list<double> lst);               // list initializer
    Vector& operator=(initializer_list<double> lst);    // list assignment

    Vector(const Vector&);                              // copy constructor
    Vector& operator=(const vector&);                   // copy assignment

    Vector(vector&&);                                   // move constructor
    Vector& operator=(Vector&&);                        // move assignment

    ~Vector() { delete[] elem; }                        // destructor

    double& operator[ ](int n) { return elem[n]; }      // access: return reference
    const double& operator[](int n) const { return elem[n]; }

    int size() const { return sz; }
    int capacity() const { return space; }

    void resize(int newsize);                           // growth
    void push_back(double d);
    void reserve(int newalloc);

    double* begin() const { return elem; }              // iteration support
    double* end() const { return elem+sz; }
};

bool operator==(Vector& v1, Vector &v2);
bool operator!=(Vector& v1, Vector &v2);
```

Note how it has the essential operations (§17.5): constructor, default constructor, copy operations, move operations, and destructor. It has an operation for accessing data (subscripting: `[ ]`) and for

providing information about that data (size() and capacity()) and for controlling growth (resize(), push_back(), and reserve()).

Drill

Write a class Ptr that has as a double* private member called p. Give Ptr the essential operations as described in §17.5. A constructor should take a double argument, allocate a double on the free store, assign the pointer to it to p, and copy the argument into *p. Give Ptr an operator * that allows you to read and write *p. Test Ptr.

Review

[1] What is the default meaning of copying for class objects?
[2] When is the default meaning of copying of class objects appropriate? Inappropriate?
[3] What is a copy constructor?
[4] What is a copy assignment?
[5] What is a move constructor?
[6] What is a move assignment?
[7] What is a default constructor?
[8] What is the difference between a copy constructor and a move constructor?
[9] What is the difference between a copy constructor and a copy assignment?
[10] What is shallow copy? What is deep copy?
[11] How does the copy of a vector compare to its source?
[12] What is the point of copy elision?
[13] What are the essential operations for a class?
[14] What is an explicit constructor?
[15] When would you prefer a constructor not to be explicit?
[16] How do you define traversal for a container?
[17] What operations may be invoked implicitly for a class object?
[18] What operators are often user-defined?
[19] What is *the rule of zero*?
[20] What is *the rule of all*?
[21] Why don't we just always define a vector with a large enough size for all eventualities?
[22] Which vector operations can change the size of a vector after construction?
[23] What is the difference between reserve() and resize()?
[24] How much spare space do we allocate for a new vector?
[25] When must we copy vector elements to a new location?
[26] What is the value of a vector after a copy?

Terms

reserve()	deep copy	shallow copy	move assignment
capacity	default constructor	move construction	list initialization
copy assignment	essential operations	begin()	end()
copy constructor	explicit constructor	&&	push_back()
resize()	size	comparison	traversal
rule of zero	rule of all	== and !=	

Exercises

[1] Define class Matrix to represent a two-dimensional matrix of doubles. A constructor should take two integer arguments specifying the number of rows and columns, e.g.,Matrix{3,4} has 3 rows and 4 columns. Provide Matrix with operators = (assignment), == (equality), [ ] (subscript), and + (addition of corresponding elements). The subscript operator should take a pairs of indices, e.g., m[2,3] yields the element 3 of the 2nd row. Indexing should be zero-based. Range check your indices. Reject operations on two Matrixs with different dimensions. If your compiler doesn't allow multiple arguments for [ ], use ( ) instead. Store the elements of your Matrix in a single vector. Test Matrix.

[2] Provide << and >> for your Matrix.

[3] Make the representation of Matrix private. What would be a more complete set of members for Matrix? For example, would you like a += operator? What would be a good set of constructors? Make a list and give a brief argument for each operation. Implement and test your more complete Matrix.

[4] Implement a row(i) member function that returns a vector that is a copy of the ith row. Implement a column(i) member function that returns a vector that is a copy of the ith column.

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 the essential operations: initialization, copying, moving, and destruction. In addition, we should consider what further operations are needed for that kind of type. For most types: == and !=. For containers: list initialization and assignment, begin(), end(), and swap(). The ultimate aim of such operations is to give the type a coherent and familiar semantics. Finally, we dramatically increase the flexibility of our Vector by adding operations that allow us to change a Vector's size: push_back(), resize(), and reserve().

The essential operations allow us to control the lifecycle of an object and to move objects between scopes. It is the foundation of reliable and efficient resource management. Other operators allow us to model concepts from application domains as types or sets of types. That's the basis for C++'s direct support for the kind of concepts we work with, the kind of entities that get drawn on our doodlepads and whiteboards.